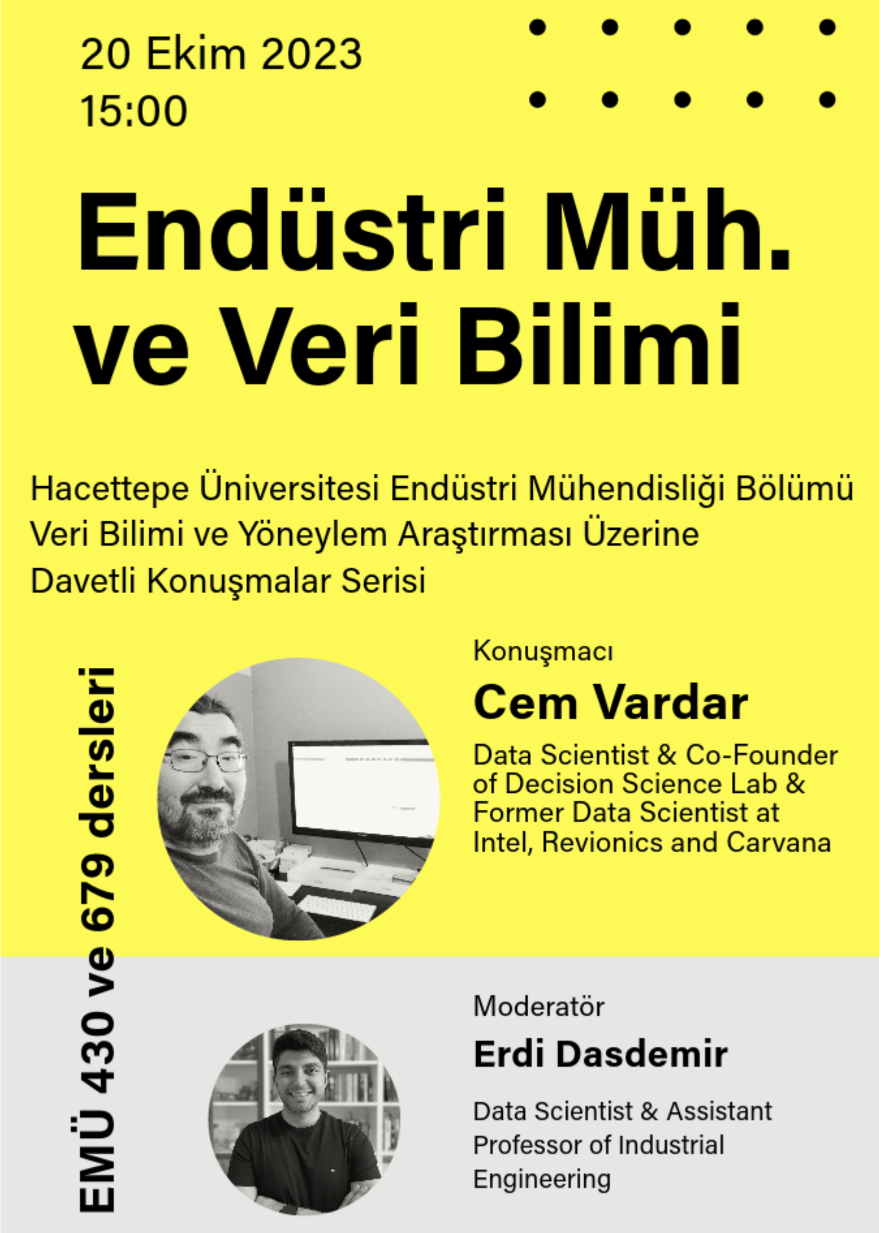
Hacettepe University
Department of Industrial Engineering
Undergraduate Program
2023-2024 Fall

EMU 430 – Data Analytics
Week3
October 20, 2023

Instructor: Erdi Dasdemir

edasdemir@hacettepe.edu.tr
www.erdidasdemir.com



20 Ekim 2023
15:00

Endüstri Müh. ve Veri Bilimi

Hacettepe Üniversitesi Endüstri Mühendisliği Bölümü
Veri Bilimi ve Yöneylem Araştırması Üzerine
Davetli Konuşmalar Serisi

EMÜ 430 ve 679 dersleri

Konuşmacı

Cem Vardar

Data Scientist & Co-Founder of Decision Science Lab & Former Data Scientist at Intel, Revionics and Carvana

Moderatör

Erdi Dasdemir

Data Scientist & Assistant Professor of Industrial Engineering

Programming Basics

- We learn R because it greatly facilitates data analysis and implementation of analytical approaches.

