Why use data.table?

- Concise and consistent syntax
  - Think in terms of `rows` , `columns` and `groups`
  - Provides a placeholder for each

```
# General form of data.table syntax
DT[i, j, by]
   |  |  |
   |  |  --> grouped by what?
   |  -----> what to do?
   --------> on which rows?
```

Creating a data.table

```r
library(data.table)
x_dt <- data.table(id = 1:2, name = c("a", "b"))
x_dt
```

```
id name
 1    a
 2    b
```

```r
y <- list(id = 1:2, name = c("a", "b"))
x <- as.data.table(y)
x
```

```
id name
 1    a
 2    b
```

Using column names to select columns

```r
ans <- batrips[, c("trip_id", "duration")]
head(ans, 2)
```

```
trip_id duration
139545      435
139546      432
```

```r
ans <- batrips[, "trip_id"]
# Still a data.table, not a vector
head(ans, 2)
```

```
trip_id
139545
139546
```

data.tables and data.frames (II)

Functions used to query data.frames also work on data.tables

```
nrow(x)
```

```
2
```

```
ncol(x)
```

```
2
```

```
dim(x)
```

```
2 2
```

Row numbers

```
# Subset 3rd and 4th rows from batrips
batrips[3:4]

# Same as
batrips[3:4, ]
```

```
# Subset everything except first five rows
batrips[-(1:5)]

# Same as
batrips[!(1:5)]
```

Logical expressions

```
# Subset rows where subscription_type is "Subscriber"
batrips[subscription_type == "Subscriber"]

# Subset rows where start_terminal = 58 and end_terminal is not 65
batrips[start_terminal == 58 & end_terminal != 65]

# Subset all rows where start_station starts with San Francisco
batrips[start_station %like% "^San Francisco"]

# Subset all rows where duration is between 2000 and 3000
batrips[duration %between% c(2000, 3000)]

# Subset all rows where start_station is "Japantown", "Mezes Park" or "MLK Library"
batrips[start_station %chin% c("Japantown", "Mezes Park", "MLK Library")]
```

Using column names to select columns

`j` argument accepts a character vector of column names

Column numbers instead of names work just ne

```r
ans <- batrips[, c(2, 4)]
head(ans, 2)
```

```
duration  start_station
435       San Francisco City Hall
432       San Francisco City Hall
```

However, we consider this a bad practice

```r
# If the order of columns changes, the result is wrong
batrips[, c(2, 4)]

# The result is always correct, no matter the order
batrips[, c("duration", "start_station")]
```

Deselecting columns with character vectors

- `-c("col1", "col2", ...)` deselects the speci ed columns
- Convenience feature only in data.table
- Using `!` instead of `-` works the same way

```
# Select all cols *except* those shown below
ans <- batrips[, -c("start_date", "end_date", "end_station")]
head(ans, 1)
```

```
trip_id  duration  start_station             start_terminal  bike_id  end_terminal
139545   435       San Francisco City Hall   58              65       473

subscription_type  zip_code
Subscriber         94612
```

Selecting columns the data.table way

Remember how columns were used as if they are variables in `i` argument in the last chapter?

```
# Recap the "i" argument
# All trips more than an hour
batrips[duration > 3600]
```

Similarly, you can use a list of variables (column names) to select columns

```
ans <- batrips[, list(trip_id, dur = duration)]
head(ans, 2)
```

```
trip_id    dur
139545     435
139546     432
```

When selecting a single column, not wrapping the variable by `list()` returns a `vector`

```
# Select a single column and return a data.table
ans <- batrips[, list(trip_id)]
head(ans ,2)
```

```
trip_id
139545
139546
```

```
# Select a single column and return a vector
ans <- batrips[, trip_id]
head(ans, 2)
```

```
139545 139546
```

Selecting columns the data.table way

`.()` is an alias to `list()`, for convenience

```
# .() is the same as list()
ans <- batrips[, .(trip_id, duration)]
head(ans, 2)
```

```
trip_id duration
139545      435
139546      432
```

Computing on rows and columns

Combining `i` and `j` is straightforward

```
# Compute mean of duration column for "Japantown" start station
batrips[start_station == "Japantown", mean(duration)]
```

```
2464.331
```

```
# How many trips started from "Japantown"?
batrips[start_station == "Japantown", .N]
```

```
902
```

Compute in j and return a data.table

Recall that you can select multiple columns using `.()`

```
# Recap: Select trip_id and duration columns
ans <- batrips[, .(trip_id, dur = duration)]
head(ans, 2)
```

```
trip_id      dur
139545      435
139546      432
```

You can compute on multiple columns and return a data.table the same way

```
# Get mean and median of duration
batrips[, .(mn_dur = mean(duration),
            med_dur = median(duration))]
```

```
mn_dur    med_dur
1131.967  511
```

The by argument

The `by` argument allows computations for each unique value of the (grouping) columns speci ed in `by`

```
# How many trips happened from each start_station?
ans <- batrips[, .N, by = "start_station"]
head(ans, 3)
```

```
           start_station        N
San Francisco City Hall      2145
 Embarcadero at Sansome     12879
      Steuart at Market     11579
```

The by argument

`by` argument accepts both `character` vector of column names as well as a `list` of variables/expressions

```
# Same as batrips[, .N, by = "start_station"]
ans <- batrips[, .N, by = .(start_station)]
head(ans, 3)
```

```
           start_station         N
San Francisco City Hall       2145
 Embarcadero at Sansome      12879
      Steuart at Market      11579
```

The by argument

Allows renaming grouping columns on the y

```
ans <- batrips[, .(no_trips = .N), by = .(start = start_station)]
head(ans, 3)
```

```
                  start  no_trips
San Francisco City Hall      2145
 Embarcadero at Sansome     12879
      Steuart at Market     11579
```

Expressions in by

The `list()` or `.()` expression in `by` allows for grouping variables to be computed on the   y

```
# Get number of trips for each start_station for each month
ans <- batrips[ , .N, by = .(start_station, mon = month(start_date))]
head(ans, 3)
```

```
          start_station mon    N
San Francisco City Hall   1  193
 Embarcadero at Sansome   1  985
      Steuart at Market   1  813
```

Chaining expressions

data.table expressions can be chained together, i.e., `x[...][...][...]`

```
# Same as
batrips[duration > 3600]
batrips[duration > 3600][order(duration)]
batrips[duration > 3600][order(duration)][1:3]
```

```
# Three start stations with the lowest mean duration
batrips[, .(mn_dur = mean(duration)),
        by = "start_station"][order(mn_dur)][1:3]
```

uniqueN()

- `uniqueN()` is a helper function that returns an integer value containing the number of unique values in the input object
- It accepts vectors as well as `data.frames` and `data.tables` .

```
id <- c(1, 2, 2, 1)
uniqueN(id)
```

```
2
```

```
x <- data.table(id, val = 1:4)
```

```
id val
 1   1
 2   2
 2   3
 1   4
```

```
uniqueN(x)
```

```
4
```

```
uniqueN(x, by = "id")
```

```
2
```

uniqueN() together with by

Calculate the total number of unique bike ids for every month

```
ans <- batrips[, uniqueN(bike_id), by = month(start_date)]
head(ans, 3)
```

```
month    V1   ## <~~ auto naming of cols
    1   605
    2   608
    3   631
```

Subset of Data, .SD

- `.SD` is a special symbol which stands for Subset of Data
- Contains subset of data corresponding to each group; which itself is a data.table
- By default, the grouping columns are excluded for convenience

```
x <- data.table(id = c(1, 1 ,2, 2, 1, 1),
                val1 = 1:6, val2 = letters[6:1])
```

```
id val1 val2
 1    1    f
 1    2    e
 2    3    d
 2    4    c
 1    5    b
 1    6    a
```

Subset of Data, .SD

```
x[, print(.SD), by = id]
```

```
val1 val2
   1    f
   2    e
   5    b
   6    a
val1 val2
   3    d
   4    c
Empty data.table (0 rows) of 1 col: id
```

Subset of Data, .SD

```
x[, .SD[1], by = id]
```

```
id val1 val2
 1    1    f
 2    3    d
```

Subset of Data, .SD

```
x[, .SD[.N], by = id]
```

```
id val1 val2
 1    6    a
 2    4    c
```

.SDcols

`.SDcols` holds the columns that should be included in `.SD`

```
batrips[, .SD[1], by = start_station]
```

```
          start_station  trip_id  duration          start_date
 San Francisco City Hall   139545       435  2014-01-01 00:14:00
  Embarcadero at Sansome   139547      1523  2014-01-01 00:17:00
```

```
# .SDcols controls the columns .SD contains
batrips[, .SD[1], by = start_station, .SDcols = c("trip_id", "duration")]
```

```
          start_station  trip_id  duration
San Francisco City Hall   139545       435
 Embarcadero at Sansome   139547      1523
```

.SDcols

```
batrips[, .SD[1], by = start_station, .SDcols = - c("trip_id", "duration")]
```

```
          start_station                start_date
San Francisco City Hall    2014-01-01 00:14:00
 Embarcadero at Sansome    2014-01-01 00:17:00
```

data.frame internals

- In v3.1.0, improvements were made to deep copy only the column that is updated
- In this case, just columns `a` and `b` are deep copied in the operation performed on `df` below

```
df <- data.frame(a = 1:3, b = 4:6, c = 7:9, d = 10:12)
df[1:2] <- lapply(df[1:2], function(x) ifelse(x%%2, x, NA))
df
```

```
 a  b c  d
 1 NA 7 10
NA  5 8 11
 3 NA 9 12
```

data.table internals

- `data.table` updates columns in place, i.e., by reference
- This means, you don't need the assign the result back to a variable
- No copy of any column is made while their values are changed
- `data.table` uses a new operator `:=` to add/update/delete columns by reference

LHS := RHS form

```
batrips[, c("is_dur_gt_1hour", "week_day") := list(duration > 3600,
                                                    wday(start_date))]
# When adding a single column quotes aren't necessary
batrips[, is_dur_gt_1hour := duration > 3600]
```

Functional form

```
batrips[, `:=`(is_dur_gt_1hour = NULL,
               start_station = toupper(start_station))]
```

Let's practice!

DATA MANIPULATION WITH DATA.TABLE IN R

Grouped aggregations

DATA MANIPULATION WITH DATA.TABLE IN R

Ma Dowle, Arun Srinivasan
Instructors, DataCamp

Combining ":=" with by

```
ncol(batrips)
```

```
11
```

```
batrips[, n_zip_code := .N, by = zip_code]
ncol(batrips)
```

```
12
```

```
batrips[, n_zip_code := .N, by = zip_code][]
```

```
trip_id duration ... zip_code  n_zip_code
 139545      435 ...    94612        1228
 139546      432 ...    94107       36061
 139547     1523 ...    94112        2168
```

Combining ":=" with by

```
batrips[, n_zip_code := .N, by = zip_code][]
```

```
trip_id duration ... zip_code  n_zip_code
 139545      435 ...    94612        1228
 139546      432 ...    94107       36061
 139547     1523 ...    94112        2168
```

```
batrips[n_zip_code > 1000]
```

```
bike_id subscription_type zip_code n_zip_code
    473        Subscriber    94612       1228
    395        Subscriber    94107      36061
    331        Subscriber    94112       2168
    335          Customer    94109       6980
    580          Customer                1541
    ...               ...      ...        ...
    677        Subscriber    94107      36061
    604        Subscriber    94133      15687
    480          Customer    94109       6980
    277          Customer    94109       6980
     56        Subscriber    94105      19899
```

Combining ":=" with by

```
batrips[, n_zip_code := .N, by = zip_code]


zip_1000 <- batrips[n_zip_code > 1000][, n_zip_code := NULL]
```

```
# Same as
zip_1000 <- batrips[, n_zip_code := .N,
                    by = zip_code][n_zip_code > 1000][, n_zip_code := NULL]
```

Let's practice!

DATA MANIPULATION WITH DATA.TABLE IN R

Advanced aggregations

DATA MANIPULATION WITH DATA.TABLE IN R

Ma Dowle, Arun Srinivasan
Instructors, DataCamp

Recap

```
# Same example as seen before
## LHS := RHS Form
batrips[, c("is_dur_gt_1hour", "week_day") :=
            .(duration > 3600, wday(start_date)]
```

```
# Same as above, but in `:=`() functional form
batrips[, `:=`(is_dur_gt_1hour = duration > 3600,
               week_day = wday(start_date))]
```

```
# Update by reference with by
batrips[, n_zip_code := .N, by = zip_code]
```

Adding multiple columns by reference by group

```
# Functional form
batrips[, `:=`(end_dur_first = duration[1],
               end_dur_last  = duration[.N]),
        by = end_station]
# LHS := RHS form
batrips[, c("end_dur_first",
            "end_dur_last") := list(duration[1], duration[.N]),
        by = end_station]
batrips[1:5]
```

```
   trip_id duration ...        end_station ... end_dur_first end_dur_last
    139545      435 ...    Townsend at 7th ...           435          660
    139546      432 ...    Townsend at 7th ...           435          660
    139547     1523 ...    Beale at Market ...          1523          229
    139549     1620 ... Powell Street BART ...          1620          540
    139550     1617 ... Powell Street BART ...          1620          540
```

Binning values

For each unique combination of `start_station` and `end_station`, if median duration:

- less than 600, "short"
- between 600 and 1800, "medium"
- "long", otherwise

Multi-line expressions in j

```
batrips[, trip_category := {
             med_dur = median(duration, na.rm = TRUE)
             if (med_dur < 600) "short"
             else if (med_dur >= 600 & med_dur <= 1800) "medium"
             else "long"
           },
        by = .(start_station, end_station)]
batrips[1:3]
```

```
trip_id duration ... zip_code trip_category
 139545      435 ...    94612         short
 139546      432 ...    94107         short
 139547     1523 ...    94112         short
```

Alternative way

```
bin_median_duration <- function(dur) {
  med_dur <- median(dur, na.rm = TRUE)
  if (med_dur < 600) "short"
  else if (med_dur >= 600 & med_dur <= 1800) "medium"
  else "long"
}

batrips[, trip_category := bin_median_duration(duration),
           by = .(start_station, end_station)]
```

All together - i, j and by

```
batrips[duration > 500, min_dur_gt_500 := min(duration),
           by = .(start_station, end_station)]
batrips[1:3]
```

```
trip_id duration ... zip_code min_dur_gt_500
 139545      435 ...    94612             NA
 139546      432 ...    94107             NA
 139547     1523 ...    94112            502
```

Let's practice!

