

R Bootcamp Part 1

Dani Navarro
Amy Perfors

The plan for today

- Me: intro material
- Dani: bringing it all together
- Very introductory (we presume nothing) but know you're smart :)

This is meant to be very interactive!!

Lots of exercises throughout



To make an anonymous comment, go to: www.menti.com

Code 78 13 31

First, a quick question to gauge where everyone is coming from:

Which of these statements best describes your background?

www.menti.com

Code 90 48 78

Structure to my lectures

- About R and Rstudio
- Basic commands
- Using functions
- Introducing variables
- Introducing vectors
- Installing and loading packages
- Managing the workspace
- Loading a workspace file
- Data frames
- Factors
- Saving data to a workspace file
- Importing data from a text file
- Storing commands as a script
- Getting around the computer



About R and RStudio

What is R?

- R is a statistical programming language
- You can use it to
 - Do basic calculations
 - Do statistical analyses
 - Draw graphs
 - Write programs
 - Etc.

Why do we teach R?

- Pluses:

- It's open source and **costs nothing**
- It's very powerful (way more powerful than this class suggests)
- It's rapidly becoming the most popular data analysis tool
- It's also (secretly) an introduction to programming (a valuable skill!)

- Minuses:

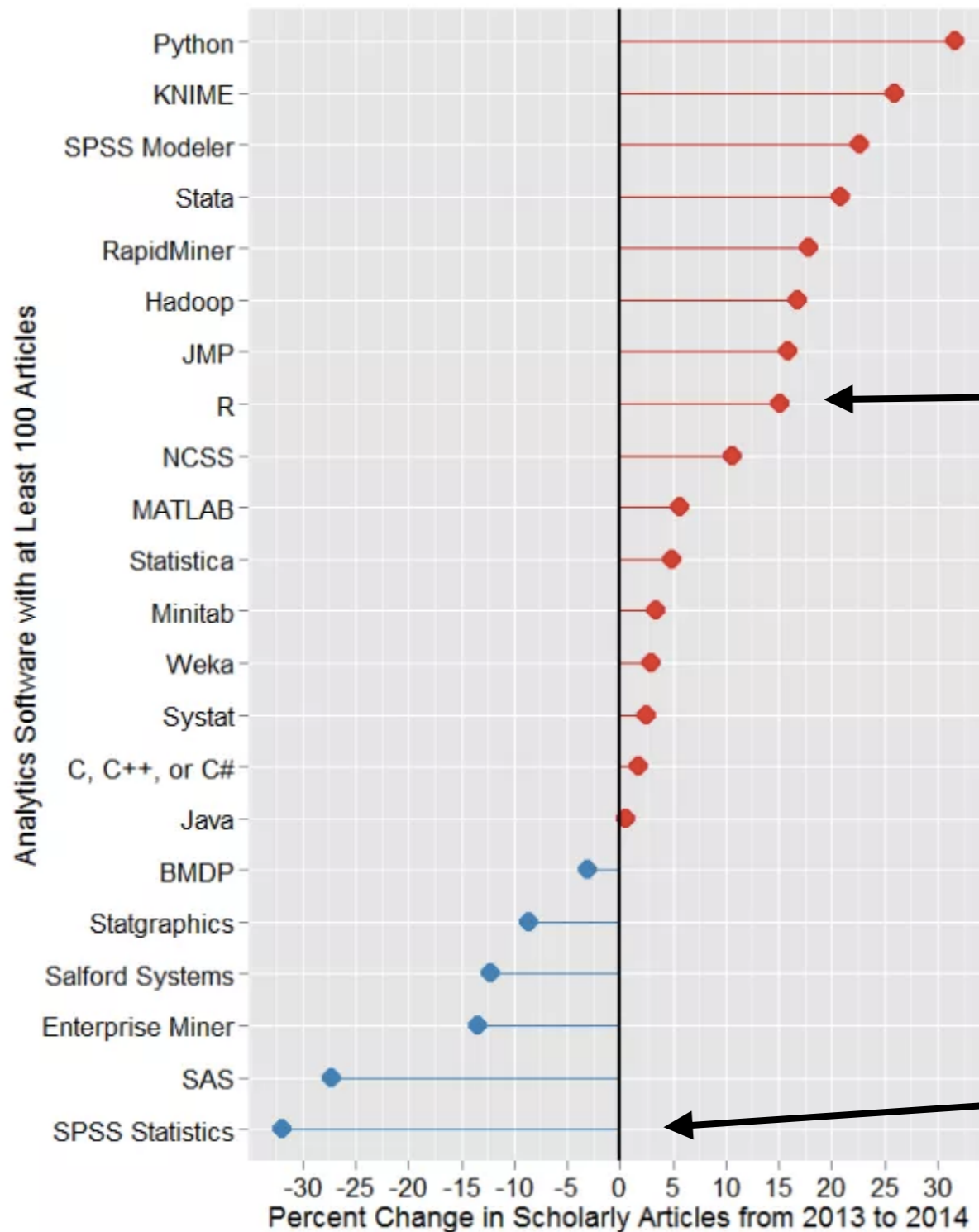
- It's got a steeper learning curve than some alternatives
- Which is why we'll spend a few lectures introducing it
- (As always: don't panic. Previous classes did just fine on this!)

R is where the jobs are...



Job trends from indeed.com. SPSS is another major statistical software (which used to be used here), not open-source.

