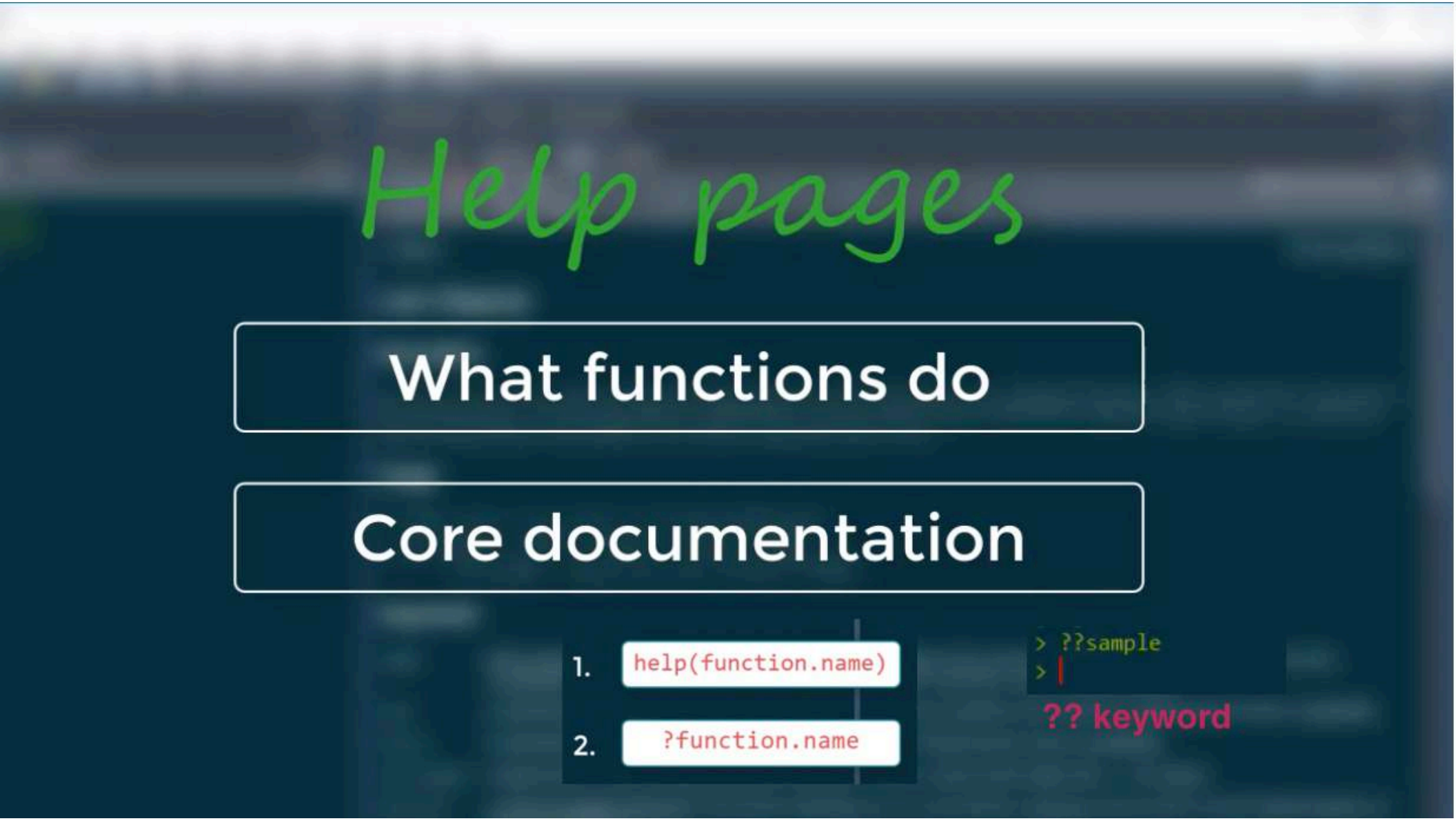
1.30

What does the pirate say?

RRRR


Help pages
What functions do
Core documentation
1. help(function.name)
2. ?function.name
> ??sample
>
?? keyword

Random Samples and Permutations

Description

`sample` takes a sample of the specified size from the elements of `x` using either with or without replacement.

a brief overview of what a function does

Usage

```
sample(x, size, replace = FALSE, prob = NULL)

sample.int(n, size = n, replace = FALSE, prob = NULL,
           useHash = (!replace && is.null(prob) && size <= n/2 && n > 1e7))
```

an example of how the function can be written

Arguments

`x`	either a vector of one or more elements from which to choose, or a positive integer. See 'Details.'
`n`	a positive number, the number of items to choose from. See 'Details.'
`size`	a non-negative integer giving the number of items to choose.
`replace`	should sampling be with replacement?
`prob`	a vector of probability weights for obtaining the elements of the vector being sampled.
`useHash`	`logical` indicating if the hash-version of the algorithm should be used. Can only be used for `replace = FALSE`, `prob = NULL`, and `size <= n/2`, and really should be used for large `n`, as `useHash=FALSE` will use memory proportional to `n`.

365 DataScience

Value

For `sample` a vector of length `size` with elements drawn from either `x` or from the integers `1:x`.

For `sample.int`, an integer vector of length `size` with elements from `1:n`, or a double vector if `n >= 2^31`.

what the function will return once the code is executed

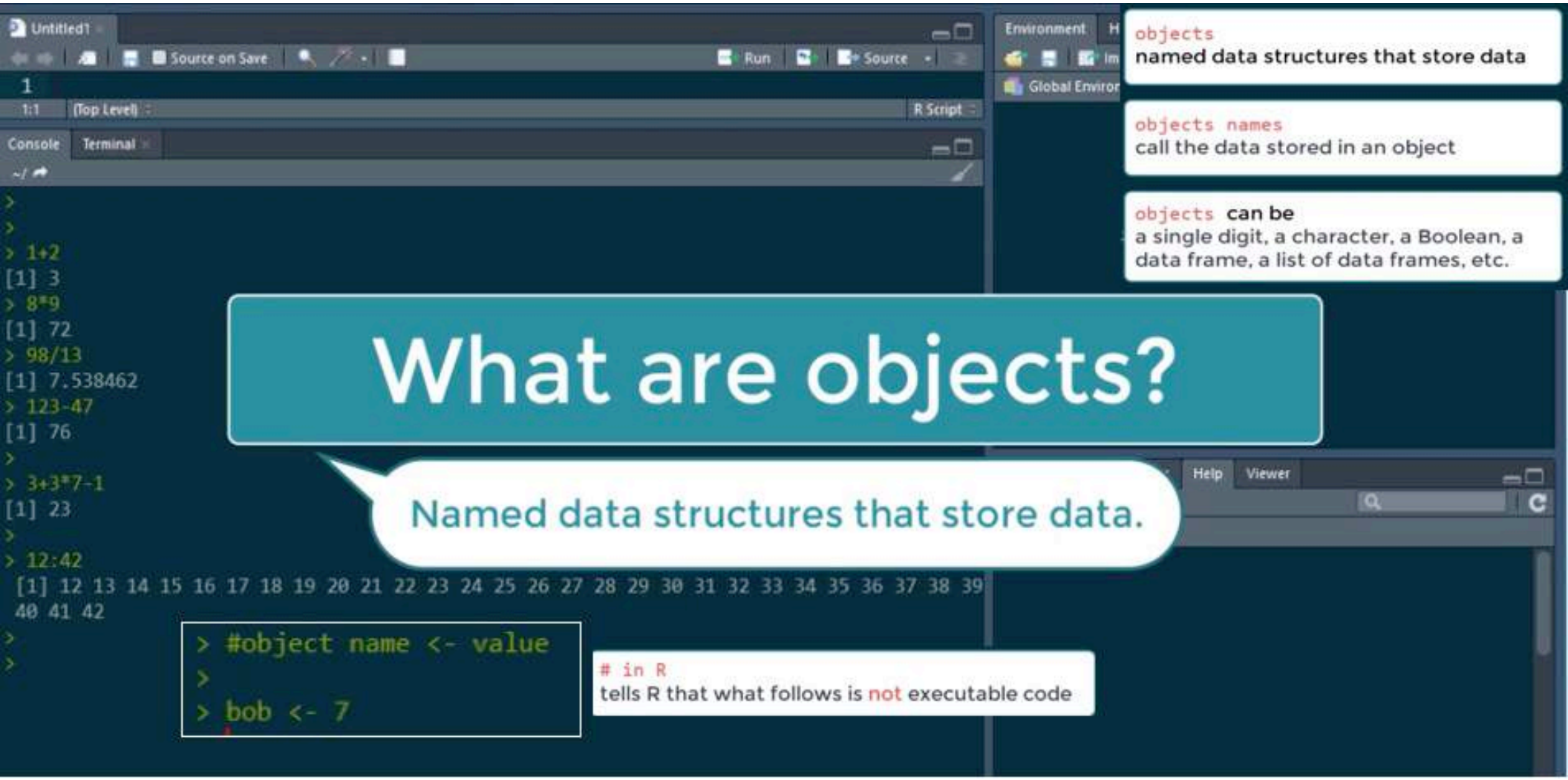
Examples

```
x <- 1:12
# a random permutation
sample(x)
# bootstrap resampling -- only if length(x) > 1 !
sample(x, replace = TRUE)

# 100 Bernoulli trials
sample(c(0,1), 100, replace = TRUE)

## More careful bootstrapping --  Consider this when using sample()
## programmatically (i.e., in your function or simulation)!
```

Guaranteed to work!


Untitled1
Source on Save
Run
Source
1
1:1
(Top Level)
R Script
Console
Terminal
~/
> 1+2
[1] 3
> 8*9
[1] 72
> 98/13
[1] 7.538462
> 123-47
[1] 76
> 3+3*7-1
[1] 23
> 12:42
[1] 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42
> #object name <- value
>
> bob <- 7
Environment
Global Enviror
Help
Viewer
objects
named data structures that store data
objects names
call the data stored in an object
objects can be
a single digit, a character, a Boolean, a data frame, a list of data frames, etc.
What are objects?
Named data structures that store data.
in R
tells R that what follows is not executable code

Naming objects

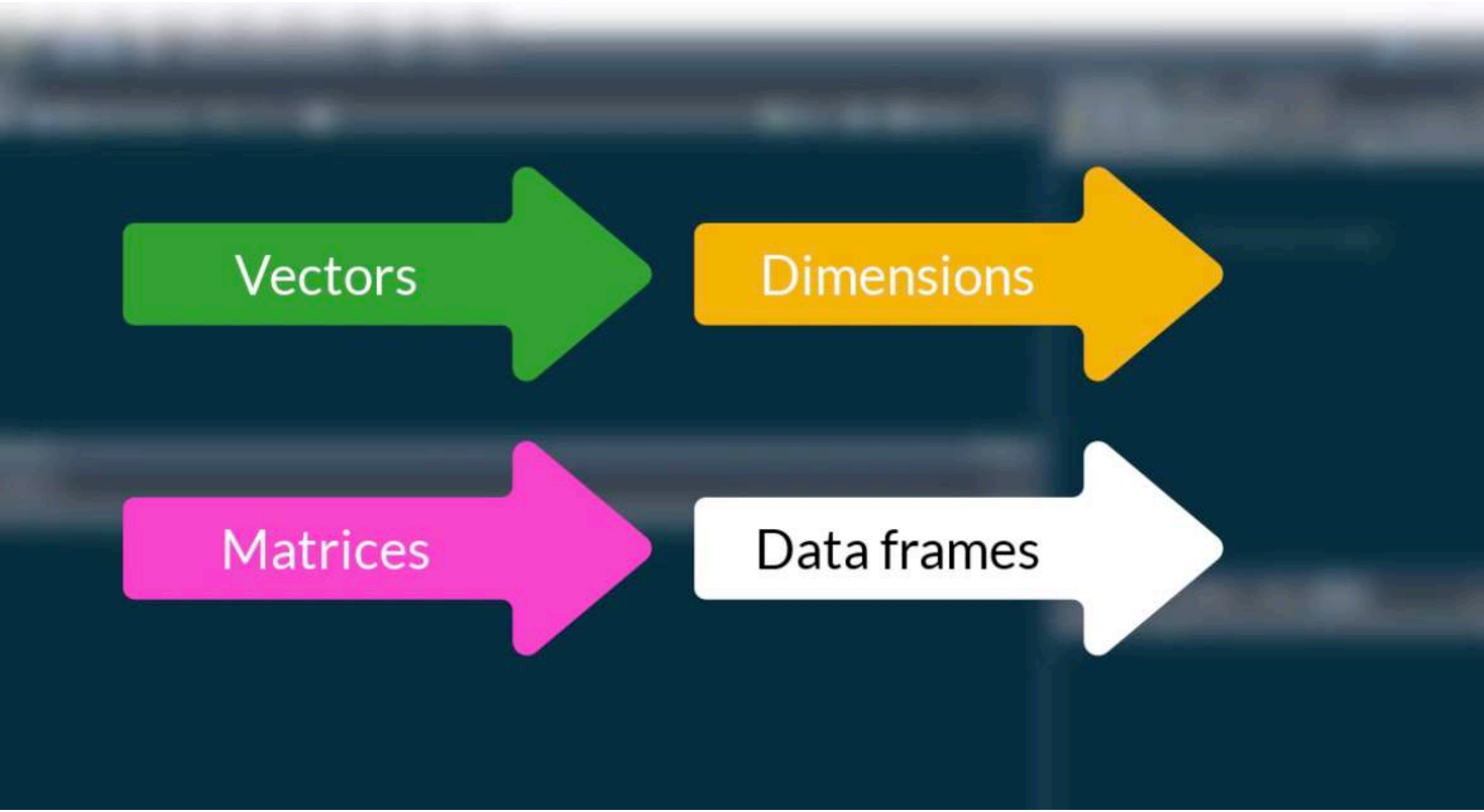
small letters

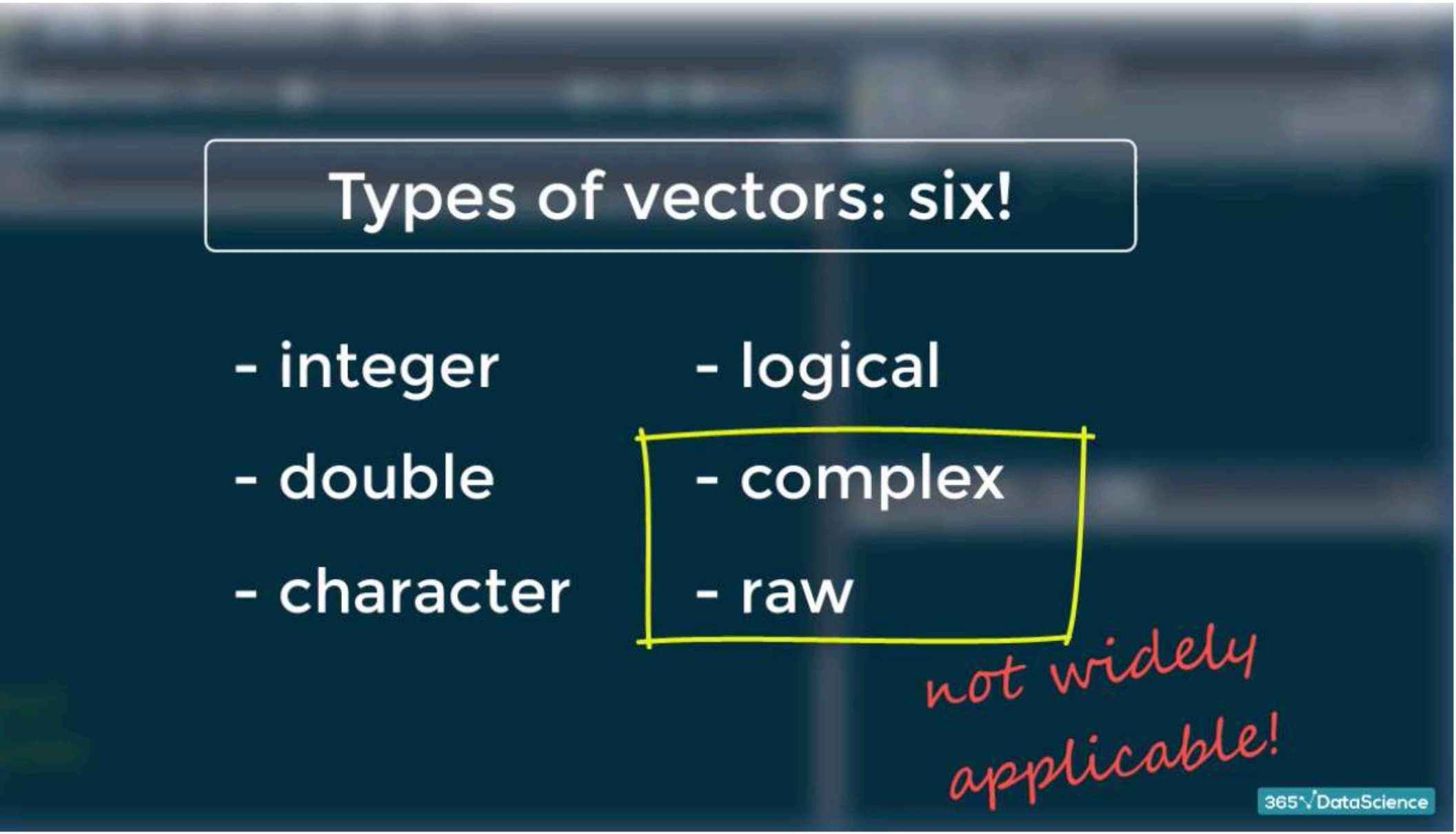
abcdefghijklmnopqrstuvwxyz

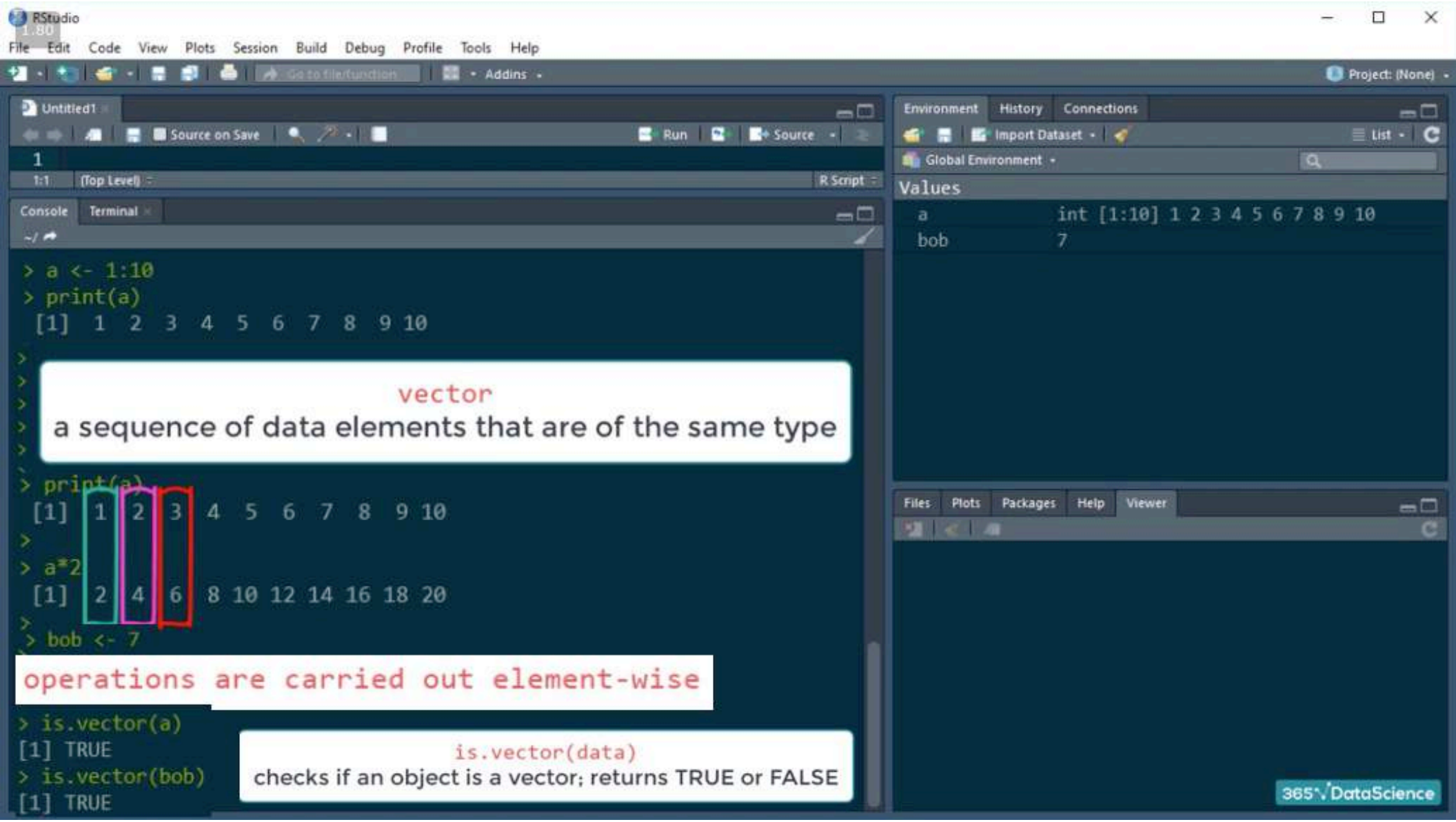
0123456789

.