DATA MANIPULATION WITH DATA.TABLE IN R

Fast data reading with fread()

DATA MANIPULATION WITH DATA.TABLE IN R

Ma Dowle, Arun Srinivasan
Instructors, DataCamp

Blazing FAST!

- Fast and parallel le reader
- Argument `nThread` controls the number of threads to use

User-friendly

- Can import local les, les from the web, and strings
- Intelligent defaults - `colClasses` , `sep` , `nrows` etc.
- Note: Dates and Datetimes are read as character columns but can be converted later with the excellent `fasttime` or `anytime` packages

Fast and friendly file reader

```r
# File from URL
DT1<-fread("https://bit.ly/2RkBXhV")
DT1
```

```
a b
1 2
3 4
```

```r
# Local file
DT2 <- fread("data.csv")
DT2
```

```
a b
1 2
3 4
```

```r
# String
DT3 <- fread("a,b\n1,2\n3,4")
DT3
```

```
a b
1 2
3 4
```

```r
# String without col names
DT4 <- fread("1,2\n3,4")
DT4
```

```
V1 V2
1  2
3  4
```

nrows and skip arguments

```
# Read only first line (after header)
fread("a,b\n1,2\n3,4", nrows = 1)
```

```
a b
1 2
```

```
# Skip first two lines containing metadata
str <- "# Metadata\nTimestamp: 2018-05-01 19:44:28 GMT\na,b\n1,2\n3,4"
fread(str, skip = 2)
```

```
a b
1 2
3 4
```

More on nrows and skip arguments

```
str <- "# Metadata\nTimestamp: 2018-05-01 19:44:28 GMT\na,b\n1,2\n3,4"
fread(str, skip = "a,b")
```

```
a b
1 2
3 4
```

```
fread(str, skip = "a,b", nrows = 1)
```

```
a b
1 2
```

select and drop arguments

```r
str <- "a,b,c\n1,2,x\n3,4,y"
fread(str, select = c("a", "c"))

# Same as
fread(str, drop = "b")
```

```
a c
1 x
3 y
```

```r
str <- "1,2,x\n3,4,y"
fread(str, select = c(1, 3))

# Same as
fread(str, drop = 2)
```

```
V1 V3
 1  x
 3  y
```

Let's practice!

DATA MANIPULATION WITH DATA.TABLE IN R

Advanced file reading

DATA MANIPULATION WITH DATA.TABLE IN R

Ma Dowle, Arun Srinivasan
Instructors, DataCamp

Reading big integers using integer64 type

- By default, R can only represent numbers less than or equal to 2^31 - 1 = 2147483647
- Large integers are automatically read in as `integer64` type, provided by the `bit64` package

```
ans <- fread("id,name\n1234567890123,Jane\n5284782381811,John\n")
ans
```

```
           id name
1234567890123 Jane
5284782381811 John
```

```
class(ans$id)
```

```
"integer64"
```

Specifying column class types with colClasses

```
str <- "x1,x2,x3,x4,x5\n1,2,1.5,true,cc\n3,4,2.5,false,ff"
ans <- fread(str, colClasses = c(x5 = "factor"))
str(ans)
```

```
Classes ‘data.table’ and 'data.frame':    2 obs. of  5 variables:
 $ x1: int  1 3
 $ x2: int  2 4
 $ x3: num  1.5 2.5
 $ x4: logi  TRUE FALSE
 $ x5: Factor w/ 2 levels "cc","ff": 1 2
```

Specifying column class types with colClasses

```
ans <- fread(str, colClasses = c("integer", "integer",
                                 "numeric", "logical", "factor"))
str(ans)
```

```
Classes ‘data.table’ and 'data.frame':    2 obs. of  5 variables:
 $ x1: int  1 3
 $ x2: int  2 4
 $ x3: num  1.5 2.5
 $ x4: logi  TRUE FALSE
 $ x5: Factor w/ 2 levels "cc","ff": 1 2
```

Specifying column class types with colClasses

```
str <- "x1,x2,x3,x4,x5,x6\n1,2,1.5,2.5,aa,bb\n3,4,5.5,6.5,cc,dd"
ans <- fread(str, colClasses = list(numeric = 1:4, factor = c("x5", "x6")))
str(ans)
```

```
Classes ‘data.table’ and 'data.frame': 2 obs. of 6 variables:
 $ x1: num  1 3
 $ x2: num  2 4
 $ x3: num  1.5 5.5
 $ x4: num  2.5 6.5
 $ x5: Factor w/ 2 levels "aa","cc": 1 2
 $ x6: Factor w/ 2 levels "bb","dd": 1 2
```

The fill argument

```
str <- "1,2\n3,4,a\n5,6\n7,8,b"
fread(str)
```

```
V1 5 6
 7 8 b
Warning message:
In fread(str) :
  Detected 2 column names but the data has 3 columns (i.e. invalid file).
  Added 1 extra default column name for the first column which is guessed to
  be row names or an index.
  Use setnames() afterwards if this guess is not correct,
  or fix the file write command that created the file to create a valid file.
```

The fill argument

```r
fread(str, fill = TRUE)
```

```
V1 V2 V3
 1  2
 3  4  a
 5  6
 7  8  b
```

The na.strings argument

Missing values are commonly encoded as: `"999"` or `"##NA"` or `"N/A"`

```
str <- "x,y,z\n1,###,3\n2,4,###\n#N/A,7,9"
ans <- fread(str, na.strings = c("###", "#N/A"))
ans
```

```
x   y   z
1  NA   3
2   4  NA
NA  7   9
```

Let's practice!

DATA MANIPULATION WITH DATA.TABLE IN R

Fast data writing with fwrite()

DATA MANIPULATION WITH DATA.TABLE IN R

Ma Dowle, Arun Srinivasan
Instructors, DataCamp

fwrite

Ability to write `list` columns using secondary separator (`|`)

```
dt <- data.table(id = c("x", "y", "z"), val = list(1:2, 3:4, 5:6))
fwrite(dt, "fwrite.csv")
fread("fwrite.csv")
```

```
id  val
 x  1|2
 y  3|4
 z  5|6
```

date and datetime columns (ISO)

- `fwrite()` provides three additional ways of writing date and datetime format - `ISO`, `squash` and `epoch`
- Encourages the use of ISO standards with `ISO` as default

Date and times

```
now <- Sys.time()
dt  <- data.table(date = as.IDate(now),
                  time = as.ITime(now),
                  datetime = now)
dt
```

```
      date     time            datetime
2018-12-17 19:54:51  2018-12-17 14:54:51
```

date and datetime columns (ISO)

```
# "ISO" is default
fwrite(dt, "datetime.csv", dateTimeAs = "ISO")


fread("datetime.csv")
```

```
      date     time                    datetime
2018-12-17 19:55:39 2018-12-17T19:55:39.735036Z
```

date and datetime columns (Squash)

- `squash` writes `yyyy-mm-dd hh:mm:ss` as `yyyymmddhhmmss` , for example
- Read in as integer. Very useful to extract month, year etc by simply using modulo arithmetic. e.g., `20160912 %/% 10000 = 2016`
- Also handles milliseconds (ms) resolution
- POSIXct type (17 digits with ms resolution) is automatically read in as `integer64` by `fread`

date and datetime columns (Squash)

```
fwrite(dt, "datetime.csv", dateTimeAs = "squash")

fread("datetime.csv")

       date   time           datetime
1: 20181217 195539 20181217195539735
```

```
20181217 %/% 10000
[1] 2018
```

date and datetime columns (Epoch)

- `epoch` counts the number of `days` (for dates) or seconds (for time and datetime) since relevant epoch
- Relevant epoch is `1970-01-01` , `00:00:00` and `1970-01-01T00:00:00Z` for `date` , `time` and `datetime` , respectively

date and datetime columns (Epoch)

```
fwrite(dt, "datetime.csv", dateTimeAs = "epoch")
fread("datetime.csv")
```

```
 date  time   datetime
17882 71871 1545076672
```

Let's practice!

DATA MANIPULATION WITH DATA.TABLE IN R

Welcome to the course

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie
Postdoctoral Researcher in Systems Genomics

Joining data.tables

- Combine information from two data.tables into a single data.table

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39

+

shipping:

name	address
Trey	12 High street
Matthew	7 Mill road
Angela	33 Pacific boulevard

→

name	gender	age	address
Trey	NA	54	12 High street
Matthew	M	43	7 Mill road
Angela	F	39	33 Pacific boulevard

Course overview

- Chapter 1: Joining data with `merge()`
- Chapter 2: Joins in the `data.table` work ow
- Chapter 3: Troubleshooting joins
- Chapter 4: Concatenating and reshaping `data.table` s

Table keys

Columns that link information across two tables

```r
library(data.table)
demographics <- data.table(name   = c("Trey", "Matthew", "Angela"),
                           gender = c(NA, "M", "F"),
                           age    = c(54, 43, 39))

shipping <- data.table(name    = c("Matthew", "Trey", "Angela"),
                       address = c("7 Mill road", "12 High street",
                                   "33 Pacific boulevard"))
```

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39

+

name	address
Matthew	7 Mill road
Trey	12 High street
Angela	33 Pacific boulevard

→

name	gender	age	address
Trey	NA	54	12 High street
Matthew	M	43	7 Mill road
Angela	F	39	33 Pacific boulevard

Inspecting `data.tables` in your R session

The `tables()` function will show you all `data.tables` loaded in your R session

```
tables()
```

```
             NAME NROW NCOL MB             COLS KEY
1: demographics    3    3  0 name,gender,age
2:     shipping    3    2  0    name,address
Total: 0MB
```

Inspecting `data.tables` in your R session

The `str()` will show you the type of each column in a single `data.table`

```
str(demographics)
```

```
Classes ‘data.table’ and 'data.frame':    3 obs. of  3 variables:
 $ name  : chr  "Trey" "Matthew" "Angela"
 $ gender: chr  NA "M" "F"
 $ age   : num  54 43 39
 - attr(*, ".internal.selfref")=<externalptr>
```

Inspecting `data.tables` in your R session

```
demographics_all
```

```
         name sex age
  1:     Trey  NA  54
  2:  Matthew   M  43
  3:   Angela   F  39
  4: Michelle   F  63
  5:  Mohamed   M  26
 ---
102:  Patrick   M  27
103:      Wei   F  41
104:     Adam   M  33
105:  Somchai   M  53
106:     Alma   F  19
```

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

The merge function

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie
Postdoctoral Researcher in Systems Genomics

Joins

- Concept of joins come from database query languages (e.g. SQL).
- Four standard joins:
  - inner
  - full
  - le
  - right
- All four can be done using `merge()`

Inner join

Only keep observations that have information in both `data.tables`

```r
merge(x = demographics, y = shipping,
      by.x = "name", by.y = "name")
```

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

+

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

→

name	gender	age	address
Angela	F	39	33 Pacific boulevard
Matthew	M	43	7 Mill road
Trey	M	NA	12 High street

The by argument

Use `by` to avoid repeated typing of the same column name

```
merge(x = demographics, y = shipping,
      by = "name")
```

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

name	gender	age	address
Angela	F	39	33 Pacific boulevard
Matthew	M	43	7 Mill road
Trey	M	NA	12 High street

Full join

Keep all observations that are in either `data.table`

```
merge(x = demographics, y = shipping,
      by = "name", all = TRUE)
```

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

+

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

→

name	gender	age	address
Abdullah	NA	NA	3a Union street
Angela	F	39	33 Pacific boulevard
Matthew	M	43	7 Mill road
Michelle	F	63	NA
Trey	M	NA	12 High street

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

Left and right joins

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie
Postdoctoral Researcher in Systems Genomics

Left joins

Add information from the right `data.table` to the le `data.table`

```
merge(x = demographics, y = shipping, by = "name", all.x = TRUE)
```

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

+

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

→

name	gender	age	address
Angela	F	39	33 Pacific boulevard
Matthew	M	43	7 Mill road
Michelle	F	63	NA
Trey	M	NA	12 High street

Right joins

Add information from the le `data.table` to the right `data.table`

```
merge(x = demographics, y = shipping, by = "name", all.y = TRUE)
```

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

+

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

→

name	gender	age	address
Abdullah	NA	NA	3a Union street
Angela	F	39	33 Pacific boulevard
Matthew	M	43	7 Mill road
Trey	M	NA	12 High street

Right joins - Left joins

```
# Right join
merge(x = demographics, y = shipping, by = "name", all.y = TRUE)

# Same as
merge(x = shipping, y = demographics, by = "name", all.x = TRUE)
```

Default values

- Default values for `all` , `all.x` and `all.y` are `FALSE` in the `merge()` function
- Look up function argument defaults using `help("merge")`

Exercise instructions

Le join `shipping` to `demographics`:

```
merge(demographics, shipping, by = "name", all.x = TRUE)
```

Right join `shipping` to `demographics`:

```
merge(demographics, shipping, by = "name", all.y = TRUE)
```

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

data.table syntax

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

datacamp

Recap of the data.table syntax

General form of `data.table` syntax

```
DT[i, j, by]
   |  |  |
   |  |  --> grouped by what?
   |  -----> what to do?
   --------> on which rows?
```

Joins

General form of `data.table` syntax joins

```
DT[i, on]
   |   |
   |   ----> join key columns
   --------> join to which data.table?
```

Right joins

The default join is a right join

```
demographics[shipping, on = .(name)]
```

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

NA

name	gender	age	address
Matthew	M	43	7 Mill road
Trey	NA	54	12 High street
Abdullah	NA	NA	3a Union street
Angela	F	39	33 Pacific boulevard

The on argument

Variables inside `list()` or `.()` are looked up in the column names of both `data.tables`

```
shipping[demographics, on = list(name)]
shipping[demographics, on = .(name)]
```

Character vectors can also be used

```
join_key <- c("name")
shipping[demographics, on = join_key]
```

Left joins

Remember, a le join is the same as a right join with the order swapped:

```
shipping[demographics, on = .(name)]
```

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

NA

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

name	address	gender	age
Trey	12 High street	NA	54
Matthew	7 Mill road	M	43
Angela	33 Pacific boulevard	F	39
Michelle	NA	F	63

Inner joins

Set `nomatch = 0` to perform an inner join:

```
shipping[demographics, on = .(name), nomatch = 0]
```

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

name	address	gender	age
Trey	12 High street	NA	54
Matthew	7 Mill road	M	43
Angela	33 Pacific boulevard	F	39

Full joins

Not possible with the `data.table` syntax, use the `merge()` function:

```
merge(demographics, shipping, by = "name", all = TRUE)
```

demographics:

name	gender	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

+

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

→

name	gender	age	address
Abdullah	NA	NA	3a Union street
Angela	F	39	33 Pacific boulevard
Matthew	M	43	7 Mill road
Michelle	F	63	NA
Trey	M	NA	12 High street

Anti-joins

Filter a `data.table` to rows that have no match in another `data.table`

```
demographics[!shipping, on = .(name)]
```

demographics:

name	sex	age
Trey	NA	54
Matthew	M	43
Angela	F	39
Michelle	F	63

anti join →

shipping:

name	address
Matthew	7 Mill road
Trey	12 High street
Abdullah	3a Union street
Angela	33 Pacific boulevard

→

name	sex	age
Michelle	F	63

Let's practice!

