Programming Camp Day 4

Augustus Kmetz

2025-09-12

Welcome back!

Plans for today:

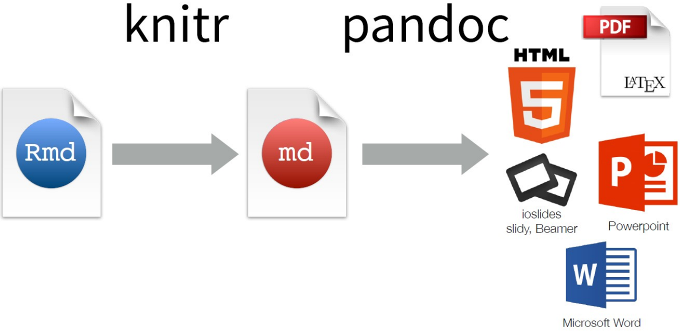
- Rmarkdown
- Control flow
- Functions
- Functionals

Rmarkdown

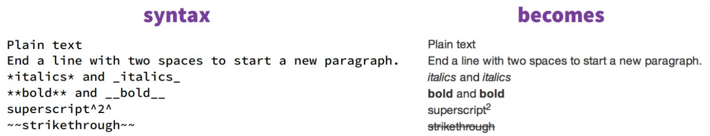
Logistics

1. Knitr runs the document in a fresh R session which means you need to load the libraries that the document uses **in the document**
2. Objects made in one code chunk will be available in later code chunks

Logistics


knitr
pandoc
Rmd
md
HTML
PDF
LaTeX
ioslides
slidy, Beamer
Powerpoint
Microsoft Word

Text



syntax

```
Plain text
End a line with two spaces to start a new paragraph.
*italics* and _italics_
**bold** and __bold__
superscript^2^
~~strikethrough~~
```

becomes

Plain text

End a line with two spaces to start a new paragraph.

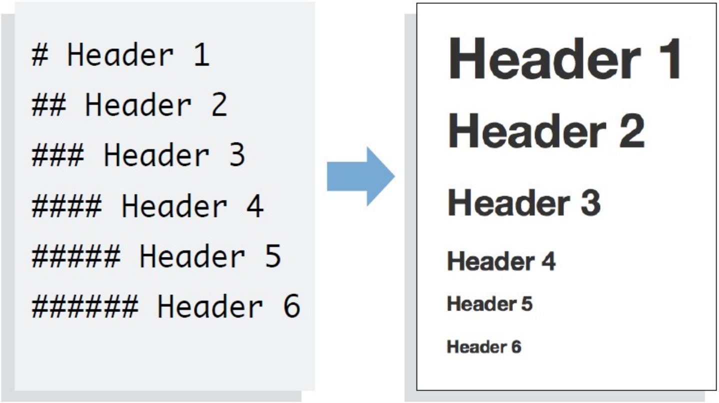
italics and *italics*

bold and **bold**

superscript^2

~~strikethrough~~

Headers



```
# Header 1
## Header 2
### Header 3
#### Header 4
##### Header 5
###### Header 6
```

Header 1

Header 2

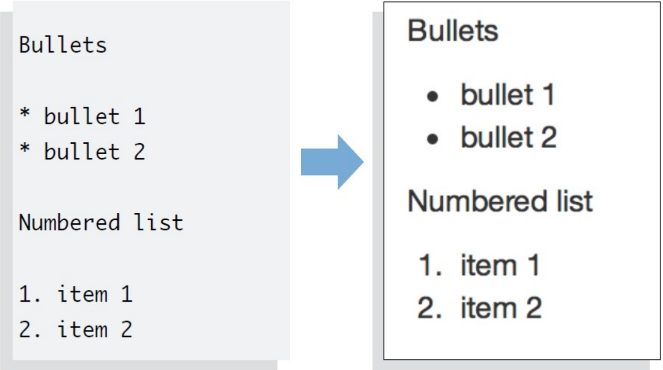
Header 3

Header 4

Header 5

Header 6

Lists



```
Bullets

* bullet 1
* bullet 2

Numbered list

1. item 1
2. item 2
```

