Introduction

Sergio Mosquim Junior

9/4/2019

Prerequisites

- R here
- R Studio here
- packages: tidyverse, forcats, stringr, readr, purrr (should be loaded with tidyverse)

Overview

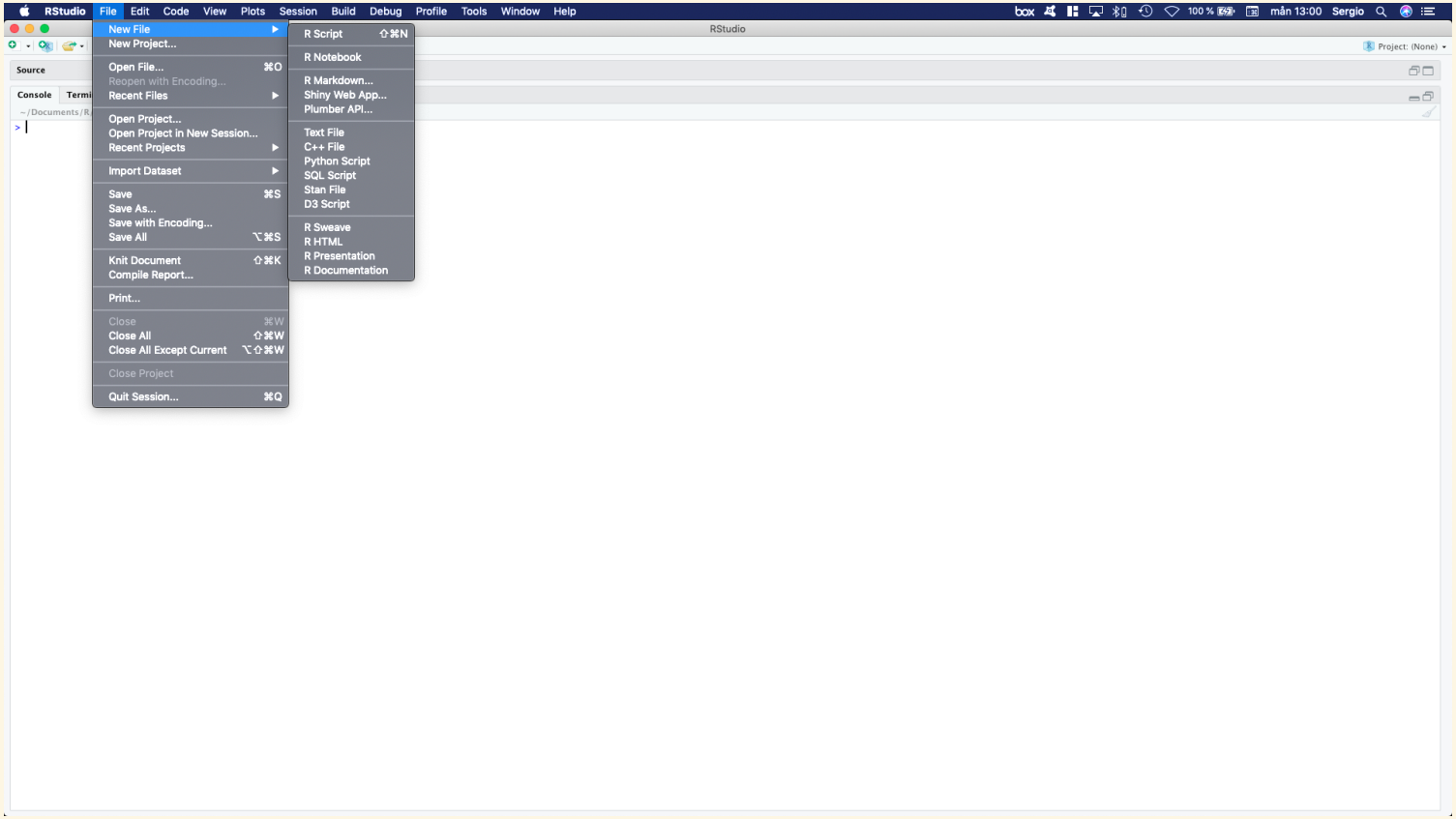
1. Rmarkdown
2. Tidyverse
3. Data Structures
4. Data manipulation