Setting and viewing data.table keys

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

datacamp

Setting `data.table` keys

Se ing keys means you don't need the `on` argument when performing a join

- Useful if you need to use a `data.table` in many di erent joins

Sorts the `data.table` in memory by the key column(s)

- Makes ltering and join operations faster

Multiple columns can be set and used as keys

The `setkey()` function

Key columns are passed as arguments

```
setkey(DT, ...)
```

```
setkey(DT, key1, key2, key3)
setkey(DT, "key1", "key2", "key3")
```

```
# To set all columns in DT as keys
setkey(DT)
```

The `setkey()` function

Set the keys of both `data.tables` before a join

```
setkey(dt1, dt1_key)
setkey(dt2, dt2_key)
```

Perform an inner, right, and le join:

```
# Inner join dt1 and dt2
dt1[dt2, nomatch = 0]
# Right join dt1 and dt2
dt1[dt2]
# Left join dt1 and dt2
dt2[dt1]
```

Setting keys programmatically

Key columns are provided as a character vector

```
keys <- c("key1", "key2", "key3")
setkeyv(dt, keys)
```

Getting keys

`haskey()` checks whether you have set keys

```
haskey(dt1)
```

```
TRUE
```

`key()` returns the key columns you have set

```
key(dt1)
```

```
"dt1_key"
```

Getting keys

When no keys are set

```
haskey(dt_no_key)
```

```
FALSE
```

```
key(dt_no_key)
```

```
NULL
```

Viewing all `data.tables` and their keys

```
tables()
```

```
     NAME         NROW NCOL MB COLS                         KEY
[1,] dt              3    4  1 key1,key2,key3,value         key1,key2,key3
[2,] dt1         1,000    3  1 dt1_key_column,value,group dt1_key
[3,] dt2         1,000    2  1 dt2_key_column,time          dt2_key
[4,] dt_no_key       5    2  1 id,color
Total: 4MB
```

Let's practice!

Incorporating joins into your data.table workflow

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

datacamp

Chaining data.table expressions

data.table expressions can be chained in sequence:

```
demographics[...][...]
```

General form of chaining a join:

```
DT1[DT2, on][i, j, by]
     |    |   |  |  |
     |    |   |  |   --> grouped by what?
     |    |   |   -----> what to do?
     |    |    --------> on which rows?
     |     ------------> join key columns
      -----------------> join to which data.table?
```

Join then compute

```r
customers <- data.table(name   = c("Mark", "Matt", "Angela", "Michelle"),
                        gender = c("M", "M", "F", "F"),
                        age    = c(54, 43, 39, 63))
customers
```

```
       name gender age
1:     Mark      M  54
2:     Matt      M  43
3:   Angela      F  39
4: Michelle      F  63
```

Join then compute

```
purchases <- data.table(name  = c("Mark", "Matt", "Angela", "Michelle"),
                        sales = c(1, 5, 4, 3),
                        spent = c(41.70, 41.78, 50.77, 60.01))
purchases
```

```
       name sales spent
1:     Mark     1 41.70
2:     Matt     5 41.78
3:   Angela     4 50.77
4: Michelle     3 60.01
```

Join then compute

```
customers[purchases,
          on = .(name)][sales > 1,
                        j = .(avg_spent = sum(spent) / sum(sales)),
                        by = .(gender)]
```

```
   gender avg_spent
1:      M  13.91333
2:      F  20.00333
```

Computation with joins

Computation with joins:

```
DT1[DT2, on, j]
     |     |    |
     |     |     ----> what to do on the join result?
     |      ---------> using which columns as keys?
      --------------> join to which data.table?
```

E cient for large `data.tables` !

Joining and column creation

Column creation takes place in the main `data.table`:

```
customers[purchases, on = .(name), return_customer := sales > 1]
customers
```

```
       name gender age return_customer
1:     Mark      M  54           FALSE
2:     Matt      M  43            TRUE
3:   Angela      F  39            TRUE
4: Michelle      F  63            TRUE
```

Grouping by matches

`by = .EACHI` groups `j` by each row from DT2

```
DT1[DT2, on, j, by = .EACHI]
    |    |   |  |
    |    |   |   --> grouped by each match in DT1.
    |    |    -----> what to do on the join result?
    |     ---------> using which columns as keys?
     --------------> join to which data.table?
```

Grouping by matches

```
shipping[customers, on = .(name),
         j = .("# of shipping addresses" = .N),
         by = .EACHI]
```

shipping:

name	type	address
Mark	residential	34 Yarra drive
Matt	residential	2 Sunshine crescent
Matt	business	12 Commercial road
Angela	residential	33 Pacific boulevard

customers:

name	gender	age
Mark	M	54
Matt	M	43
Angela	F	39
Michelle	F	63

name	# of shipping addresses
Mark	1
Matt	2
Angela	1
Michelle	0

Grouping by columns with joins

Grouping by columns in the `by` restricts computation to the main data.table:

```
DT1[DT2, on, j, by]
     |    |   |  |
     |    |   |   --> grouped by what columns in DT1?
     |    |    -----> what to do on columns in DT1?
     |     ---------> using which columns as keys?
      --------------> join to which data.table?
```

Grouping by columns with joins

Join and calculate by group in `customers`:

```
customers[shipping, on = .(name),
          .(avg_age = mean(age)), by = .(gender)]
```

```
   gender  avg_age
1:      M 46.66667
2:      F 39.00000
```

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

Complex keys

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

datacamp

Misspecified joins

What happens when you don't use the correct columns for join keys?

- An error is thrown
- The result is a malformed `data.table`

Column type mismatch

Using join key columns with di erent types will error

```
customers[web_visits, on = .(age = name)]
```

```
Error in bmerge(i, x, leftcols, rightcols, io, xo, roll, rollends,
nomatch,  :
  typeof x.age (double) != typeof i.name (character)
```

customers:

name	gender	age	address
Madeline Martin	F	54	5 Market lane
Madeline Bernard	F	45	4 Jacaranda crescent
George Dimakos	M	39	2a Park square

+

web_visits:

name	date	duration
Madeline Martin	2018-05-02	5
Madeline Martin	2018-05-03	32
Madeline Bernard	2018-05-03	12
George Dimakos	2018-04-27	45

Column type mismatch

```
customers[web_visits, on = .(id)]
```

```
Error in bmerge(i, x, leftcols, rightcols, io, xo, roll, rollends,
nomatch,  :
  typeof x.id (integer) != typeof i.id(character)
```

customers:

id	name	gender	age	address
1	"Madeline Martin"	"F"	54	"5 Market lane"
2	"Madeline Bernard"	"F"	45	"4 Jacaranda crescent"
3	"George Dimakos"	"M"	39	"2a Park square"

+

web_visits:

id	name	date	duration
"1"	"Madeline Martin"	2018-05-02	5
"1"	"Madeline Martin"	2018-05-03	32
"2"	"Madeline Bernard"	2018-05-03	12
"3"	"George Dimakos"	2018-04-27	45

Malformed full joins - no common key values

```
merge(customers, web_visits, by.x = "address", by.y = "name", all = TRUE)
```

customers:

name	gender	age	address
Madeline Martin	F	54	5 Market lane
Madeline Bernard	F	45	4 Jacaranda crescent
George Dimakos	M	39	2a Park square

+

web_visits:

name	date	duration
Madeline Martin	2018-05-02	5
Madeline Martin	2018-05-03	32
Madeline Bernard	2018-05-03	12
George Dimakos	2018-04-27	45

=

address	name	gender	age	date	duration
2a Park square	George Dimakos	M	39	NA	NA
4 Jacaranda crescent	Madeline Bernard	F	45	NA	NA
5 Market lane	Madeline Martin	F	54	NA	NA
George Dimakos	NA	NA	NA	2018-04-27	45
Madeline Bernard	NA	NA	NA	2018-05-03	12
Madeline Martin	NA	NA	NA	2018-05-02	5
Madeline Martin	NA	NA	NA	2018-05-03	32

Malformed right and left joins - no common key values

```
customers[web_visits, on = .(address = name)]
```

customers:

name	gender	age	address
Madeline Martin	F	54	5 Market lane
Madeline Bernard	F	45	4 Jacaranda crescent
George Dimakos	M	39	2a Park square

+

web_visits:

name	date	duration
Madeline Martin	2018-05-02	5
Madeline Martin	2018-05-03	32
Madeline Bernard	2018-05-03	12
George Dimakos	2018-04-27	45

=

name	gender	age	address	date	duration
NA	NA	NA	Madeline Martin	2018-05-02	5
NA	NA	NA	Madeline Martin	2018-05-03	32
NA	NA	NA	Madeline Bernard	2018-05-03	12
NA	NA	NA	George Dimakos	2018-04-27	45

Malformed inner joins - no common key values

```
customers[web_visits, on = .(address = name), nomatch = 0]
```

customers:

name	gender	age	address
Madeline Martin	F	54	5 Market lane
Madeline Bernard	F	45	4 Jacaranda crescent
George Dimakos	M	39	2a Park square

+

web_visits:

name	date	duration
Madeline Martin	2018-05-02	5
Madeline Martin	2018-05-03	32
Madeline Bernard	2018-05-03	12
George Dimakos	2018-04-27	45

=

name	gender	age	address	date	duration

Malformed joins - coincidental common key values

```
customers[web_visits, on = .(age = duration), nomatch = 0]
```

customers:

name	gender	age	address
Madeline Martin	F	54	5 Market lane
Madeline Bernard	F	45	4 Jacaranda crescent
George Dimakos	M	39	2a Park square

\+

web_visits:

name	date	duration
Madeline Martin	2018-05-02	5
Madeline Martin	2018-05-03	32
Madeline Bernard	2018-05-03	12
George Dimakos	2018-04-27	45

=

name	gender	age	address	i.name	date
Madeline Bernard	F	45	4 Jacaranda crescent	George Dimakos	2018-04-27

Avoiding misspecified joins

> Learning what each column represents before joins will help you avoid errors

Keys with different column names

customers:

name	gender	age	address
Madeline Martin	F	54	5 Market lane
Madeline Bernard	F	45	4 Jacaranda crescent
George Dimakos	M	39	2a Park square

web_visits:

person	date	duration
Madeline Martin	2018-05-02	5
Madeline Martin	2018-05-03	32
Madeline Bernard	2018-05-03	12
George Dimakos	2018-04-27	45

```
merge(customers, web_visits, by.x = "name", by.y = "person")
customers[web_visits, on = .(name = person)]
customers[web_visits, on = c("name" = "person")]
key <- c("name" = "person")
customers[web_visits, on = key]
```

Multi-column keys

customers:

first	last	gender	age	address
Madeline	Martin	F	54	5 Market lane
Madeline	Bernard	F	45	4 Jacaranda crescent
George	Dimakos	M	39	2a Park square

web_visits:

first	last	date	duration
Madeline	Martin	2018-05-02	5
Madeline	Martin	2018-05-03	32
Madeline	Bernard	2018-05-03	12
George	Dimakos	2018-04-27	45

Multi-column keys

purchases:

name	date	item	units	price
Madeline Martin	2018-05-03	book	2	$15.00
Arthur Smith	2018-05-03	shelf	1	$30.00
Jaqueline Mary	2018-05-03	CD	1	$12.00
George Dimakos	2018-05-03	plant	3	$16.00
George Dimakos	2018-04-27	shelf	1	$30.00

web_visits:

name	date	duration
Madeline Martin	2018-05-02	5
Madeline Martin	2018-05-03	32
Madeline Bernard	2018-05-03	12
George Dimakos	2018-04-27	45

Specifying multiple keys with merge()

```
merge(purchases, web_visits, by = c("name", "date"))
```

```
merge(purchases, web_visits,
      by.x = c("name", "date"),
      by.y = c("person", "date"))
```

Specifying multiple keys with the data.table syntax

```
purchases[web_visits, on = .(name, date)]
purchases[web_visits, on = c("name", "date")]
```

```
purchases[web_visits, on = .(name = person, date)]
purchases[web_visits, on = c("name" = "person", "date")]
```

Final Slide

JOINING DATA WITH DATA.TABLE IN R

Problem columns

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

Common column names

parents:

name	gender	age
Sarah	F	41
Max	M	43
Qin	F	36

children:

parent	name	gender	age
Sarah	Oliver	M	5
Max	Sebastian	M	8
Qin	Kai-lee	F	7

Common column names

Using the `data.table` syntax

```
parents[children, on = .(name = parent)]
```

```
    name gender age    i.name i.gender i.age
1: Sarah      F  41    Oliver        M     5
2:   Max      M  43 Sebastian        M     8
3:   Qin      F  36   Kai-lee        F     7
```

Common column names with merge()

Using the `merge()` function

```
merge(x = children, y = parents, by.x = "parent", by.y = "name")
```

```
   parent      name gender.x age.x gender.y age.y
1:    Max Sebastian        M     8        M    43
2:    Qin   Kai-lee        F     7        F    36
3:  Sarah    Oliver        M     5        F    41
```