- However, R is not just a data analysis environment, but a programming language.

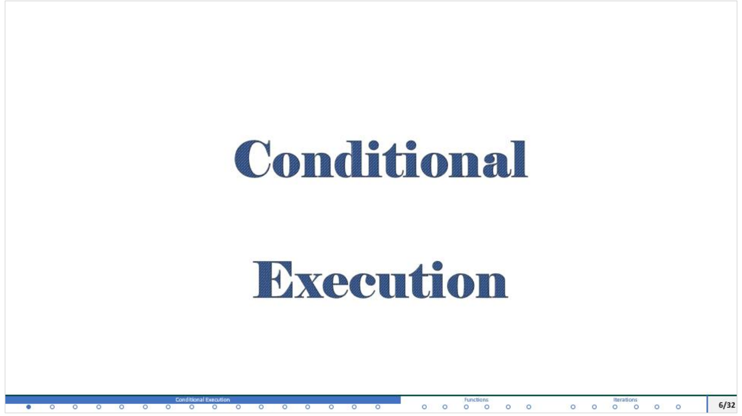
- Advanced R programmers can develop user applications and perform other complex programming tasks.

Programming Basics

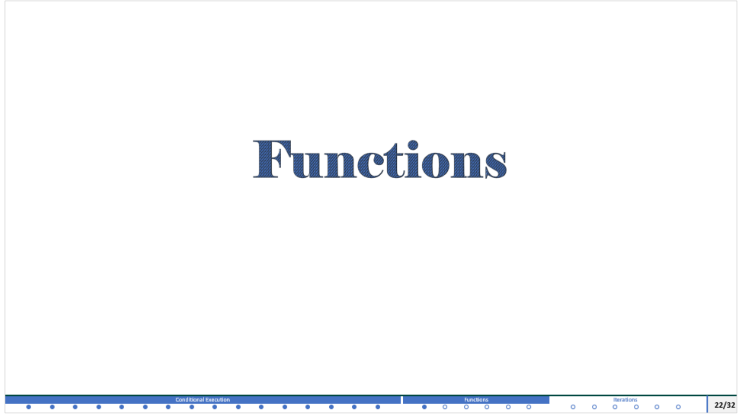
- Three fundamental programming concepts:
  - conditional execution,
  - iteration,
  - and creating functions.

- These are not only foundational elements of computer programming but are frequently useful in the context of data analysis.

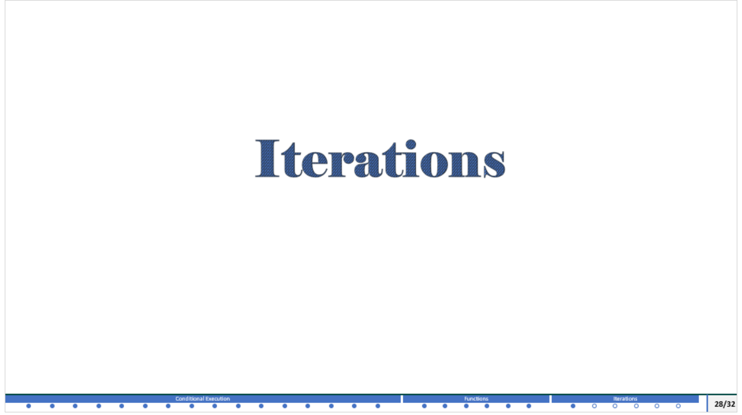
Outline



Conditional Execution



Functions



Iterations

Conditional Execution

Logical (boolean) expressions

- A logical (boolean) expression is an expression that is either **true** or **false**.

```
430 == 430
[1] TRUE


430 == 679
[1] FALSE
```