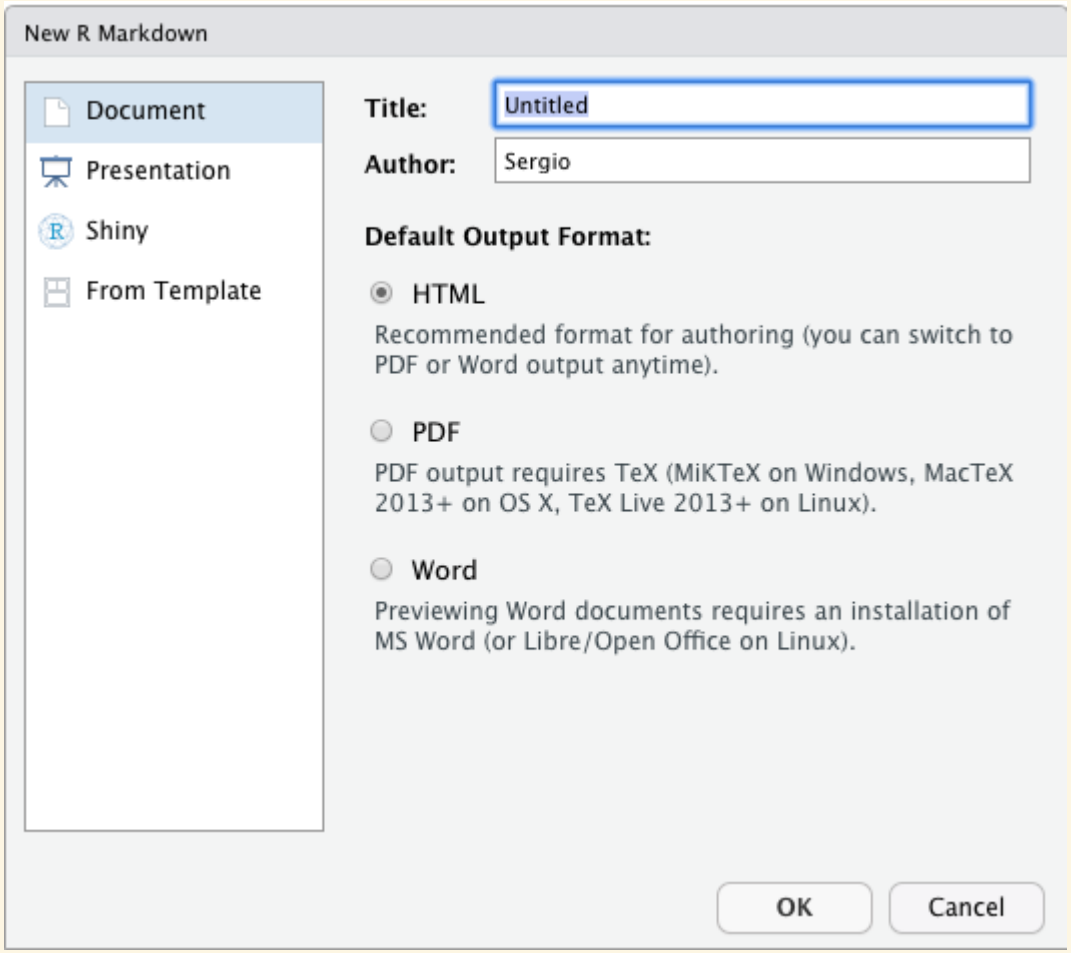
Rmarkdown

Why?

- Save and execute code
- Generate reports which can easily shared
- Support for different documents (notebooks, presentations, books, scientific articles, etc.)
- Code and text in the same document

How?