Adding context with your own suffixes

The `suffixes` argument can add useful context:

```
merge(children, parents, by.x = "parent", by.y = "name",
      suffixes = c(".child", ".parent"))
```

```
   parent       name gender.child age.child gender.parent age.parent
1:    Max  Sebastian            M         8             M         43
2:    Qin    Kai-lee            F         7             F         36
3:  Sarah     Oliver            M         5             F         41
```

Renaming columns

Rename all columns using `setnames()`

```
setnames(parents, c("parent", "parent.gender", "parent.age"))
setnames(parents, old = c("gender", "age"),
         new = c("parent.gender", "parent.age"))
parents
```

```
   parent parent.gender parent.age
1:  Sarah             F         41
2:    Max             M         43
3:    Qin             F         36
```

Joining with `data.frames`

Join keys for `data.frames` may be in the rownames

```
parents
```

```
      gender age
Sarah      F  41
Max        M  43
Qin        F  36
```

```
parents <- as.data.table(parents, keep.rownames = "parent")
parents
```

```
   parent gender age
1:  Sarah      F  41
2:    Max      M  43
3:    Qin      F  36
```

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

Duplicate matches

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

Join key duplicates

```
# Which bacteria could be found at both sites using any method?
site1_ecology[site2_ecology, on = .(genus)]
```

site1_ecology:

genus	count	method
Nitrosomonas	500	WGS
Nitrosomonas	620	16S
Rhizobium	360	WGS
Rhizobium	300	16S

site2_ecology:

genus	present	method
Nitrosomonas	TRUE	WGS
Nitrosomonas	TRUE	16S
Nitrosomonas	TRUE	Culture
Rhizobium	TRUE	WGS
Rhizobium	TRUE	16S
Rhizobium	FALSE	Culture

Error from multiplicative matches

```
site1_ecology[site2_ecology, on = .(genus)]
```

```
Error in vecseq(f__, len__, if (allow.cartesian || notjoin ||
!anyDuplicated(f__,  :
  Join results in 12 rows; more than 10 = nrow(x)+nrow(i). Check for
  duplicate key values in i each of which join to the same group in x over
  and over again. If that's ok, try by=.EACHI to run j for each group to
  avoid the large allocation. If you are sure you wish to proceed, rerun
  with allow.cartesian=TRUE. Otherwise, please search for this error message
  in the FAQ, Wiki, Stack Overflow and data.table issue tracker for advice.
```

Allowing multiplicative matches

`allow.cartesian = TRUE` allows the join to proceed:

```
# data.table syntax
site1_ecology[site2_ecology, on = .(genus), allow.cartesian = TRUE]
```

```
# merge()
merge(site1_ecology, site2_ecology, by = "genus", allow.cartesian = TRUE)
```

Allowing multiplicative matches

```
site1_ecology[site2_ecology, on = .(genus), allow.cartesian = TRUE]
```

```
           genus count method present i.method
 1: Nitrosomonas   500    WGS    TRUE      WGS
 2: Nitrosomonas   620    16S    TRUE      WGS
 3: Nitrosomonas   500    WGS    TRUE      16S
 4: Nitrosomonas   620    16S    TRUE      16S
 5: Nitrosomonas   500    WGS    TRUE  Culture
 6: Nitrosomonas   620    16S    TRUE  Culture
 7:    Rhizobium   360    WGS    TRUE      WGS
 8:    Rhizobium   300    16S    TRUE      WGS
 9:    Rhizobium   360    WGS    TRUE      16S
10:    Rhizobium   300    16S    TRUE      16S
11:    Rhizobium   360    WGS   FALSE  Culture
12:    Rhizobium   300    16S   FALSE  Culture
```

Missing values

Missing values (`NA`) will match all other missing values:

site1_ecology:

genus	count	method
Nitrosomonas	500	WGS
Rhizobium	360	WGS
NA	1000	WGS
NA	150	WGS
NA	0	WGS

site2_ecology:

genus	present	method
Nitrosomonas	TRUE	Culture
Rhizobium	TRUE	Culture
NA	TRUE	Culture
NA	TRUE	Culture
Azotobacter	TRUE	Culture
NA	TRUE	Culture

Filtering missing values

`!is.na()` can be used to lter rows with missing values

```
site1_ecology <- site1_ecology[!is.na(genus)]
site1_ecology
```

```
          genus count method
1: Nitrosomonas   500    WGS
2:    Rhizobium   360    WGS
```

```
site2_ecology <- site2_ecology[!is.na(genus)]
site2_ecology
```

```
          genus present  method
1: Nitrosomonas    TRUE Culture
2:    Rhizobium    TRUE Culture
3:  Azotobacter    TRUE Culture
```

Keeping only the first match

```
site1_ecology[site2_ecology, on = .(genus), mult = "first"]
```

site1_ecology:

genus	Year	count	method
Nitrosomonas	2018	620	16S
Nitrosomonas	2017	603	16S
Nitrosomonas	2016	591	16S
Rhizobium	2018	290	16S
Rhizobium	2017	300	16S
Rhizobium	2016	280	16S
Azotobacter	2018	1230	16S
Azotobacter	2017	0	16S
Azotobacter	2016	0	16S

site2_ecology:

genus	present	method
Nitrosomonas	TRUE	WGS
Rhizobium	TRUE	WGS
Azotobacter	FALSE	WGS

Keeping only the last match

```
children[parents, on = .(parent = name), mult = "last"]
```

site1_ecology:

genus	Year	count	method
Nitrosomonas	2018	620	16S
Nitrosomonas	2017	603	16S
Nitrosomonas	2016	591	16S
Rhizobium	2018	290	16S
Rhizobium	2017	300	16S
Rhizobium	2016	280	16S
Azotobacter	2018	1230	16S
Azotobacter	2017	0	16S
Azotobacter	2016	0	16S

site2_ecology:

genus	present	method
Nitrosomonas	TRUE	WGS
Rhizobium	TRUE	WGS
Azotobacter	FALSE	WGS

Identifying and removing duplicates

`duplicated()` : what rows are duplicates?

`unique()` : lter a `data.table` to just unique rows

The duplicated() function

Using values in all columns:

```
duplicated(site1_ecology)
```

```
FALSE FALSE FALSE FALSE
```

Using values in a subset of columns:

```
duplicated(site1_ecology,
           by = "genus")
```

```
FALSE TRUE FALSE TRUE
```

site1_ecology:

genus	count	method
Nitrosomonas	500	WGS
Nitrosomonas	620	16S
Rhizobium	360	WGS
Rhizobium	300	16S

The unique() function

```
unique(site1_ecology, by = "genus")
```

site1_ecology:

genus	count	method
Nitrosomonas	500	WGS
Nitrosomonas	620	16S
Rhizobium	360	WGS
Rhizobium	300	16S

genus	count	method
Nitrosomonas	500	WGS
Rhizobium	360	WGS

Changing the search order

`fromLast = TRUE` changes the direction of the search to start from the last row

```
duplicated(site1_ecology, by = "genus", fromLast = TRUE)
```

```
TRUE FALSE TRUE FALSE
```

```
unique(site1_ecology, by = "genus", fromLast = TRUE)
```

site1_ecology:

genus	count	method
Nitrosomonas	500	WGS
Nitrosomonas	620	16S
Rhizobium	360	WGS
Rhizobium	300	16S

genus	count	method
Nitrosomonas	620	16S
Rhizobium	300	16S

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

Concatenating data.tables

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie
Postdoctoral Researcher in Systems Genomics

datacamp

Same columns, different data.tables

Concatenating `data.tables`

sales_2015:

quarter	amount
1	$3,200,100
2	$2,950,000
3	$2,980,700
4	$3,420,000

sales_2016:

quarter	amount
1	$3,350,000
2	$3,000,300
3	$3,120,200
4	$3,670,000

sales:

year	quarter	amount
2015	1	$3,200,100
2015	2	$2,950,000
2015	3	$2,980,700
2015	4	$3,420,000
2016	1	$3,350,000
2016	2	$3,000,300
2016	3	$3,120,200
2016	4	$3,670,000

Concatenation functions

`rbind()` : concatenate rows from `data.tables` stored in di erent variables

`rbindlist()` : concatenate rows from a `list` of `data.tables`

The rbind() function

Concatenate two or more `data.tables` stored as variables

```
# ... takes any number of arguments
rbind(...)
rbind(sales_2015, sales_2016)
```

```
   quarter  amount
1:       1 3200100
2:       2 2950000
3:       3 2980700
4:       4 3420000
5:       1 3350000
6:       2 3000300
7:       3 3120200
8:       4 3670000
```

Adding an identifier column

The `idcol` argument adds a column indicating the `data.table` of origin

```
rbind("2015" = sales_2015, "2016" = sales_2016, idcol = "year")
```

```
   year quarter  amount
1: 2015       1 3200100
2: 2015       2 2950000
3: 2015       3 2980700
4: 2015       4 3420000
5: 2016       1 3350000
6: 2016       2 3000300
7: 2016       3 3120200
8: 2016       4 3670000
```

Adding an identifier column

```
rbind(sales_2015, sales_2016, idcol = "year")
```

```
   year quarter  amount
1:    1       1 3200100
2:    1       2 2950000
3:    1       3 2980700
4:    1       4 3420000
5:    2       1 3350000
6:    2       2 3000300
7:    2       3 3120200
8:    2       4 3670000
```

Adding an identifier column

```
rbind(sales_2015, sales_2016, idcol = TRUE)
```

```
   .id quarter  amount
1:   1       1 3200100
2:   1       2 2950000
3:   1       3 2980700
4:   1       4 3420000
5:   2       1 3350000
6:   2       2 3000300
7:   2       3 3120200
8:   2       4 3670000
```

Handling missing columns

```
rbind("2015" = sales_2015, "2016" = sales_2016, idcol = "year",
      fill = TRUE)
```

sales_2015:

quarter	profit
1	$3,200,100
2	$2,950,000
3	$2,980,700
4	$3,420,000

sales_2016:

quarter	profit	revenue
1	$3,350,000	$1,860,000
2	$3,000,300	$1,500,000
3	$3,120,200	$1,307,000
4	$3,670,000	$2,400,000

fill = TRUE

sales:

year	quarter	profit	revenue
2015	1	$3,200,100	NA
2015	2	$2,950,000	NA
2015	3	$2,980,700	NA
2015	4	$3,420,000	NA
2016	1	$3,350,000	$1,860,000
2016	2	$3,000,300	$1,500,000
2016	3	$3,120,200	$1,307,000
2016	4	$3,670,000	$2,400,000

Handling missing columns

```
rbind(sales_2015, sales_2016, idcol = "year")
```

```
Error in rbindlist(l, use.names, fill, idcol) :
  Item 2 has 3 columns, inconsistent with item 1 which has 2 columns.
  If instead you need to fill missing columns, use set argument 'fill'
  to TRUE.
```

The rbindlist() function

Concatenate rows from a `list` of `data.tables`

```
# Read in a list of data.tables
table_files <- c("sales_2015.csv", "sales_2016.csv")
list_of_tables <- lapply(table_files, fread)
rbindlist(list_of_tables)
```

```
   quarter  amount
1:       1 3200100
2:       2 2950000
3:       3 2980700
4:       4 3420000
5:       1 3350000
6:       2 3000300
7:       3 3120200
8:       4 3670000
```

Adding an identifier column

The `idcol` argument takes names from the input list

```
names(list_of_tables) <- c("2015", "2016")
rbindlist(list_of_tables, idcol = "year")
```

```
   year quarter  amount
1: 2015       1 3200100
2: 2015       2 2950000
3: 2015       3 2980700
4: 2015       4 3420000
5: 2016       1 3350000
6: 2016       2 3000300
7: 2016       3 3120200
8: 2016       4 3670000
```

Handling different column orders

```
rbind("2015" = sales_2015, "2016" = sales_2016, idcol = "year",
      use.names = TRUE)
```

sales_2015:

quarter	amount
1	$3,200,100
2	$2,950,000
3	$2,980,700
4	$3,420,000

sales_2016:

amount	quarter
$3,350,000	1
$3,000,300	2
$3,120,200	3
$3,670,000	4

use.names = TRUE

sales:

year	quarter	amount
2015	1	$3,200,100
2015	2	$2,950,000
2015	3	$2,980,700
2015	4	$3,420,000
2016	1	$3,350,000
2016	2	$3,000,300
2016	3	$3,120,200
2016	4	$3,670,000

`data.tables` with different column names

```
rbind("2015" = sales_2015, "2016" = sales_2016, idcol = "year",
      use.names = FALSE)
```

sales_2015:

quarter	amount
1	$3,200,100
2	$2,950,000
3	$2,980,700
4	$3,420,000

sales_2016:

quarter	profit
1	$3,350,000
2	$3,000,300
3	$3,120,200
4	$3,670,000

use.names = FALSE

sales:

year	quarter	amount
2015	1	$3,200,100
2015	2	$2,950,000
2015	3	$2,980,700
2015	4	$3,420,000
2016	1	$3,350,000
2016	2	$3,000,300
2016	3	$3,120,200
2016	4	$3,670,000

Pitfalls of `use.names = FALSE`

```
rbind("2015" = sales_2015, "2016" = sales_2016, idcol = "year",
      use.names = FALSE)
```

sales_2015:

quarter	amount
1	$3,200,100
2	$2,950,000
3	$2,980,700
4	$3,420,000

sales_2016:

amount	quarter
$3,350,000	1
$3,000,300	2
$3,120,200	3
$3,670,000	4

use.names = FALSE

sales:

year	quarter	amount
2015	1	$3,200,100
2015	2	$2,950,000
2015	3	$2,980,700
2015	4	$3,420,000
2016	$3,350,000	1
2016	$3,000,300	2
2016	$3,120,200	3
2016	$3,670,000	4

Differing defaults

- Default for `rbind()` is `use.names = TRUE`
- Default for `rbindlist()` is `use.names = FALSE` unless `fill = TRUE` .

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

Set operations

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

Set operation functions

Given two `data.tables` with the same columns:

- `fintersect()` : what rows do these two `data.tables` share in common?
- `funion()` : what is the unique set of rows across these two `data.tables` ?
- `fsetdiff()` : what rows are unique to this `data.table` ?