Comparison Operators

```
x == y # x is equal to y

x != y # x is not equal to y

x > y # x is greater than y

x < y # x is less than y

x >= y # x is greater than or equal to y

x <= y # x is less than or equal to y
```

Logical Operators

& (and), |(or), !(not)

```
X <- 500
x > 430 & x < 679
[1] TRUE

x <- 300
x > 430 & x < 679
[1] FALSE
```

General form:

```
if (boolean condition) {
  expressions
} else {
  alternative expressions
}
```

Conditional statements

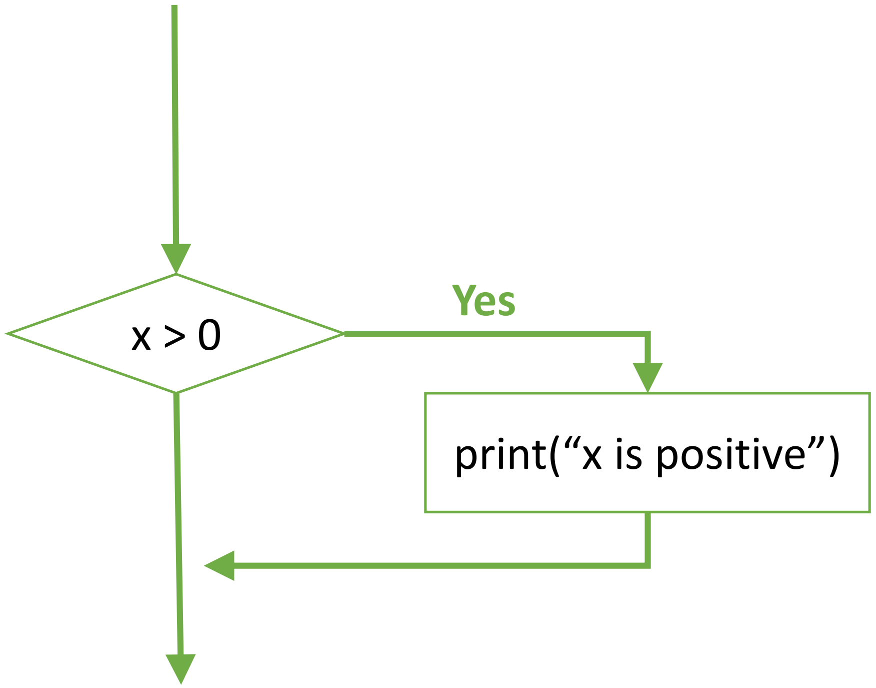
```
x <- 430

if (x > 0) {
  print("x is positive")
}
```