RStudio
File
Edit
Code
View
Plots
Session
Build
Debug
Profile
Tools
Window
Help
mån 13:00
Sergio
RStudio
New File
New Project...
Open File... ⌘O
Reopen with Encoding...
Recent Files
Open Project...
Open Project in New Session...
Recent Projects
Import Dataset
Save ⌘S
Save As...
Save with Encoding...
Save All ⌥⌘S
Knit Document ⇧⌘K
Compile Report...
Print...
Close ⌘W
Close All ⇧⌘W
Close All Except Current ⌥⇧⌘W
Close Project
Quit Session... ⌘Q
R Script ⇧⌘N
R Notebook
R Markdown...
Shiny Web App...
Plumber API...
Text File
C++ File
Python Script
SQL Script
Stan File
D3 Script
R Sweave
R HTML
R Presentation
R Documentation
Source
Console
Project: (None)

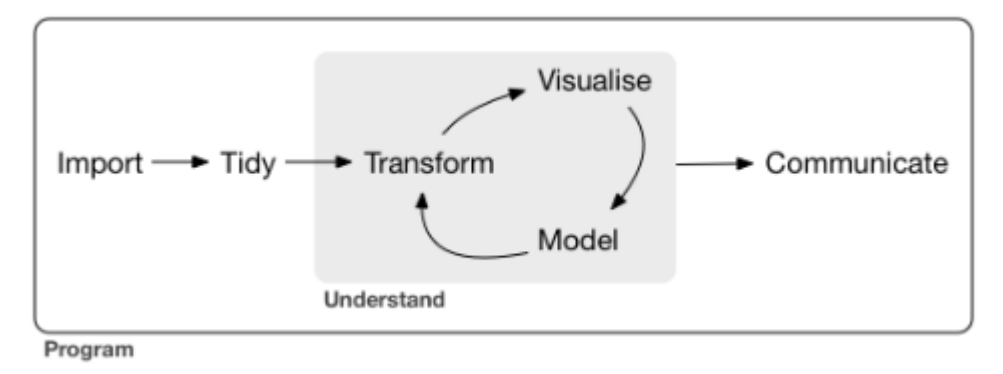


- Read more here
- Let me know if interested in a workshop on RMarkdown

Tidyverse

What is?

- Collection of packages for data science
- They share a common language
- Made to work well together



Tidyverse pipeline

Tidy Data

- Three rules:
  1. Each variable must have its own column
  2. Each observation must have its own row
  3. Each value must have its own cell

Data Structures

Vectors

- Two types of vectors
  1. Atomic Vectors
     - logical
     - numeric (double + integer)
     - character
     - complex
     - raw
  2. Lists
  3. Augmented vectors
     - Factors
     - Dates
     - Date-times
     - Tibbles

Atomic Vectors

Logical Vectors

- Three values
  - FALSE
  - TRUE
  - NA

Example:

```
1:10 %% 3 == 0
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

Numeric Vectors

- Include both integer and double vectors
- Default is double

Example:

```
sqrt(2)**2 == 2
```

```
## [1] FALSE
```

```
as.integer(sqrt(2)**2)==2
```

```
## [1] TRUE
```

Character Vectors

- Complex

Example

```
x <- "This is a reasonably long string."
pryr::object_size(x)
```

```
## 152 B
```

```
y <- rep(x, 1000)
pryr::object_size(y)
```

```
## 8.14 kB
```

Using atomic vectors

Converting between types

```
x <- 1
str(x)
```

```
##  num 1
```

```
# Numeric
str(as.integer(x))
```

```
##  int 1
```

```
# Character
str(as.character(x))
```

```
##  chr "1"
```

```
# Logical
str(as.logical(x))
```

```
##  logi TRUE
```

Getting vector type

- There are a couple of functions one can use in this case. Example:

```
typeof(x)
```

```
## [1] "double"
```

```
is_atomic(x)
```

```
## [1] TRUE
```

```
is_double(x)
```

```
## [1] TRUE
```

```
is_logical(x)
```

```
## [1] FALSE
```

Rename

- Example:

```
x <- c(a=1, b=2, c=3)
x
```

```
## a b c
## 1 2 3
```

```
set_names(x,c('x','y','z'))
```

```
## x y z
## 1 2 3
```

Get elements of interest

- Use single '[' for that
- Example:

```
x[1]
```

```
## a
## 1
```

```
x[2]
```

```
## b
## 2
```

Vector recycling

- Some operations recycle vectors if they're of different sizes, for example:

```
data.frame(x=1:4,y=1:2)
##   x y
## 1 1 1
## 2 2 2
## 3 3 1
## 4 4 2
```

- However, tidyverse prevents this from happening to anything which is not a scalar