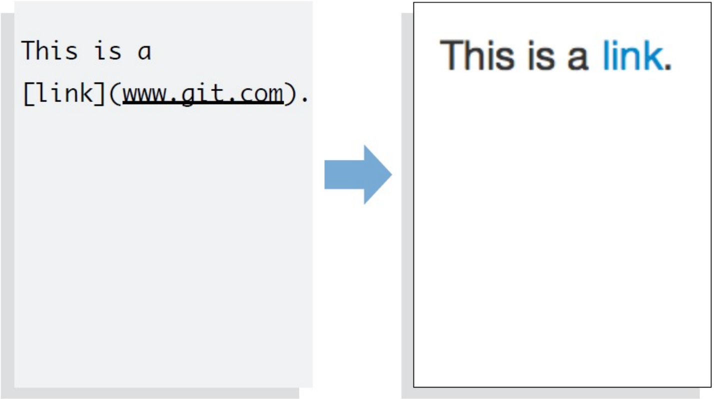
Bullets

- bullet 1
- bullet 2

Numbered list

1. item 1
2. item 2

Hyperlinks

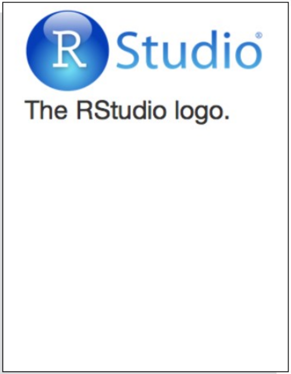


```
This is a
[link](www.git.com).
```

This is a link.

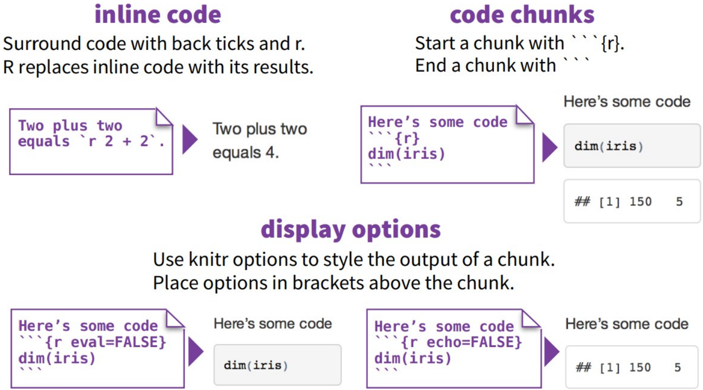
Images

```
![](images/1.png)
The RStudio logo.
```



The RStudio logo.

Code



inline code

Surround code with back ticks and r.
R replaces inline code with its results.

```
Two plus two
equals `r 2 + 2`.
```

Two plus two
equals 4.

code chunks

Start a chunk with ```{r}.
End a chunk with ```

~~~
Here’s some code
```{r}
dim(iris)
```
~~~

Here’s some code

```
dim(iris)
```

```
## [1] 150   5
```

display options

Use knitr options to style the output of a chunk.
Place options in brackets above the chunk.

~~~
Here’s some code
```{r eval=FALSE}
dim(iris)
```
~~~

Here’s some code

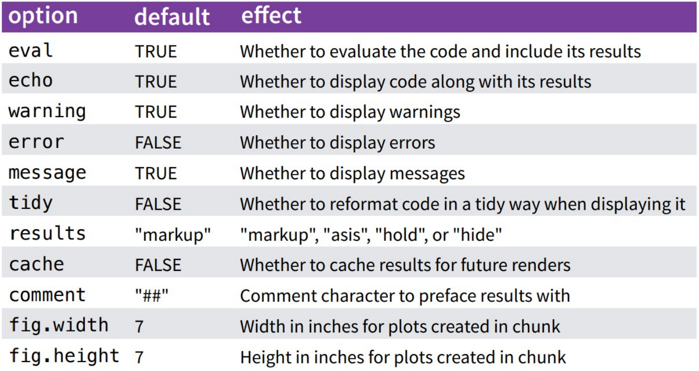
```
dim(iris)
```

~~~
Here’s some code
```{r echo=FALSE}
dim(iris)
```
~~~

Here’s some code

```
## [1] 150   5
```

Chunk options



option	default	effect
`eval`	TRUE	Whether to evaluate the code and include its results
`echo`	TRUE	Whether to display code along with its results
`warning`	TRUE	Whether to display warnings
`error`	FALSE	Whether to display errors
`message`	TRUE	Whether to display messages
`tidy`	FALSE	Whether to reformat code in a tidy way when displaying it
`results`	"markup"	"markup", "asis", "hold", or "hide"
`cache`	FALSE	Whether to cache results for future renders
`comment`	"##"	Comment character to preface results with
`fig.width`	7	Width in inches for plots created in chunk
`fig.height`	7	Height in inches for plots created in chunk

(Regression) tables

stargazer package is great for making ice regression tables that work well with markdown

```
stargazer(weather_delay_model_fe, header = F)
```

helpful tips for formatting tables

When making a latex document need to set the chunk option as result = asis

Your turn!