_


Vectors
Dimensions
Matrices
Data frames


Types of vectors: six!
- integer
- double
- character
- logical
- complex
- raw
not widely applicable!
365 DataScience


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Addins
Project: (None)
Untitled1
Source on Save
Run
Source
1
1:1
(Top Level)
R Script
Console
Terminal
~/
> a <- 1:10
> print(a)
[1] 1 2 3 4 5 6 7 8 9 10
vector
a sequence of data elements that are of the same type
> print(a)
[1] 1 2 3 4 5 6 7 8 9 10
> a*2
[1] 2 4 6 8 10 12 14 16 18 20
> bob <- 7
operations are carried out element-wise
> is.vector(a)
[1] TRUE
> is.vector(bob)
[1] TRUE
is.vector(data)
checks if an object is a vector; returns TRUE or FALSE
Environment
History
Connections
Import Dataset
List
Global Environment
Values
a int [1:10] 1 2 3 4 5 6 7 8 9 10
bob 7
Files
Plots
Packages
Help
Viewer
365 DataScience

`integers`
whole numbers; with nothing after the decimal

`double`
doubles store regular numbers
(large, small, positive, negative, with digits after the decimal, or without)

```
a <- 1:10
```

`c(...)`
combines the passed data to form a vector

```
a <- c(1, 2, 3, 4, ..., 10)
```

`typeof(...)`
returns the basic type of an object

```
> typeof(int)
[1] "integer"
```

Why do we need the "L"?

Because R saves numeric variables into doubles by default.

`ls()`
lists all visible objects in your environment

```
int <- c(5, 6, 7, 8)
int <- c(5L, 6L, 7L, 8L)
```

```
int          num [1:4] 5 6 7 8
int          int [1:4] 5 6 7 8
```

R rewrites objects

without asking for permission or issuing a warning

`creating an integer with "L"`
ensures the object will store integers instead of the default double

`character vectors`
store text data; can be a single character or a longer string

`object.name <- c("text", "text", ...)`
text values are denoted by quotation marks

`strings`
the elements of character vectors; can be letters, numbers, or symbols

```
> char <- c("R", "for", "life")
> typeof(char)
[1] "character"
>
> char2 <- c("The answer to life", "the universe", "and everything", "is", "42")
> typeof(char2)
[1] "character"
```

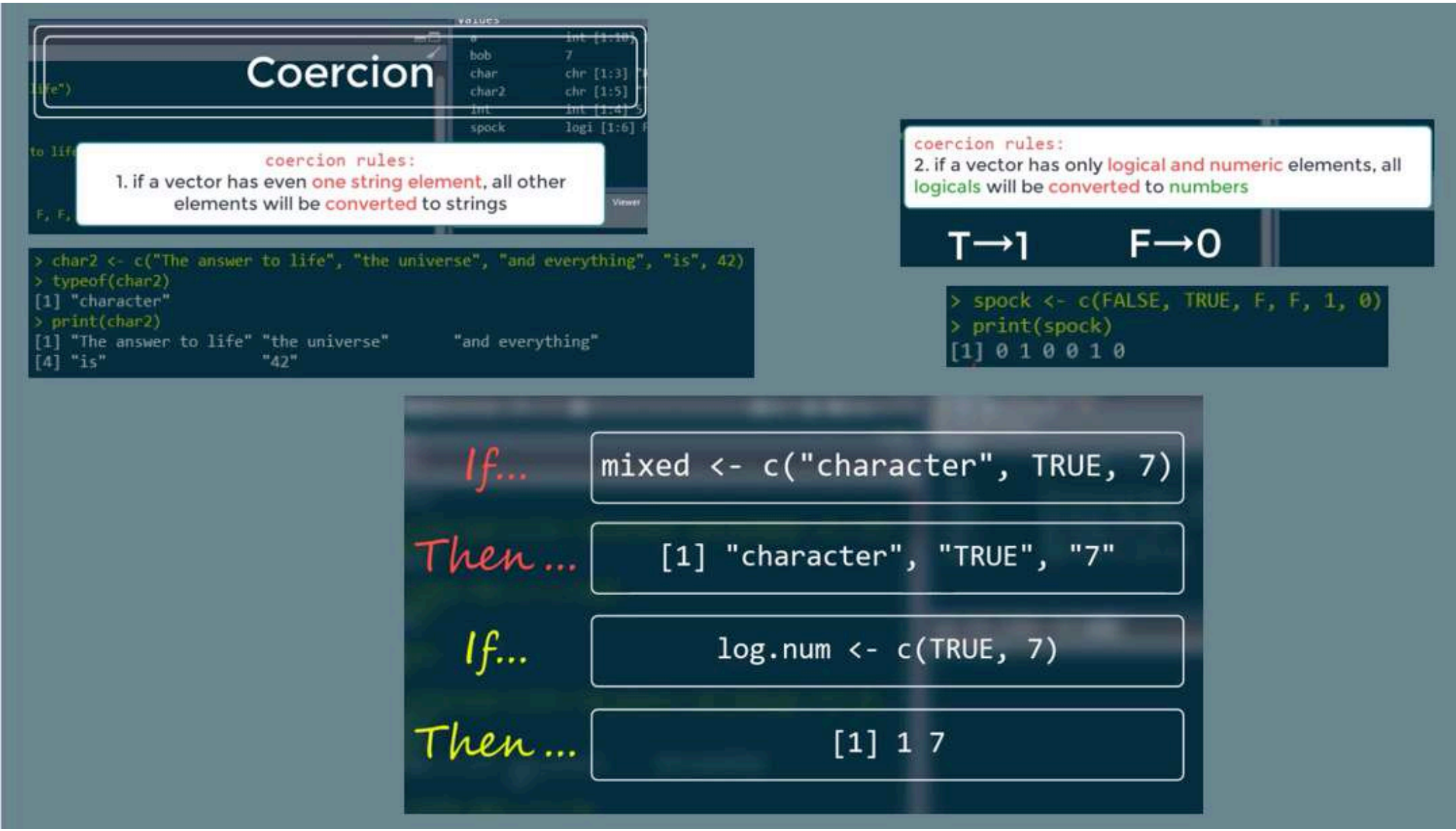
`vectors (recap)`
a sequence of data elements that are of the same type

`logical vectors`
store Boolean data (TRUE & FALSE values)
T & F = TRUE & FALSE

```
>
> spock <- c(FALSE, TRUE, F, F, T, FALSE)
>
```

no quotation marks
ALL CAPS

```
?is.complex()
=
help(is.complex)
```


Coercion
coercion rules:
1. if a vector has even one string element, all other elements will be converted to strings
> char2 <- c("The answer to life", "the universe", "and everything", "is", 42)
> typeof(char2)
[1] "character"
> print(char2)
[1] "The answer to life" "the universe" "and everything"
[4] "is" "42"
coercion rules:
2. if a vector has only logical and numeric elements, all logicals will be converted to numbers
T→1 F→0
> spock <- c(FALSE, TRUE, F, F, 1, 0)
> print(spock)
[1] 0 1 0 0 1 0
If...
mixed <- c("character", TRUE, 7)
Then...
[1] "character", "TRUE", "7"
If...
log.num <- c(TRUE, 7)
Then...
[1] 1 7

VECTOR RECYCLING

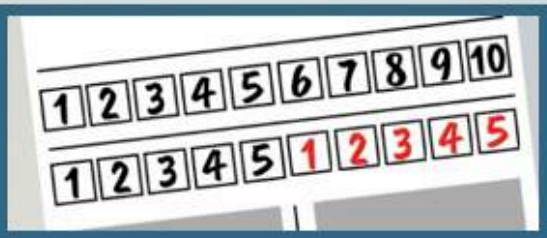
`element-wise execution`
ensures variables (or groups of values) are manipulated in an orderly fashion

```
vec <- c(1,2,3)
vic <- c(11,12,13)

vec + vic
vec - vic

vec*vic
vec/vic
```

```
> vec + vic
[1] 12 14 16
> vec - vic
[1] -10 -10 -10
> vec*vic
[1] 11 24 39
> vec/vic
[1] 0.09090909 0.16666667 0.23076923
```



New script
Ctrl + Shift + N

`mean(x, ...)`
returns the arithmetic mean

`median(x, ...)`
computes the middle value

`sd(x, ...)`
finds the standard deviation

`sum(x, ...)`
adds up all values in an object

`prod(x, ...)`
multiplies all values in an object

Vector attributes

- Names
- Dimensions
- Classes

```
> age <- c(23, 26, 24, 26)
> attributes(age)
NULL
> names(age)
NULL
```

NULL
signifies that the object is empty
(in this case, has no attributes)

`names(x) <- c("name1", "name2", ...)`
assigning names to a vector

`print(x)` or `x`
you can print an object by typing its name instead of using the `print()` function

```
> names(age) <- c("George", "John", "Paul", "Ringo")
> age
George   John   Paul  Ringo
    23     26     24     26
```

```
> attributes(age)
$names
[1] "George" "John"   "Paul"   "Ringo"
```

```
> names(age)
[1] "George" "John"   "Paul"   "Ringo"
```

```
names(age) <- c("George Harrison", "John Lennon", "Paul McCartney", "Ringo Starr")
age

names(age) <- NULL
```

```
> names(age) <- NULL
> age
[1] 23 26 24 26
> attributes(age)
NULL
```

Indexing & slicing

For selecting values from a vector

- New variable
- Editing variables
- Matrices & data frames

How do you select a value from an object?

Indexing

```
vector.name[n]
```

```
vector.name[-n]
```

selects all values but the one indexed

```
> deck <- c("Duke", "Assassin", "Captain", "Ambassador", "Contessa")
> deck[4]
[1] "Ambassador"
> deck[-4]
[1] "Duke"      "Assassin" "Captain"  "Contessa"
```

```
vector.name[c(n1,n2,n3)]
```

```
> deck[c(1,3,5)]
[1] "Duke"      "Captain"  "Contessa"
```

```
names(x) <- c("name1", "name2", ...)
```

assigning names to a vector

Indexing by name

```
named.vector["value name"]
```

```
n.deck<-c(6,7,8,9,10)
```

```
> names(n.deck) <- deck
> n.deck
      Duke  Assassin   Captain Ambassador   Contessa
         6         7         8         9        10
>
```

```
> n.deck["Contessa"]
Contessa
      10
```

```
> n.deck[c("Contessa", "Duke", "Ambassador")]
  Contessa       Duke Ambassador
        10          6          9
```

Slicing

Selecting several consecutive values at once

```
vector.name[n1:nk]
```

```
> n.deck[3:5]
   Captain Ambassador   Contessa
         8          9         10
```

How do you select all other elements but a slice?

By using the "-" sign!

```
vector.name[-(n1:nk)]
```

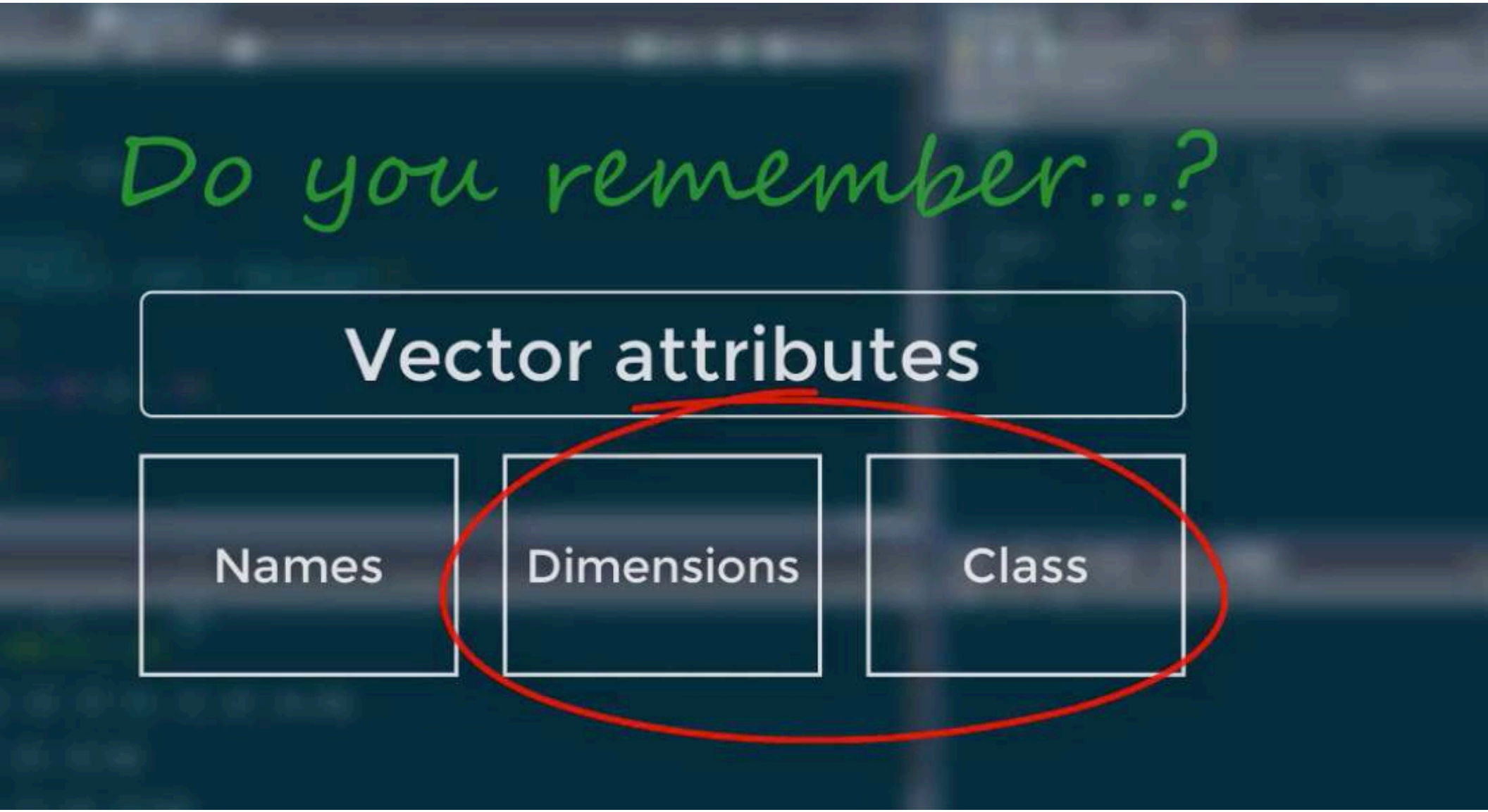
```
> lv <- seq(10, 100, by = 10)
> lv
 [1]  10  20  30  40  50  60  70  80  90 100
```

```
> lv[-(4:7)]
[1]  10  20  30  80  90 100
```

Slicing with a comprison operator

< >

```
> lv[lv > 30]
[1]  40  50  60  70  80  90 100
```


Do you remember...?
Vector attributes
Names
Dimensions
Class

`sample(from =, to =, by =, ...)`
generates regular sequences of numbers

DIMENSIONS

```
a <- seq(10, 120, by = 10)
a
[1]  10  20  30  40  50  60  70  80  90 100 110 120
```

We can tranform this... easily.

Rows come first!