```
tibble(x=1:4,y=rep(1:2,2))
## # A tibble: 4 x 2
##       x     y
##   <int> <int>
## 1     1     1
## 2     2     2
## 3     3     1
## 4     4     2
```

Lists

Lists

- Also called recursive vectors (can store multiple lists)
- Created by using **list()**

Useful commands

- **str()**: shows the structure of a list without focusing on the contents

```
str(example_list)
```

```
## List of 3
##  $ x: int [1:4] 1 2 3 4
##  $ y: chr [1:26] "a" "b" "c" "d" ...
##  $ z: logi TRUE
```

- '[' to extract sublists

```
str(example_list[2])
```

```
## List of 1
##  $ y: chr [1:26] "a" "b" "c" "d" ...
```

- '[[' to extract a single component from a list
- Can also reference by name by using '$'

```
str(example_list[[2]])
```

```
##  chr [1:26] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" ...
```

```
str(example_list$y)
```

```
##  chr [1:26] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" ...
```

```
example_list[[2]][1]
```

```
## [1] "a"
```

Augmented Vectors

Augmented Vectors

- Vectors + metadata

Factors

- Represent categorical data
- Example:

```
example_factor <- factor(c('ab','cd','ab'),levels = c('ab','cd','ef'))
example_factor
```

```
## [1] ab cd ab
## Levels: ab cd ef
```

Dates

- Numeric vectors
- Example:

```
example_date <- as.Date('1971-01-01')
unclass(example_date)
```

```
## [1] 365
```

Date-time

- Also numeric vectors
- *lubridate* package
- You also have a *tzone* attribute which you can choose depending on where you are
- Example:

```
example_datetime <- ymd_hm('1970-01-01 01:00')
unclass(example_datetime)
```

```
## [1] 3600
## attr(,"tzone")
## [1] "UTC"
```

Tibbles

- Based on lists
- Example

```
nycflights13::flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # … with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Strings

Strings

- **Stringr** package
- Created by using ' or "
- Special characters added by using '\'
- Helpful characters:
  - '\t' for tab
  - '\n' newline
  - run ?" ' " for more

Useful functions

- **str_length()** tells you the number of characters in a string
- **str_c()** to combine strings
- **str_replace()** to replace patterns
- **str_sub()** extracts parts of a string giving start and end positions
- **str_to_lower()** or **str_to_upper()** to convert to upper or lower case. Can take language in the *locale* argument
  - Language code based on ISO 639
- **str_order()** and **str_sort()** to sort strings (also take *locale* argument)

Examples

```
example_string <- c('Apple','Banana','Pear')
str_length(example_string)
## [1] 5 6 4
```

```
str_c(example_string,collapse = ' ')
## [1] "Apple Banana Pear"
```

```
str_c('Apple','Banana',sep = ', ')
## [1] "Apple, Banana"
```

```
str_sort(example_string,decreasing = TRUE,locale = 'en')
## [1] "Pear"   "Banana" "Apple"
```

```
str_sub(example_string,1,-2)
## [1] "Appl"  "Banan" "Pea"
```

```
str_to_upper(example_string,locale = 'en')
## [1] "APPLE"  "BANANA" "PEAR"
```

Matching patterns with regular expressions

- Use regular expresions
- Check this website for regular expressions

Examples:

```
str_view_all(example_string,pattern = '[aeiou]')
```

```
Apple
Banana
Pear
```

Data Import

Data Import

- For this we're again using the tidyverse, namely a package called readr
- Most functions in the package focus on turning flat files in data frames

Functions

- **read_csv()** reads comma delimited files
- **read_csv2()** reads semicolon separated files
- **read_tsv()** reads tab separated files
- **read_delim()** reads files with any delimiters

Attributes

- Use *skip=n* to skip the first *n* lines
- Use *comment= '#'* to drop all lines which start with for example #
- *col_names* uses the first ine as headings
- *na* lets you specify how to treat missing values in your data

Parsing data

- **parse_()** functions take a vector and return a more specified vector
- Example:

```
str(parse_logical(c("TRUE", "FALSE", "NA")))
```

```
##  logi [1:3] TRUE FALSE NA
```

```
str(parse_date(c("2010-01-01", "1979-10-14")))
```

```
##  Date[1:2], format: "2010-01-01" "1979-10-14"
```