Put together a documents or presentation that contains:

1. At least one graph
2. At least one equation
3. At least one table

Control Flow + Functions

Introduction

You can go a long way in R without doing things that are typically associated with computer programming like writing loops and functions

Useful to learn basic programming constructs

Choices

The basic form of an if statement in R is:

```
if (condition) true_action
if (condition) true_action else false_action
```

If `condition` is `TRUE`, `true_action` is evaluated; if `condition` is `FALSE`, the optional `false_action` is evaluated.

if, else

Typically actions are compounded statements contained within {}

```
if(<condition>) {
        ## do something
}
else {
        ## do something else
}
```

if, elseif, else

```
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
} else {
        ## do something different
}
```

if

```r
x <- 8
if (x >= 10) {
  print("x is greater than or equal to 10")
}
```

if, else

```r
x <- 8
if (x >= 10) {
  print("x is greater than or equal to 10")
} else {
  print("x is less than 10")
}
```

```
[1] "x is less than 10"
```

if, elseif, else

```
x <- 8

if (x >= 10) {

  print("x is greater than or equal to 10")

} else if (x > 5) {

  print("x is greater than 5, but less than 10")

} else {

  print("x is less than 5")

}
```

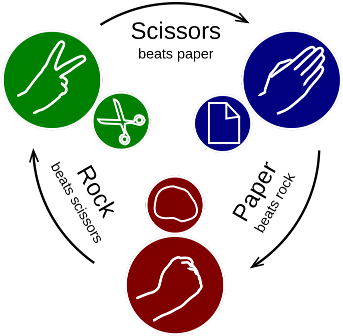
```
[1] "x is greater than 5, but less than 10"
```

Your turn!

Write a sequence of if, else, and ifelse that will correctly return the result of a rock - paper - scissors game given the values of

- player_1 is one of c("rock", "paper", "scissors")
- player_2 is one of c("rock", "paper", "scissors")

use the sample() function to generate random values for player_1 and player_2 to check your code



Your turn!

```r
player_1 <- sample(c("rock", "paper", "scissors"))
player_2 <- sample(c("rock", "paper", "scissors"))

if (player_1 == player_2){

  print("Tie")

} else if ((player_1 == "rock" & player_2 == "scissors") |
           (player_1 == "scissors" & player_2 == "paper") |
           (player_1 == "paper" & player_2 == "rock")){

  print("player 1 wins!")

} else {

  print("player 2 wins!")
}
```

ifelse() and case_when()

Two function very useful for creating conditional variables

```
df %>%
  mutate(new_variable = ifelse(test, yes, no))
```

`case_when()` is the generalized version

```
df %>%
  mutate(new_variable = ifelse(test1 ~ value_1,
                               test2 ~ value_2,
                               test3 ~ value_3,
                               T ~ all_other_cases))
```

for()

- If you want to iterate over a set of values, when the order of iteration is important, and perform the same operation on each, a for() loop is the correct tool for the job
- **Avoid** using for() loops unless the order of iteration is important: i.e. the calculation at each iteration depends on the results of previous iterations
- If the order of iteration is not important, then you should use vectorized alternatives (coming soon)

for()

```
for (iterator in set of values) {
  do a thing
}
```

```
for (i in 1:10) {
  print(i)
}
```

```
for (i in 1:5) {
  for (j in c('a', 'b', 'c', 'd', 'e')) {
    print(paste(i,j))
  }
}
```

Tips!

```r
output_vector <- c()

tic()
for (i in 1:1000) {
  for (j in c('a', 'b', 'c', 'd', 'e')) {
    temp_output <- paste(i, j)
    output_vector <- c(output_vector, temp_output)
  }
}
toc()
```

```
0.16 sec elapsed
```

Tips!

- **DO NOT** build a results object (vector, list, matrix, data frame) as your for loop progresses
- Computers are very bad at handling this
- It's much better to define an empty results object before hand of appropriate dimensions, rather than initializing an empty object without dimensions

Tips!

```r
output_matrix <- matrix(nrow = 1000, ncol = 5)
j_vector <- c('a', 'b', 'c', 'd', 'e')

tic()
for (i in 1:1000) {
  for (j in 1:5) {
    temp_j_value <- j_vector[j]
    temp_output <- paste(i, temp_j_value)
    output_matrix[i, j] <- temp_output
  }
}
toc()
```

```
0.03 sec elapsed
```

while()

```
while(this condition is true){
  do a thing
}
```

```
# As an example, here’s a while loop that generates random
# numbers from a uniform distribution (the runif() function)
# between 0 and 1 until it gets one that’s less than 0.1.
z <- 1

while(z > 0.1){
  z <- runif(1)
  cat(z, "\n")
}
```

while()

You have to be particularly careful that you don't end up stuck in an infinite loop because your condition is always met and hence the while statement never terminates

```r
z <- 1

iter <- 1

while((z > 0.1) & (iter < 1000)){

  z <- runif(1)
  cat(z, "\n")

  iter <- iter + 1

}
```

Your turn!