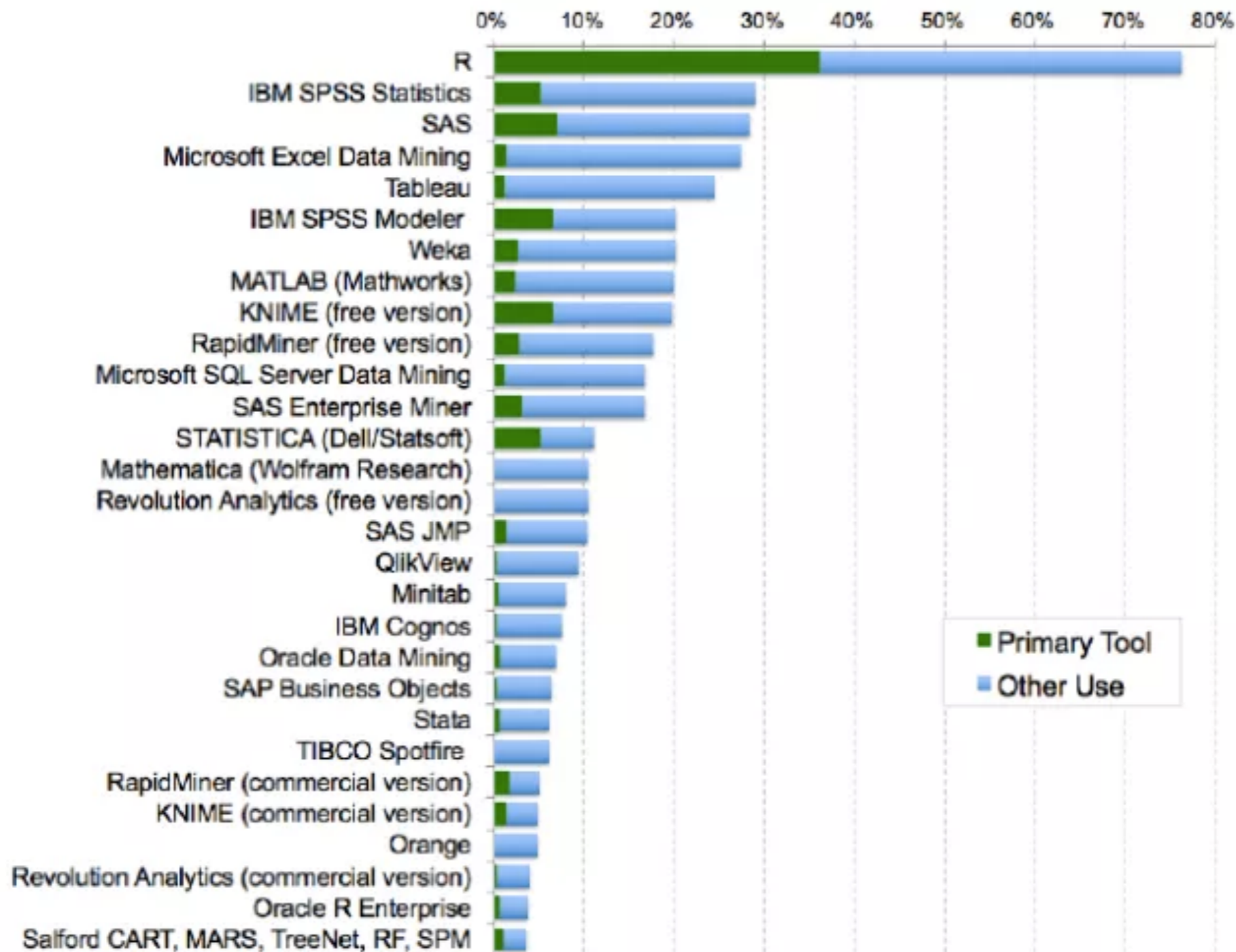
R is (increasingly) what academics use



R increased from 2013 to 2014

SPSS decreased (a lot) from 2013 to 2014

and what most current data scientists use



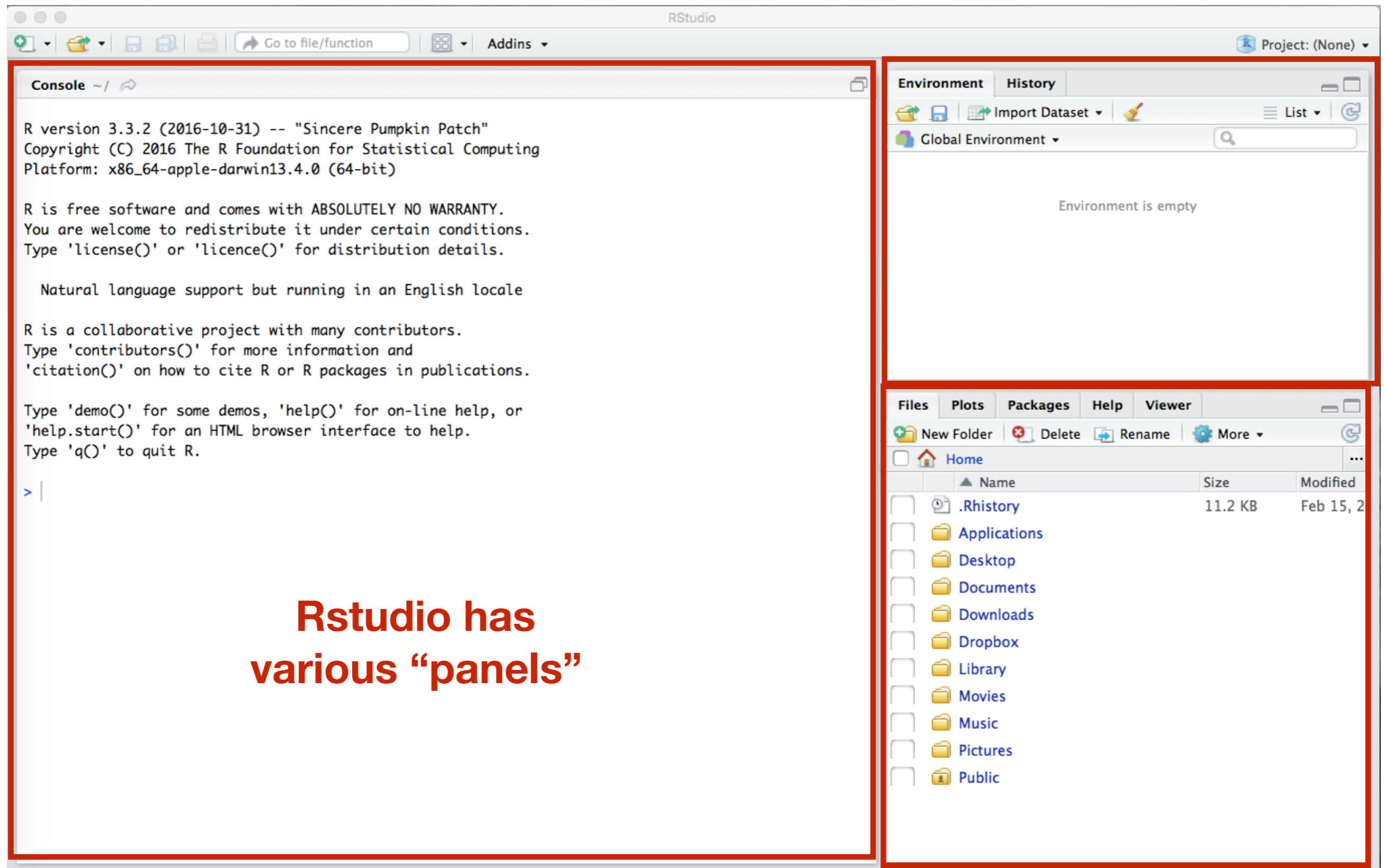
Responses
to a survey
of data
scientists

Getting R and Rstudio

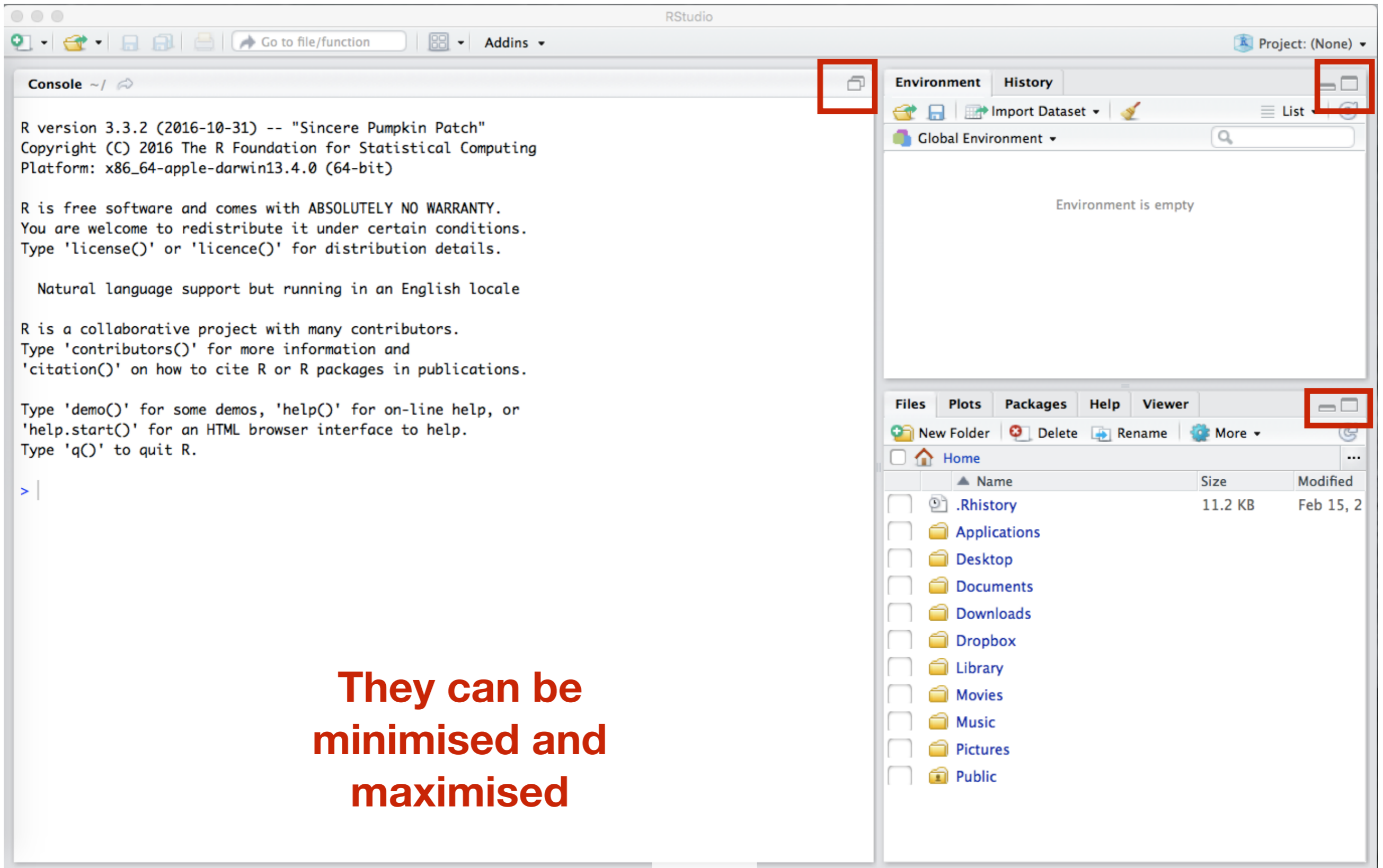
- You should have them on your computer already
- If you don't, the websites you need are:
 - <http://www.r-project.org/> (install R first)
 - <http://www.rstudio.com> (install Rstudio next)
- There are documents on our resource page describing the process in detail
- Dani's PSYR page also describes the process:
 - <http://compcogscisydney.org/psyr/#core-toolkit>