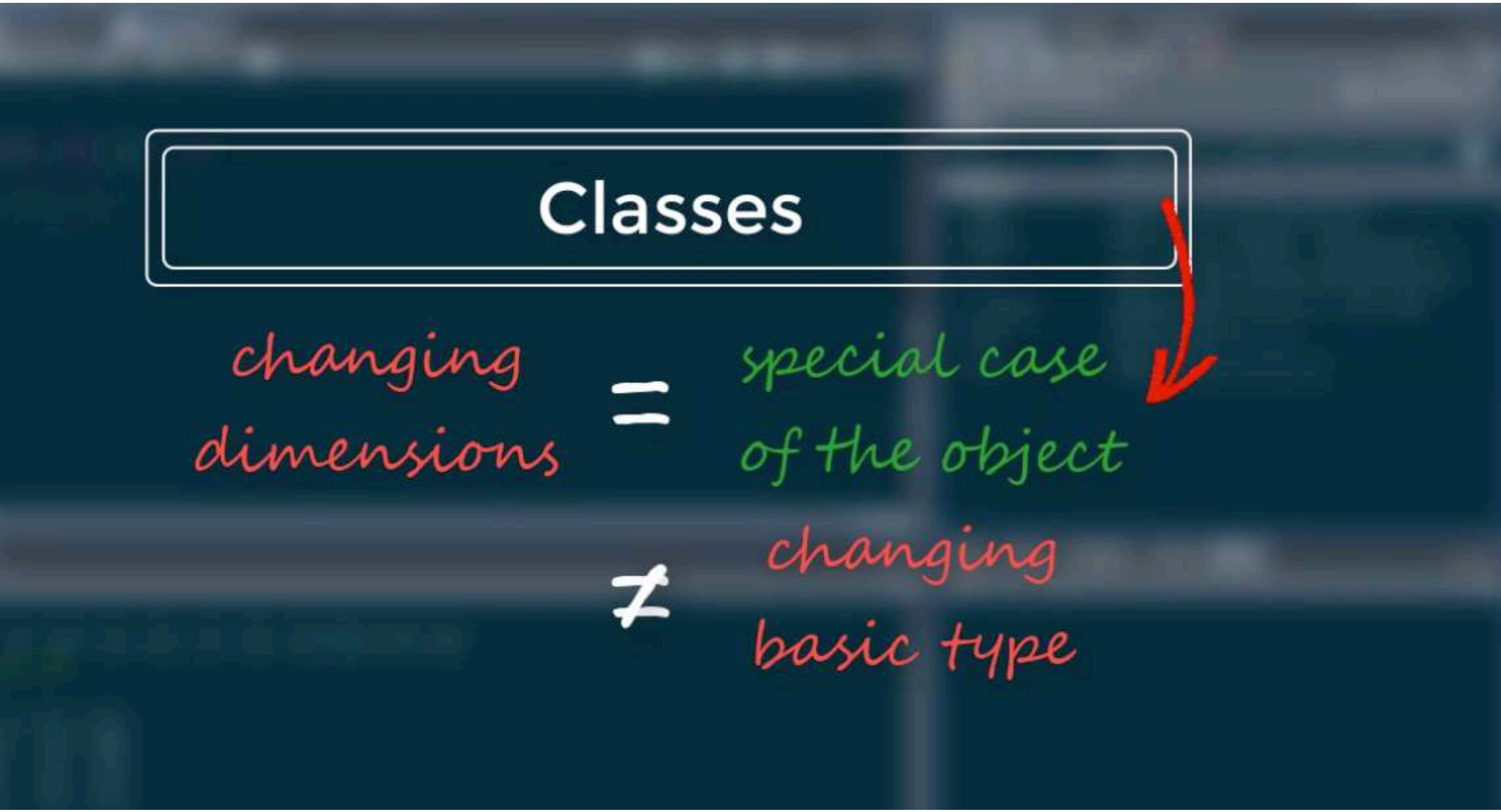
`dim(object.to.transform) <- c(n, m)`
bends the `object.to.transform` into a two-dimensional object with `n` rows and `m` columns

Vector
into array

```
a <- seq(10, 120, by = 10)
a
dim(a) <- c(3, 4)
```

"You need this 1D vector to bend into a 2D object that has 3 rows and 4 columns?"

```
> a
 [1]  10  20  30  40  50  60  70  80  90 100 110 120
> dim(a) <- c(3, 4)
> a
     [,1] [,2] [,3] [,4]
[1,]   10   40   70  100
[2,]   20   50   80  110
[3,]   30   60   90  120
>
```


Classes
changing dimensions
=
special case of the object
≠
changing basic type

Why are classes important?

concept art!

gives R the base information
needed to create an object

all objects of a class
share features

assigning and re-assigning
classes is *not advised*!

CLASSES

Re-run this to recreate the vector a

Re-run this to give a dimensions

Run this to get the basic type of vector a

Re-run this to get the basic type of the matrix a

```
> a <- seq(10, 120, by = 10)
> typeof(a)
[1] "double"
> dim(a) <- c(3, 4)
> typeof(a)
[1] "double"
```

Structure changed:

```
> a <- seq(10, 120, by = 10)
> class(a)
[1] "numeric"
> dim(a) <- c(3, 4)
> class(a)
[1] "matrix"
```

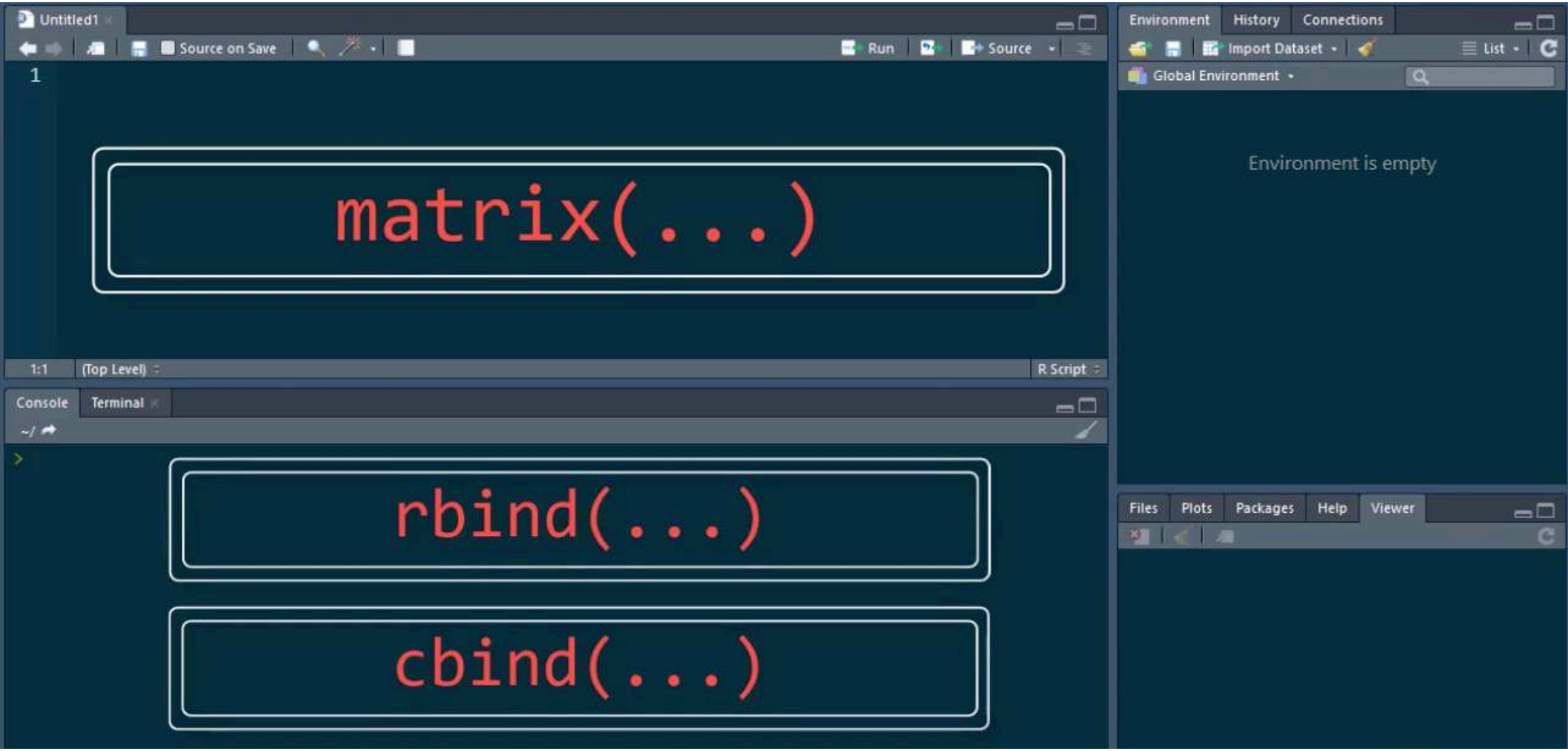
Before a is given dimensions

After a is given dimensions

Matrices

A few things!

1. Matrices are a natural extension to vectors (2D arrays)
2. They have a fixed number of rows & columns
3. Can contain only one basic data type


Untitled1
Source on Save
Run
Source
1
matrix(...)
1:1
(Top Level)
R Script
Console
Terminal
~/
rbind(...)
cbind(...)
Environment
History
Connections
Import Dataset
List
Global Environment
Environment is empty
Files
Plots
Packages
Help
Viewer

Matrices: matrix()

vector

number of rows

number of columns

```
matrix(data = , nrow = , ncol =, ...)
```

creates a matrix from a given set of values

Environment is empty

```
> mtrx <- matrix(1:12, nrow = 3)
> mtrx
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> mtrx <- matrix(1:12, ncol = 4)
> mtrx
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

Same result!

← the values are entered by columns

```
matrix(data = , nrow = , byrow = TRUE, ...)
```

organizes the values in the matrix by row

```
> mtrx <- matrix(1:12, ncol = 4, byrow = TRUE)
> mtrx
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Matrices: cbind()

```
usa <- c(1.3, 1.5, 1.2, 1.4, 1.5)
usa
de <- c(0.2, 0.4, 0.7, 0.8, 0.8)
de

ngo <- cbind(usa, de)
ngo
```

```
> ngo <- cbind(usa, de)
> ngo
     usa  de
[1,] 1.3 0.2
[2,] 1.5 0.4
[3,] 1.2 0.7
[4,] 1.4 0.8
[5,] 1.5 0.8
```


`t(x)`
transposes `x`

```
> ngo <- t(ngo)
> ngo
     2013 2014 2015 2016 2017
usa  1.3  1.5  1.2  1.4  1.5
de   0.2  0.4  0.7  0.8  0.8
```

Naming rows!

`colnames()` & `rownames()`
name the columns and the rows of a matrix

```
> rownames(ngo) <- c("2013", "2014", "2015", "2016", "2017")
> ngo
     usa  de
2013 1.3 0.2
2014 1.5 0.4
2015 1.2 0.7
2016 1.4 0.8
2017 1.5 0.8
```

`rbind(...)`
binds vectors into rows of a matrix or adds them to an existing structure

```
> ind <- c(2, 2.2, 2.4, 2.5, 2.6)
> ngo <- rbind(ngo, ind)
> ngo
     2013 2014 2015 2016 2017
usa  1.3  1.5  1.2  1.4  1.5
de   0.2  0.4  0.7  0.8  0.8
ind  2.0  2.2  2.4  2.5  2.6
```

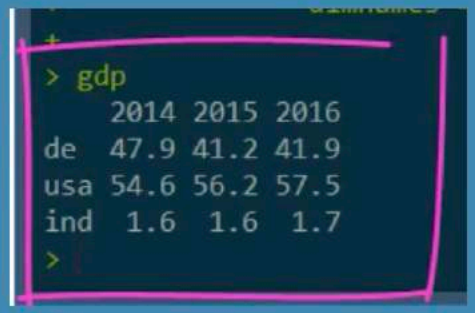
Creating a matrix: one-liners :)

`dimnames =`
an attribute argument; can contain a `list` giving the row and column names

```
gdp <- matrix(c(47.9, 41.2, 41.9, 54.6, 56.2, 57.5, 1.6, 1.6, 1.7),
              nrow = 3, byrow = TRUE,
              dimnames = list(c("de", "usa", "ind"),
                              c("2014", "2015", "2016")))
```

← row names vector

← column names vector



```
> gdp
    2014 2015 2016
de  47.9 41.2 41.9
usa 54.6 56.2 57.5
ind  1.6  1.6  1.7
>
```

Recycling

Repeating values in order to reach the necessary vector length to complete an operation

```
exmpl <- matrix(1:10, nrow = 4, ncol = 4)
exmpl
```

```
> exmpl <- matrix(1:10, nrow = 4, ncol = 4)
Warning message:
In matrix(1:10, nrow = 4, ncol = 4) :
  data length [10] is not a sub-multiple or multiple of the number of rows [4]
> exmpl
     [,1] [,2] [,3] [,4]
[1,]    1    5    9    3
[2,]    2    6   10    4
[3,]    3    7    1    5
[4,]    4    8    2    6
>
```

Building a matrix

```
matrix(data = , nrow = , ncol = , byrow = )
```

```
rbind(...)
```

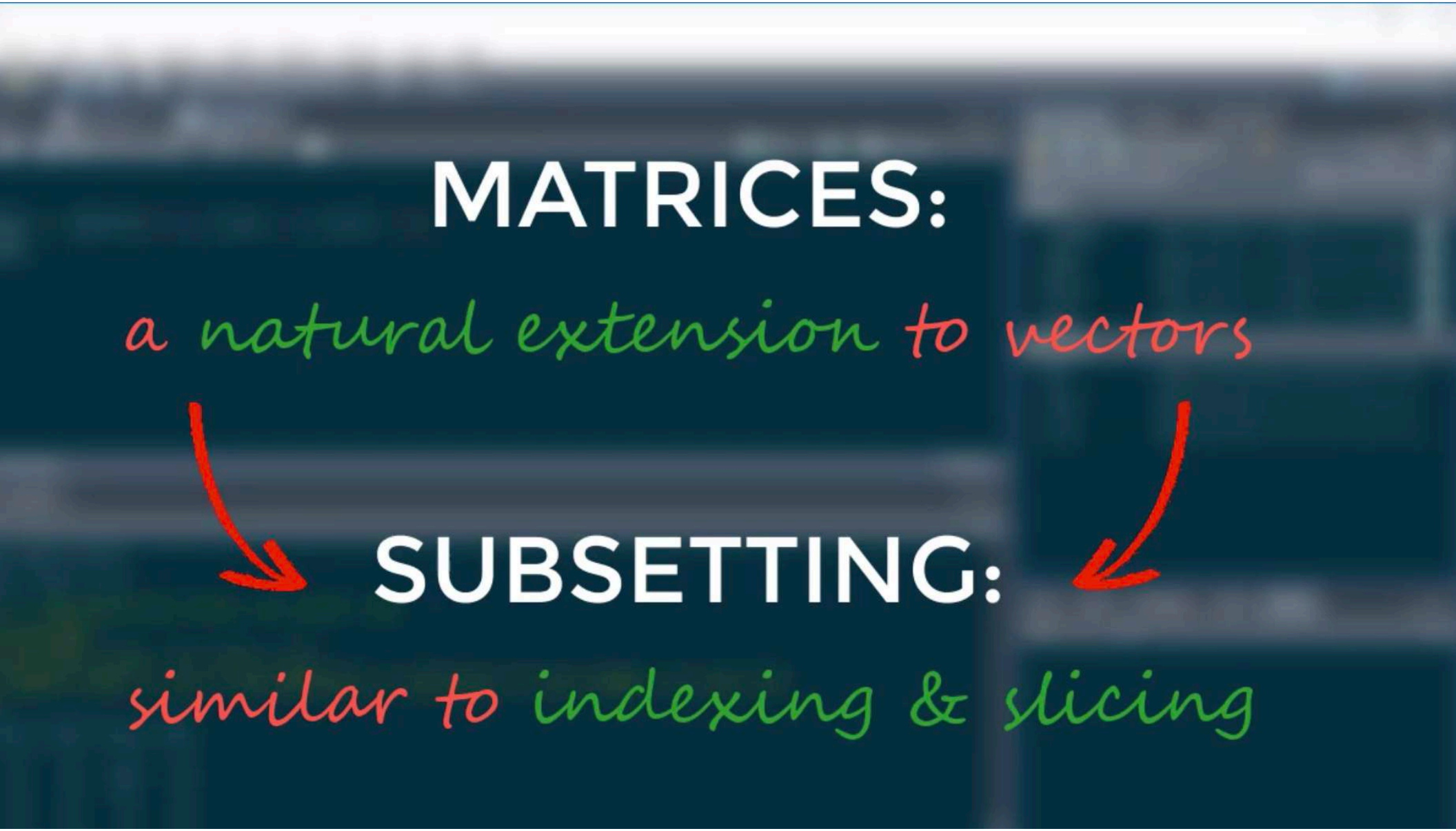
```
cbind(...)
```

Naming the matrix

```
rownames(...)
```

```
colnames(...)
```

```
dimnames =
```


MATRICES:
a natural extension to vectors
SUBSETTING:
similar to indexing & slicing

Subsetting a matrix

```
gross <- c(381, 1340, 318, 975, 396, 960, 292,
           940, 302, 934, 290, 897, 262, 879, 249, 797)
hp.mat <- matrix(gross, nrow = 8, byrow = T)
hp.mat
```

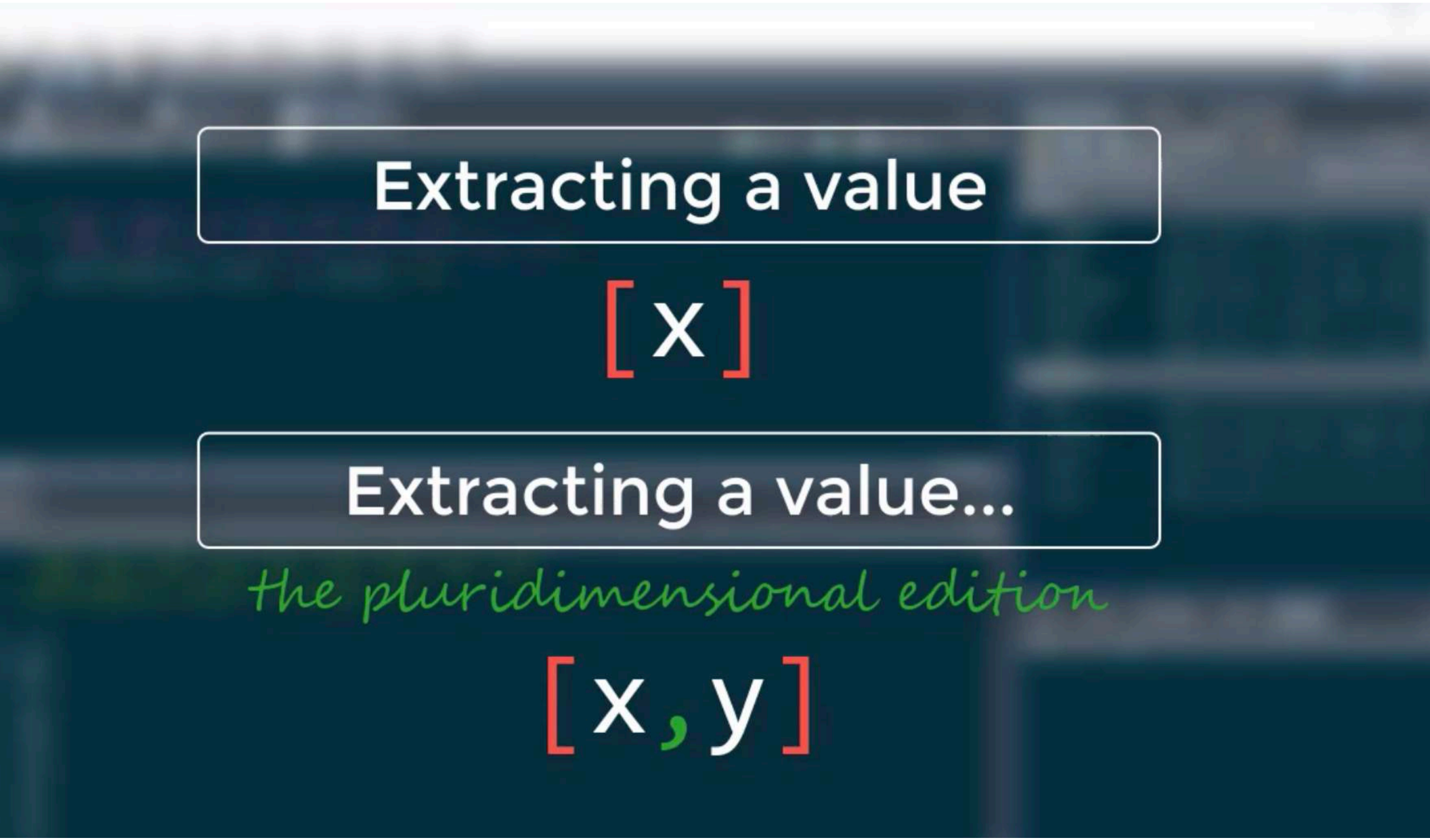
```
> hp.mat
     [,1] [,2]
[1,]  381 1340
[2,]  318  975
[3,]  396  960
[4,]  292  940
[5,]  302  934
[6,]  290  897
[7,]  262  879
[8,]  249  797
```

Box office grosses for each Harry Potter movie in the US and Worldwide

```
> hp.mat <- matrix(gross, nrow = 8, byrow = T)
> hp.mat
     [,1] [,2]
[1,]  381 1340
[2,]  318  975
[3,]  396  960
[4,]  292  940
[5,]  302  934
[6,]  290  897
[7,]  262  879
[8,]  249  797
```

[n,]
row index

[,n]
column index


Extracting a value
[x]
Extracting a value...
the pluridimensional edition
[x,y]

Subsetting a matrix

```
> hp.mat
     [,1] [,2]
[1,]  381 1340
[2,]  318  975
[3,]  396  960
[4,]  292  940
[5,]  302  934
[6,]  290  897
[7,]  262  879
[8,]  249  797
>
```

```
> hp.mat[6,2]
[1] 897
```

Calling an entire row?

```
> hp.mat
     [,1] [,2]
[1,]  381 1340
[2,]  318  975
[3,]  396  960
[4,]  292  940
[5,]  302  934
[6,]  290  897
[7,]  262  879
[8,]  249  797
```

```
> hp.mat[6]
[1] 290
```

When you tell R this...