- If parsing fails, you get a warning, and features will be missing from output
- Parsing numbers:
  - numbers are written differently in differnt parts of the world
  - numbers can be surrounded by other characters '$', '£'
  - numbers may contain grouping characters ','
  - use *locale* attribute

```
parse_double("1,23", locale = locale(decimal_mark = ","))
## [1] 1.23
parse_double("1.23", locale = locale(decimal_mark = "."))
## [1] 1.23
parse_double("1,23")
## Warning: 1 parsing failure.
## row col               expected actual
##   1  -- no trailing characters    ,23
## [1] NA
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected               actual
##   <int> <int> <chr>                  <chr>
## 1     1    NA no trailing characters ,23
```

```r
# Used in America
parse_number("$123,456,789")
```

```
## [1] 123456789
```

```r
# Used in many parts of Europe
parse_number("123.456.789", locale = locale(grouping_mark = "."))
```

```
## [1] 123456789
```

```r
# Used in Switzerland
parse_number("123'456'789", locale = locale(grouping_mark = "'"))
```

```
## [1] 123456789
```

Parsing strings

- Characters depend on encoding
  - English is well represented in ASCII
  - Readr uses UTF-8
  - understands every possible character
  - may cause problems with older systems that do not understand UTF-8

Parsing factors

- Give **parse_vector()** a *levels* attribute to give back a warning whenever an unexpected value is present
- Example:

```
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
```

```
## Warning: 1 parsing failure.
## row col           expected   actual
##   3  -- value in level set bananana

## [1] apple  banana <NA>
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected           actual
##   <int> <int> <chr>              <chr>
## 1     3    NA value in level set bananana
## Levels: apple banana
```

Parsing Problems

- **readr** guesses the parsing using the first 1000 rows
  - Might have issues after the first 1000 rows
  - First rows might contain several missing values
- Solution:
  - set *col_types* attribute in read_ function
  - for every *parse_* there's a *col_* option

Example:

```
challenge <- read_csv(readr_example('challenge.csv'))
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_logical()
## )
## Warning: 1000 parsing failures.
##  row col           expected     actual
## 1001   y 1/0/T/F/TRUE/FALSE 2015-01-16 '/Library/Frameworks/R.fran
## 1002   y 1/0/T/F/TRUE/FALSE 2018-05-18 '/Library/Frameworks/R.fran
## 1003   y 1/0/T/F/TRUE/FALSE 2015-09-05 '/Library/Frameworks/R.fran
## 1004   y 1/0/T/F/TRUE/FALSE 2012-11-28 '/Library/Frameworks/R.fran
## 1005   y 1/0/T/F/TRUE/FALSE 2020-01-13 '/Library/Frameworks/R.fran
## .... ... .................. .......... ..........................
## See problems(...) for more details.
tail(challenge)
## # A tibble: 6 x 2
##       x y
##   <dbl> <lgl>
## 1 0.805 NA
## 2 0.164 NA
## 3 0.472 NA
```

```
challenge <- read_csv(readr_example('challenge.csv'),col_types = cols
tail(challenge)
```

```
## # A tibble: 6 x 2
##       x y
##   <dbl> <date>
## 1 0.805 2019-11-21
## 2 0.164 2018-03-29
## 3 0.472 2014-08-04
## 4 0.718 2015-08-16
## 5 0.270 2020-02-04
## 6 0.608 2019-01-06
```

Data Manipulation

Overview

- Syntax
- *Dplyr* package
- *Magrittr* package
- Examples

Syntax

- In tidyverse, verbs work in a similar way
    1. Data frame
    2. Arguments on what to do with it
    3. resulst in a new data frame

Dplyr package

- Pick observations by values
- Reorder rows
- Pick variables by names
- Create new variables
- Summary of values

Pick observations by values

- use verb **filter()**
  - used to filter rows
- Example (*nycflights13* package):

```
# select flights on the 1st of January
filter(flights,month==1,day==1)
## # A tibble: 842 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # … with 832 more rows, and 12 more variables: sched_arr_time <int
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour
```