Write a `for` loop that runs 100 times. In each iteration:

- Take a random 10% sample of the `nyc_weather_delays` data (`sample_frac()` or `slice_sample()` may be useful functions)
- Run a regression of flight delay on precipitation
- Save the coefficient on precipitation (`tidy()` from the `broom` package may be a useful function)

Plot the distribution of coefficients

How long does it take the code to run? (`tic()` and `toc()` from the `tictoc` package may be helpful)

Your turn!

```
nyc_flights <-
  flights %>%
  group_by(carrier, year, month, day, origin) %>%
  summarise(dep_delay = mean(dep_delay, na.rm = T),
            max_dep_delay = max(dep_delay, na.rm = T),
            frac_delayed = sum(dep_delay > 0, na.rm = T) / n()
            frac_cancelled = sum(is.na(dep_time )) / n(),
            n = n()) %>%
  filter(n >= 10)
```

Your turn!

```
nyc_weather <-
  weather %>%
  group_by(year, month, day, origin) %>%
  summarise(temp     = mean(temp, na.rm = TRUE),
            min_temp = min(temp, na.rm = TRUE),
            max_temp = max(temp, na.rm = TRUE),
            wind     = mean(wind_speed, na.rm = TRUE),
            max_wind = max(wind_speed, na.rm = TRUE),
            precip = sum(precip, na.rm = TRUE))

nyc_weather_delays <-
  nyc_flights %>%
  inner_join(nyc_weather, by = c("year", "month", "day", "orig
```

Your turn!

```
betas <-
  tibble(n = 1:100,
         beta = 0)



for(i in 1:100){

  temp_model <-
    lm(dep_delay ~ precip,
       sample_frac(nyc_weather_delays, 0.25, replace = T)) %>%
    broom::tidy()

  betas$beta[i] <- temp_model$estimate[2]

}
```

Your turn!

```
betas %>%
  ggplot(aes(x = beta)) +
  geom_histogram(bins = 20)
```

Functions

Creating functions

The general structure of a function is

```
my_function <- function(parameters) {
  # perform action
  # return value
}
```

```
fahr_to_cel <- function(temp) {
  cel <- ((temp - 32) * (5 / 9))
  return(cel)
}

fahr_to_cel(32)
```

```
[1] 0
```

```
fahr_to_cel(100)
```

```
[1] 37.77778
```

Documenting your functions

```r
root_mean_squared_error <- function(predicted, targets){
    # Computes root mean squared error between two vectors
    #
    # Args:
    #    predicted: a numeric vector of predictions
    #    targets: a numeric vector of target values for each p
    #
    # Returns:
    #    The root mean squared error between predicted values

    sqrt(mean((targets - predicted) ^ 2))
}
```

your turn!

write a function `rock_paper_scissors()` that “plays” rock paper scissors

- The inputs should be what `player_1` and `player_2` played
- It should return the winner

your turn!

```
rock_paper_scissors <- function(player_1, player_2){

  if (player_1 == player_2){

    return("Tie")

  } else if ((player_1 == "rock" & player_2 == "scissors") |
             (player_1 == "scissors" & player_2 == "paper") |
             (player_1 == "paper" & player_2 == "rock")){

    return("player 1 wins!")

  } else {

    return("player 2 wins!")
  }

}
```

Defensive programming

- it is important to ensure that functions only work in their intended use-cases
- Checking function parameters is related to the concept of defensive programming
- Basic idea - frequently check conditions and throw an error if something is wrong
- These checks are referred to as assertion statements because we want to assert some condition is TRUE before proceeding
- They make it easier to debug because they give us a better idea of where the errors originate.

Defensive programming

```r
fahr_to_cel <- function(temp) {

  if (!is.numeric(temp)) {
    stop("temp must be a numeric vector.")
  }

  cel <- ((temp - 32) * (5 / 9))

  return(cel)
}

fahr_to_cel("augustus")
```

Modify your `rock_paper_scissors()` function to restrict inputs

your turn!

```
rock_paper_scissors <- function(player_1, player_2){

  if(!(player_1 %in% c("rock", "paper", "scissors") &
       player_2 %in% c("rock", "paper", "scissors"))){

    stop("The only valid inputs are rock, paper, or scissors")
  }

  if (player_1 == player_2){

    return("Tie")

  } else if ((player_1 == "rock" & player_2 == "scissors") |
             (player_1 == "scissors" & player_2 == "paper") |
             (player_1 == "paper" & player_2 == "rock")){

    return("player 1 wins!")
```

Your turn!