`hp.mat[6]`

R looks for the 6-th value

`hp.mat[6, ]`

calling the entire 6th row

```
> hp.mat[6, ]
[1] 290 897
```

```
> hp.mat
     [,1] [,2]
[1,]  381 1340
[2,]  318  975
[3,]  396  960
[4,]  292  940
[5,]  302  934
[6,]  290  897
[7,]  262  879
[8,]  249  797
```

```
[1] 290 897
> hp.mat[ ,2]
[1] 1340  975  960  940  934  897  879  797
>
```

vector! because our data is one-dimentional

Subsetting a matrix

Vector slicing

```
vector[c(n1, n2, n3)]
```

```
hp.mat[c(1, 3, 7), ]
```
selecting multiple rows

No "," = extracting values instead of entire rows

Selecting multiple elements

Indexing a matrix row

[n,]

Indexing a matrix column

[,n]

```
> hp.mat[c(1, 3, 7), ]
     [,1] [,2]
[1,]  381 1340
[2,]  396  960
[3,]  262  879
>
```

matrix!

Subsetting matrices with

column names

row names

```
colnames(hp.mat) <- c("USA", "Worldwide")
rownames(hp.mat) <- c("Hallows Part 2", "Sorcerer's Stone", "Hallows Part 1",
                      "Order", "Prince", "Goblet", "Chamber", "Prisoner")
hp.mat
```

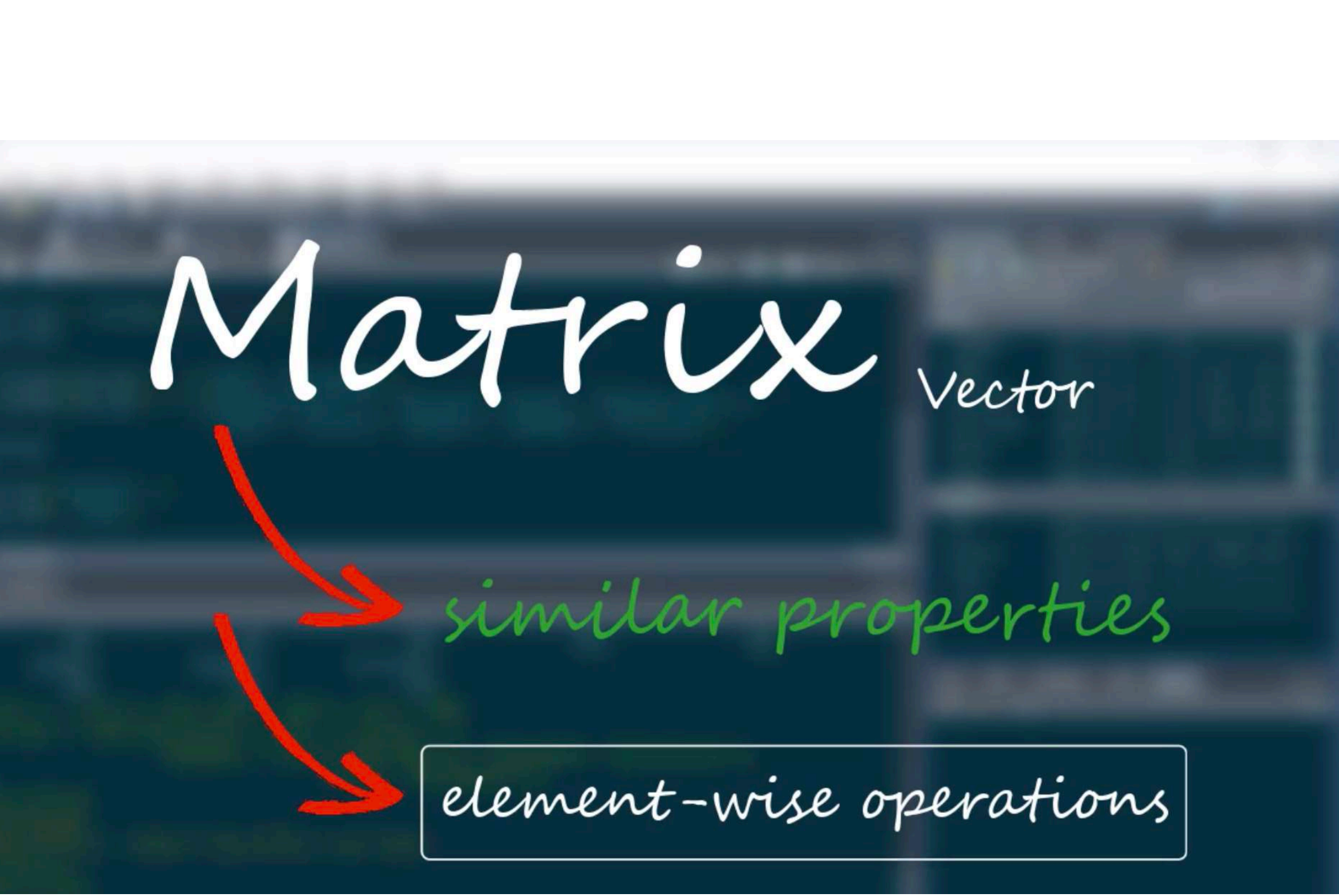
```
> hp.mat
                 USA Worldwide
Hallows Part 2   381      1340
Sorcerer's Stone 318       975
Hallows Part 1   396       960
Order            292       940
Prince           302       934
Goblet           290       897
Chamber          262       879
Prisoner         249       797
>
```

```
hp.mat["Goblet", ]
```

```
> hp.mat["Goblet", ]
     USA Worldwide
     290       897
>
```

vector! because we are calling a single row, and a 1-D object suffices

```
> hp.mat[ ,"USA"]
 Hallows Part 2 Sorcerer's Stone   Hallows Part 1            Order           Prince
            381              318              396              292              302
        Goblet          Chamber         Prisoner
            290              262              249
>
```


Matrix
Vector
similar properties
element-wise operations

All arithmetic operations use element-wise execution

```
b.office <- c(171.5, 292, 281.6, 460.6, 139.3, 288)
matrix.mat <- matrix(b.office, nrow = 3, byrow = T,
                     dimnames = list(c("The Matrix", "Reloaded", "Revolutions"),
                                     c("US","Worldwide")))
matrix.mat
```

```
> matrix.mat
            US Worldwide
The Matrix 171.5     292.0
Reloaded   281.6     460.6
Revolutions 139.3    288.0
>
```

3x2

in $ millions

What if I want to convert the millions into billions?

Use scaling!

Multiplying or dividing an entire structure by a single number

÷ 1,000

```
> matrix.scaled <- matrix.mat/1000
> matrix.scaled
            US Worldwide
The Matrix  0.1715    0.2920
Reloaded    0.2816    0.4606
Revolutions 0.1393    0.2880
>
```

What's the average margin for each Matrix movie ...given a budget of $ 121 million?

```
> avg.margin <- matrix.mat - 121
> avg.margin
            US Worldwide
The Matrix   50.5     171.0
Reloaded    160.6     339.6
Revolutions  18.3     167.0
>
```

Subtracting two matrices

```
matrix.scaled <- matrix.mat/1000
matrix.scaled

avg.margin <- matrix.mat - 121
avg.margin

budget <- matrix(c(63, 150, 150), nrow = 3, ncol = 2)
budget
margin <- matrix.mat - budget
margin
```

...creating the budget matrix

...discovering the margin

```
budget <- matrix(c(63, 150, 150), nrow = 3, ncol = 2)
budget
     [,1] [,2]
[1,]   63   63
[2,]  150  150
[3,]  150  150
```

R recycled to create this


```
> margin <- matrix.mat - budget
> margin
             US Worldwide
The Matrix  108.5     229.0
Reloaded    131.6     310.6
Revolutions -10.7     138.0
>
```

margin!

Subtracting a vector from a matrix

```
budget <- matrix(c(63, 150, 150), nrow = 3, ncol = 2)
budget
margin <- matrix.mat - budget
margin
matrix.mat - c(63, 150, 150)
```

R recycled to create this

```
> margin <- matrix.mat - budget
> margin
             US Worldwide
The Matrix  108.5      229.0
Reloaded    131.6      310.6
Revolutions -10.7      138.0
> matrix.mat - c(63, 150, 150)
             US Worldwide
The Matrix  108.5      229.0
Reloaded    131.6      310.6
Revolutions -10.7      138.0
>
```

Matrix multiplication

Element by element!

...creating the 1 to 6 matrix

...multiplying the matrices

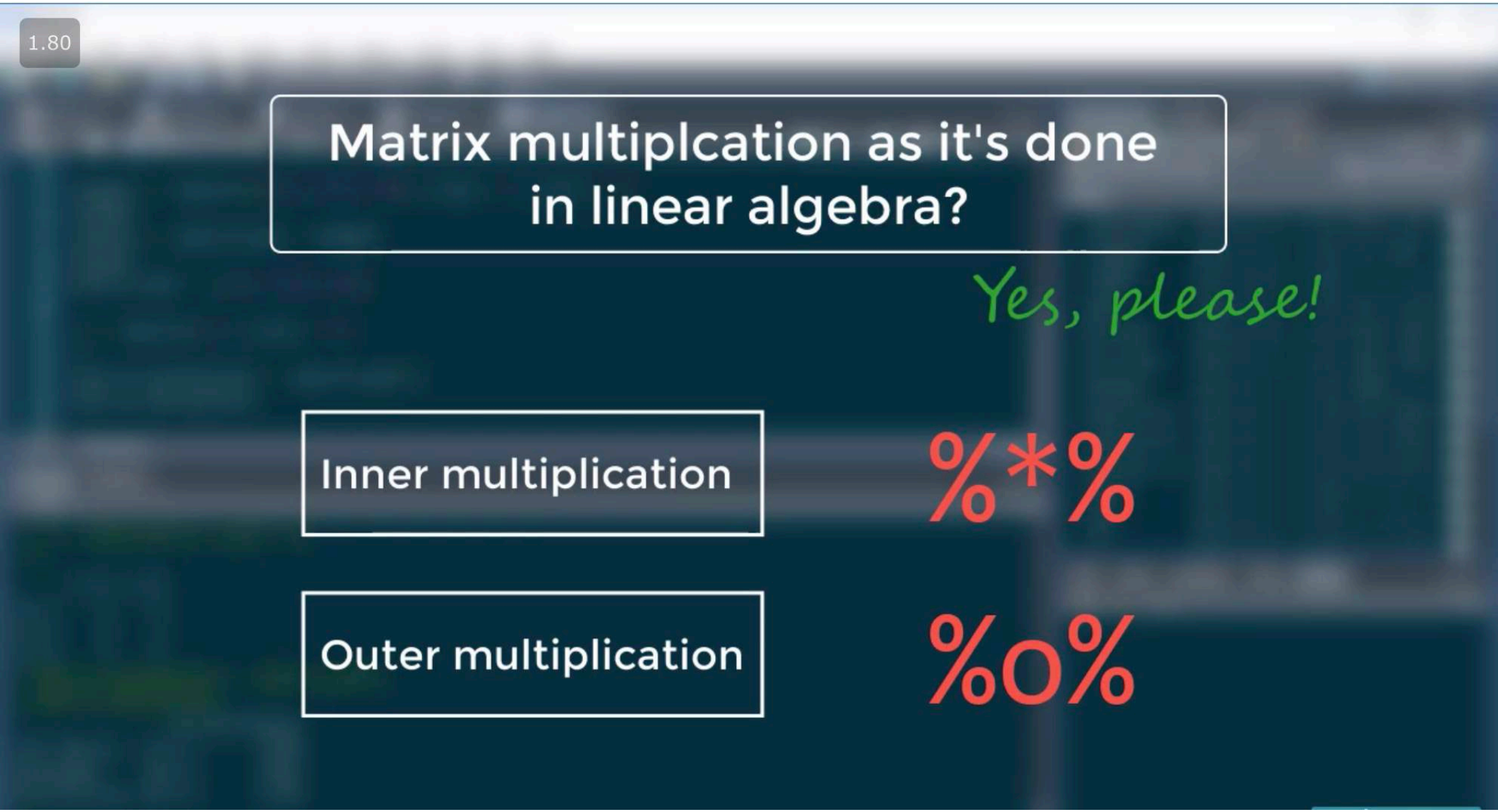
```
v <- matrix(1:6, nrow = 3)
v
matrix.multiplied <- matrix.mat*v
matrix.multiplied
```

```
> v <- matrix(1:6, nrow = 3)
> v
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> matrix.multiplied <- matrix.mat*v
> matrix.multiplied
             US Worldwide
The Matrix  171.5      1168
Reloaded    563.2      2303
Revolutions 417.9      1728
>
```

X

```
> matrix.mat
             US Worldwide
The Matrix  171.5      292.0
Reloaded    281.6      460.6
Revolutions 139.3      288.0
```

=


1.80
Matrix multiplcation as it's done in linear algebra?
Yes, please!
Inner multiplication
%*%
Outer multiplication
%o%

Sums

R is CasE-SeNSitiVe

```
# --- RECREATING matrix.mat ---

# matrix.mat <- matrix(c(171.5, 292, 281.6, 460.6,  139.3, 288), nrow =3 , byrow = T,
#                      dimnames = list(c("The Matrix", "Reloaded", "Revolutions"),
#                                      c("US", "Worldwide")))
```

```
> matrix.mat
             US Worldwide
The Matrix  171.5     292.0
Reloaded    281.6     460.6
Revolutions 139.3     288.0
```

`colSums(data)`
returns the sum for each column in your data structure

`rowSums(data)`
returns the sum for each row in your data structure

```
> colSums(matrix.mat)
       US Worldwide
    592.4    1040.6
> rowSums(matrix.mat)
 The Matrix    Reloaded Revolutions
      463.5       742.2       427.3
>
```

For each movie, US + Worldwide

Means

What about averages?

```
> matrix.mat
                US Worldwide
The Matrix  171.5     292.0
Reloaded    281.6     460.6
Revolutions 139.3     288.0
```

`colMeans(data)`
returns the mean for each column in your data structure

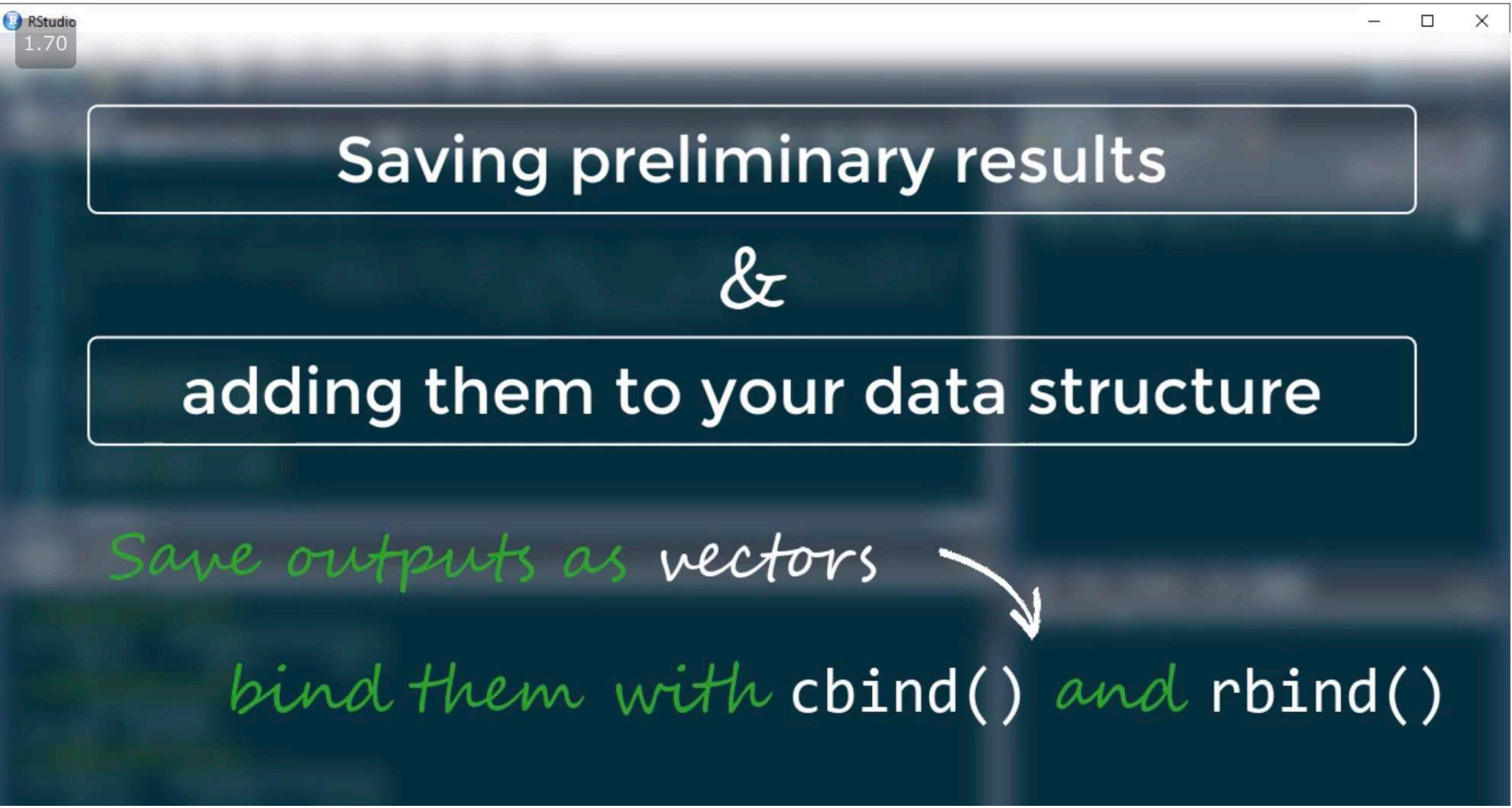
```
> colMeans(matrix.mat)
       US Worldwide
  197.4667  346.8667
>
```