This is the icon for Rstudio.
Open this to get started

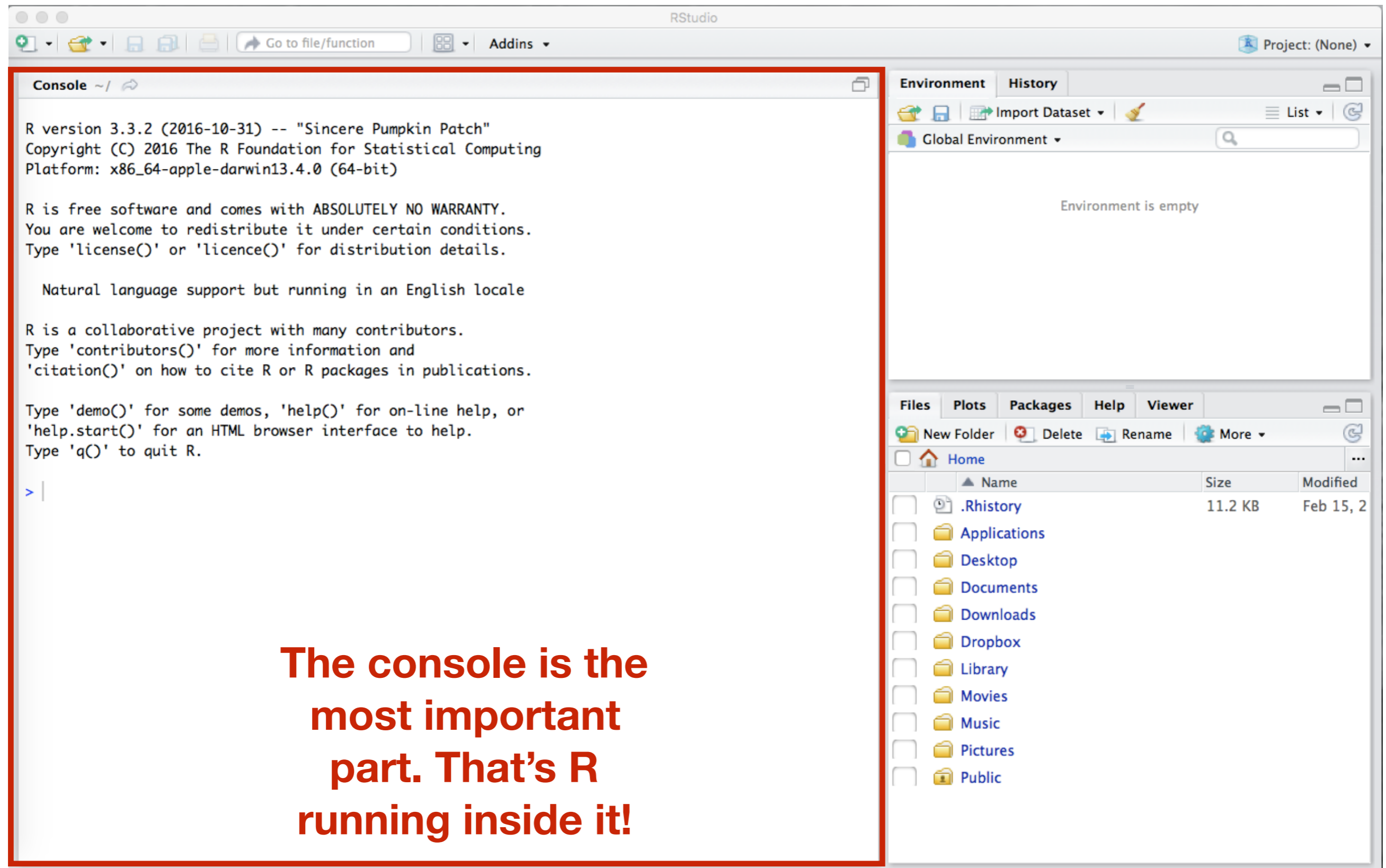


Note: I use a Mac and all my screenshots will be from that. There may be subtle differences if you have Windows, but nothing substantial will differ.



The image shows the RStudio interface. The console on the left displays the R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch" and its license information. The right-hand side of the interface is divided into several panels. The top-right panel is split into 'Environment' and 'History' tabs, both of which are highlighted with red boxes. Below this, there are tabs for 'Files', 'Plots', 'Packages', 'Help', and 'Viewer', also highlighted with red boxes. The 'Files' panel is currently active, showing a file browser view of the home directory with columns for Name, Size, and Modified. The 'Environment' panel shows 'Global Environment' and 'Environment is empty'. The 'History' panel is currently empty.

Click on these to bring up different “panels” (we’ll talk about them later)



The console is the most important part. That's R running inside it!



Basic commands

Console ~/ ↵

```
R version 3.3.2 (2016-10-31) -- "Sincere Pumpkin Patch"  
Copyright (C) 2016 The R Foundation for Statistical Computing  
Platform: x86_64-apple-darwin13.4.0 (64-bit)
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.
```

```
  Natural language support but running in an English locale
```

```
R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.
```

```
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.
```

```
>
```

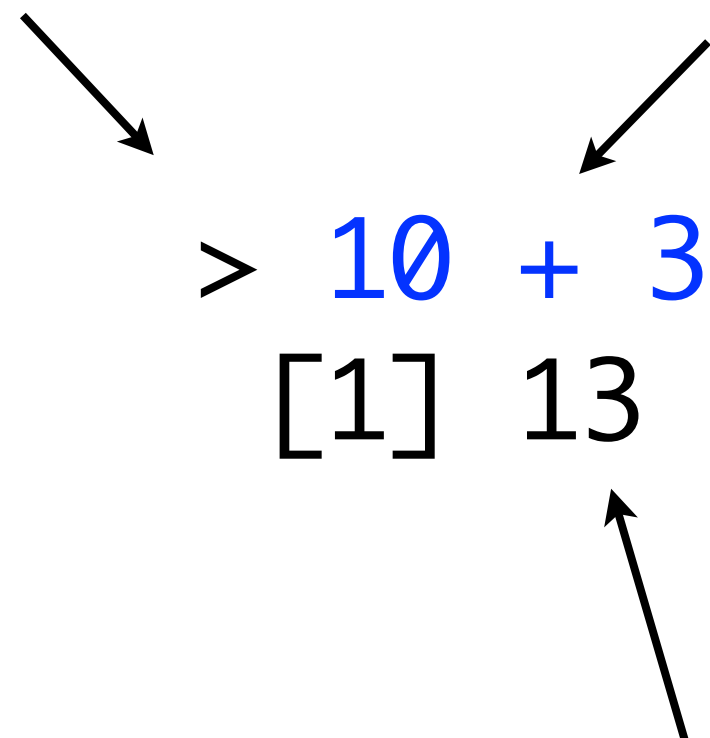


You type **commands** here, at the
“command prompt”

Our first command...

The > is the command prompt

This is a command



```
> 10 + 3  
[1] 13
```

The number 13 is the output
(don't worry about the [1] for now)

Simple calculations

+	addition
-	subtraction
*	multiplication
/	division
^	taking powers

← These are referred to as “**operators**”
(each operator is used to carry out a particular kind of operation)

```
> (6 - 4) / 2  
[1] 1
```

```
> 6 - (4/2)  
[1] 4
```

← When performing multiple calculations, use parentheses to make sure R does the calculations in the desired order

(Note: without parentheses, the order is: ^ first, then * and / second (left to right), and then + and - last (left to right). No-one remembers this at first.)