Convert your for loop for extracting betas to a function where the input is the iteration number

It should return a tibble with the beta and the iteration number

Your turn!

```r
beta <- function(n){

  temp_model <-
    lm(dep_delay ~ precip,
       sample_frac(nyc_weather_delays, 0.25, replace = T)) %>%
    broom::tidy()


  return(tibble(beta =  temp_model$estimate[2],
                n = n))
}

beta(1)
```

Functionals

purrr

- Very often we find ourselves looping over a vector, doing something for each element and saving the results
- purrr provides a family of functions to do it for you
- The two benefits are (marginal) speed improvements and clarity of code
- … but don't let beautifying code get in the way of solving the problem

map()

```
map(.x, .f, ...)
```

- .x - a vector, a list, a data frame
- .f - a function

`map()`

How many starships has each character been in?

For each person in `sw_people`, count the number of starships

```
map(sw_people, .f, ...)
```

Strategy

1. Do it for one element
2. Turn it into a recipe
3. Use `map()` to do it for all elements

Do it for one element

How many star ships has Luke been in?

```r
luke <- sw_people[[1]]
```

```r
luke$starships
```

```r
length(luke$starships)
```

Turn it into a recipe

Make it a formula!

(Use `.x` as a pronoun)

```
~ length(.x$starships)
```

Do it for all!

```r
map(sw_people, ~ length(.x$starships))
```

map()

map() always returns a list

but there are other types of output!

- map_lgl() logical vector
- map_int() integer vector
- map_dbl() double vector
- map_chr() character vector

walk() - when you want nothing at all, use a function of its side effects

Your turn!

Replace `map()` with appropriately typed function

```r
# How many starships has each character been in?
map_(sw_people, ~ length(.x$starships))

# What color is each character's hair?
map_(sw_people, ~.x[["hair_color"]])

# Is the character male?
map_(sw_people, ~.x[["gender"]] == "male")
```

.f

.f can be:

- a formula
- a string or integer

```
map_chr(sw_people, ~.x[["hair_color"]])
# for each element extract the named/numbered element
map_chr(sw_people, "hair_color")
```

- a function

```
map_int(sw_people, ~ length(.x$starships))
map(sw_people, "starships") %>% map_int(length)
```

Your turn!

Which film (see `sw_films`) has the most characters?

Your turn!

```
map(sw_films, "characters") %>%
  map_int(length) %>%
  set_names(map_chr(sw_films, "title"))
```

your turn

Use map to run your function 100 times

your turn

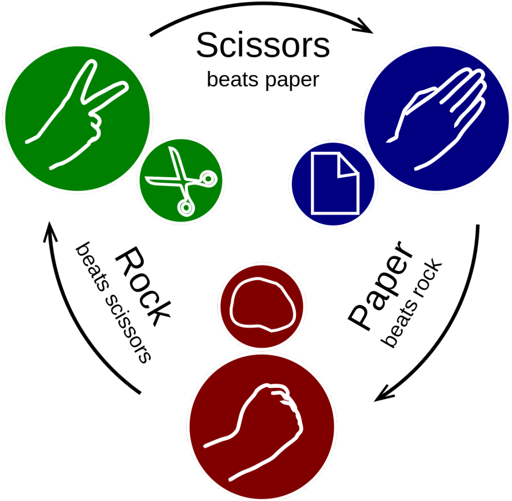
```r
tic()
estimates <- map_dfr(1:100, beta)
toc()
```

parallel computing

Parallel & distributed processing can be used to:

- speed up processing
- lower memory footprint (per machine)

purrr and furrr



quick and dirty paralleization

```r
plan(multisession, workers = 2)

future_map(c("hello", "world"), ~.x)

tic()
nothingness <- future_map(c(2, 2, 2), ~Sys.sleep(.x))
toc()
```

quick and dirty paralleization

- Remember that data has to be passed back and forth between the workers
- Whatever performance gain you might have gotten from parallelization can be crushed by moving large amounts of data around
- **Tip** consider only returning a smaller piece rather than a whole data set/model/etc

Your turn!

What are the speed gains from parallelizing your bootstrap procedure?

Your turn!

```r
furrr_options(seed = T)
set.seed(42)
tic()
estimates <-
  future_map_dfr(1:100,
                 beta,
                 .options= furrr_options(seed = 42))
toc()
```