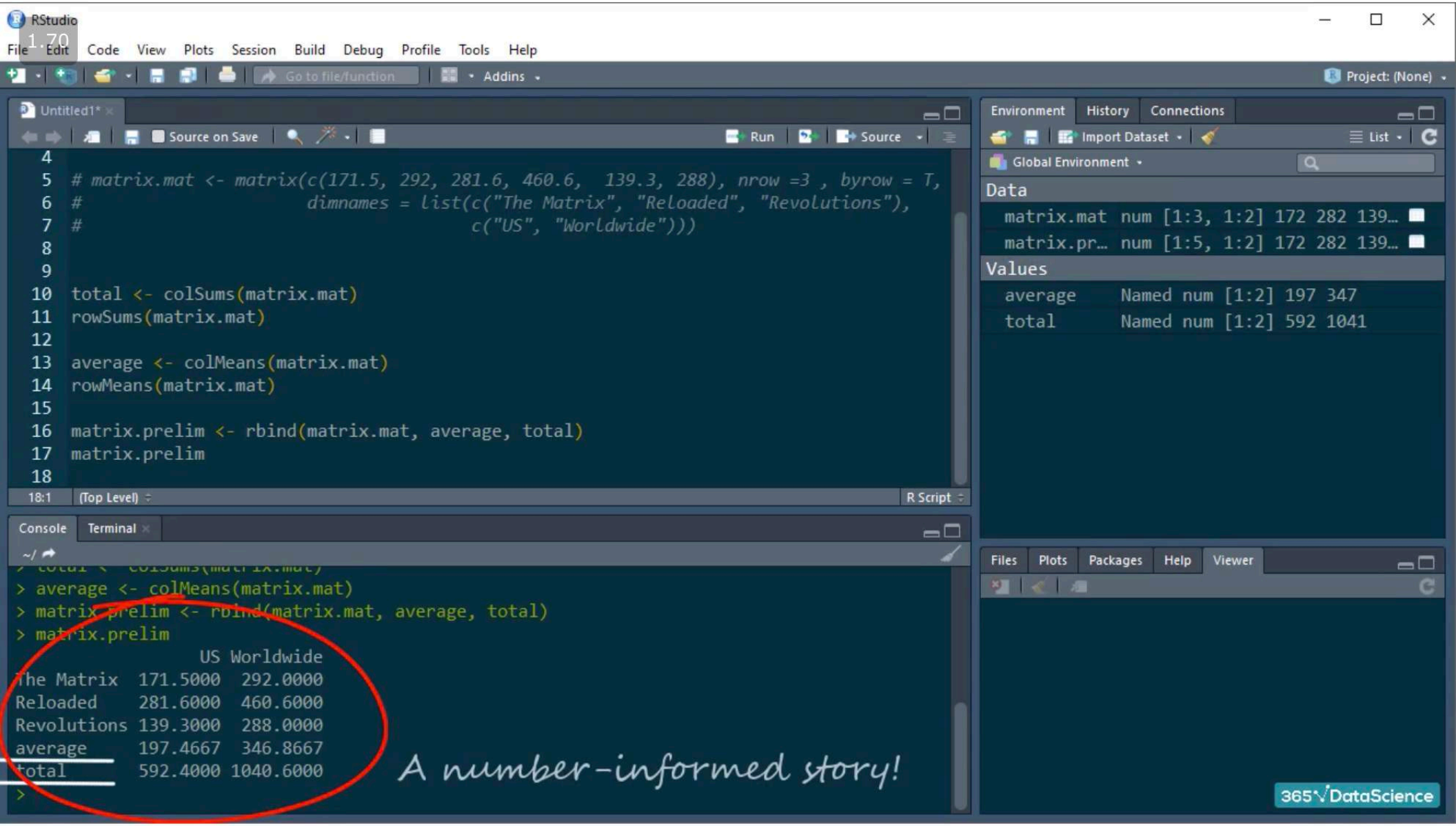
Averages for each location

`rowMeans(data)`
returns the mean for each row in your data structure

```
197.4667  346.8667
> rowMeans(matrix.mat)
The Matrix    Reloaded Revolutions
    231.75      371.10      213.65
>
```

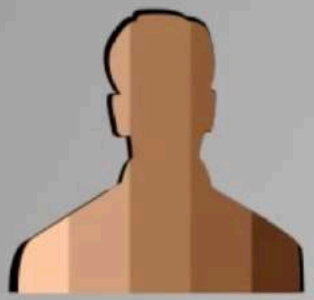
Averages for each movie

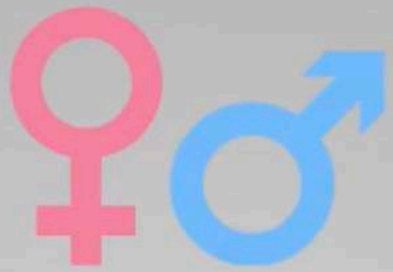

RStudio
1.70
Saving preliminary results
&
adding them to your data structure
Save outputs as vectors
bind them with cbind() and rbind()


RStudio
1.70
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Project: (None)
Untitled1*
Source on Save
Run
Source
4
5 # matrix.mat <- matrix(c(171.5, 292, 281.6, 460.6, 139.3, 288), nrow =3 , byrow = T,
6 # dimnames = list(c("The Matrix", "Reloaded", "Revolutions"),
7 # c("US", "Worldwide")))
8
9
10 total <- colSums(matrix.mat)
11 rowSums(matrix.mat)
12
13 average <- colMeans(matrix.mat)
14 rowMeans(matrix.mat)
15
16 matrix.prelim <- rbind(matrix.mat, average, total)
17 matrix.prelim
18
18:1 (Top Level) R Script
Console Terminal
~/
> average <- colMeans(matrix.mat)
> matrix.prelim <- rbind(matrix.mat, average, total)
> matrix.prelim
US Worldwide
The Matrix 171.5000 292.0000
Reloaded 281.6000 460.6000
Revolutions 139.3000 288.0000
average 197.4667 346.8667
total 592.4000 1040.6000
>
A number-informed story!
Environment History Connections
Import Dataset
List
Global Environment
Data
matrix.mat num [1:3, 1:2] 172 282 139…
matrix.pr… num [1:5, 1:2] 172 282 139…
Values
average Named num [1:2] 197 347
total Named num [1:2] 592 1041
Files Plots Packages Help Viewer
365√DataScience

CATEGORICAL VARIABLES

Groups of data that fall into a limited number of categories





Nominal

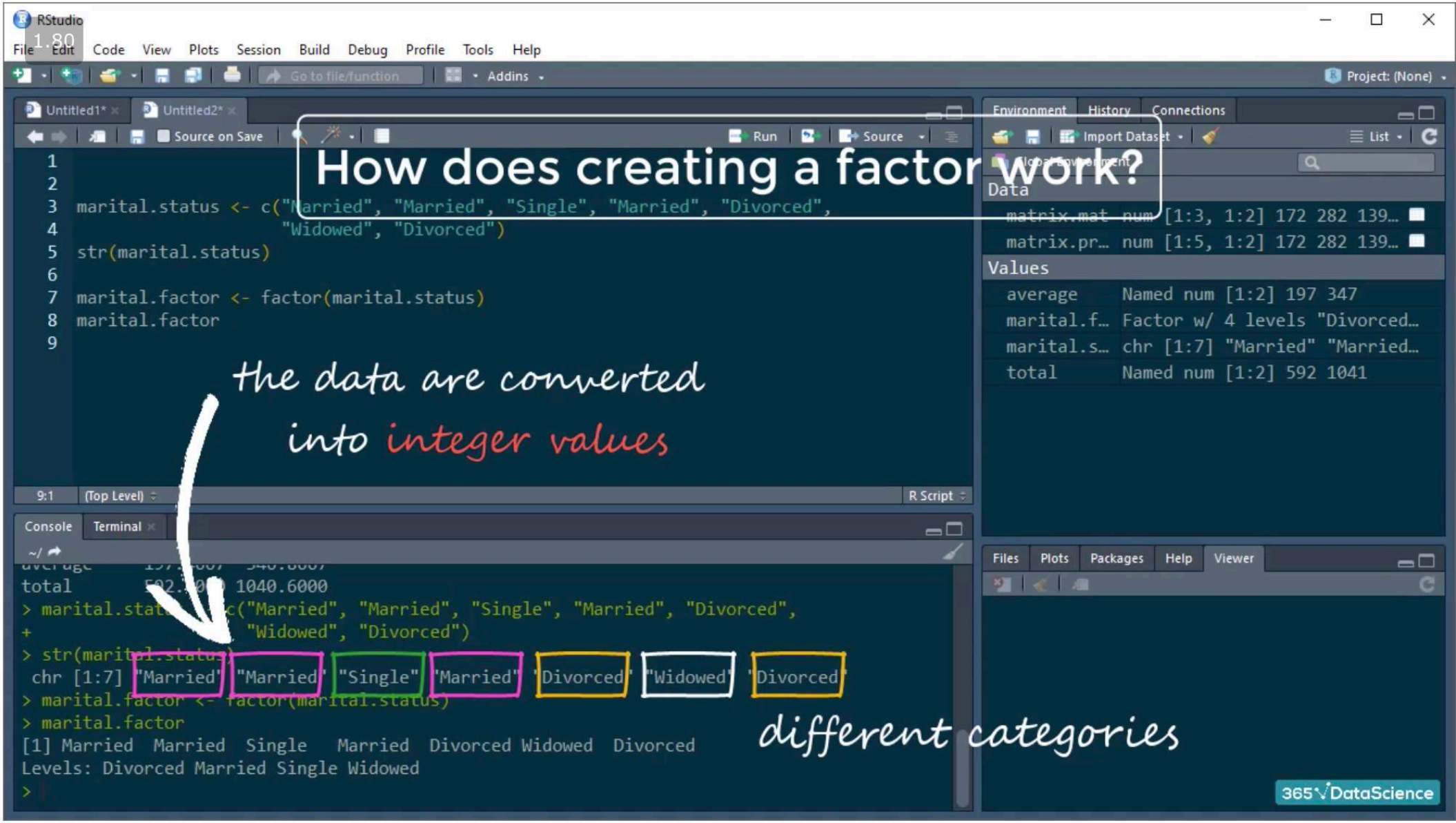
- Jewish
- Black People
- White People
- Hispanic
- Arab
- .....

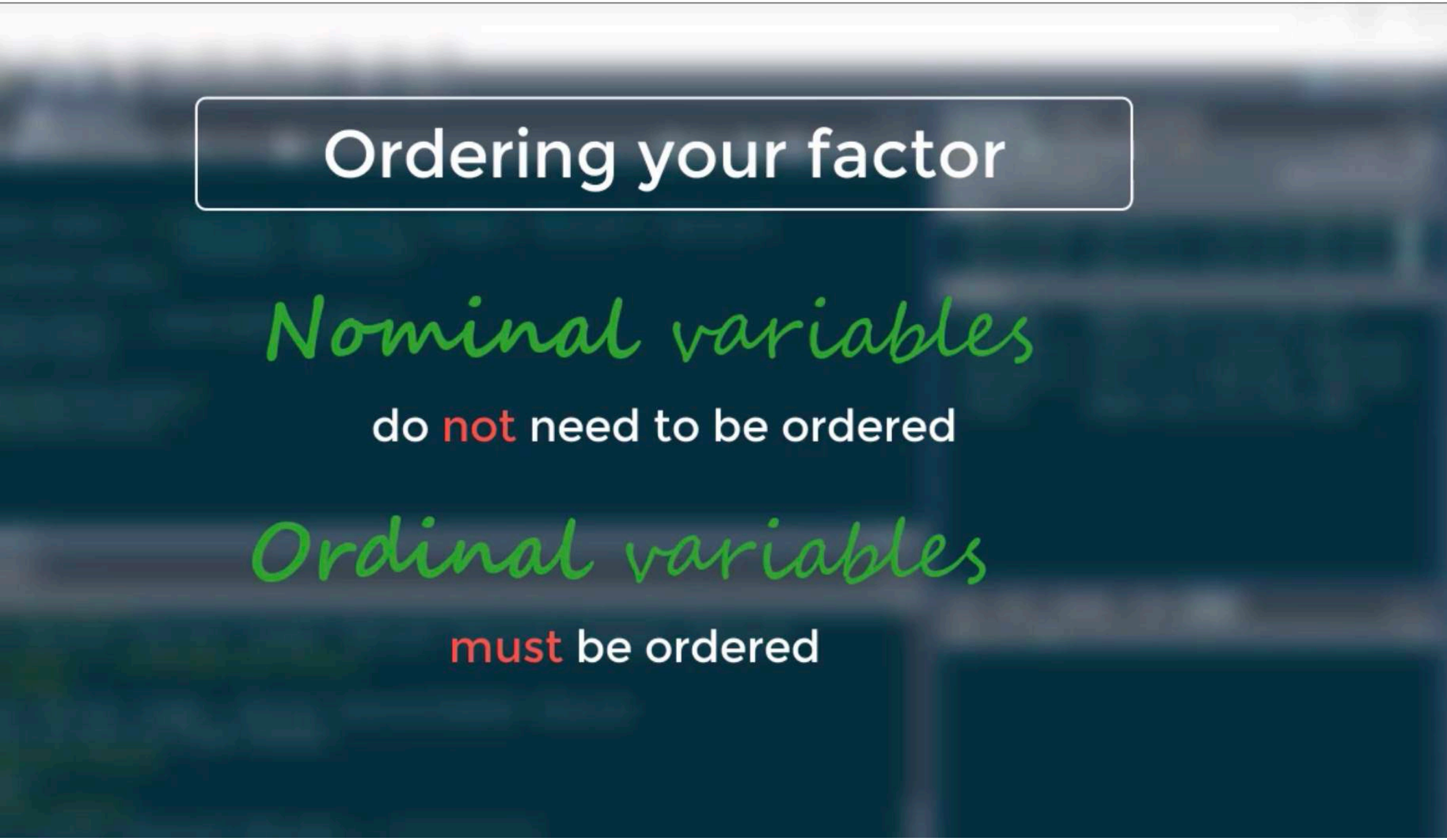
The different categories in a categorical variable are called levels

how much formal education a person has received

Ordinal

- None
- Primary
- Secondary
- Short-cycle tertiary
- Bachelor
- Master
- Doctoral


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Addins
Project: (None)
Untitled1*
Untitled2*
Source on Save
Run
Source
How does creating a factor work?
1
2
3 marital.status <- c("Married", "Married", "Single", "Married", "Divorced",
4 "Widowed", "Divorced")
5 str(marital.status)
6
7 marital.factor <- factor(marital.status)
8 marital.factor
9
the data are converted
into integer values
9:1
(Top Level)
R Script
Environment
History
Connections
Import Dataset
List
Data
matrix.mat num [1:3, 1:2] 172 282 139…
matrix.pr… num [1:5, 1:2] 172 282 139…
Values
average Named num [1:2] 197 347
marital.f… Factor w/ 4 levels "Divorced…
marital.s… chr [1:7] "Married" "Married…
total Named num [1:2] 592 1041
Console
Terminal
~/
total 1040.6000
> marital.status <- c("Married", "Married", "Single", "Married", "Divorced",
+ "Widowed", "Divorced")
> str(marital.status)
chr [1:7] "Married" "Married" "Single" "Married" "Divorced" "Widowed" "Divorced"
> marital.factor <- factor(marital.status)
> marital.factor
[1] Married Married Single Married Divorced Widowed Divorced
Levels: Divorced Married Single Widowed
>
different categories
Files
Plots
Packages
Help
Viewer
365 DataScience


Ordering your factor
Nominal variables
do not need to be ordered
Ordinal variables
must be ordered

`factor(data, levels =)`
`levels` = sets the order of encoding

`levels()`
provides acces to the levels attribute of a variable; can use it to rename the levels

```
12
13  new.factor <- factor(marital.status,
14                       levels = c("Single", "Married", "Divorced", "Widowed"))
15  str(new.factor)
16
```

16:1 (Top Level) R Script

Console Terminal

```
Levels: Divorced Married Single Widowed
> typeof(marital.factor)
[1] "integer"
> str(marital.factor)
 Factor w/ 4 levels "Divorced","Married",..: 2 2 3 2 1 4 1
> new.factor <- factor(marital.status,
+                      levels = c("Single", "Married", "Divorced", "Widowed"))
> str(new.factor)
 Factor w/ 4 levels "Single","Married",..: 2 2 1 2 3 4 3
>
```

```
> marital.factor
[1] Married  Married  Single   Married  Divorced Widowed  Divorced
```

```
,..: 2 2 1 2 3 4 3
```

Modifying factor names

```
new.factor <- factor(marital.status,
                     levels = c("Single", "Married", "Divorced", "Widowed"))
str(new.factor)

levels(new.factor) <- c("s", "m", "d", "w")
```

Ordering factor levels

`factor(data, ordered = TRUE)`
allows you to define the order of your factor levels

```
                  labels = c("Sin", "Mar", "Div", "Wid"))
str(new.factor)

ordered.factor <- factor(marital.status, ordered = TRUE,
                         levels = c("Single", "Married", "Divorced", ))
```

enabling ordering

defining the levels in the order we want them

```
+                         levels = c("Single", "Married", "Divorced", "Widowed"))
> ordered.factor
[1] Married  Married  Single   Married  Divorced Widowed  Divorced
Levels: Single < Married < Divorced < Widowed
>
```

This means our factor is now ordered!