Logical statements

`==` equality
`!=` inequality
`>` greater than
`>=` greater than or equal to
`<` less than
`<=` less than or equal to

`&` AND
`|` OR
`!` NOT

```
> 10 < 100  
[1] TRUE
```

```
> 2 + 2 == 5  
[1] FALSE
```

```
> (10 < 100) | (2 + 2 == 5)  
[1] TRUE
```

```
> (10 < 100) & (2 + 2 == 5)  
[1] FALSE
```

Exercises

1. One mile is 1.61 kilometers. Use R to figure out how many kilometers 5 miles are.
2. Use R to calculate $(8+4)*12$. How is that different from $8+4*12$?
3. TRUE or FALSE? eight is less than six
4. TRUE OR FALSE? 5^2 is equal to 81?
5. TRUE OR FALSE? (4 is greater than or equal to 22) AND (1 is less than -1 times -1)



Using functions

Functions

There are not enough symbols on the keyboard to do everything you might want to do... so there are only a few operators

Most things are **functions**

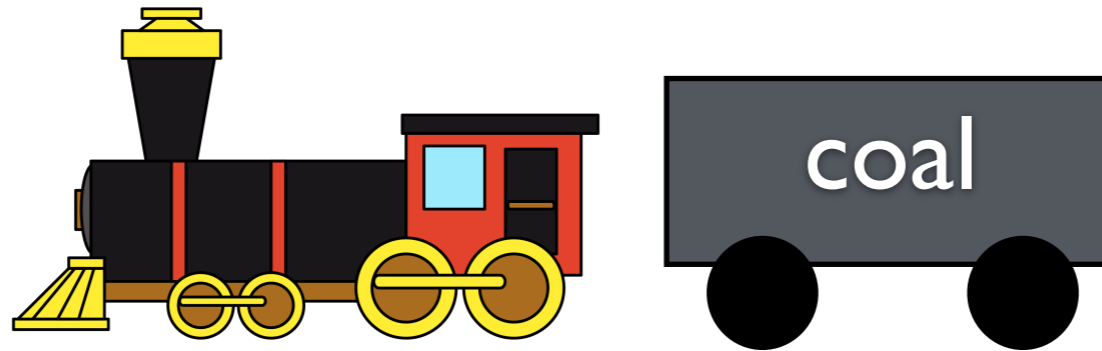
Example: square root

```
> sqrt(25)
[1] 5
```

- The function is called `sqrt()`
- The 25 is the “**argument**” to the function.

Arguments

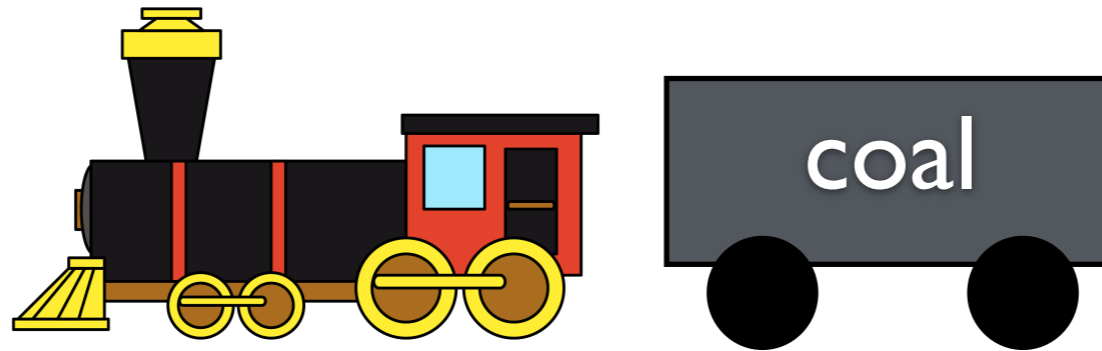
Every function has arguments. You can think of functions like empty trains at the depot. Each function has a certain number of carriages, and each carriage requires certain things.



This train cannot leave the depot if it doesn't have its carriage filled with the right thing (coal).

Arguments

Every function has arguments. You can think of functions like empty trains at the depot. Each function has a certain number of carriages, and each carriage requires certain things.



`sqrt`

`(25)`

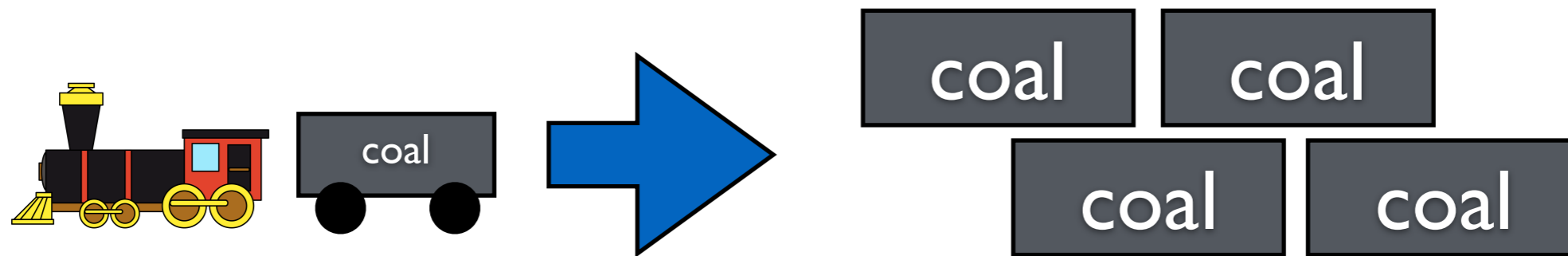
This function cannot work if it doesn't have its argument filled with the right thing (a number).

```
> sqrt()
```

```
Error in sqrt() : 0 arguments passed to 'sqrt' which requires 1
```

Arguments

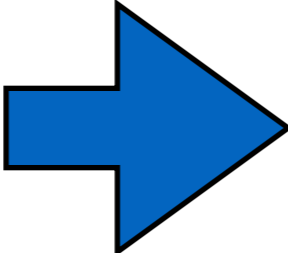
Every function has arguments. You can think of functions like empty trains at the depot. Each function has a certain number of carriages, and each carriage requires certain things.



When a train arrives successfully, you can “convert” it to money (which buys more coal).

Arguments

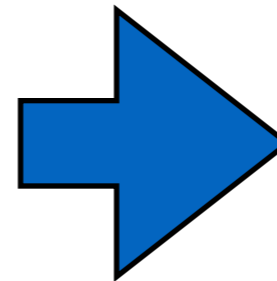
Every function has arguments. You can think of functions like empty trains at the depot. Each function has a certain number of carriages, and each carriage requires certain things.

`sqrt (25)`  5

When it runs successfully, it “converts” into an answer

Some functions that you might see on a scientific calculator

<code>sqrt()</code>	- Square root
<code>round()</code>	- Round a number
<code>log()</code>	- Logarithm
<code>exp()</code>	- Exponentiation
<code>abs()</code>	- Absolute value



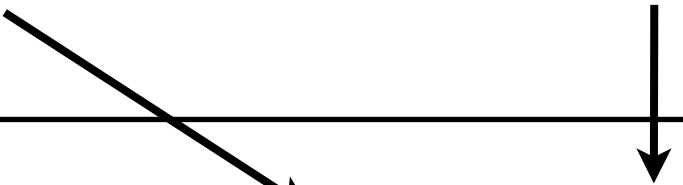
```
> sqrt(4)
[1] 2
> round(3.24)
[1] 3
> log(32)
[1] 3.465736
> exp(2)
[1] 7.389056
> abs(-3)
[1] 3
```

Functions with multiple arguments

- Many functions can “take” more than one argument;
- Separate the arguments with commas.