my recommended format

```
if (x > 0) {print("x is positive")}
```

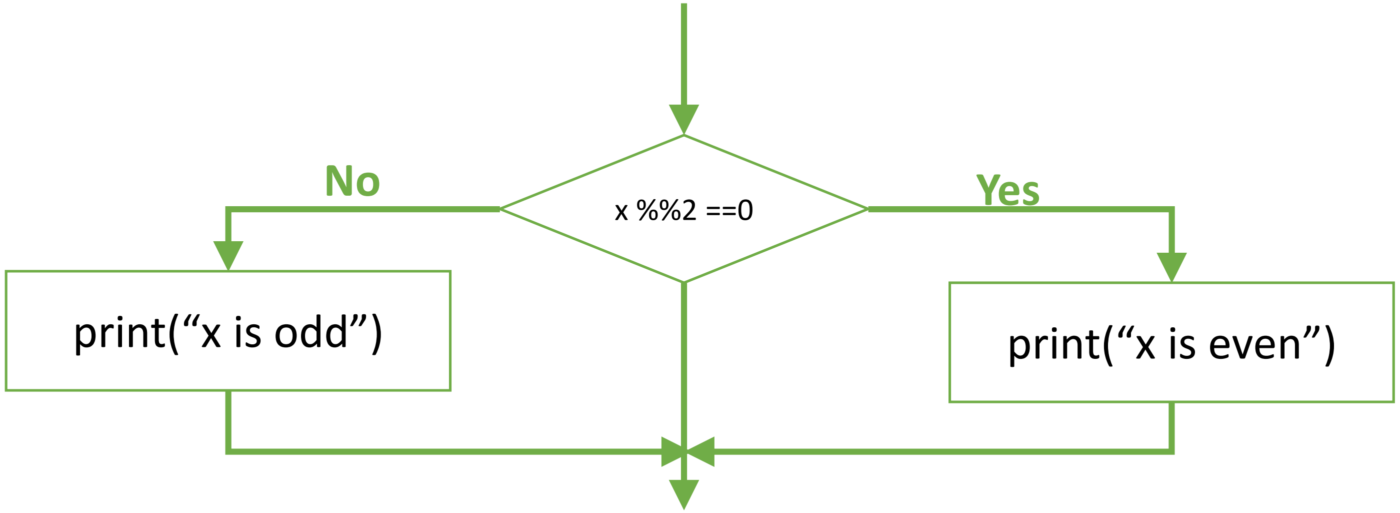
```
if (x > 0)
  print("x is positive")
```

```
if (x > 0)
print("x is positive")
```



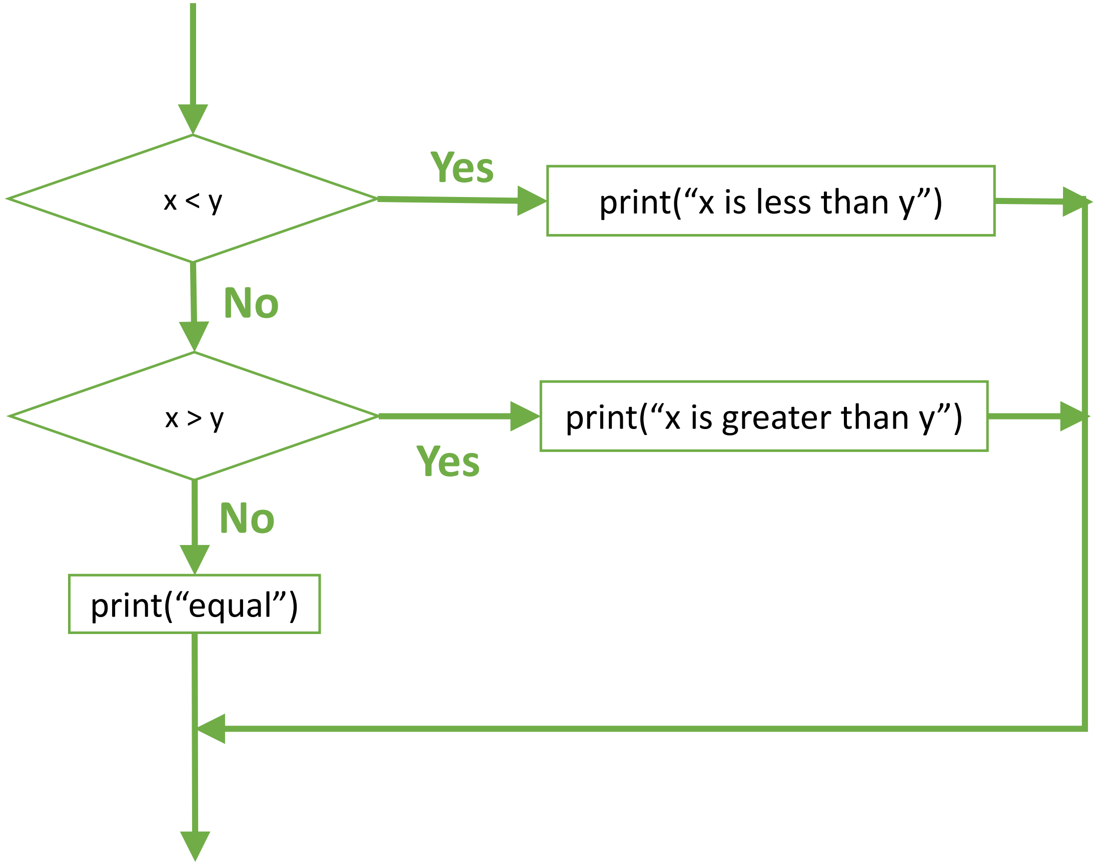
Alternative Execution

```
x <- 430

if (x %% 2 == 0) {
  print("x is even")
} else {
  print("x is odd")
}
```