Ordering fac

Lists

1. a.k.a., recursive vectors
2. group R objects into a set
3. can store other lists
4. one-dimensional
5. can store elements of different basic types
6. no inherent structure

Lists

hierarchical structures

tree-like elements

List

```
my.book <- list("1984", "George Orwell", 1949,
                list(c(1:8), c(1:10), c(1:6), "Newspeak"))
my.book
```

```
> my.book
[[1]]
[1] "1984"

[[2]]
[1] "George Orwell"

[[3]]
[1] 1949

[[4]]
[[4]][[1]]
[1] 1 2 3 4 5 6 7 8

[[4]][[2]]
 [1]  1  2  3  4  5  6  7  8  9 10

[[4]][[3]]
[1] 1 2 3 4 5 6

[[4]][[4]]
[1] "Newspeak"
```

Element

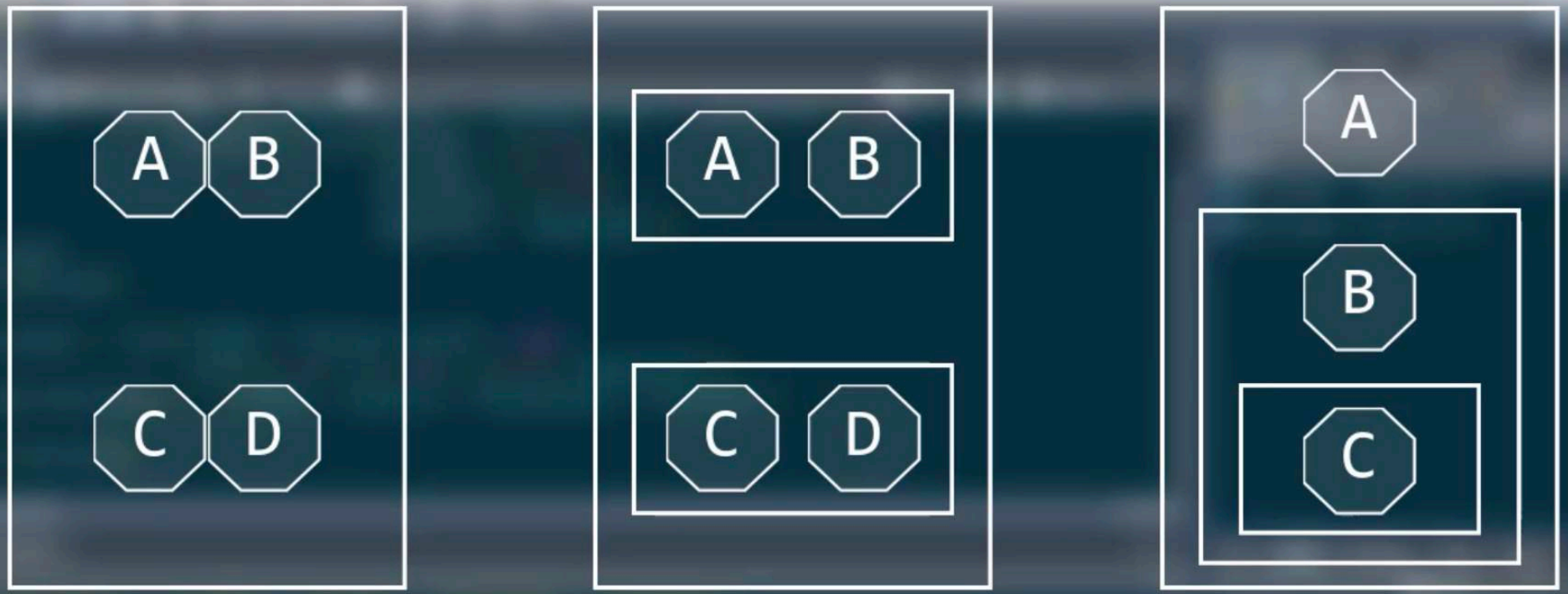
Sub-Element

```
> str(my.book)
List of 4
 $ : chr "1984"
 $ : chr "George Orwell"
 $ : num 1949
 $ :List of 4
  ..$ : int [1:8] 1 2 3 4 5 6 7 8
  ..$ : int [1:10] 1 2 3 4 5 6 7 8 9 10
  ..$ : int [1:6] 1 2 3 4 5 6
  ..$ : chr "Newspeak"
>
```

Naming a list

```
my.book <- list(Name = "1984", Author = "George Orwell", Published = 1949,
                Contents = list(PartOne = c(1:8),
                                PartTwo = c(1:10),
                                PartThree = c(1:6),
                                Appendix = "Newspeak"))
my.book
```

```
> str(my.book)
List of 4
 $ Name     : chr "1984"
 $ Author   : chr "George Orwell"
 $ Published: num 1949
 $ Contents :List of 4
  ..$ PartOne  : int [1:8] 1 2 3 4 5 6 7 8
  ..$ PartTwo  : int [1:10] 1 2 3 4 5 6 7 8 9 10
  ..$ PartThree: int [1:6] 1 2 3 4 5 6
  ..$ Appendix : chr "Newspeak"
```




```
list(c("A","B"),
     c("C","D"))
```

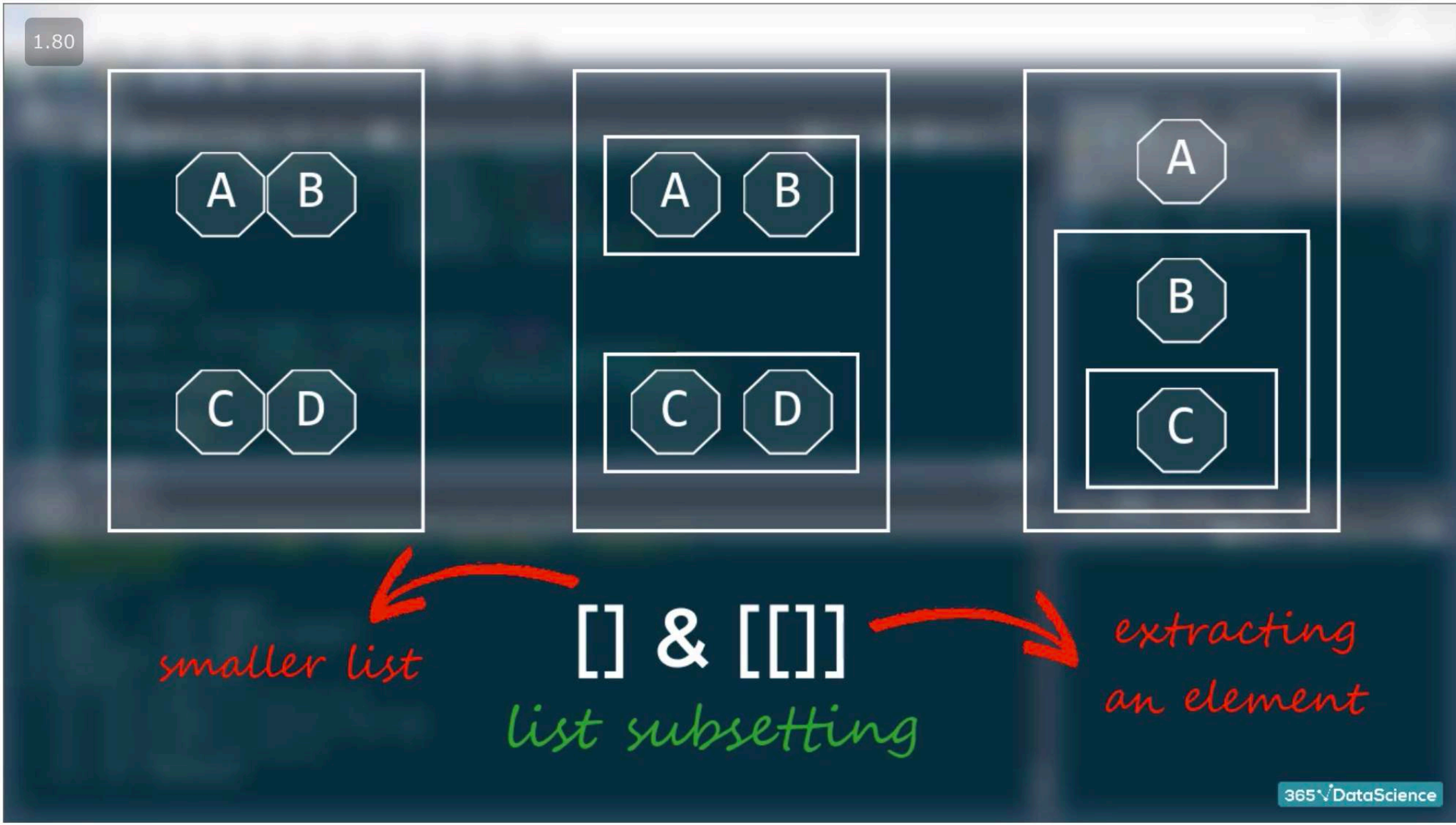
x

```
list(list("A","B"),
     list("C","D"))
```

y

```
list(A", list("B",
     list("C")))
```

z


1.80
A
B
C
D
A
B
C
D
A
B
C
smaller list
[] & [[]]
list subsetting
extracting
an element
365 DataScience

[] & [[]]

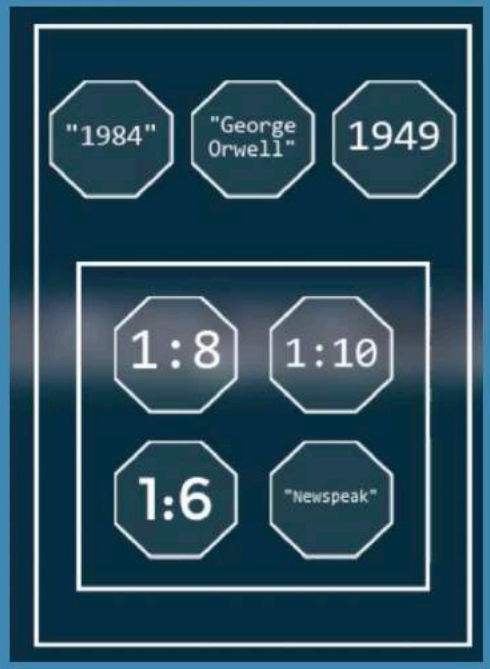
list subsetting

```
my.book <- list(Name = "1984", Author = "George Orwell", Published = 1949,
                Contents = list(PartOne = c(1:8),
                                PartTwo = c(1:10),
                                PartThree = c(1:6),
                                Appendix = "Newspeak"))
my.book
```

```
> str(my.book)
List of 4
 $ Name     : chr "1984"
 $ Author   : chr "George Orwell"
 $ Published: num 1949
 $ Contents :List of 4
  ..$ PartOne  : int [1:8] 1 2 3 4 5 6 7 8
  ..$ PartTwo  : int [1:10] 1 2 3 4 5 6 7 8 9 10
  ..$ PartThree: int [1:6] 1 2 3 4 5 6
  ..$ Appendix : chr "Newspeak"
```

```
> my.book[1:2]
$Name
[1] "1984"

$Author
[1] "George Orwell"
```




```
> my.book[4]
$Contents
$Contents$PartOne
[1] 1 2 3 4 5 6 7 8

$Contents$PartTwo
 [1]  1  2  3  4  5  6  7  8  9 10

$Contents$PartThree
[1] 1 2 3 4 5 6

$Contents$Appendix
[1] "Newspeak"
```

List with a list in it. Not the content itself.

```
> my.book[[4]]
$PartOne
[1] 1 2 3 4 5 6 7 8

$PartTwo
 [1]  1  2  3  4  5  6  7  8  9 10

$PartThree
[1] 1 2 3 4 5 6

$Appendix
[1] "Newspeak"
```

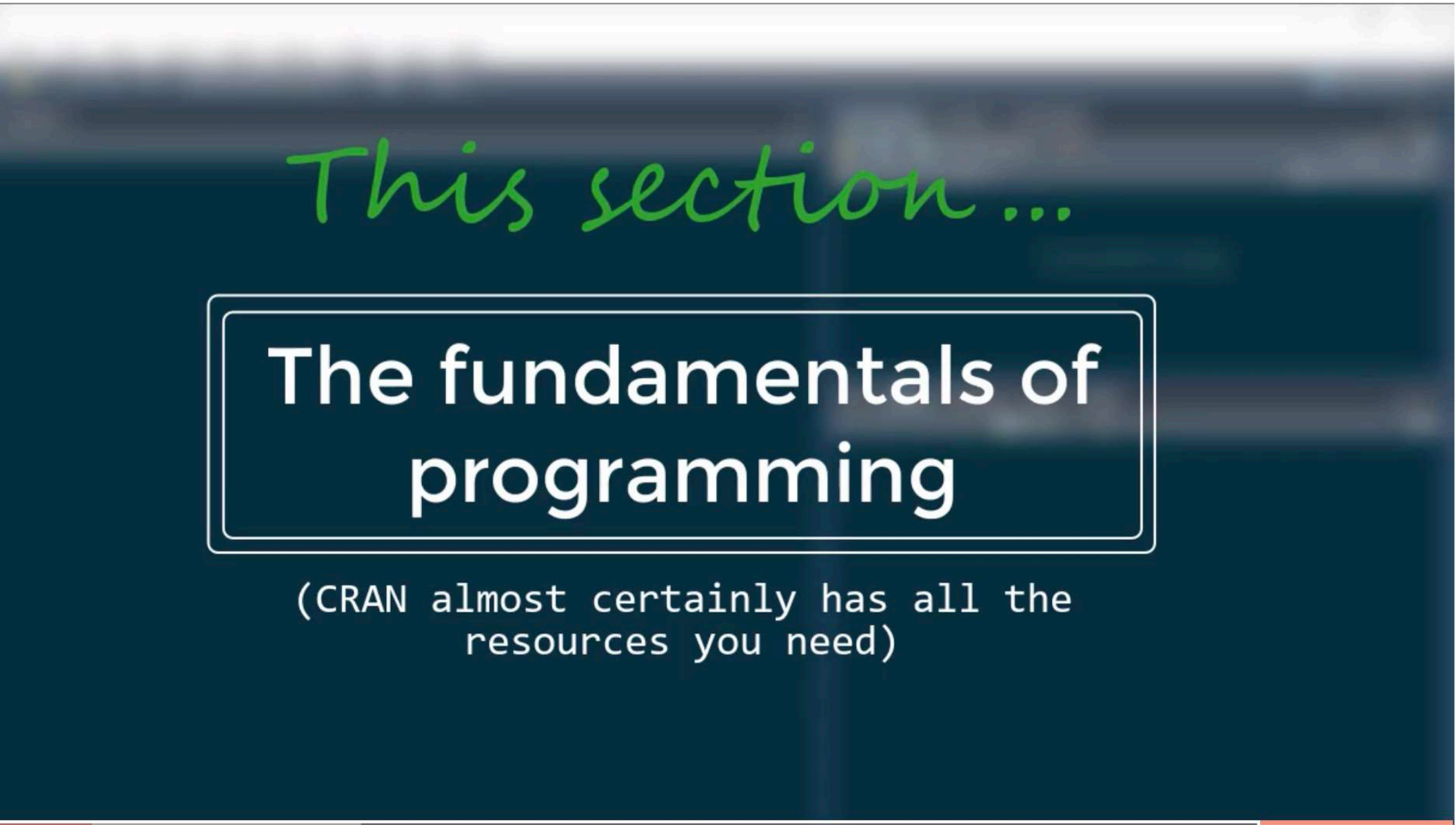
The content itself.

```
> my.book[[4]][2]
$PartTwo
 [1]  1  2  3  4  5  6  7  8  9 10
```

Elements of position 2 as list inside of a vector

```
> my.book[[4]][[2]]
 [1]  1  2  3  4  5  6  7  8  9 10
```

Elements of position 2 as a vector


This section ...
The fundamentals of programming
(CRAN almost certainly has all the resources you need)

Relational operators

for evaluating objects in relation to one another

`==` `!=` `<` `>` `<=` `>=`

!! The "=" has a different function !!

it's another version of "<-"

can be used with different types of objects

```
> 3 == 3
[1] TRUE
> "cat" == "cat"
[1] TRUE
> "cat" == "car"
[1] FALSE
> TRUE == TRUE
[1] TRUE
> TRUE == FALSE
[1] FALSE
> "sugar" != "salt"
[1] TRUE
> 3 != 3
[1] FALSE
> TRUE != FALSE
[1] TRUE
```

!=

checks if something isn't equal to something else

```
> 17 > 21
[1] FALSE
> 17 > 5
[1] TRUE
> "rat" > "cat"
[1] TRUE
> TRUE < FALSE
[1] FALSE
> 7 >= 7
[1] TRUE
> 7 >= 4
[1] TRUE
> 7 <= 4
[1] FALSE
```

alphabetical order

Cercion?

T = 1
F = 0

Logical operators

and "&"

returns TRUE iff all values are TRUE
& is an exclusive operator

```
> TRUE & TRUE
[1] TRUE
> TRUE & FALSE
[1] FALSE
> FALSE & TRUE
[1] FALSE
> FALSE & FALSE
[1] FALSE
```

or "|"

an inclusive operator

as long as there is a TRUE value in the expression, | evaluates to TRUE

```
> TRUE | TRUE
[1] TRUE
> TRUE | FALSE
[1] TRUE
> FALSE | TRUE
[1] TRUE
> FALSE | FALSE
[1] FALSE
```

not "!"

flips the result of a logical test

```
> !TRUE
[1] FALSE
> !(4 < 3)
[1] TRUE
```

Logical Operators AND Vectors

ELEMENT-BY-ELEMENT

```
> v <- c(1, 3, 5, 7)
> w <- c(1, 2, 3, 4)
>
> 3 == v
[1] FALSE  TRUE FALSE FALSE
> 3 == w
[1] FALSE FALSE  TRUE FALSE
```

```
> 12 > c(11, 11, 13, 14)
[1]  TRUE  TRUE FALSE FALSE
> "catch" == c("catch", 22, "is", "fantastic")
[1]  TRUE FALSE FALSE FALSE
>
> c(11, 12, 13) >= c(10, 12, 14)
[1]  TRUE  TRUE FALSE
```

Logical operators, comparisons, and vectors

```
> (v >= 3) & (v < 7)
[1] FALSE  TRUE  TRUE FALSE
> v >= 3
[1] FALSE  TRUE  TRUE  TRUE
> v < 7
[1]  TRUE  TRUE  TRUE FALSE
>
> (v >= 3) | (v < 7)
[1] TRUE TRUE TRUE TRUE
> !(v > 3)
[1]  TRUE  TRUE FALSE FALSE
```

If, else, else if statements

1. `if(A){Z}`
2. The condition must evaluate to a single logical value
3. Adding an `else` statement gives the program a plan B in case plan A didn't evaluate to TRUE
4. You can define an `else-if` statement if you want to specify a special case

If, else, else if statements

If statement

an instruction to R to do something if a condition is met

```
if(A){
  Z
}
```

```
> num <- -3
> if(num < 0){
+   print("Your number is negative")
+ }
[1] "Your number is negative"
```

Else statement

the plan B given to R in case the if condition isn't satisfied

```
if(A){
  Scenario 1
} else {
  Scenario 2
}
```

```
v <- 8
if(v < 0){
  v <- v*-1
  print("I have made your object positive. Look:")
  print(v)
} else {
  print("Your object was already positive or zero, so I did nothing")
}
```

Else-If statement

```
v <- 23
if(v < 0){
  print("The object was less than 0, but I am working on it!")
  print("Wait for it...")
  v <- v*-1
  print("Your object is now positive. Check it out!")
  print(v)
} else if(v == 0){
  print("Your object is exactly zero")
} else if((v > 0) & (v < 12)){
  print("The object is positive and less than 12")
} else {
  print("Your object is positive, and larger than 12")
  print(v)
}
```

```
[1] "Your object is positive, and larger than 12"
[1] 23
```

The keep-in-mind's

1. A condition for an if-statement must have only one element that evaluates to a logical

IF there is a vector, the statement will only look at the first element

2. An if-statement only needs one condition to evaluate to TRUE to stop its search

Conditions that are a vector of logical values

```
z <- c(1, 0, -3, 5)
w <-6

if(z < 1){
  print("My parrot sings 'God save the queen'")
}
```

```
+ }
Warning message:
In if (z < 1) { :
  the condition has length > 1 and only the first element will be used
>
```

```
> z < 1
[1] FALSE  TRUE  TRUE FALSE
>
```

First element evaluates False and R does not execute condition

Conditions that are not mutually exclusive

```
if(w < 7){
  print("w is less than 7")
} else if(w == 6){
  print("w is precisely 6")
} else {
  print("w is more than 7")
}
```

```
+   print("w is more than
+ }
[1] "w is less than 7"
>
```

Only first condition is evaluated as only one statement is evaluated when condition is True.