Argument #1: The number that needs to be rounded

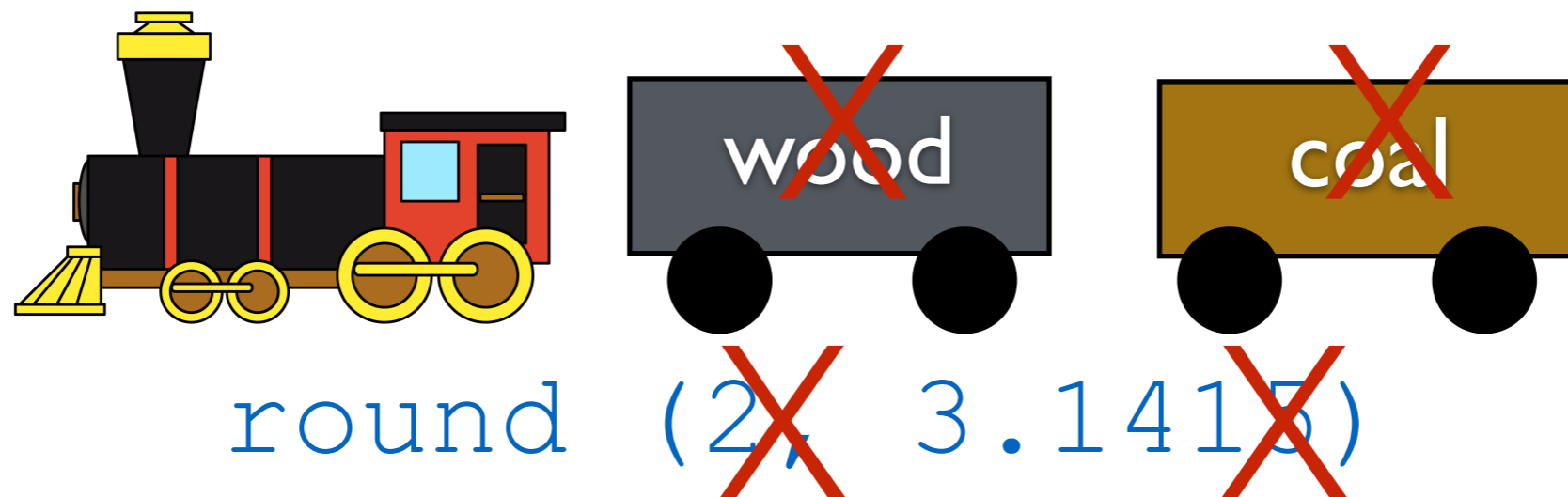
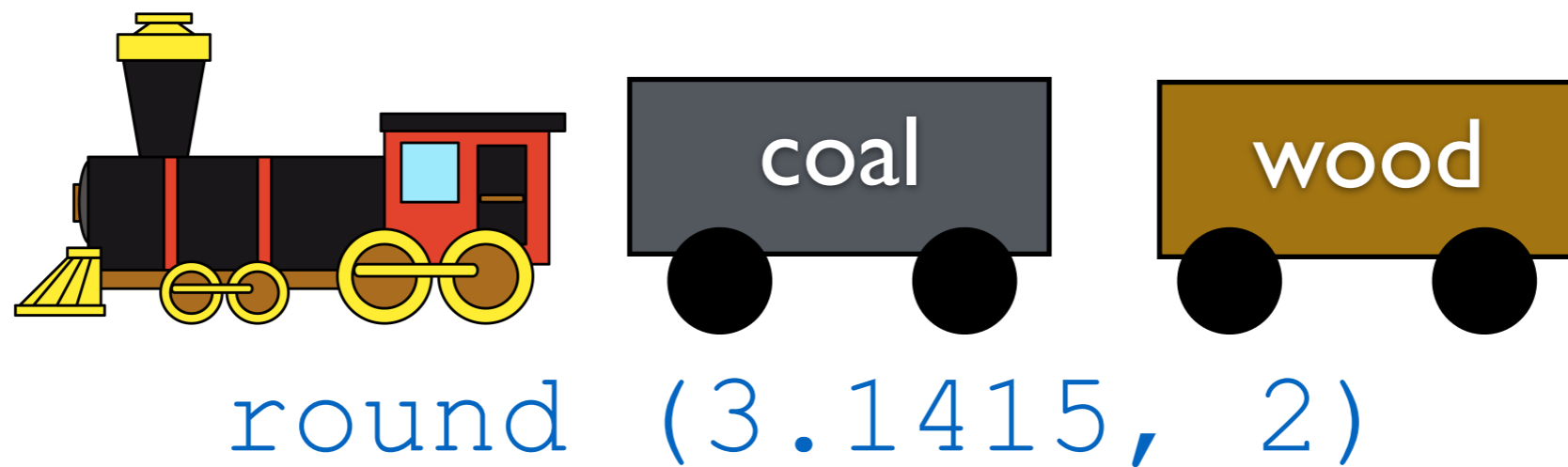
Argument #2: How many digits to round it to?



```
> round(3.14159,2)  
[1] 3.14
```

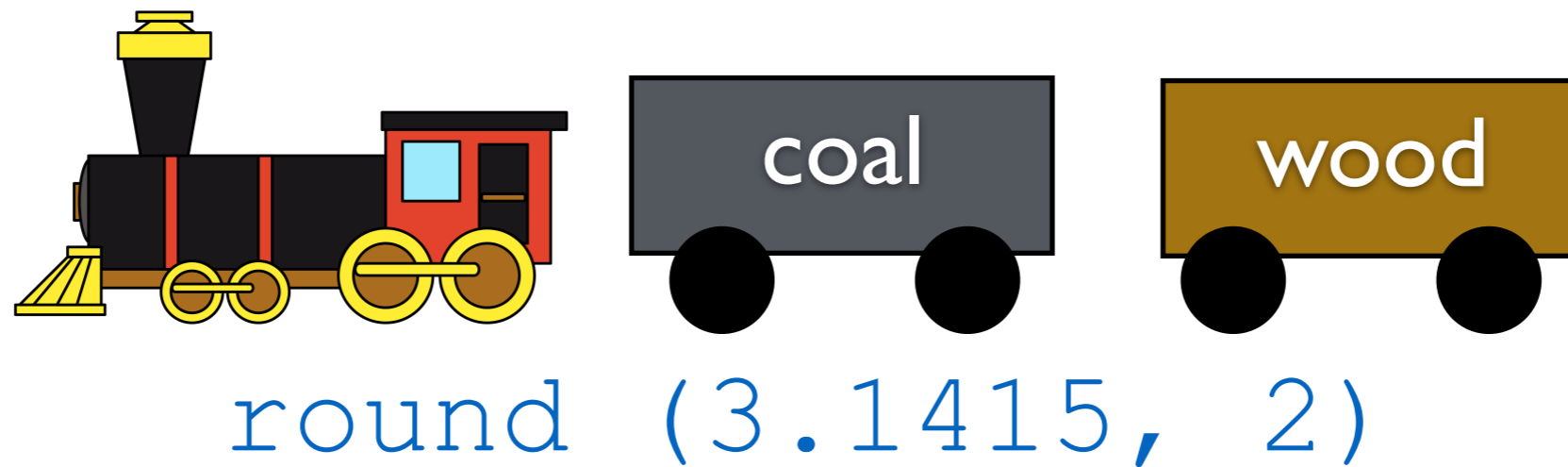
Arguments

This is like a train with multiple carriages. Each carriage can only take the thing it is designed to take (order matters).



Arguments

This is like a train with multiple carriages. Each carriage can only take the thing it is designed to take.



Note that *some* arguments are required, and some are not!

```
> round(3.14159)
```

```
[1] 3
```

```
> round()
```

```
Error: 0 arguments passed to 'round' which requires 1 or 2 arguments
```

Some arguments have defaults

- A lot of arguments have “**default values**”.
- If you don't tell R what value to use, it uses the default

```
> round( x = 3.1415 )  
[1] 3
```


The default number of digits to round to is zero, so that's what R uses here

Arguments have names

- Most of the time, the arguments have “**names**”,
- You can use the names when typing commands

The number that needs to be rounded is called **x**

The number of digits to round to is called **digits**



```
> round( x = 3.1415, digits = 2 )  
[1] 3.14
```

Arguments have names

If you specify the names, the order doesn't matter. But if you don't, it does!