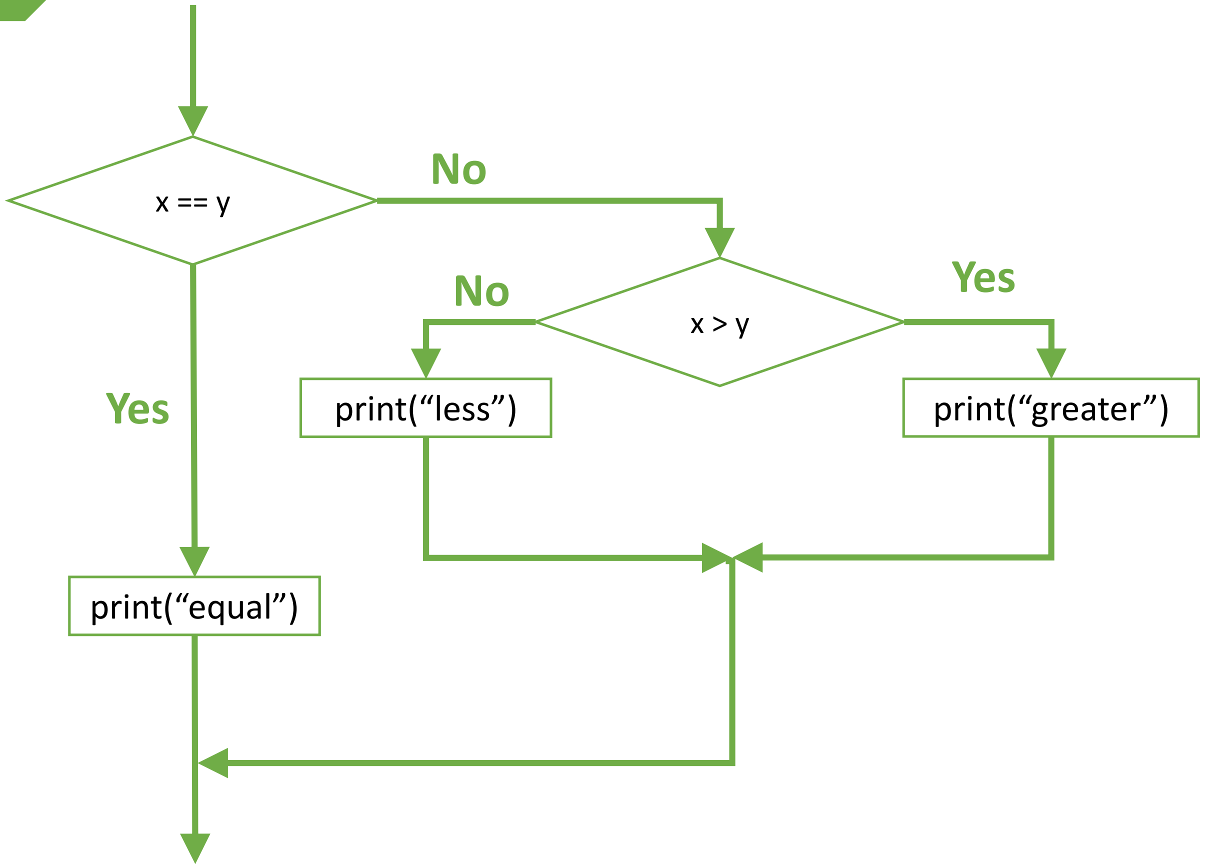
Chained Conditionals

```
if (x < y) {
  print("x is less than y")
} else if (x > y) {
  print("x is greater than y")
} else {
  print("x and y are equal")
}
```