Set operations: `fintersect()`

Extract rows that are present in both `data.tables`

```
fintersect(dt1, dt2)
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
3	antelope	brown
4	mouse	grey

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
6	cassowary	black

fintersect()

id	animal	color
2	lion	yellow
4	mouse	grey

`fintersect()` and duplicate rows

Duplicate rows are ignored by default:

```
fintersect(dt1, dt2)
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
3	antelope	brown
4	mouse	grey
2	lion	yellow
2	lion	yellow

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
6	cassowary	black
2	lion	yellow

fintersect()

id	animal	color
2	lion	yellow
4	mouse	grey

`fintersect()` and duplicate rows

`all = TRUE` : keep the number of copies present in both `data.tables` :

```
fintersect(dt1, dt2, all = TRUE)
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
3	antelope	brown
4	mouse	grey
2	lion	yellow
2	lion	yellow

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
6	cassowary	black
2	lion	yellow

fintersect()

id	animal	color
2	lion	yellow
4	mouse	grey
2	lion	yellow

Set operations: `fsetdiff()`

Extract rows found exclusively in the rst `data.table`

```
fsetdiff(dt1, dt2)
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
3	antelope	brown
4	mouse	grey

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
6	cassowary	black

fsetdiff()

id	animal	color
1	giraffe	yellow
3	antelope	brown

`fsetdiff()` and duplicates

Duplicate rows are ignored by default:

```
fsetdiff(dt1, dt2)
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
3	antelope	brown
4	mouse	grey
2	lion	yellow
2	lion	yellow
3	antelope	brown

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
6	cassowary	black
2	lion	yellow

fsetdiff()

id	animal	color
1	giraffe	yellow
3	antelope	brown

`fsetdiff()` and duplicates

`all = TRUE` : return all extra copies:

```
fsetdiff(dt1, dt2, all = TRUE)
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
3	antelope	brown
4	mouse	grey
2	lion	yellow
2	lion	yellow
3	antelope	brown

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
6	cassowary	black
2	lion	yellow

fsetdiff()

id	animal	color
1	giraffe	yellow
3	antelope	brown
2	lion	yellow
3	antelope	brown

Set operations: `funion()`

Extract all rows found in either `data.table` :

```
funion(dt1, dt2)
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
3	antelope	brown
4	mouse	grey

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
6	cassowary	black

funion()

id	animal	color
1	giraffe	yellow
3	antelope	brown
2	lion	yellow
4	mouse	grey
5	whale	blue
6	cassowary	black

`funion()` and duplicates

Duplicate rows are ignored by default:

```
funion(dt1, dt2)
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
2	lion	yellow
2	lion	yellow

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
2	lion	yellow

funion() →

id	animal	color
1	giraffe	yellow
2	lion	yellow
4	mouse	grey
5	whale	blue

`funion()` and duplicates

`all = TRUE` : return all rows:

```
funion(dt1, dt2, all = TRUE) # rbind()
```

dt1:

id	animal	color
1	giraffe	yellow
2	lion	yellow
2	lion	yellow
2	lion	yellow

dt2:

id	animal	color
2	lion	yellow
4	mouse	grey
5	whale	blue
2	lion	yellow

funion()

id	animal	color
1	giraffe	yellow
2	lion	yellow
2	lion	yellow
2	lion	yellow
2	lion	yellow
4	mouse	grey
5	whale	blue
2	lion	yellow

Removing duplicates when combining many `data.tables`

Two `data.tables` :

1. Use `funion()` to concatenate unique rows

Three or more:

1. Concatenate all `data.tables` using `rbind()` or `rbindlist()`
2. Identify and remove duplicates using `duplicated()` and `unique()`

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

Melting data.tables

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

datacamp

Melting a wide data.table

sales_wide:

quarter	2015	2016
1	$3,200,100	$3,350,000
2	$2,950,000	$3,000,300
3	$2,980,700	$3,120,200
4	$3,420,000	$3,670,000

sales_long:

quarter	year	amount
1	2015	$3,200,100
2	2015	$2,950,000
3	2015	$2,980,700
4	2015	$3,420,000
1	2016	$3,350,000
2	2016	$3,000,300
3	2016	$3,120,200
4	2016	$3,670,000

The `melt()` function

Use `measure.vars` to specify columns to stack:

```
melt(sales_wide, measure.vars = c("2015", "2016"))
```

```
   quarter variable   value
1:       1     2015 3200100
2:       2     2015 2950000
3:       3     2015 2980700
4:       4     2015 3420000
5:       1     2016 3350000
6:       2     2016 3000300
7:       3     2016 3120200
8:       4     2016 3670000
```

The `melt()` function

Use `variable.name` and `value.name` to rename these columns in the result:

```
melt(sales_wide, measure.vars = c("2015", "2016"),
     variable.name = "year", value.name = "amount")
```

```
   quarter year  amount
1:       1 2015 3200100
2:       2 2015 2950000
3:       3 2015 2980700
4:       4 2015 3420000
5:       1 2016 3350000
6:       2 2016 3000300
7:       3 2016 3120200
8:       4 2016 3670000
```

The `melt()` function

Use `id.vars` to specify columns to keep aside

```
melt(sales_wide, id.vars = "quarter",
     variable.name = "year", value.name = "amount")
```

```
   quarter year  amount
1:       1 2015 3200100
2:       2 2015 2950000
3:       3 2015 2980700
4:       4 2015 3420000
5:       1 2016 3350000
6:       2 2016 3000300
7:       3 2016 3120200
8:       4 2016 3670000
```

The `melt()` function

Use both to keep only a subset of columns

```
melt(sales_wide, id.vars = "quarter", measure.vars = "2015",
     variable.name = "year", value.name = "amount")
```

```
   quarter year  amount
1:       1 2015 3200100
2:       2 2015 2950000
3:       3 2015 2980700
4:       4 2015 3420000
```

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

Casting data.tables

JOINING DATA WITH DATA.TABLE IN R

Sco Ritchie

Postdoctoral Researcher in Systems Genomics

Casting a long data.table

```
sales_wide <- dcast(sales_long, quarter ~ year, value.var = "amount")
```

sales_long:

quarter	year	amount
1	2015	$3,200,100
2	2015	$2,950,000
3	2015	$2,980,700
4	2015	$3,420,000
1	2016	$3,350,000
2	2016	$3,000,300
3	2016	$3,120,200
4	2016	$3,670,000

sales_wide:

quarter	2015	2016
1	$3,200,100	$3,350,000
2	$2,950,000	$3,000,300
3	$2,980,700	$3,120,200
4	$3,420,000	$3,670,000

The dcast() function

The general form of `dcast()`:

```
dcast(DT, ids ~ group, value.var = "values")
      |   |     |                   |
      |   |     |                    --> column to split
      |   |      ----------------------> group labels to split by
      |    ----------------------------> rows to keep behind as identifiers
       --------------------------------> data.table to reshape
```

The dcast() function

```
sales_wide <- dcast(sales_long, quarter ~ year, value.var = "amount")
```

sales_long:

quarter	year	amount
1	2015	$3,200,100
2	2015	$2,950,000
3	2015	$2,980,700
4	2015	$3,420,000
1	2016	$3,350,000
2	2016	$3,000,300
3	2016	$3,120,200
4	2016	$3,670,000

sales_wide:

quarter	2015	2016
1	$3,200,100	$3,350,000
2	$2,950,000	$3,000,300
3	$2,980,700	$3,120,200
4	$3,420,000	$3,670,000

Splitting multiple value columns

```
dcast(profit_long, quarter ~ year, value.var = c("revenue", "profit"))
```

profit_long:

quarter	year	revenue	profit
1	2015	$3,200,100	$640,020
2	2015	$2,950,000	$590,000
3	2015	$2,980,700	$596,140
4	2015	$3,420,000	$684,000
1	2016	$3,350,000	$670,000
2	2016	$3,000,300	$600,060
3	2016	$3,120,200	$624,040
4	2016	$3,670,000	$734,000

quarter	revenue_2015	revenue_2016	profit_2015	profit_2016
1	$3,200,100	$3,350,000	$640,020	$670,000
2	$2,950,000	$3,000,300	$590,000	$600,060
3	$2,980,700	$3,120,200	$596,140	$624,040
4	$3,420,000	$3,670,000	$684,000	$734,000

Multiple row identifiers

Keep multiple columns as row identi ers:

```
dcast(sales_long, quarter + season ~ year, value.var = "amount")
```

sales_long:

quarter	season	year	amount
1	Winter	2015	$3,200,100
2	Spring	2015	$2,950,000
3	Summer	2015	$2,980,700
4	Autumn	2015	$3,420,000
1	Winter	2016	$3,350,000
2	Spring	2016	$3,000,300
3	Summer	2016	$3,120,200
4	Autumn	2016	$3,670,000

quarter	season	2015	2016
1	Winter	$3,200,100	$3,350,000
2	Spring	$2,950,000	$3,000,300
3	Summer	$2,980,700	$3,120,200
4	Autumn	$3,420,000	$3,670,000

Dropping columns

Only columns included in the formula or `value.var` will be in the result:

```
sales_wide <- dcast(sales_long, quarter ~ year, value.var = "amount")
```

sales_long:

quarter	season	year	amount
1	Winter	2015	$3,200,100
2	Spring	2015	$2,950,000
3	Summer	2015	$2,980,700
4	Autumn	2015	$3,420,000
1	Winter	2016	$3,350,000
2	Spring	2016	$3,000,300
3	Summer	2016	$3,120,200
4	Autumn	2016	$3,670,000

quarter	2015	2016
1	$3,200,100	$3,350,000
2	$2,950,000	$3,000,300
3	$2,980,700	$3,120,200
4	$3,420,000	$3,670,000

Multiple groupings

Split on multiple group columns:

```
dcast(sales_long, quarter ~ department + year, value.var = "amount")
```

sales_long:

quarter	department	year	amount
1	retail	2015	$3,200,100
3	retail	2015	$2,980,700
1	retail	2016	$3,350,000
3	retail	2016	$3,120,200
1	consulting	2015	$100,400
3	consulting	2015	$130,200
1	consulting	2016	$125,000
3	consulting	2016	$150,400

quarter	retail_2015	retail_2016	consulting_2015	consulting_2016
1	$3,200,100	$3,350,000	$100,400	$125,000
3	$2,980,700	$3,120,200	$130,200	$150,400

Converting to a matrix

```
sales_wide <- dcast(sales_long, season ~ year, value.var = "amount")
sales_wide
```

```
   season    2015    2016
1: Autumn 3420000 3670000
2: Spring 2950000 3000300
3: Summer 2980700 3120200
4: Winter 3200100 3350000
```

Converting to a matrix

`as.matrix()` can take one of the columns to use as the matrix rownames:

```
mat <- as.matrix(sales_wide, rownames = "season")
mat
```

```
       2015    2016
Autumn 3420000 3670000
Spring 2950000 3000300
Summer 2980700 3120200
Winter 3200100 3350000
```

Let's practice!

JOINING DATA WITH DATA.TABLE IN R

Introduction to the course

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

A data frame is a general-purpose data structure

- A data frame is not something unique to R!
- It's a common data structure that meets these properties:
  - List of lists
  - All lists are of equal length
  - Value type must be the same within each list (column)
  - Value types can be di erent across columns

```
someDF <- data.frame(x = rnorm(10), y = rep(TRUE, 100))
str(someDF)
```

```
'data.frame':    100 obs. of  2 variables:
 $ x: num  -1.5456 -1.1905 0.6055 0.9489 0.0023 ...
 $ y: logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
```

data.table is an extension on data.frame

- `data.frame` = R's default data frame implementation
- `data.table` = extension of that base class
- `data.table` improvements:
  - more expressive syntax
  - more e cient memory use via pass-by-reference operators

```
library(data.table)
someDT <- data.table(x = rnorm(100), y = rep(TRUE, 100))
str(someDT)
```

```
Classes ‘data.table’ and 'data.frame':    100 obs. of  2 variables:
 $ x: num  -0.474 -0.944 0.382 -0.505 -1.128 ...
 $ y: logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
```

Selecting columns with .()

You can select columns from a `data.table` with `.()` :

```
baseballDT[, .(timestamp, winning_team)]
```

```
             timestamp winning_team
1: 2018-01-01 00:00:00          BOS
2: 2018-01-01 00:00:36          CWS
3: 2018-01-01 00:01:12          MIL
```

Column selection with .SD

Use `.SD` (Subset of Data) to reference a subset of columns.

```
cols <- c("timestamp", "winning_team")
baseballDT[, .SD, .SDcols = cols]
```

This is identical:

```
baseballDT[, .SD, .SDcols = c("timestamp", "winning_team")]
```

"new data.table with speci c columns"

```
             timestamp winning_team
1: 2018-01-01 00:00:00          BOS
2: 2018-01-01 00:00:36          CWS
3: 2018-01-01 00:01:12          MIL
```

Brief review of grep()

`grep()` returns indexes of strings matching a pa ern.

```
grep(pattern = 'art', c('artistic', 'colorful'))
```

```
1
```

Use `value = TRUE` to get values instead of indexes.

```
grep(pattern = 'art', c('artistic', 'colorful'), value = TRUE)
```

```
"artistic"
```

Use column su xes to group columns.

```
   innings_pitched_COUNT runs_allowed_COUNT era_AVERAGE
1:                    10                  8         7.2
2:                    20                  4         1.8
3:                    30                 22         6.6
```

Get just the count data

```
count_cols <- grep('COUNT$', names(baseballDT), value = TRUE)
countDT <- baseballDT[, .SD, .SDcols = count_cols]
countDT
```

```
   innings_pitched_COUNT runs_allowed_COUNT
1:                    10                  8
2:                    20                  4
3:                    30                 22
```

Combining row and column selection

Expressive subset statements with row selectors

```
cols <- c("timestamp", "winning_team")
baseballDT[
    which.max(timestamp),
    .SD,
    .SDcols = cols
]
```

"Get the most recent observation"

```
             timestamp winning_team
1: 2018-01-01 01:00:00          BOS
```

Let's practice!