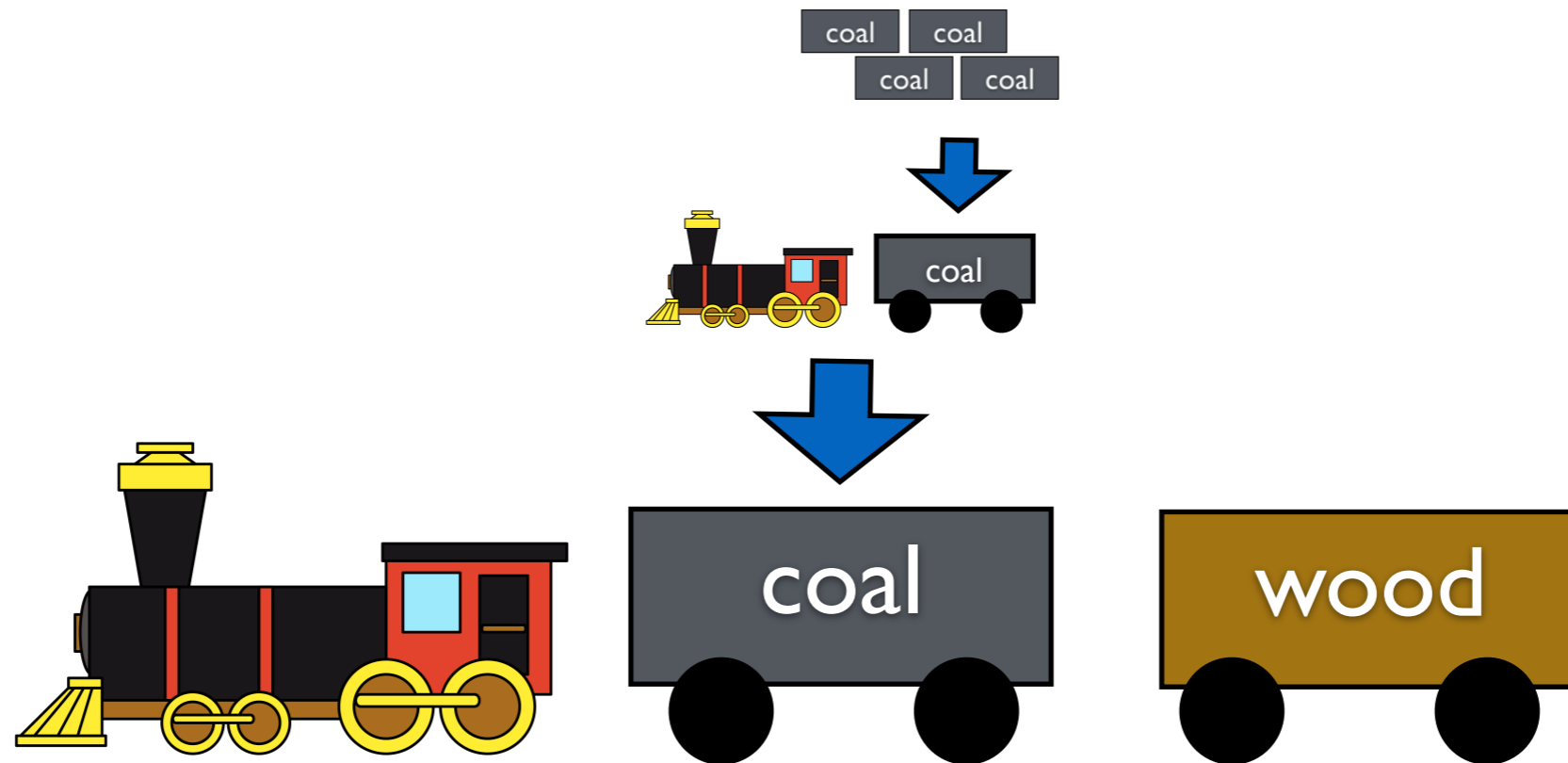
```
> round (x=3.14159, digits=2)
[1] 3.14
> round (digits=2, x=3.14159)
[1] 3.14
```

```
> round(3.14159,2)
[1] 3.14
> round(2,3.14159)
[1] 2
```

This is known as a “silent fail.” What we input didn't make sense, so it just went with the default, with no warning. Be careful of these!

“Nesting” functions

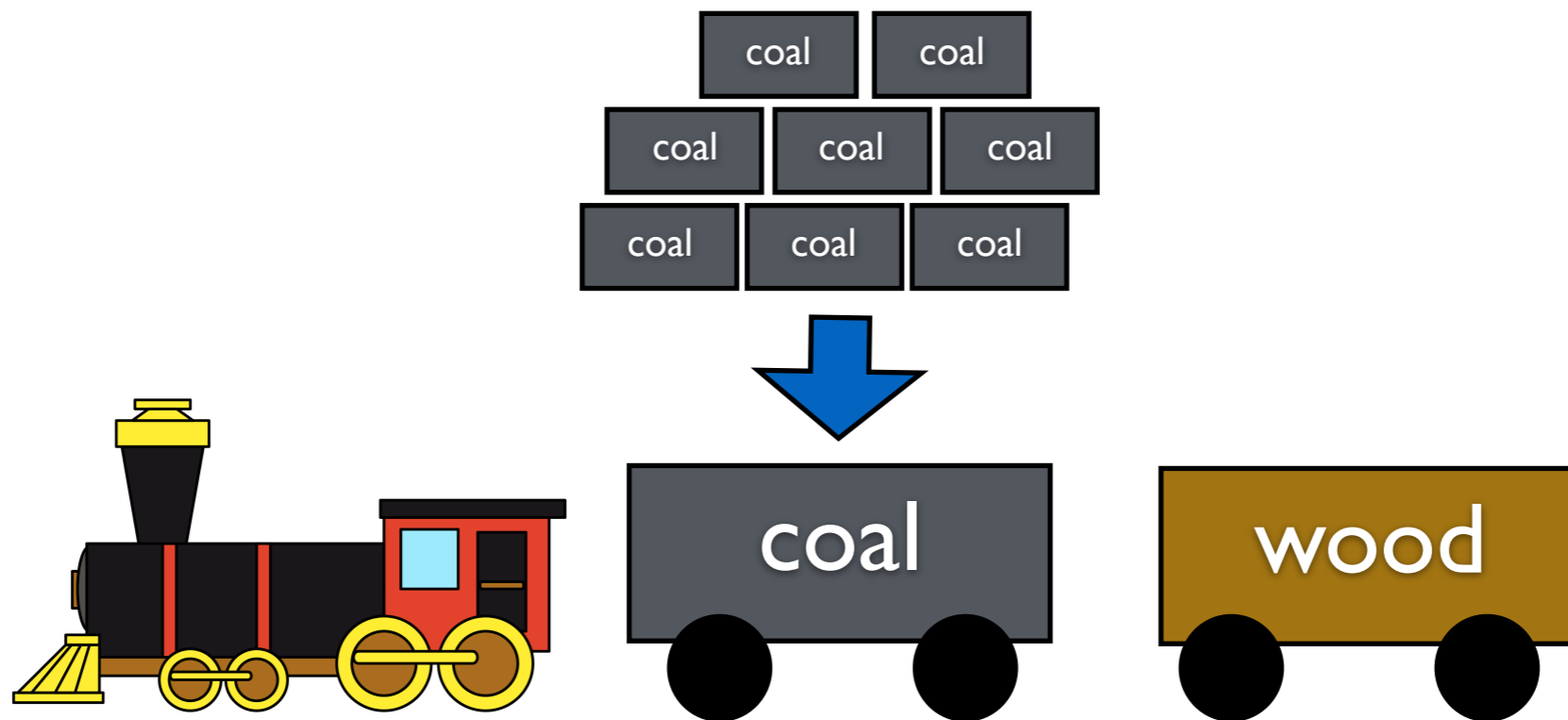
Functions can take other functions as arguments!



This is where the analogy sort of breaks down, but remember that you can effectively convert “trains” to “coal...”

“Nesting” functions

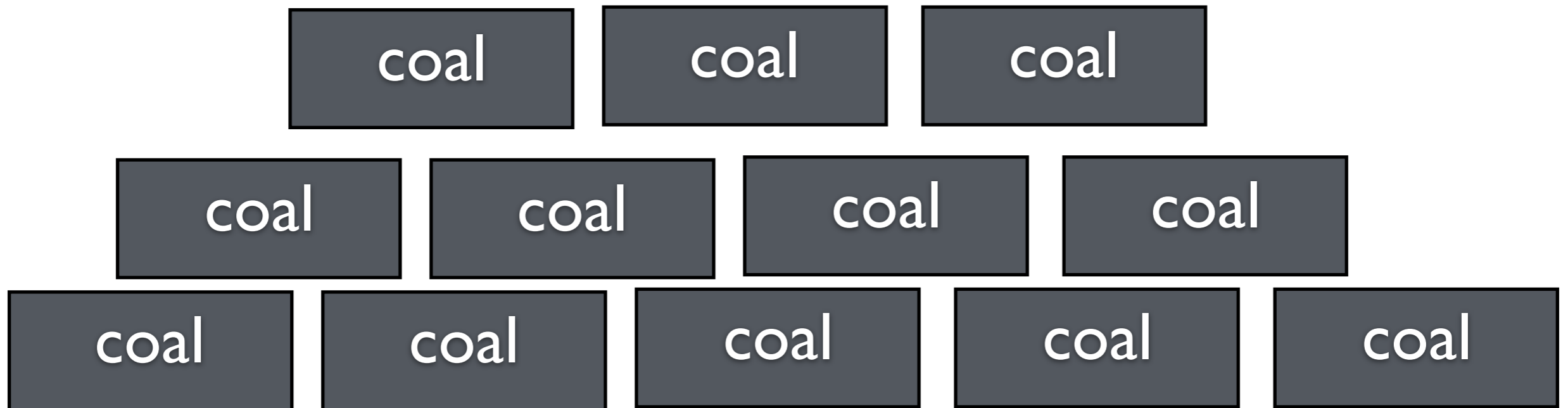
Functions can take other functions as arguments!