Reorder rows

- Use verb **arrange()**
  - similar to filter, but changes the order of rows
- Example:

```
arrange(flights, year, month, sort(day,decreasing = TRUE))
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1    31     1446           1450        -4     1650
##  2  2013     1    31     1447           1450        -3     1811
##  3  2013     1    31     1447           1415        32     1540
##  4  2013     1    31     1448           1303       105     1635
##  5  2013     1    31     1449           1445         4       NA
##  6  2013     1    31     1450           1329        81     1804
##  7  2013     1    31     1452           1330        82     1753
##  8  2013     1    31     1453           1455        -2     1623
##  9  2013     1    31     1454           1455        -1     1833
## 10  2013     1    31     1454           1445         9     1802
## # … with 336,766 more rows, and 12 more variables: sched_arr_time
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour
## #   minute <dbl>, time_hour <dttm>
```

Pick variables by name

- Use verb **select()**
  - select columns
  - good to narrow down values
  - helper function:
    - *starts_with*
    - *ends_with*
    - *contains*
    - *matches* (regular expression)
    - *num_range*
    - *everything*

Examples

```
select(flights, year:day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # … with 336,766 more rows
```

```
select(flights, starts_with('dep'))
```

```
## # A tibble: 336,776 x 2
##    dep_time dep_delay
##       <int>     <dbl>
##  1      517         2
##  2      533         4
##  3      542         2
##  4      544        -1
##  5      554        -6
##  6      554        -4
##  7      555        -5
##  8      557        -3
##  9      557        -3
## 10      558        -2
## # … with 336,766 more rows
```

```
select(flights, contains('time'))
```

```
## # A tibble: 336,776 x 6
##    dep_time sched_dep_time arr_time sched_arr_time air_time
##       <int>          <int>    <int>          <int>    <dbl>
##  1      517            515      830            819      227
##  2      533            529      850            830      227
##  3      542            540      923            850      160
##  4      544            545     1004           1022      183
##  5      554            600      812            837      116
##  6      554            558      740            728      150
##  7      555            600      913            854      158
##  8      557            600      709            723       53
##  9      557            600      838            846      140
## 10      558            600      753            745      138
## # … with 336,766 more rows, and 1 more variable: time_hour <dttm>
```

Create new variables

- Use verb **mutate**
  - adds new columns to the end of the dataset
  - also possible to use variables just created
- Verb **transmute** only keeps new variables
- Examples:

```
mutate(flights, gain= dep_delay - arr_delay, speed= distance/air_time
## # A tibble: 336,776 x 22
##     year month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>
##  1  2013     1     1      517            515         2      830
##  2  2013     1     1      533            529         4      850
##  3  2013     1     1      542            540         2      923
##  4  2013     1     1      544            545        -1     1004
##  5  2013     1     1      554            600        -6      812
##  6  2013     1     1      554            558        -4      740
##  7  2013     1     1      555            600        -5      913
##  8  2013     1     1      557            600        -3      709
##  9  2013     1     1      557            600        -3      838
## 10  2013     1     1      558            600        -2      753
## # … with 336,766 more rows, and 15 more variables: sched_arr_time
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
```

```
transmute(flights, gain= dep_delay - arr_delay, speed= distance/air_t
## # A tibble: 336,776 x 3
##     gain speed gain_per_hour
##    <dbl> <dbl>         <dbl>
##  1    -9  370.        -1.8
##  2   -16  374.        -3.2
##  3   -31  408.        -6.2
##  4    17  517.         3.4
##  5    19  394.         3.17
##  6   -16  288.        -3.2
##  7   -24  404.        -4
##  8    11  259.         1.83
##  9     5  405.         0.833
## 10   -10  319.        -1.67
## # … with 336,766 more rows
```

Summaries

- Use verb **summarise**
  - collapses data
  - use together with **group_by**

Examples:

```
by_day <- group_by(flights, year
by_day
## # A tibble: 336,776 x 19
## # Groups:   year, month, day
##     year month   day dep_time
##    <int> <int> <int>    <int>
##  1  2013     1     1      517
##  2  2013     1     1      533
##  3  2013     1     1      542
##  4  2013     1     1      544
##  5  2013     1     1      554
##  6  2013     1     1      554
##  7  2013     1     1      555
##  8  2013     1     1      557
##  9  2013     1     1      557
## 10  2013     1     1      558
## # … with 336,766 more rows, a
## #   arr_delay <dbl>, carrier
## #   origin <chr>, dest <chr>,
## #   minute <dbl>, time_hour <
```

```
summarise(by_day, delay= mean(de
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day delay
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
## 10  2013     1    10  2.84
## # … with 355 more rows
```