Nested Conditionals

```
if (x == y) {
  print("x and y are equal")
} else {
  if (x < y) {
    print("x is less than y")
  } else {
    print("x is greater than y")
  }
}
```



Nested Conditionals

```
if (0 < x){
  if (x < 10) {
    print("x is a positive single-digit number")
  }
}
```

```
if (0 < x & x < 10) {
  print("x is a positive single-digit number")
}
```

Nested Conditionals

```
if (0 < x){
  if (x < 10) {
    print("x is a positive single-digit number")
  }
}
```

```
if (0 < x & x < 10) {
  print("x is a positive single-digit number")
}
```

Murders Data Set

```
library(dslabs)
data(murders)
murder_rate <- (murders$total/murders$population)*100000
```

If the murder rate of the state with the lowest murder rate is lower than 0.5, print the name of that state.

BONUS

20 sec

```
ind <- which.min(murder_rate)
if (murder_rate[ind] < 0.5) {
  print (murders$state[ind])
} else {
  print("No state has murder rate that low")
}
Output: [1] Vermont
```

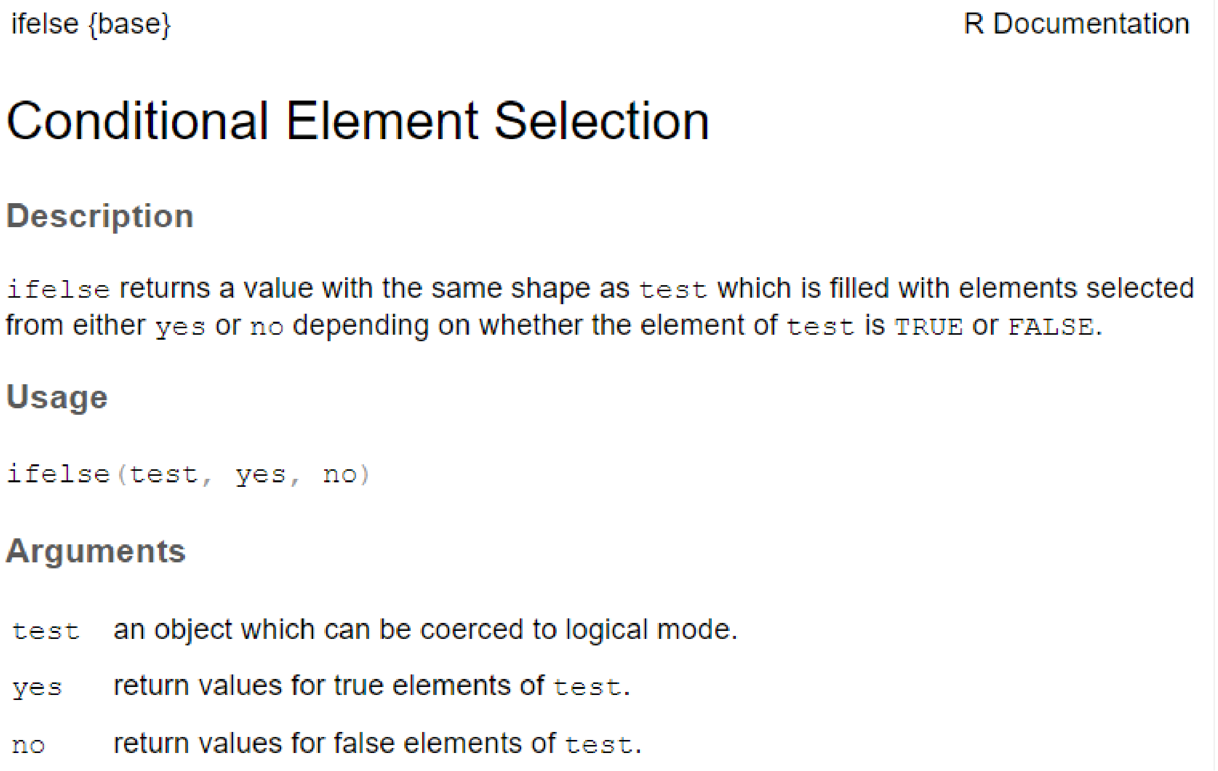
Murders Data Set

If we change the threshold level to 0.25

```
ind <- which.min(murder_rate)
if (murder_rate[ind] < 0.25) {
  print (murders$state[ind])
} else {
  print("No state has murder rate that low")
}
Output: [1] “No state has murder rate that low”
```

ifelse statement

```
a <- 0
ifelse(a > 0, 1/a, NA)
Output: [1] NA

a <- 5
ifelse(a > 0, 1/a, NA)
Output: [1] 0.2
```



ifelse {base} R Documentation

Conditional Element Selection