This is where the analogy sort of breaks down, but remember that you can effectively convert “trains” to “coal...”

“Nesting” functions

Functions can take other functions as arguments!

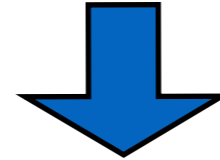


This is where the analogy sort of breaks down, but remember that you can effectively convert “trains” to “coal...”

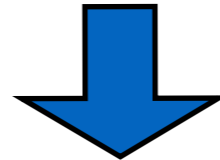
“Nesting” functions

Functions can take other functions as arguments!

```
sqrt ( round(4.45) )
```



```
sqrt ( 4 )
```



```
2
```

In the same way, the inside functions “convert” to their answers. So to evaluate it, you work from the inside out.

“Nesting” functions

Functions can take other functions as arguments!

```
sqrt ( round(4.45) )
```

Note that the parentheses are balanced. If they aren't, this can cause a problem because R won't know what goes inside what.

```
> sqrt(round4.33))  
Error: unexpected ')' in "sqrt(round4.33))"
```

```
> sqrt(round(4.33))  
+ |
```

Exercises

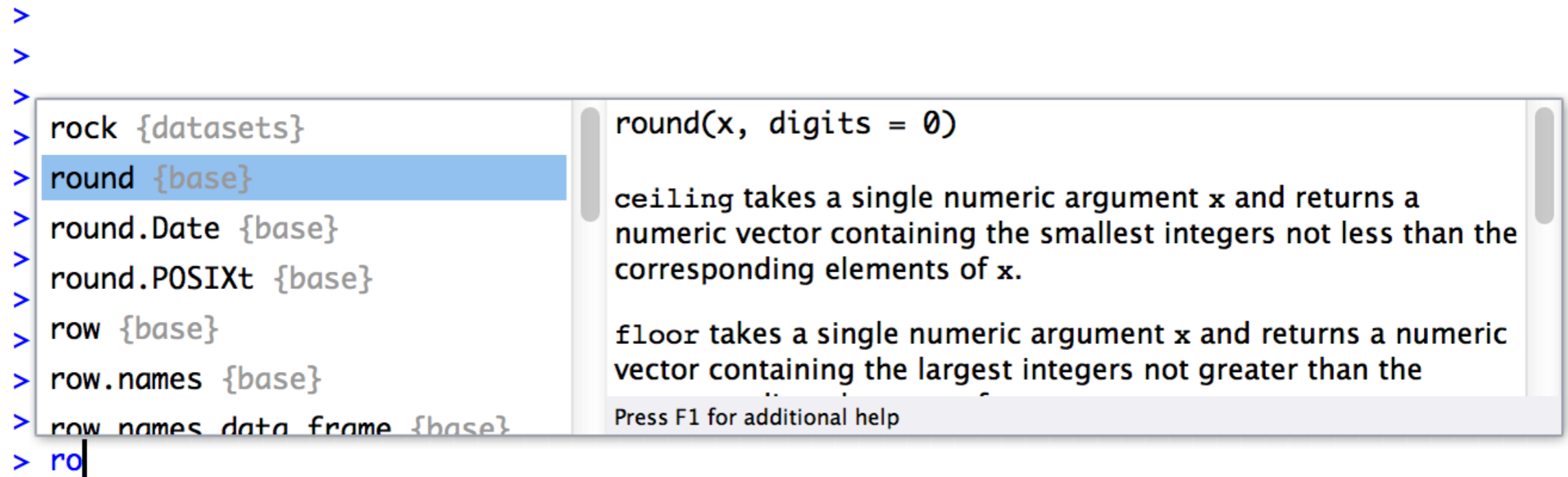
1. Calculate the square root of 81.
2. What happens if you try to take the square root of -81? Use the `abs()` function to make it non-negative first.
3. What is the exponent of the log of 2?
4. Round 4328.29874 to two digits after the decimal place. How can you make it round it to 4000?
5. See if you can figure out how the `floor()` function is different from `round()`.



Navigation hints

“Tab” autocomplete

```
>
>
>
> rock {datasets}
> round {base}
> round.Date {base}
> round.POSIXt {base}
> row {base}
> row.names {base}
> row.names.data.frame {base}
> ro|
```

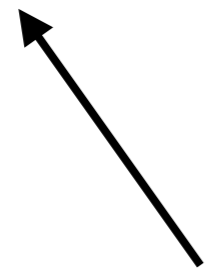
A screenshot of an R console window. The prompt is '> ro|'. A dropdown menu is open, listing several R functions: 'rock {datasets}', 'round {base}', 'round.Date {base}', 'round.POSIXt {base}', 'row {base}', 'row.names {base}', and 'row.names.data.frame {base}'. The 'round {base}' option is highlighted in blue. To the right of the dropdown, a help window for the 'round' function is visible. It shows the function signature 'round(x, digits = 0)' and two paragraphs of text: 'ceiling takes a single numeric argument x and returns a numeric vector containing the smallest integers not less than the corresponding elements of x.' and 'floor takes a single numeric argument x and returns a numeric vector containing the largest integers not greater than the'. At the bottom of the help window, it says 'Press F1 for additional help'.

round(x, digits = 0)

ceiling takes a single numeric argument `x` and returns a numeric vector containing the smallest integers not less than the corresponding elements of `x`.

floor takes a single numeric argument `x` and returns a numeric vector containing the largest integers not greater than the

Press F1 for additional help



Type `ro` and then hit tab.

Brings up a window showing possible commands you might like to use

The up arrow

```
> 3+4+5  
[1] 12  
> 3*4*5  
[1] 60  
> sqrt(4)  
[1] 2  
|
```

If you type the up arrow it will let you go through all your previous commands in reverse order.

(This would first show `sqrt(4)`, then `3*4*5`, then `3+4+5`)



Variables

This is a box.



Inside the box is a cat.



The box is storage.
It could store many things.
It is a “**variable**”

The cat is the thing stored
The thing stored in a
variable is its “**value**”

Variable

Value

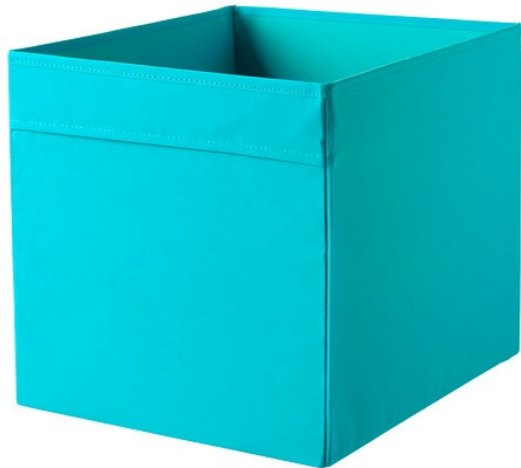


```
box <- "cat"
```

The variable **box** “gets”
the value **"cat"**



There are many different colours of boxes



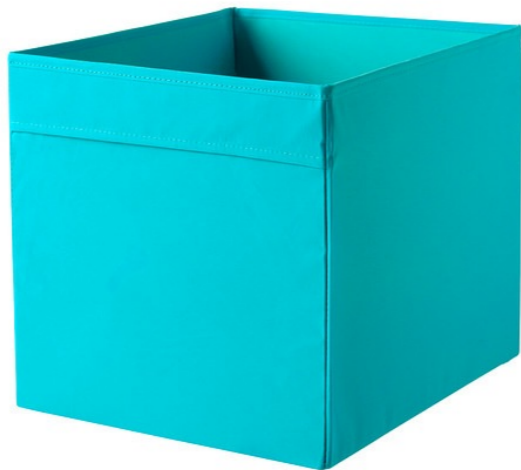
There are many different **classes** of variable