Magrittr Package

- Pipe operator ' %>% '
- Useful to tie operations together
- Easier to read code
- '.' is a place holder

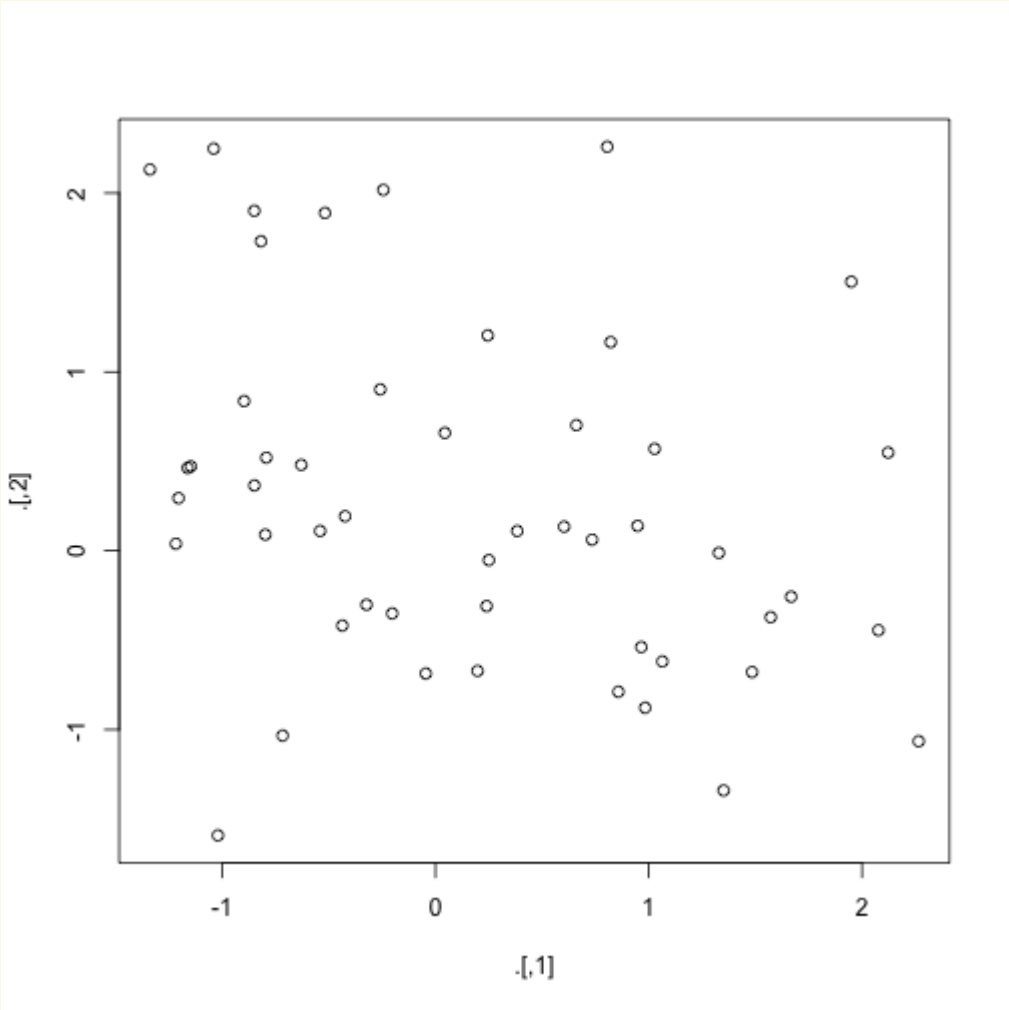
- Example:

```
flights %>%
  group_by(., year, month, day)
  summarise(.,delay= mean(dep_de
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##     year month   day delay
##    <int> <int> <int> <dbl>
##  1  2013     1     1 11.5
##  2  2013     1     2 13.9
##  3  2013     1     3 11.0
##  4  2013     1     4  8.95
##  5  2013     1     5  5.73
##  6  2013     1     6  7.15
##  7  2013     1     7  5.42
##  8  2013     1     8  2.55
##  9  2013     1     9  2.28
## 10  2013     1    10  2.84
## # … with 355 more rows
```