Description

`ifelse` returns a value with the same shape as `test` which is filled with elements selected from either `yes` or `no` depending on whether the element of `test` is `TRUE` or `FALSE`.

Usage

```
ifelse(test, yes, no)
```

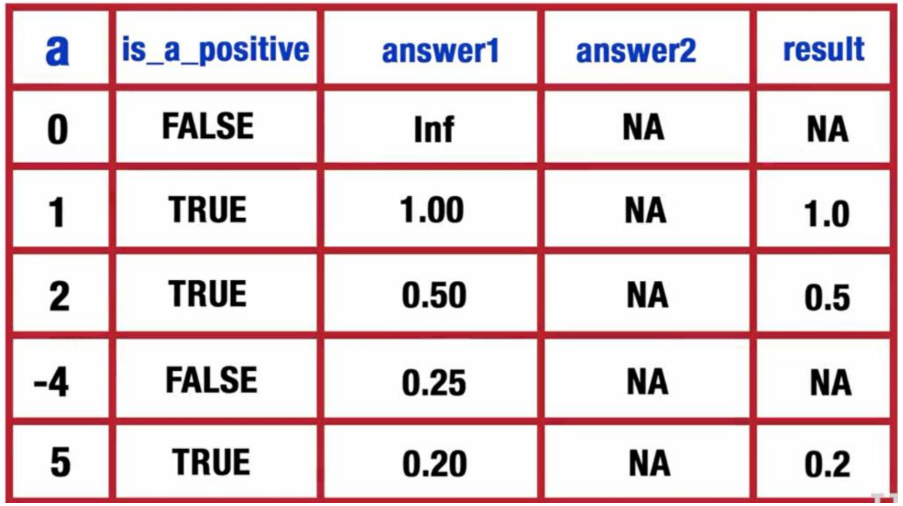
Arguments

`test`	an object which can be coerced to logical mode.
`yes`	return values for true elements of `test`.
`no`	return values for false elements of `test`.

ifelse is particularly useful because it works with vectors.

ifelse statement with vectors

```
a <- c(0, 1, 2, -4, 5)
ifelse(a > 0, 1/a, NA)
Output: [1] NA 1.0 0.5 NA 0.2
```



a	is_a_positive	answer1	answer2	result
0	FALSE	Inf	NA	NA
1	TRUE	1.00	NA	1.0
2	TRUE	0.50	NA	0.5
-4	FALSE	0.25	NA	NA
5	TRUE	0.20	NA	0.2

A common usage of ifelse is replacing NAs with some other value.

```
data(na_example)
sum(is.na(na_example)) # there are 145 Nas
Output: [1] 145
# convert na_example to a vector that does not have any Nas
no_nas <- ifelse(is.na(na_example), 0, na_example) # note that the last argument is also a vector
sum(is.na(no_nas))
Output: [1] 0
```

any and *all* functions

```
# any function takes a vector of logical and returns if any element is true

z <- c(TRUE, TRUE, TRUE)
any(z) → Output: [1] TRUE
all(z) → Output: [1] TRUE

z <- c(TRUE, TRUE, FALSE)
any(z) → Output: [1] TRUE
all(z) → Output: [1] FALSE
```

Functions

Programming Basics Functions

- Perform the same operations over and over.