Flexible data selection

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Explicit references

Use direct name references in `[]`

```
locDT <- data.table(
    cities = c("Chicago", "Boston", "Milwaukee"),
    ppl_mil = c(2.7, 0.673, 0.595)
)
locDT[, cities]
```

```
"Chicago"   "Boston"    "Milwaukee"
```

Calling functions

Functions in the `i` block to select rows

```
locDT[which.max(ppl_mil)]
```

```
    cities ppl_mil
1: Chicago     2.7
```

Using get()

`get()` : evaluate a string as a column reference

```
locDT <- data.table(
    cities = c("Chicago", "Boston", "Milwaukee"),
    ppl_mil = c(2.7, 0.673, 0.595)
)
city_col <- "cities"
locDT[, get(city_col)]
```

```
"Chicago"   "Boston"    "Milwaukee"
```

get() is great when writing functions

Write reusable functions without hard-coded column names:

```
square_col <- function(DT, col_name){
    return(DT[, get(col_name) ^ 2])
}
```

```
square_col(locDT, "ppl_mil")
```

```
7.290000 0.452929 0.354025
```

Using()

Problem: get people in thousands from the `ppl_mil` column.

```
locDT[, ppl_bil := ppl_mil * 1000]
locDT[, ppl_bil]
```

```
2700  673  595
```

But what if you want to parameterize the new column name?

```
add_bil_ppl <- function(DT, new_name){
    DT[, (new_name) := ppl_mil * 1000
}
add_bil_ppl(locDT, "some_rand_name")
print(locDT)
```

```
      cities ppl_mil some_rand_name
1:   Chicago   2.700           2700
2:    Boston   0.673            673
3: Milwaukee   0.595            595
```

Combining () and get()

Function to create features by adding 10 to existing columns

```
add10 <- function(DT, cols){
    for (col in cols){
        new_name <- paste0(col, "_plus10")
        DT[, (new_name) := get(col) + 10]
    }
}
add10(locDT, cols = "ppl_mil")
locDT
```

```
      cities ppl_mil ppl_mil_plus10
1:   Chicago   2.700         12.700
2:    Boston   0.673         10.673
3: Milwaukee   0.595         10.595
```

Changing names with setnames()

Change a single column's name:

```
locDT <- data.table(
    cities = c("Chicago", "Boston", "Milwaukee"),
    ppl_mil = c(2.7, 0.673, 0.595)
)
setnames(locDT, old = "cities", new = "city_names")
names(locDT)
```

```
"city_names" "ppl_mil"
```

setnames() in functions

```
tag_important_columns <- function(DT, cols){
    setnames(DT, old = cols, new = paste0(cols, "_important"))
}
```

Calling this function is e cient and doesn't copy the data!

```
tag_important_columns(locDT, "ppl_mil")
locDT
```

```
      cities ppl_mil_important
1:   Chicago             2.700
2:    Boston             0.673
3: Milwaukee             0.595
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Executing functions inside data.tables

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Use functions in the "i" block to select rows

```
stockDT <- data.table(
    close_date = seq.POSIXt(as.POSIXct("2017-01-01"), as.POSIXct("2017-01-30"), length.out = 100),
    MSFT = runif(100, 70, 80),
    AAPL = runif(100, 140, 180)
)
stockDT[which.max(MSFT)]   # Best day for Microsoft
```

```
            close_date    MSFT     AAPL
1: 2017-01-08 07:45:27 79.9235 159.9928
```

```
stockDT[close_date > max(close_date) - 60 * 60 * 8]   # Final 8 hours of the dataset
```

```
            close_date     MSFT     AAPL
1: 2017-01-29 16:58:10 73.78340 157.9154
2: 2017-01-30 00:00:00 71.51727 141.8897
```

`cor()` creates a correlation matrix between columns

```
cor(stockDT[, .SD, .SDcols = c('AAPL', 'MSFT')])
```

```
           AAPL       MSFT
AAPL 1.00000000 0.05680504
MSFT 0.05680504 1.00000000
```

You can call this directly inside a `data.table`!

```
corr_mat <- stockDT[, cor(.SD), .SDcols = c('AAPL', 'MSFT')]
print(corr_mat)
```

```
           AAPL       MSFT
AAPL 1.00000000 0.05680504
MSFT 0.05680504 1.00000000
```

Use functions in the "j" block to generate new columns

Add a new column:

```
stockDT[, rand_noise := AAPL + rnorm(100)]
```

```
              close_date     MSFT     AAPL rand_noise
  1: 2017-01-01 00:00:00 76.46907 163.6131   162.4594
  2: 2017-01-01 07:01:49 78.68001 174.1177   174.9193
```

```r
# Two-step process to generate "mean price by hour of the day"
stockDT[, hour_of_day := as.integer(strftime(close_date, "%H"))]
stockDT[, mean(AAPL), by = hour_of_day][order(hour_of_day)]
```

```
   hour_of_day       V1
1:           0 155.4853
2:           1 163.5479
3:           2 152.5203
```

```r
# 1-step process to generate "mean price by hour of day"
stockDT[, mean(AAPL), by = .(
    hour_of_day = as.integer(strftime(close_date, "%H"))
    )][order(hour_of_day)]
```

```
   hour_of_day       V1
1:           0 155.4853
2:           1 163.5479
3:           2 152.5203
```

Applying a function over every column with .SD

- Use `lapply()` if you want a `data.table` back
- Use `sapply()` if you want a vector or list back

```
# Count percent missing values by column
stockDT[, lapply(.SD, function(x){mean(is.na(x))})]
```

```
   close_date MSFT AAPL
1:          0  0.1 0.26
```

```
# Count non-NA values
num_obs <- stockDT[, sapply(.SD, function(x){sum(!is.na(x), na.rm = TRUE)})]
print(num_obs)
```

```
close_date       MSFT       AAPL
       100         90         74
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Overview of the POSIXct type

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

History of POSIX

POSIX = Portable Operating System for Unix

`POSIXlt` = a list object with date-time components like `year` and `day` stored in individual a ributes

```
lt <- as.POSIXlt("2017-01-01", tz = "UTC")
print(attributes(lt))
```

```
$names
"sec"   "min"   "hour"  "mday"  "mon"   "year"  "wday"  "yday"  "isdst"
```

History of POSIX

`POSIXct` = a signed integer representing seconds since 1970-01-01, with a single a ribute capturing timezone.

```
ct <- as.POSIXct("2017-01-01", tz = "UTC")
print(as.numeric(ct))
```

```
1483228800
```

Converting other formats to POSIXct

```
# String conversion
as.POSIXct("2004-10-27", tz = "UTC")
```

```
"2004-10-27 UTC"
```

```
# Integer conversion
as.POSIXct(1540153601, origin = "1970-01-01", tz = "UTC")
```

```
"2018-10-21 20:26:41 UTC"
```

```
# Excel dates
as.POSIXct(as.Date(42885, origin = "1900-01-01"), tz = "UTC")
```

```
"2017-06-01 00:00:00 UTC"
```

as.POSIXct is vectorized!

Apply to a vector

```
dates <- c("2004-10-24", "2004-10-25", "2004-10-26")
as.integer(as.POSIXct(dates, tz = "UTC"))
```

```
1098576000 1098662400 1098748800
```

as.POSIXct is vectorized!

Code looks the same on a `data.table` column

```
someDT <- data.table(dates = c("2004-10-24", "2004-10-25", "2004-10-26"))
someDT[, posix := as.POSIXct(dates, tz = "UTC")]
str(someDT)
```

```
Classes ‘data.table’ and 'data.frame':    3 obs. of  2 variables:
 $ dates: chr  "2004-10-24" "2004-10-25" "2004-10-26"
 $ posix: POSIXct, format: "2004-10-24" ...
```

Creating POSIXct dates out of data frame columns

- `:=` can be used to add or modify columns
- `as.POSIXct()` is vectorized

Sample dataset:

```
gameDT <- data.table(
    game_date = c("2004-10-23", "2004-10-24", "2004-10-26", "2004-10-27")
)
```

Add a new column:

```
gameDT[, posix_date := as.POSIXct(game_date, tz = "UTC")]
```

```
the_date <- "10-27-2004 22:29:00"
```

`as.POSIXct()` can't handle this, `lubridate` makes it easy!

```
lubridate::mdy_hms(the_date)
```

```
"2004-10-27 10:29:00 UTC"
```

Other common `lubridate` functions:

- `ymd_hms()` : ex. "2017-01-10 00:00:00"
- `dmy_hms()` : ex. "10-01-2017 00:00:00"
- `ymd_h()` : ex. "2017-01-10 06"
- `ymd()` : ex. "2017-01-10"

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Creating data.tables from vectors

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Creating data.tables from scratch

Creating a `data.table` is as easy as calling `data.table()` !

```
candyDT <- data.table(
    color = c("red", "blue", "green"),
    size = c("S", "L", "S"),
    num = c(100, 50, 210)
)
```

```
   color size num
1:   red    S 100
2:  blue    L  50
3: green    S 210
```

If you can make vectors, you can make a data.table!

Use all your favorite vector-making functions to make `data.table` s!

```
testDT <- data.table(
    rand_numbers = rnorm(100),
    rand_strings = sample(LETTERS, n = 100, replace = TRUE),
    simple_index = 1:100,
    sample_dates = seq.POSIXt(
        from = as.POSIXct("1990-01-01"),
        to = as.POSIXct("1992-08-01"),
        length.out = 100),
    fifty_fifty_split = c(rep(TRUE, 50), rep(FALSE, 50)))
```

`c()` , `rep()` , `seq()` , `sample()` , `rnorm()` and more will be valuable!

More on seq.POSIXt()

`seq.POSIXt()` is the `POSIXt` variant of R's `seq()` family

```
# Date range defining one day
start <- as.POSIXct("2010-06-17", tz = "UTC")
end <- as.POSIXct("2010-06-18", tz = "UTC")
```

`length.out` : the secret to changing the frequency of your test data

```
# Hourly timestamps
hourlyDT <- data.table(
    timestamp = seq.POSIXt(start, end, length.out = 1 + 24))
# Minute timestamps
minuteDT <- data.table(
    timestamp = seq.POSIXt(start, end, length.out = 1 + 24 * 60))
```

Dynamic resizing with .N

could hard code the number of elements everywhere

```
# Hourly stock price dataset
hourlyDT <- data.table(
    close_time = seq.POSIXt(start, end, length.out = 1 + 24),
    COMPANY1 = rnorm(n = 1 + 24),
    COMPANY2 = rnorm(n = 1 + 24)
)
```

But `.N` means you don't have to!

```
add_stock_data <- function(DT){
    DT[, COMPANY1 := rnorm(n = .N)]
    DT[, COMPANY2 := rnorm(n = .N)]
}
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Coercing from xts

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Creating xts objects

Two required things:

- `x` = a vector of input data
- `order.by` = a vector of date-times to use as index

```
dates <- seq.POSIXt(
    from = as.POSIXct("2017-06-15"),
    to = as.POSIXct("2017-06-16"),
    length.out = 24
)
ex_tee_ess<- xts::xts(
   x = rnorm(24),
   order.by = dates
)
```

Creating xts objects

Complex object with a ributes.

- `tclass` = R class for the date-time index
- `tzone` = timezone for date-time index

```
attr(ex_tee_ess, "tclass")
```

```
"POSIXct" "POSIXt"
```

```
attr(ex_tee_ess, "tzone")
```

```
""
```

Expressive subsetting

Friendly subse ing makes data scientists happy.

- `['/']` = "the whole dataset"
- `['2017']` = "data from 2017"
- `['2017-01/']` = "data from January 2017 to the end of the data"
- `['2014/2015']` = "data from 2014 to 2015"

Subsetting example

```
# Entire dataset
str(hourlyXTS)
```

```
An ‘xts’ object on
  2017-06-15/2017-06-18 containing:
      Data: num [1:73, 1] -0.118 ...
```

```
# Observations on or after June 16
str(hourlyXTS["2017-06-16/"])
```

```
An ‘xts’ object on
  2017-06-16/2017-06-18 containing:
      Data: num [1:49, 1] 0.495 ...
```

Easy aggregations

How to create a time-series aggregation:

- bucket your dataset into equal-sized windows by time
- evaluate one or more functions over the values that fall within each window

Examples include `to.minutes()`, `to.minutes10()`, `to.daily()`

```
xts::to.daily(hourlyXTS)
```

```
           hourlyXTS.Open hourlyXTS.High hourlyXTS.Low hourlyXTS.Close
2017-06-16      0.3511835       1.783355     -1.750838      0.09564442
2017-06-17     -1.0457750       3.182890     -3.039372     -1.43888466
2017-06-18      0.7893328       2.396728     -1.770283      0.69979482
2017-06-18      1.7245329       1.724533      1.724533      1.72453289
```

Converting from xts to data.table

- `xts` : powerful for speci c tasks
- `data.table` : exible to custom processing

Converting is as easy as `as.data.table()` !