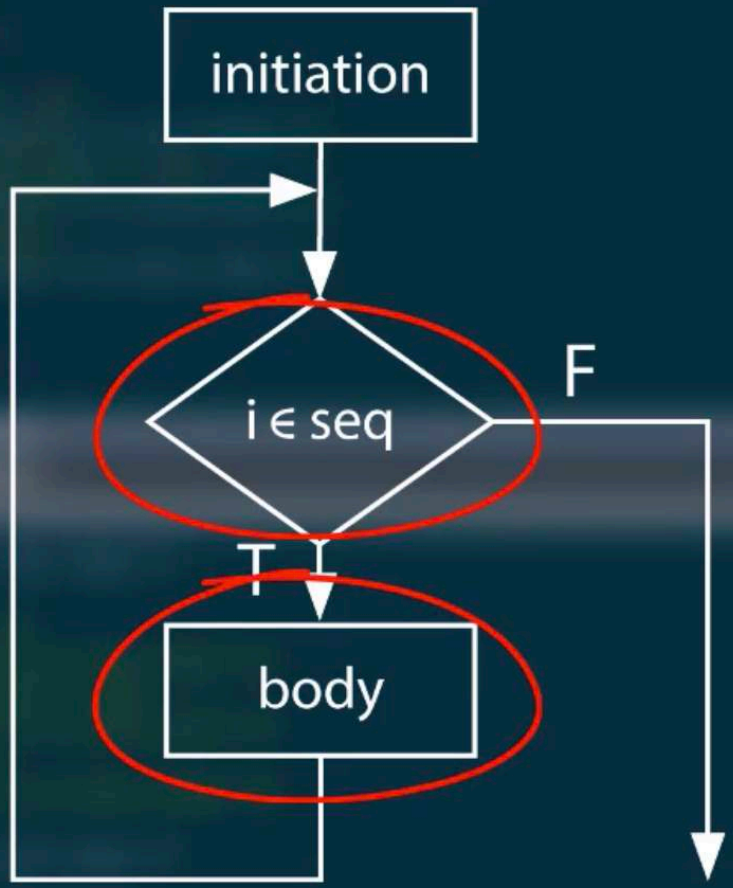
LOOPS

cycles iterations

a method to automate a task that has multiple steps

1.90

for loop



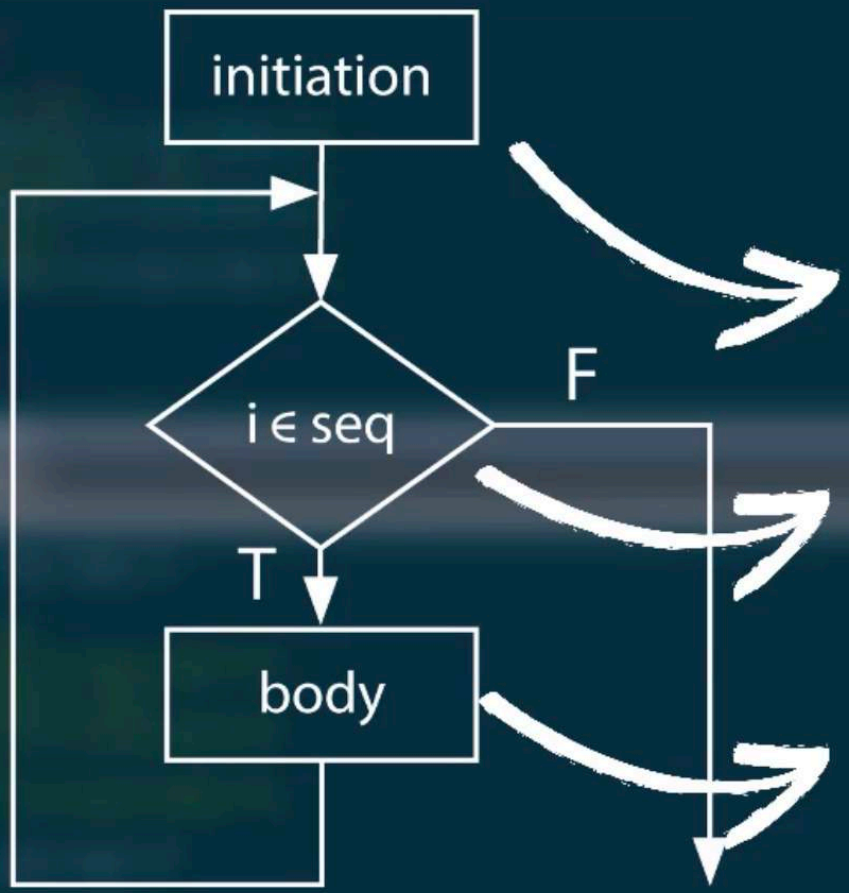
repeats the same operation many times each time with a different value from an input sequence

for every value of x

do y

1.90

for loop



the instructions: define the object with the set of values to use as a reference

f(x): for Every value of x do y

the decision: does i belong to the sequence in x?

the code to loop through: if i belongs to x, execute for i

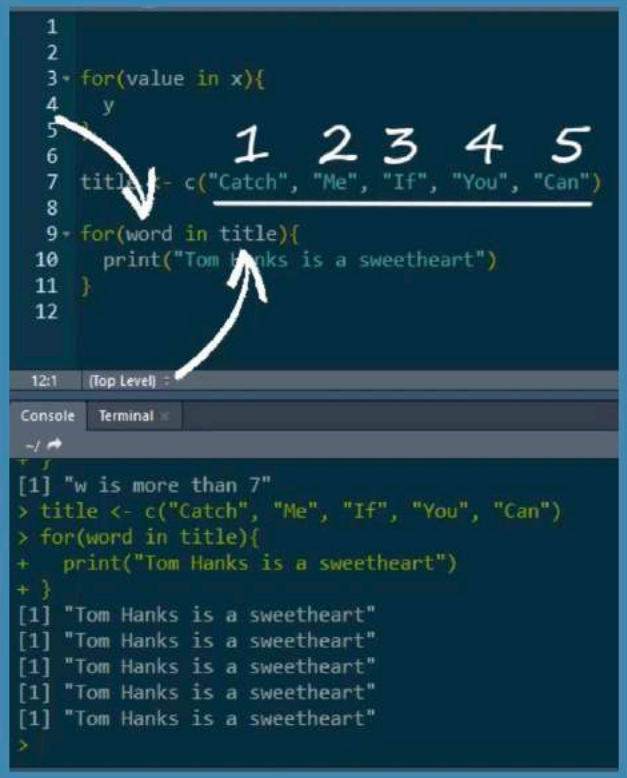
This is the y

for loop

```
for(value in x){
  y
}
```

```
title <- c("Catch", "Me", "If", "You", "Can")

for(word in title){
  print("Tom Hanks is a sweetheart")
}
```



The loop stops when we it had assigned every element in title to word

```
> for(i in title){
+   print(i)
+ }
[1] "Catch"
[1] "Me"
[1] "If"
[1] "You"
[1] "Can"
```

Creating an empty object

```
> new.title <- vector(length = 5)
> new.title
[1] FALSE FALSE FALSE FALSE FALSE
```

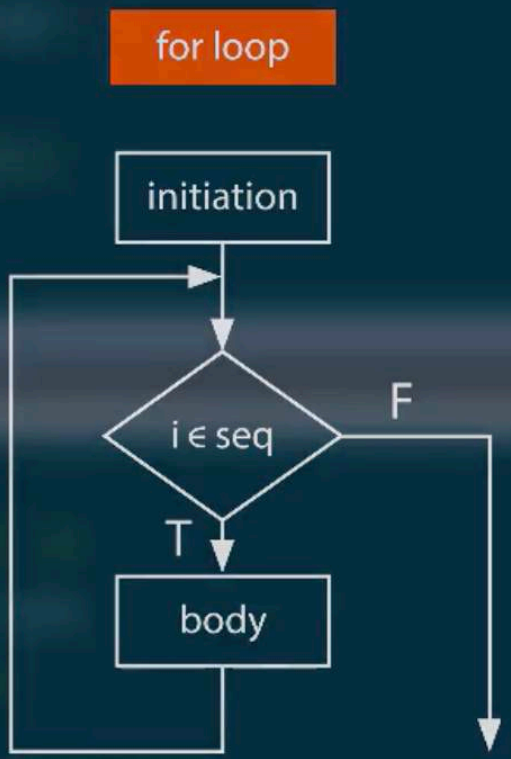
Assigning loop output to a new object

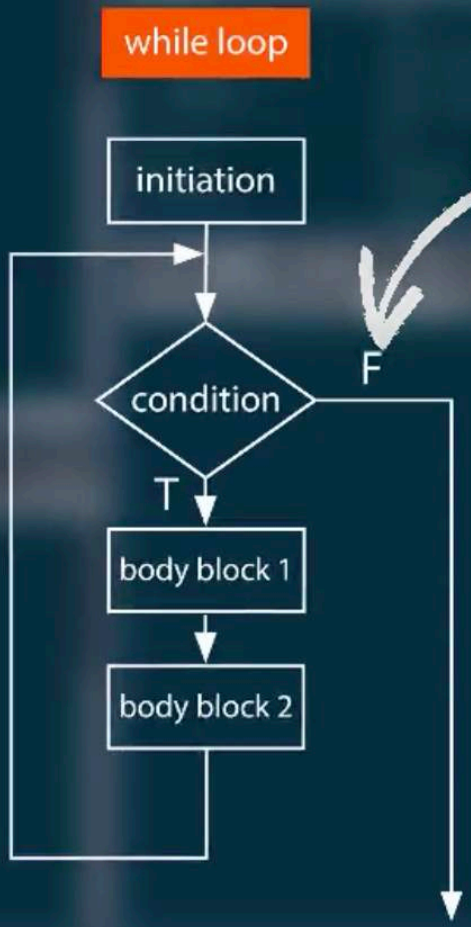
```
for(i in 1:5){
  new.title[i] <- title[i]
}

new.title
```

```
> new.title
[1] "Catch" "Me"    "If"    "You"   "Can"
```

WHILE loops

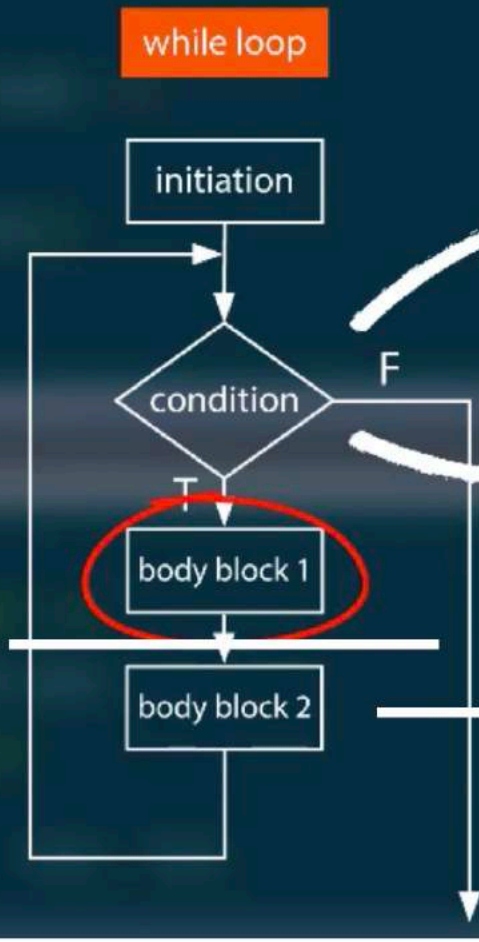




as long as the condition evaluates to TRUE, the loop will keep iterating through the code

365 DataScience

WHILE loops



must evaluate to a logical

compares a value to a control

(n > 5)

Intended to alter the value the condition is checking so it eventually evaluates to FALSE

WHILE loops

```
n <- -12
while(n < 0){
  print("Your value n stands for Negative")
}
```

```
n <- -12
while(n < 0){
  print("Your value n stands for Negative")
}
```

Infinite loop

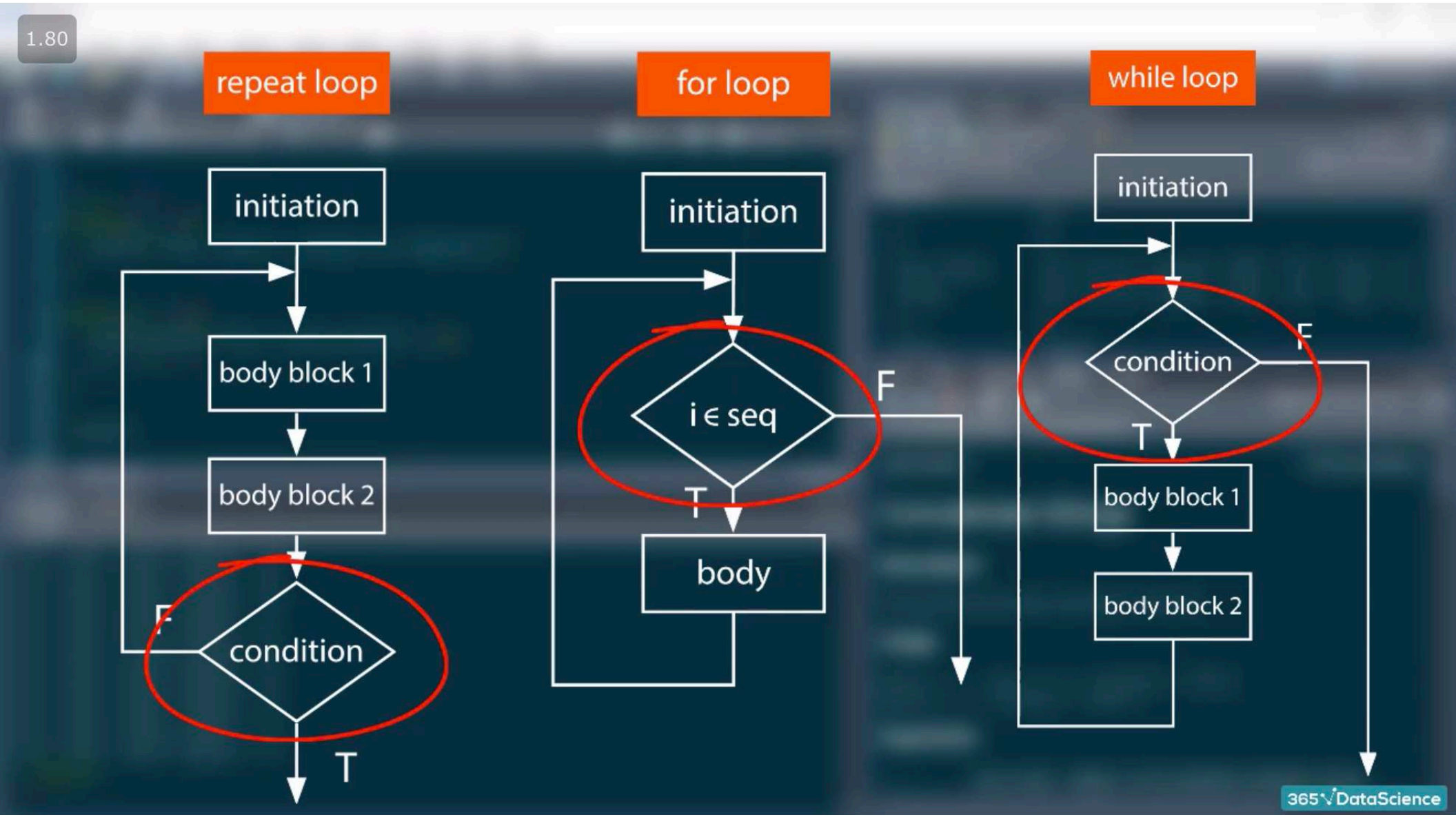
no second block of instructions

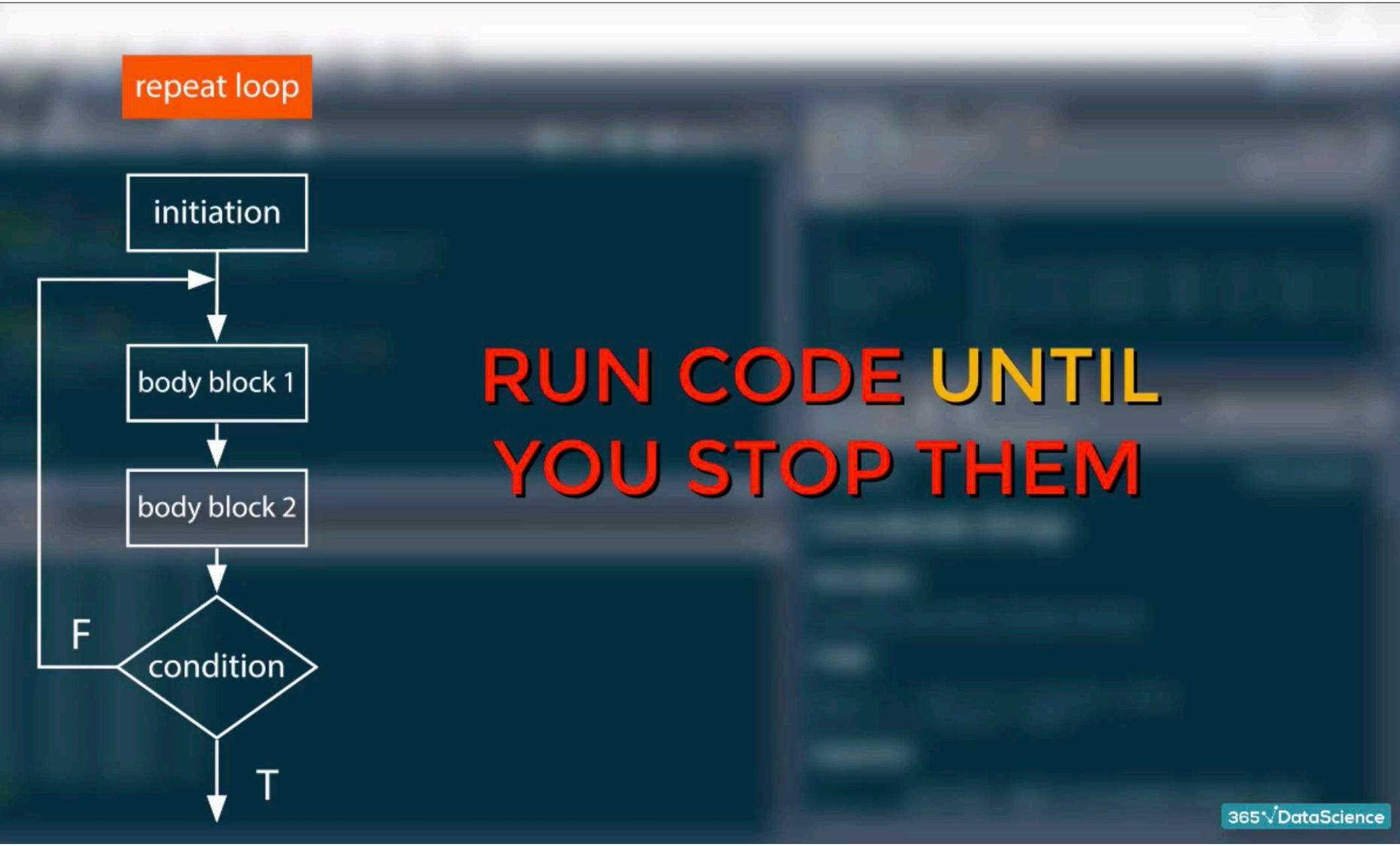
`paste(...)`
turns vectors into string type and combines them

```
while(n < 0){
  print(paste("Your value equals", n))
  n <- n + 1
}
```