- Example:

  You compute the average all the time when you are doing data science

  ```
  sum(x) / length(x)
  ```

  Write a function that does this calculation:

  `mean()` already exists.

- In many situations, the function that you need is not defined. → You have to write your own.

```
avg <- function(x){
  s <- sum(x)
  n <- length(x)
  s/n
}
```

```
x <- 1:100
avg(x)
[1] 50.5

identical(mean(x), avg(x))
[1] TRUE
```

○ General form:

```
my_function <- function(x, y, z) {
    operations that operate on x, y, z, which are defined by the user
    when they call this function.
}
```

- ➢ Functions are objects, so we assign them to variable names.

- ➢ Define a function that does arithmetic or geometric mean calculation

```
avg <- function(x, arithmetic = TRUE){
  n <- length(x)
  ifelse(arithmetic, sum(x)/n, prod(x)^(1/n))
}
```

- **Lexical Scope:**
- Variables defined inside a function are not saved in the workspace.

```
avg <- function(x){
  s <- sum(x)
  n <- length(x)
  s/n
}

s <- 3
avg(1:10)
s
Output:[1] 3
```

Inside the function, an s is created that is not 3, it is something else.
But that only happens inside the function.

- **Return Value**

```
# Version 1: A function to calculate the area of a rectangle
calculate_rect_area <- function(width, height){
    return(width * height) # return a specific result
}

# Version 2: A function to calculate the area of a rectangle
calculate_rect_area <- function(width, height){
    width * height # return a specific result
}

# Version 3: A function to calculate the area of a rectangle
calculate_rect_area <- function(width, height){
    area <- width * height # calculate area
     area
 }
```

- **Debugging Functions**

```
# Version 3: A function to calculate the area of a rectangle
calculate_rect_area <- function(width, height){
    area <- width * height # calculate area
     area
 }
```

assign sample values to your arguments, and then run through the function line by line.

Iterations

- **for-loops**

```
for (i in range of values) {
  operations that use i, which is changing across the
range of values
}
```

- **Example**

$$1 + 2 + \ ... + n = \frac{n(n+1)}{2}$$

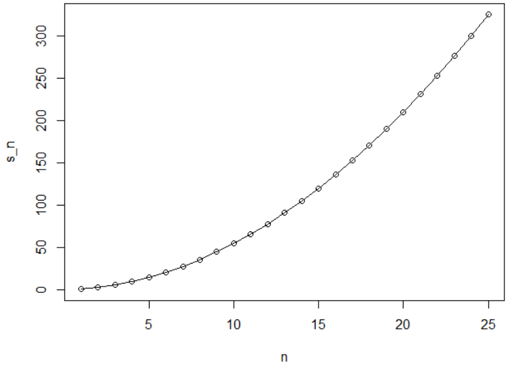
We want to do this calculation for $n = 1, 2, \dots, 25$

1. Define a function:

```
compute_s_n <- function(n){
  x <- 1:n
  sum(x)
}
compute_s_n(3)
[1] 6

compute_s_n(100)
[1] 5050
```

2. calculation for $\boldsymbol{n = 1, 2, \dots, 25}$

```
m <- 25
# create an empty vector
s_n <- vector(length = m)
for (i in 1:m){
  s_n[i] <- compute_s_n(i)
}
n <- 1:m
plot(n, s_n)
lines(n, n*(n+1)/2)
```



We normally rarely use for loops in R

R functions:

- apply
- sapply
- lapply
- tapply
- mapply

```
m <- 25
# create an empty vector
s_n <- vector(length = m)
for (i in 1:m){
  s_n[i] <- compute_s_n(i)
}
```

```
sapply(1:m, compute_s_n)
```

while

```
n <- 430
while (n > 0){
  print(n)
  n <- n - 50
}
```

```
[1] 430
[1] 380
[1] 330
[1] 280
[1] 230
[1] 180
[1] 130
[1] 80
[1] 30
```

Any
Questions