```r
# Convert with as.data.table()
hourlyDT <- data.table::as.data.table(
    hourlyXTS)
head(hourlyDT, n = 2)
```

```
                  index         V1
 1: 2017-06-15 00:00:00 -0.4448620
 2: 2017-06-15 01:00:00  0.5558520
```

```r
# Change names
data.table::setnames(hourlyDT, "V1", "stock_price")
head(hourlyDT, n = 2)
```

```
                 index stock_price
1: 2017-06-15 00:00:00  -0.4448620
2: 2017-06-15 01:00:00   0.5558520
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Combining datasets with merge and rbindlist

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Two timestamps might look the same printed...

```
sec <- as.POSIXct("2010-04-06 19:00:00", tz = "UTC")
milli <- as.POSIXct("2010-04-06 19:00:00.005", tz = "UTC")
print(c(sec, milli))
```

```
"2010-04-06 14:00:00 CDT" "2010-04-06 14:00:00 CDT"
```

...but have different underlying values!

```
options(digits = 16)
print(as.numeric(sec))
print(as.numeric(milli))
```

```
1270580400
1270580400.005
```

Precision-safe merges

The naive approach returns a checkerboard join result:

```
merge(secDT, milliDT, by = "timestamp", all = TRUE)
```

```
              timestamp abc  def
1: 2010-04-06 19:00:00 1.5   NA
2: 2010-04-06 19:00:00  NA TRUE
```

Use round() for safer merges

Instead, use `round()` to get to the nearest second.

```
secDT[, timestamp := as.POSIXct(round(as.numeric(timestamp)),
                                origin = "1970-01-01")]
milliDT[, timestamp := as.POSIXct(round(as.numeric(timestamp)),
                                  origin = "1970-01-01")]
```

```
merge(secDT, milliDT, by = "timestamp", all = TRUE)
```

```
             timestamp abc  def
1: 2010-04-06 19:00:00 1.5 TRUE
```

Downsampling

`data.table` functions for extracting integer date-parts:

- `year()` = 4-digit year
- `mday()` = day-of-the-month (1-31)
- `hour()` = hour (1-24)

Example:

```
salesDT[, .(ts, year = year(ts), mday = mday(ts), hour = hour(ts))]
```

```
                    ts year mday hour
1: 2018-01-02 00:45:06 2018    2    0
2: 2018-01-03 10:15:08 2018    3   10
```

Merging across frequencies

Get a daily aggregation of the hourly price data:

```
dailySalesDT[, day_int := mday(timestamp)]
dailyPriceDT <- hourlyPriceDT[, .(price = mean(price)),
                                 by = mday(timestamp)]
```

Merge daily sales with daily prices:

```
mergeDT <- merge(
    dailySalesDT,
    dailyPriceDT,
    by.x = "day_int",
    by.y = "day"
)
```

Stacking datasets with rbindlist()

Ok, so you have a few data.tables

```
DT1 <- fread("2014.csv")
DT2 <- fread("2015.csv")
DT3 <- fread("2016.csv")
```

Just `rbindlist()` them up!

```
allDT <- rbindlist(list(DT1, DT2, DT3), fill = TRUE)
```

A warning with rbindlist()

When using `rbindlist()` , watch out for:

- Different column names
- Timestamps with different types (e.g. `Date` vs. `POSIXct`)

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Generating lags

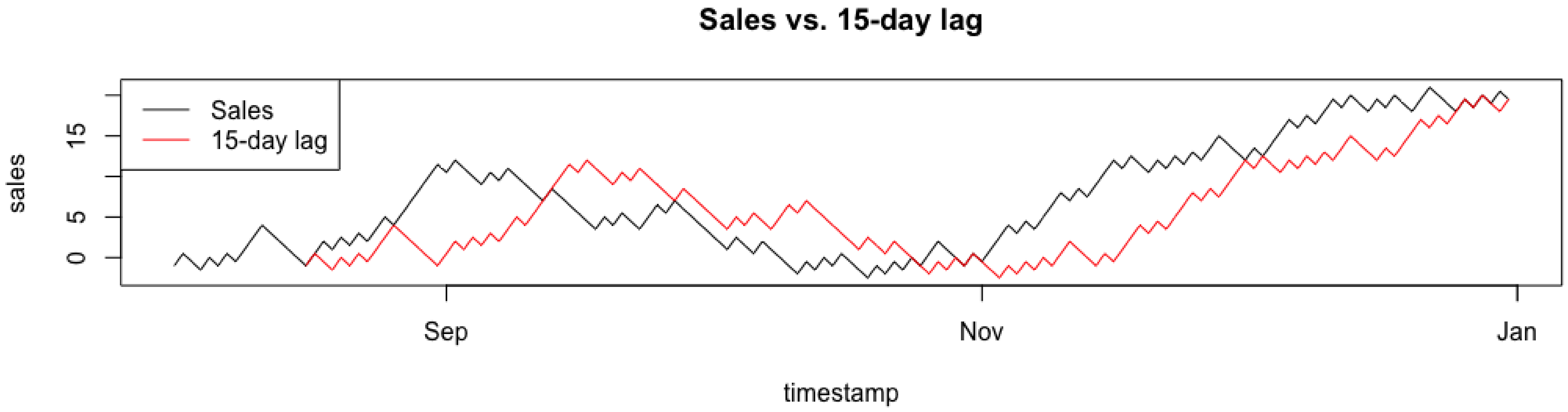
TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Introduction to lags

"lag" = "the value of this variable _n_ periods ago"

```
dailyDT[, lag15 := shift(sales, type = "lag",  n = 15)]
```



- `type = "lag"` : move earlier data forward
- `type = "lead"` : move later data backwards

Check it out!

```
someDT <- data.table(col1 = c("a", "b", "c", "d", "e"))
someDT[, col1_lag1 := shift(col1, n = 1, type = "lag")]
someDT[, col1_lag2 := shift(col1, n = 2, type = "lag")]
someDT[, col1_lead1 := shift(col1, n = 1, type = "lead")]
someDT[, col1_lead2 := shift(col1, n = 2, type = "lead")]
someDT
```

```
   col1 col1_lag1 col1_lag2 col1_lead1 col1_lead2
1:    a      <NA>      <NA>          b          c
2:    b         a      <NA>          c          d
3:    c         b         a          d          e
4:    d         c         b          e       <NA>
5:    e         d         c       <NA>       <NA>
```

Keying / sorting by time

`shift()` takes vector as-is

```
backwardsDT[, somenums_lag1 := shift(somenums, type = "lag", n = 1)]
backwardsDT
```

```
             timestamp somenums somenums_lag1
1: 2017-06-20 00:00:00        1            NA
2: 2017-06-19 10:40:00        2             1
3: 2017-06-18 21:20:00        3             2
4: 2017-06-18 08:00:00        4             3
5: 2017-06-17 18:40:00        5             4
```

Always use setorderv before shift

Use `setorderv()` to x this!

```r
setorderv(backwardsDT, "timestamp")
backwardsDT[, somenums_lag1 := shift(somenums, type = "lag", n = 1)]
```

```
             timestamp somenums somenums_lag1
1: 2017-06-15 00:00:00       10            NA
2: 2017-06-15 13:20:00        9            10
3: 2017-06-16 02:40:00        8             9
```

Using lags in linear models

If you have lags in your `data.table` , you can drop them right into a linear model:

```
mod <- lm(sales ~ lag15, data = dailyDT)
summary(mod)
```

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.02777    0.58156   5.206 6.96e-07 ***
lag15        0.83273    0.06929  12.018  < 2e-16 ***
```

Making lags on the fly in models

But even cooler...make them on the y!

```
mod <- lm(sales ~ shift(sales, n = 21), data = dailyDT)
summary(mod)
```

```
Coefficients:
                     Estimate Std. Error t value Pr(>|t|)
(Intercept)           4.57565    0.71704   6.381 2.84e-09 ***
shift(sales, n = 21)  0.69558    0.09491   7.329 2.20e-11 ***
```

```r
# Fit models with 1 and 2 lags
mod1 <- lm(price ~ lag1, data = aluminumDT)
mod2 <- lm(price ~ lag1 + lag2, data = aluminumDT)
# Compare with stargazer
stargazer::stargazer(list(mod1, mod2), type = "text")
```

```
=========================================================
                         Dependent variable: price
                           (1)                 (2)
<hr />-----------------------------------------------------
lag1                      -0.015              -0.035
lag2                                          0.046
Constant                  0.162*              0.169*
<hr />-----------------------------------------------------
Observations               99                  98
R2                        0.0002              0.003
Adjusted R2               -0.010              -0.018
=========================================================
Note:                         *p<0.1; **p<0.05; ***p<0.01
```

Caution with long datasets

Wrong approach - shi   ing across subjects:

```
experimentDT[, lag1 := shift(result, type = "lag", n = 1)]
experimentDT
```

```
   day result subject_id lag1
1:   1    1.0          A   NA
2:   2    3.3          A  1.0
3:   3    2.5          A  3.3
4:   1    1.1          B  2.5
5:   2    3.9          B  1.1
6:   3    3.8          B  3.9
```

Use "by" with long datasets

Correct approach - with "by":

```
experimentDT[, lag1 := shift(result, type = "lag", n = 1), by = subject_id]
```

```
   day result subject_id lag1
1:   1    1.0          A   NA
2:   2    3.3          A  1.0
3:   3    2.5          A  3.3
4:   1    1.1          B   NA
5:   2    3.9          B  1.1
6:   3    3.8          B  3.9
```

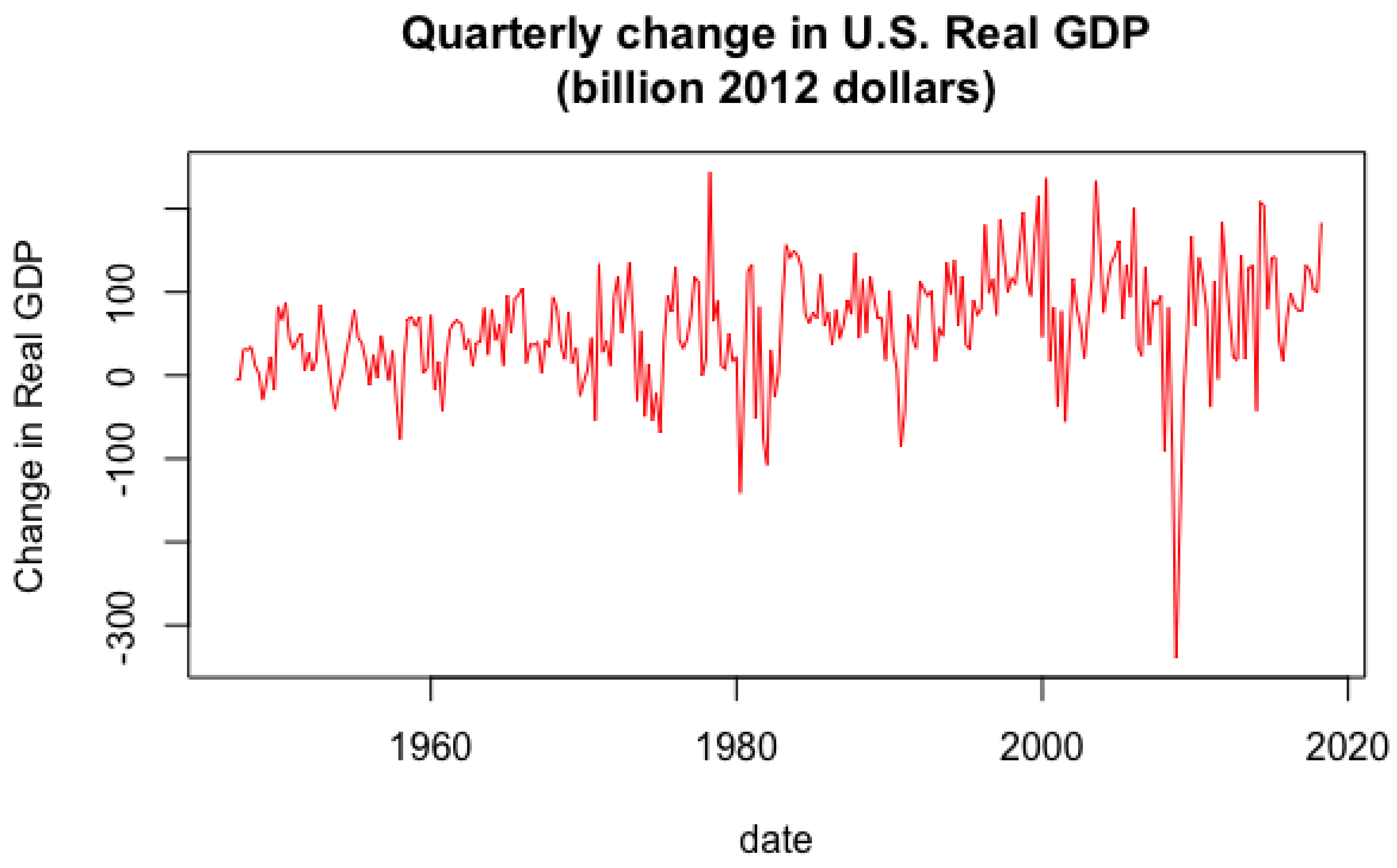
Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Generating growth rates and differences

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor


Quarterly change in U.S. Real GDP
(billion 2012 dollars)
Change in Real GDP
100
0
-100
-300
1960
1980
2000
2020
date

Computing differences (math)

The formula for an n-period di erence:

$$x_t - x_{t-n}$$

Where:

- x_t = the value of x at time t
- x_{t-n} = the value x n periods prior to time t

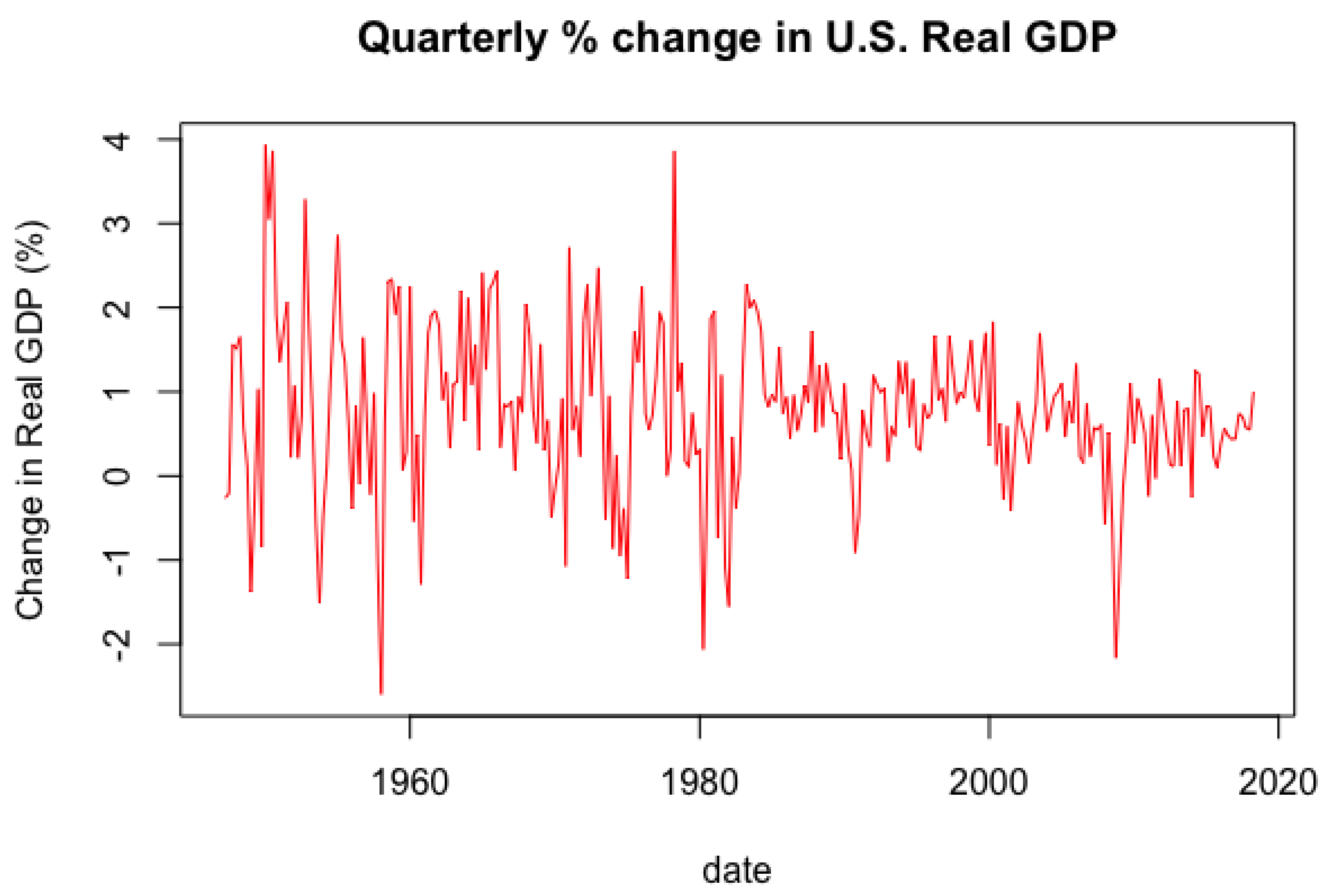
Computing differences (code)

That x_{t-n} term is just the _n_-period lag!