WHILE LOOPS DO NOT SAVE THE RESULT OF WHAT HAPPENS IN THE LOOP

you'd need to do that manually


1.80
repeat loop
initiation
body block 1
body block 2
condition
F
T
for loop
initiation
i ∈ seq
F
T
body
while loop
initiation
condition
F
T
body block 1
body block 2
365 DataScience


repeat loop
initiation
body block 1
body block 2
condition
F
T
RUN CODE UNTIL
YOU STOP THEM
365 DataScience

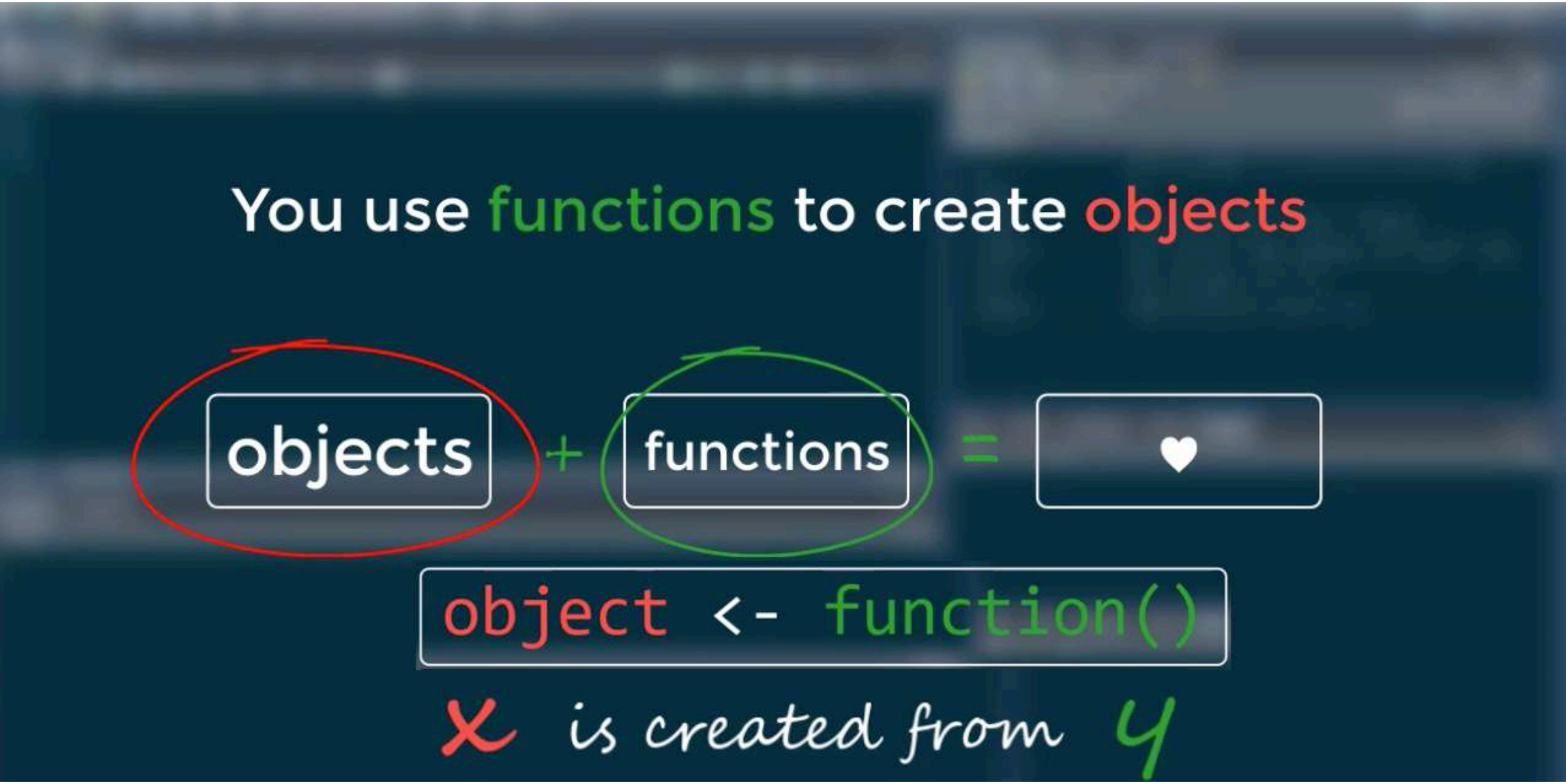
repeat loop

```
while(n < 0){
  print(paste("Your value equals", n))
  n <- n + 1
}
```

```
n <- -12

repeat{
  print(paste("Your value equals", n))
  n <- n + 1
  if(n >= 0){
    print("N is now either 0 or a positive number")
    print("The loop will be broken")
    break
  }
}
```

```
[1] "Your value equals -9"
[1] "Your value equals -8"
[1] "Your value equals -7"
[1] "Your value equals -6"
[1] "Your value equals -5"
[1] "Your value equals -4"
[1] "Your value equals -3"
[1] "Your value equals -2"
[1] "Your value equals -1"
[1] "N is now either 0 or a positive number"
[1] "The loop will be broken"
>
```


You use functions to create objects
objects
+
functions
=
♥
object <- function()
x is created from y

`function.name(x)`
runs the function called `function.name` on the data `x`

```
> round(2,4271)
[1] 2
```

You use functions to create objects

```
object <- function()
```

Because we save the results of functions into separate objects!

`function arguments`
what we pass in the parentheses after the function name

More operations! | Cleaner code!

```
> mean(1:10)
[1] 5.5
> mean(a)
[1] 5.5
```

Data arguments can be:

- raw data
- an R object
- the result of another function

```
a <- (31:60)

b <- mean(a)

c <- round(b)

print(b)
print(c)
```

```
mean(a)

round(mean(a))
```

first, execute `mean(a)` = 5.5

then, round the result with `round()` = 6

`args(function.name)`
returns the arguments a function takes

optional arguments have a default value

```
> args(round)
function (x, digits = 0)
NULL
>
```

digits after the decimal (optional)

the data you are applying the function to

R has a predefined argument order

The more arguments there are, the higher the chance of getting the order wrong


```
> round(2.4271, digits = 2)
[1] 2.43
> round(2.4271, 2)
[1] 2.43
```

Same result!

argument name + switched order

```
round(x, digits = 0)
```

```
> round(digits = 2, 2.4271)
[1] 2.43
>
```

the argument order is switched

but the result is the same!

```
> round(digits = 2, 2.4271)
[1] 2.43
> round(2, 2.4271)
[1] 2
>
```

R's interpretation:
the digits after the decimal

R's interpretation:
the data you are applying the function to

data number of elements you want to sample

```
sample(x, size)
```
sampling function

```
sample(x, size, replace = FALSE)
```
replace is an optional argument; default = FALSE

sets aside any character you draw so you cannot draw it again

```
hand <- sample(deck, size=3, replace=T)
```
hand gets assigned the value `sample` outputs only once

```
# ---------- BUILDING A FUNCTION ----------

deck <- c("Duke", "Assassin", "Captain", "Ambassador", "Contessa")
print(deck)

hand <- sample(deck, size = 3, replace = T)
print(hand)
```

```
> print(hand)
[1] "Ambassador" "Duke"       "Captain"
> print(hand)
[1] "Ambassador" "Duke"       "Captain"
```

`functions in R`
like objects, but store a command instead of data

`functions in R`
name + body of code + arguments

```
function.name <- function(){
    body.of.code
}
```

`function.name()`
functions must be executed with parentheses

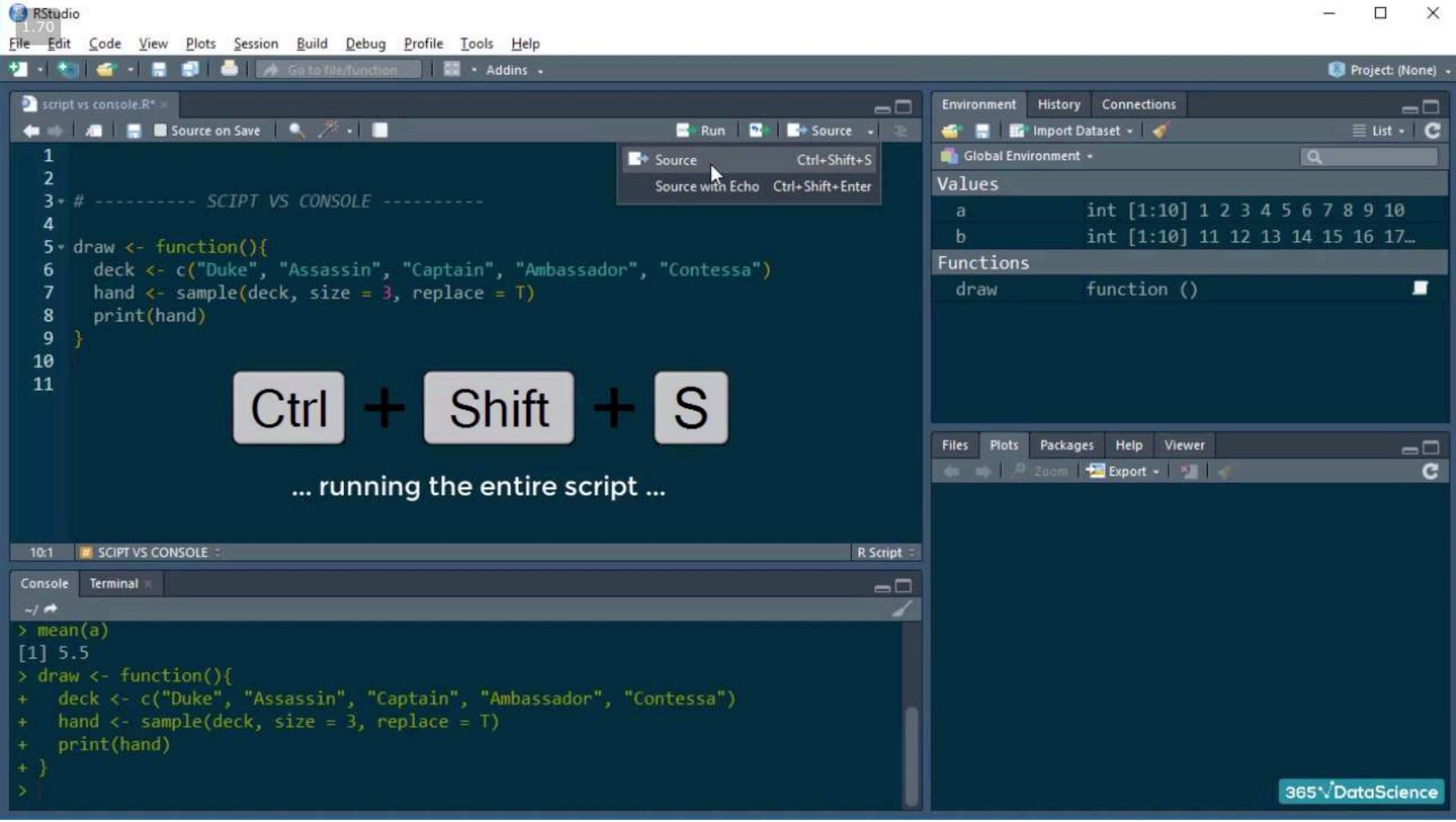
Objectives:

1. Create a deck every time
2. Sample 3 cards from it
3. Allow repetition

```
draw <- function(){
  deck <- c("Duke", "Assassin", "Captain", "Ambassador", "Contessa")
  hand <- sample(deck, size = 3, replace = T)
  print(hand)
}

draw()
```

```
> draw()
[1] "Captain"   "Assassin" "Duke"
> draw()
[1] "Contessa"    "Ambassador" "Contessa"
```


RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
script vs console.R*
Source on Save
Run
Source
Source Ctrl+Shift+S
Source with Echo Ctrl+Shift+Enter
# ---------- SCIPT VS CONSOLE ----------
draw <- function(){
deck <- c("Duke", "Assassin", "Captain", "Ambassador", "Contessa")
hand <- sample(deck, size = 3, replace = T)
print(hand)
}
Ctrl + Shift + S
... running the entire script ...
10:1
SCIPT VS CONSOLE
R Script
Console
Terminal
> mean(a)
[1] 5.5
> draw <- function(){
+ deck <- c("Duke", "Assassin", "Captain", "Ambassador", "Contessa")
+ hand <- sample(deck, size = 3, replace = T)
+ print(hand)
+ }
Environment
History
Connections
Import Dataset
List
Global Environment
Values
a int [1:10] 1 2 3 4 5 6 7 8 9 10
b int [1:10] 11 12 13 14 15 16 17…
Functions
draw function ()
Files
Plots
Packages
Help
Viewer
Export
365 DataScience

Defining the arguments of a function

```
# draw <- function(){
#   deck <- c("Duke", "Assassin", "Captain", "Ambassador", "Contessa")
#   hand <- sample(deck, size=3, replace = T)
#   print(hand)
# }
```

```
# draw <- function(deck){
#   hand <- sample(deck, size=3, replace = T)
#   print(hand)
# }
```

```
coup <- matrix(rep(c("Duke", "Assassin", "Captain",
                     "Ambassador", "Contessa"), 3), ncol = 1)

deal <- function(deck){
  cards <- deck[1:3,]
  print(cards)
}

deal(deck = coup)
```

```
> deal(deck = coup)
[1] "Duke"      "Assassin" "Captain"
```

```
shuffle <- function(deck){
  random <- sample(1:23, size = 23)
  deck.s <- deck[random, , drop = FALSE]
  print(deck.s)
}
```

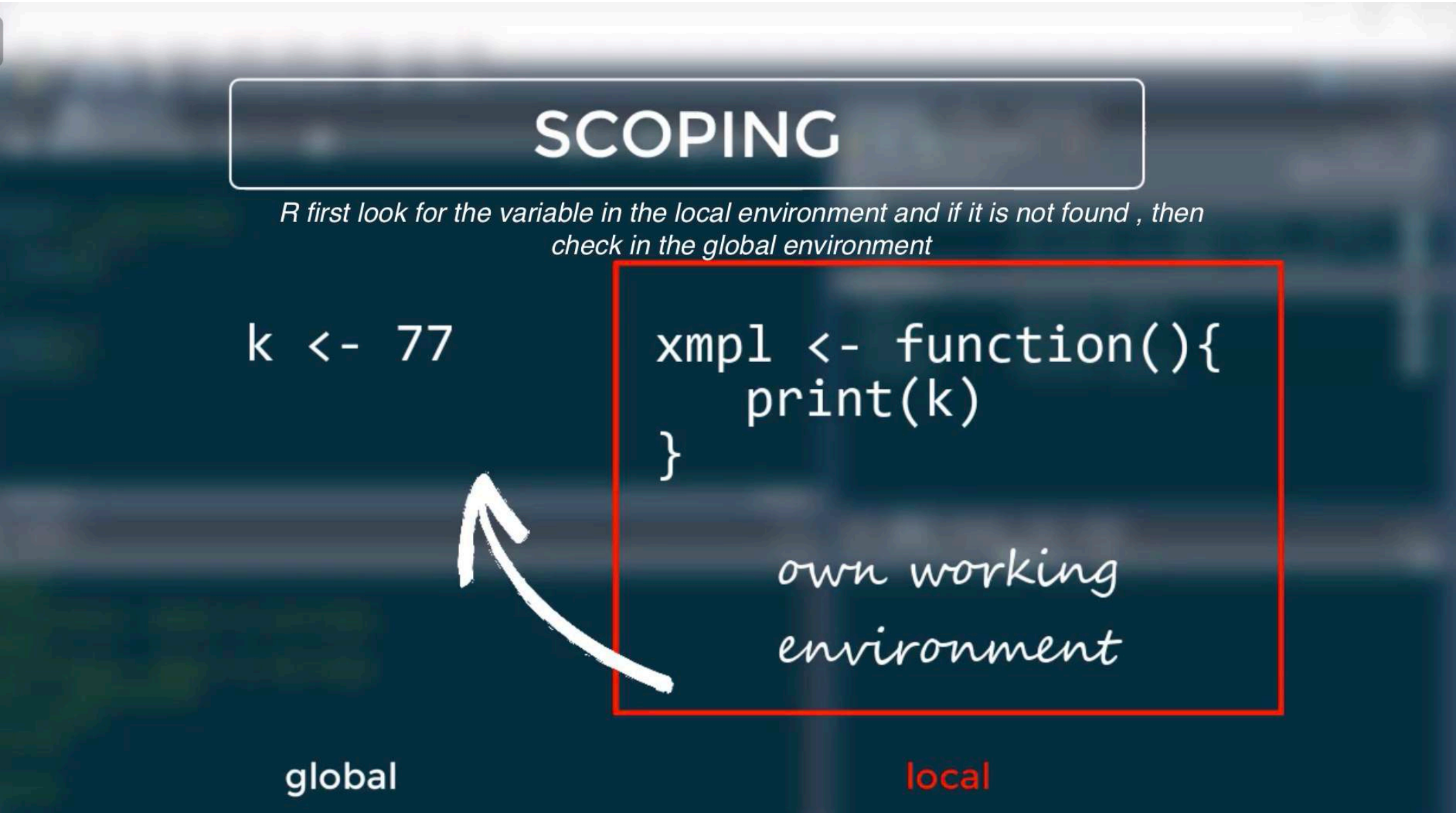
```
cit <- matrix(c("Magistrate", "Thief", "Wizard", "Patrician", "Bishop",
                "Queen", "Witch", "Blackmailer", "Magician", "Emperor",
                "Tax Collector", "Spy", "Seer", "Merchant", "Scholar",
```

```
shuffle(cit)
```

```
deal(cit)
```

SCOPING

R first look for the variable in the local environment and if it is not found , then check in the global environment



Function in a function

Local environment

if an object is created inside the body of a function, it exists and can be accessed only there

```
mult12 <- function(x){
  y <- (x*12)
}

mult12(3)
print(y)
```

```
> mult12 <- function(x){
+   y <- (x*12)
+ }
> mult12(3)
> print(y)
Error in print(y) : object 'y' not found
```

```
mult12 <- function(x){
  y <- (x*12)
  print(y)
}

mult12(3)
```

```
> mult12 <- function(x){
+   y <- (x*12)
+   print(y)
+ }
> mult12(3)
[1] 36
```

Saving the results of a function

Functions returns by default the last evaluated expression & if calling the print statement.

return()

```
mult12 <- function(x){
  y <- (x*12)
  return(y)
}
```

```
new.var <- mult12(3)
new.var
```

```
> new.var
[1] 36
```

```
shuffle <- function(deck){
  random <- sample(1:23, size = 23)
  deck.s <- deck[random, , drop = FALSE]
  return(deck.s)
}

deal <- function(deck){
  shuffled <- shuffle(deck)
  cards <- shuffled[1:3, ]
  return(cards)
}
```

```
> deal(cit)
[1] "Tax Collector" "Bishop"        "Magician"
> deal(cit)
[1] "Emperor"    "Magistrate" "Bishop"
```

Running a custom function without passing an argument

default values

```
deal <- function(deck = matrix(1:23, nrow = 23)){
  shuffled <- shuffle(deck)
  cards <- shuffled[1:3, ]
  return(cards)
}
```

```
> deal()
[1]  6 20 16
```