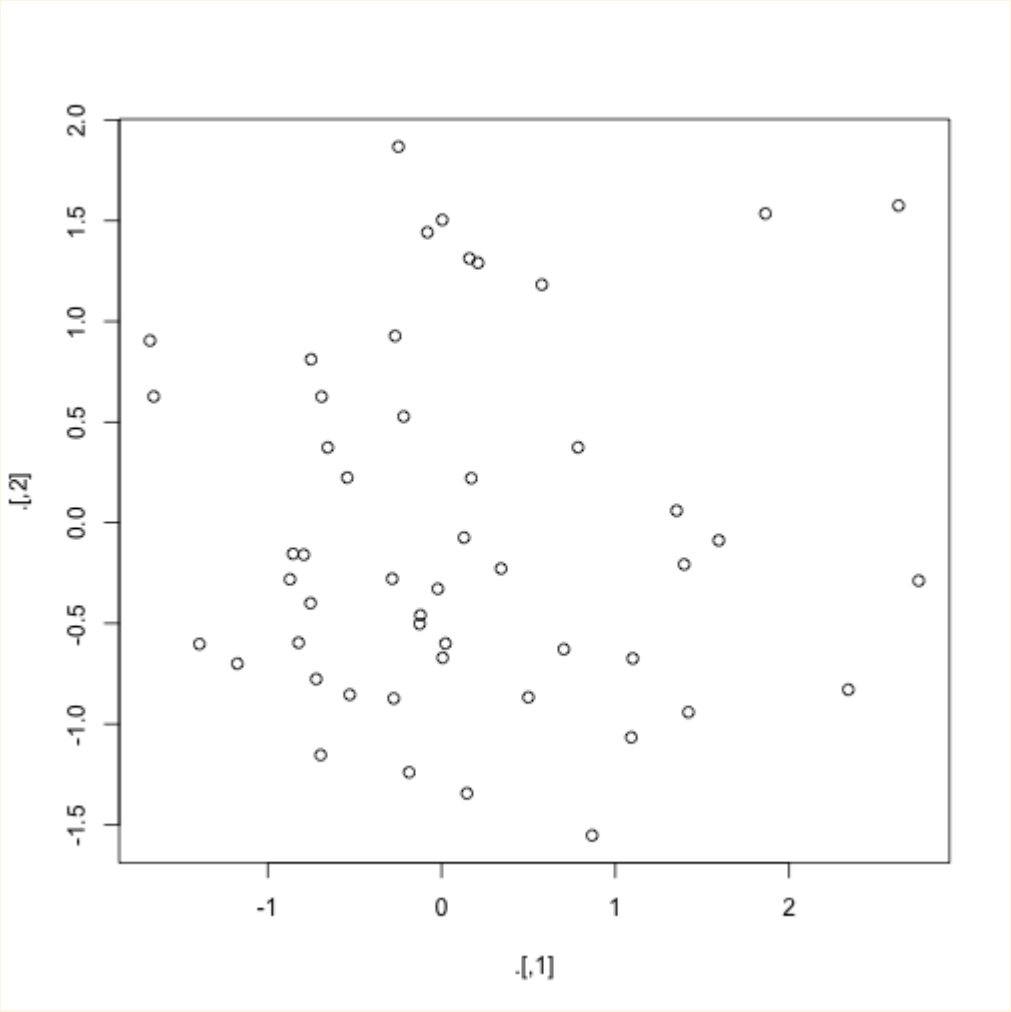
Other pipes

- **%T>%** returns the right-hand side instead of the left
  - useful with functions that don't return anything, e.g. plot
- **%$%** 'explodes' the data
  - Useful to refer to variables by name when working with base R

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
```



```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
```



```
mtcars %$%
  cor(disp,mpg)
## [1] -0.8475514
```