```
gdpDT[, lag1 := shift(gdp, type = "lag", n = 1)]
gdpDT[, diff1 := gdp - lag1]
```

You can also do this in one shot:

```
gdpDT[, diff1 := gdp - shift(gdp, type = "lag", n = 1)]
```


Quarterly % change in U.S. Real GDP
Change in Real GDP (%)
4
3
2
1
0
-1
-2
1960
1980
2000
2020
date

Computing growth rates (math)

The formula for an n-period di erence:

$$\frac{x_t - x_{t-n}}{x_{t-n}}$$

Where:

- x_t = the value of x at time t
- x_{t-n} = the value x n periods prior to time t

Computing growth rates (code)

That x_{t-n} term is just the n-period lag!

```
gdpDT[, lag1 := shift(gdp, type = "lag", n = 1)]
gdpDT[, diff1 := gdp - lag1]
gdpDT[, growth1 := diff1 / lag1]
```

You can also do this in one shot:

```
gdpDT[, growth1 :=
    (gdp - shift(gdp, type = "lag", n = 1)) /
    shift(gdp, type = "lag", n = 1)
]
```

A simpler growth formula

The growth rate formula can be re-wri en

$$\frac{x_t}{x_{t-n}} - 1$$

This simpli es the code:

```
gdpDT[, growth1 := (gdp / shift(gdp, type = "lag", n = 1)) - 1 ]
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Windowing with j and by

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Why you should care about windowed aggregations

1. Creating features for machine learning models. For example:
   - "hourly average click volume"
   - "1-day volatility in price"
   - "1-month count of failed inspections"
2. Downsampling for plo ing

Creating a grouping indicator

"group by month"

```
salesDT[, nearest_month := month(timestamp)]
```

```
    timestamp   sales nearest_month
1: 2018-08-01 543.183             8
2: 2018-08-02 546.341             8
3: 2018-09-19 576.842             9
4: 2018-10-19 510.838            10
5: 2018-11-08 472.143            11
```

Applying aggregate functions

Windowed aggregations, one set of values per month:

```
aggDT <- salesDT[, .(
        min = min(sales),
        total = sum(sales),
        num_obs = length(sales)),
    by = nearest_month]
```

```
   nearest_month     min    total num_obs
1:             8 358.099 15202.14      31
2:             9 420.018 15067.15      30
3:            10 404.858 15872.85      31
4:            11 403.295 14733.55      30
5:            12 372.442 15695.31      31
```

Windowing on the fly

Windowing and aggregation in one expression:

```
aggDT <- salesDT[, .(
        min = min(sales),
        total = sum(sales),
        num_obs = length(sales)),
    by = month(timestamp)]
```

```
   month     min    total num_obs
1:     8 358.099 15202.14      31
2:     9 420.018 15067.15      30
3:    10 404.858 15872.85      31
4:    11 403.295 14733.55      30
5:    12 372.442 15695.31      31
```

Word of caution: statistical validity

A system issue wiped out most of our August-October data!

```
aggDT <- malfunctionDT[, .(
    min = min(sales),
    total = sum(sales),
    num_obs = length(sales)),
    by = month(timestamp)]
```

Be sure to look at those observation counts:

```
   month     min     total variance num_obs
1:     8 475.030  1564.554 1623.344       3
2:    10 423.986  6672.959 2158.440      13
3:    11 403.295 14733.546 2337.096      30
4:    12 372.442 15695.306 2474.622      31
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Getting Started

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Getting data from Quandl

Quandl provides an R package for pulling data

```
aluminumDF <- Quandl::Quandl(
    code = "LME/PR_AL",
    start_date = "2001-12-31",
    end_date = "2018-03-12")
head(aluminumDF, n = 2)
```

```
        Date Cash Buyer Cash Seller & Settlement 3-months Buyer 3-months Seller 15-months Buyer
1 2018-03-12     2096.5                   2097.0         2117.0            2118              NA
2 2018-03-09     2078.0                   2078.5         2098.5            2099              NA
  15-months Seller Dec 1 Buyer Dec 1 Seller Dec 2 Buyer Dec 2 Seller Dec 3 Buyer Dec 3 Seller
1               NA        2168         2173        2188         2193        2208         2213
2               NA        2148         2153        2168         2173        2188         2193
```

Convert to a data.table

Use `as.data.table()` to convert a `data.frame` to a `data.table`

```
aluminumDT <- as.data.table(aluminumDF)
```

Now you have a `data.table` !

```
str(aluminumDT)
```

```
Classes ‘data.table’ and 'data.frame':    1552 obs. of  13 variables:
 $ Date                    : Date, format: "2018-03-12" "2018-03-09" ...
 $ Cash Buyer              : num  2096 2078 2082 2112 2136 ...
 $ Cash Seller & Settlement: num  2097 2078 2082 2112 2136 ...
 $ 3-months Buyer          : num  2117 2098 2104 2132 2154 ...
 $ 3-months Seller         : num  2118 2099 2104 2132 2155 ...
```

Clean up column names

```
aluminumDT[, .(Date, `Cash Seller & Settlement`)]  # Spaces are cumbersome
```

```
         Date Cash Seller & Settlement
1: 2018-03-12                   2097.0
2: 2018-03-09                   2078.5
```

Use `setnames()` to clean up

```
setnames(aluminumDT, "Cash Seller & Settlement", "aluminum_price")
aluminumDT[, .(Date, aluminum_price)]
```

```
         Date aluminum_price
1: 2018-03-12         2097.0
2: 2018-03-09         2078.5
```

Renaming columns during a subset

Use `.()` to select and rename columns

```
newDT <- aluminumDT[, .(obstime = Date,
                        aluminum_price = `Cash Seller & Settlement`
                    )]
```

Now you'll have a new table to work with!

```
      obstime aluminum_price
1: 2018-03-12         2097.0
2: 2018-03-09         2078.5
3: 2018-03-08         2082.5
```

Applying functions with .()

Subset, rename columns, AND change types!

```
newDT <- aluminumDT[, .(obstime = as.POSIXct(Date, tz = "UTC"),
                        aluminum_price = `Cash Seller & Settlement`
                   )]
```

Look at that new dataset:

```
str(newDT)
```

```
Classes ‘data.table’ and 'data.frame':    1552 obs. of  2 variables:
 $ obstime       : POSIXct, format: "2018-03-11 19:00:00" "2018-03-08 18:00:00" ...
 $ aluminum_price: num  2097 2078 2082 2112 2136 ...
```

Merging on timestamps

Select:

- Two `data.tables`
- One or more columns to merge on
- A merge strategy

```
mergedDT <- merge(
    x = aluminumDT,
    y = nickelDT,
    all = TRUE,
    by = "obstime"
)
```

```
               obstime aluminum_price nickel_price
1: 2012-01-02 18:00:00         2006.0        18430
2: 2012-01-03 18:00:00         2052.0        18705
3: 2012-01-04 18:00:00         2003.5        18590
4: 2012-01-05 18:00:00         2020.0        18680
5: 2012-01-08 18:00:00         2061.5        18855
```

Using Reduce with merge()

```
Reduce(
    f = function(x,y){paste0(x, y, "|")},
    x = c("a", "b", "c")
)
```

```
"ab|c|"
```

Use it to merge `data.tables` !

```
Reduce(
    f = function(x, y){merge(x, y, by = "obstime
    x = list(someDT, otherDT)
)
```

```
               obstime   col1   col2
1: 2017-01-01 00:01:00 -0.873 -0.286
2: 2017-01-01 00:08:00  1.571  0.320
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

datacamp

Timeseries feature engineering

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Differences review

Math:

$$x_t - x_{t-1}$$

Code:

```
gdpDT[, diff1 := gdp - shift(gdp, type = "lag", n = 1)]
```

Hardcoded difference function

The code from the previous slide, as a function:

```
add_diffs <- function(DT){
    DT[, diff1 := gdp - shift(gdp, type = "lag", n = 1)]
    return(invisible(NULL))
}
```

Drawbacks:

- assumes that column called `"gdp"` exists
- assumes you want to always compute a 1-period di erence
- assumes you want to store the di erence in a column called `"diff1"`

Improvement 1: configure new column name

Recall: you can pass in a variable with a column name to `()`

```
colname <- "abc"
someDT[, (colname) := rnorm(10)]
```

Update the function:

```
add_diffs <- function(DT, newcol){
    DT[, (newcol) := gdp - shift(gdp, type = "lag", n = 1)]
    return(invisible(NULL))}
```

Call it:

```
add_diffs(DT, "diff1")
```

Improvement 2: choose the column to difference

Use `get()` to evaluate a column reference:

```
colname <- "def"
someDT[, random_stuff := get(colname) * rnorm(10)]
```

Update the function:

```
add_diffs <- function(DT, newcol, dcol){
    DT[, (newcol) := get(dcol) - shift(get(dcol), type = "lag", n = 1)]
    return(invisible(NULL))}
```

Call it:

```
add_diffs(DT, "diff1", "cpi")
```

Improvement 3: configure number of periods

Update the function:

```
add_diffs <- function(DT, newcol, dcol, ndiff){
    DT[, (newcol) := get(dcol) - shift(get(dcol), type = "lag", n = ndiff)]
    return(invisible(NULL))
}
```

Call it:

```
add_diffs(DT, "diff1", "cpi", 2)
```

Growth rates review

Math:

$$\frac{x_n}{x_{n-1}} - 1$$

Code:

```
gdpDT[, growth1 := (gdp / shift(gdp, type = "lag", n = 1)) - 1 ]
```

Extending to growth rates

Di erences:

```
get(dcol) - shift(get(dcol), type = "lag", n = ndiff)
```

Growth rates:

```
(get(dcol) / shift(get(dcol), type = "lag", n = ndiff)) - 1
```

The function:

```
add_growth_rates <- function(DT, newcol, dcol, ndiff){
    DT[, (newcol) :=
        (get(dcol) / shift(get(dcol), type = "lag", n = ndiff)) - 1
      ]
    return(invisible(NULL))}
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

EDA and model building

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Feature selection

Terms:

- Feature engineering = taking some columns and making more columns
- Feature selection = choosing which columns to show to a model

Strategies for feature selection in time series problems

Strategies:

- Hand-picking features based on domain knowledge
- Dropping 0-variance or low-variance variables
- Highest (absolute) linear correlation with the target
- Model families that do it automatically
  - Penalized regression
  - Tree-based models

Computing correlations

Correlation matrices from data.tables

`cor()` can take a `data.table` directly

```
someDT <- data.table(x = rnorm(100), y = rnorm(100), z = rnorm(100))
```

Correlations are bounded between -1 and 1:

```
cor(someDT)
```

```
            x         y           z
x  1.00000000 0.1294980 -0.05782045
y  0.12949804 1.0000000  0.11575081
z -0.05782045 0.1157508  1.00000000
```

Problem with missing values

Add in one missing value...

```
someDT <- data.table(x = c(NA, rnorm(99)), y = rnorm(100), z = rnorm(100))
```

...and this is what you get:

```
cor(someDT)
```

```
   x          y          z
x  1         NA         NA
y NA 1.00000000 0.03368368
z NA 0.03368368 1.00000000
```

Handling missing values

Given a `data.table` with missing values...

```
       x y     z
1:    NA 1 green
2:  TRUE 2   red
3: FALSE 3  <NA>
```

```
complete.cases(someDT) # ...get a logical vector telling you which rows have no NAs
someDT[complete.cases(someDT)] # and subset with it!
```

```
FALSE  TRUE FALSE


      x y   z
1: TRUE 2 red
```

Correlation matrix una ected by NAs:

```
someDT <- data.table(x = c(NA, rnorm(99)), y = rnorm(100), z = rnorm(100))
# Get correlation matrix
cmat <- cor(someDT[complete.cases(someDT)])
```

```
            x         y           z
x  1.00000000 0.1294980 -0.05782045
y  0.12949804 1.0000000  0.11575081
z -0.05782045 0.1157508  1.00000000
```

See what, if anything, is strongly correlated with `x`:

```
cmat[, "x"]
```

```
         x         y           z
1.00000000 0.1294980 -0.05782045
```

Pseudocode for a regression training pipeline

Hand picking features:

```
# Select features
feat_cols <- c("var_1", "var_5")
# Fit model
mod1 <- lm(target ~ ., data = trainDT[, .SD, .SDcols = feat_cols])
```

Some fancy strategy you put in a function:

```
# Select features
feat_cols <- select_features(trainDT)
# Fit model
mod2 <- lm(target ~ ., data = trainDT[, .SD, .SDcols = feat_cols)
```

Let's practice!

TIME SERIES WITH DATA.TABLE IN R

Congratulations

TIME SERIES WITH DATA.TABLE IN R

James Lamb
Instructor

Congratulations!

TIME SERIES WITH DATA.TABLE IN R