“**numeric**” variables store numbers

```
blackBox <- 212
```



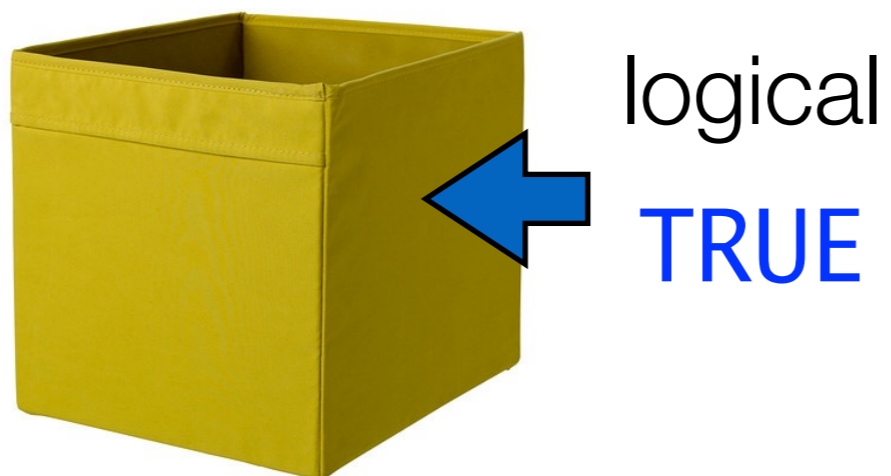
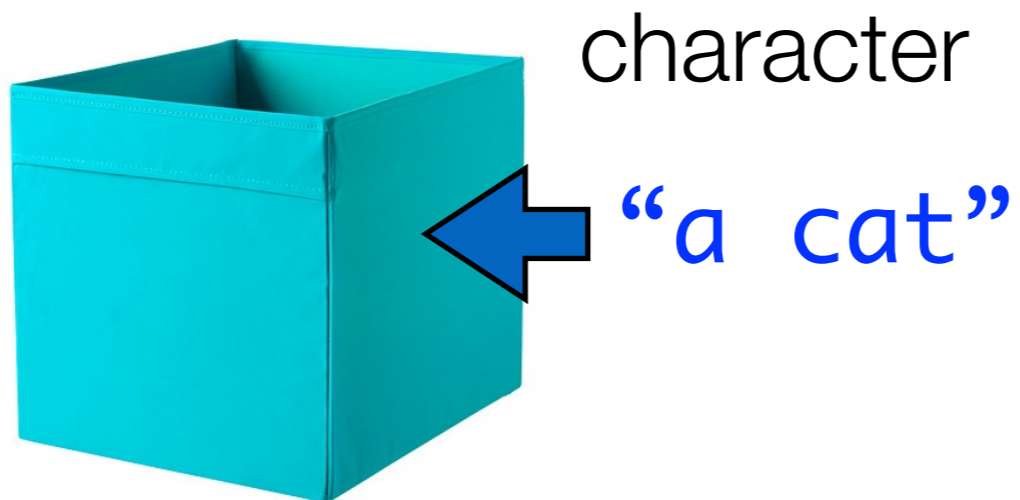
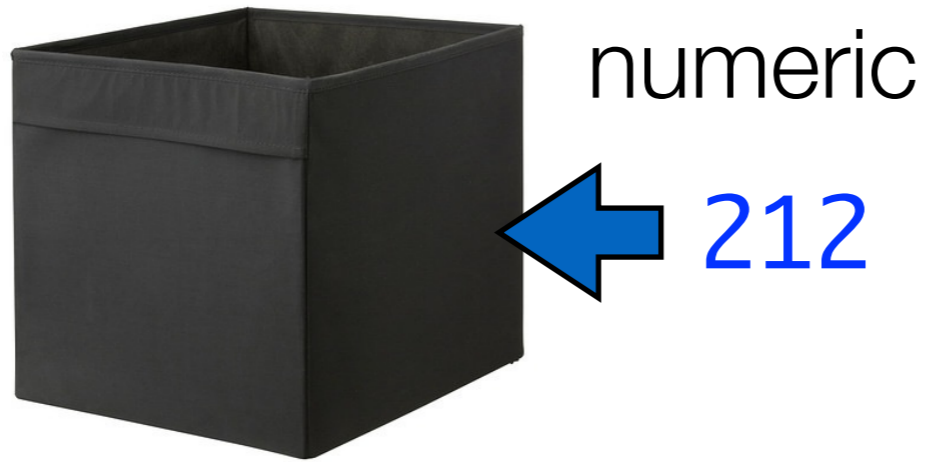
“**character**” variables store text

```
blueBox <- “a cat”
```



“**logical**” variables store true/false

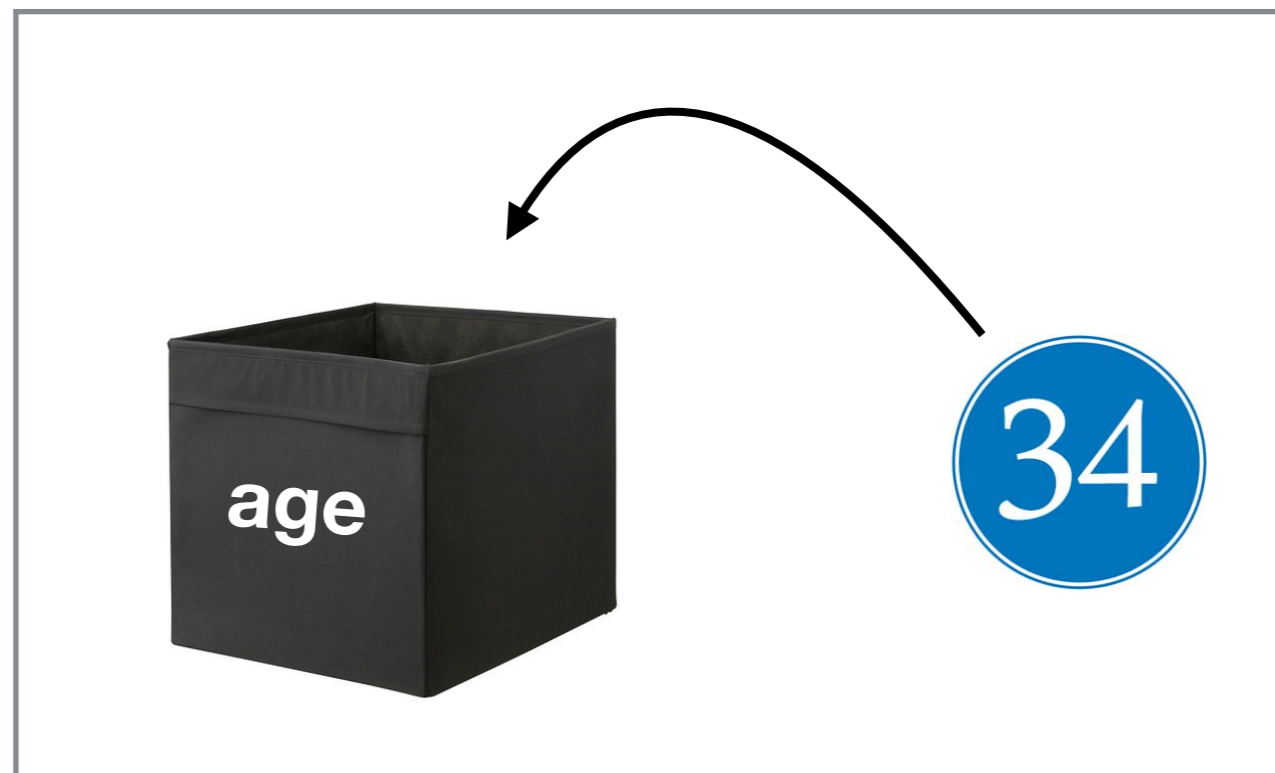
```
yellowBox <- TRUE
```

Creating variables

- Variables are used to store information
- They provide a way of labelling information
- They refer to the contents of a block of computer memory
- Use the “assignment operator” `<-` to create one

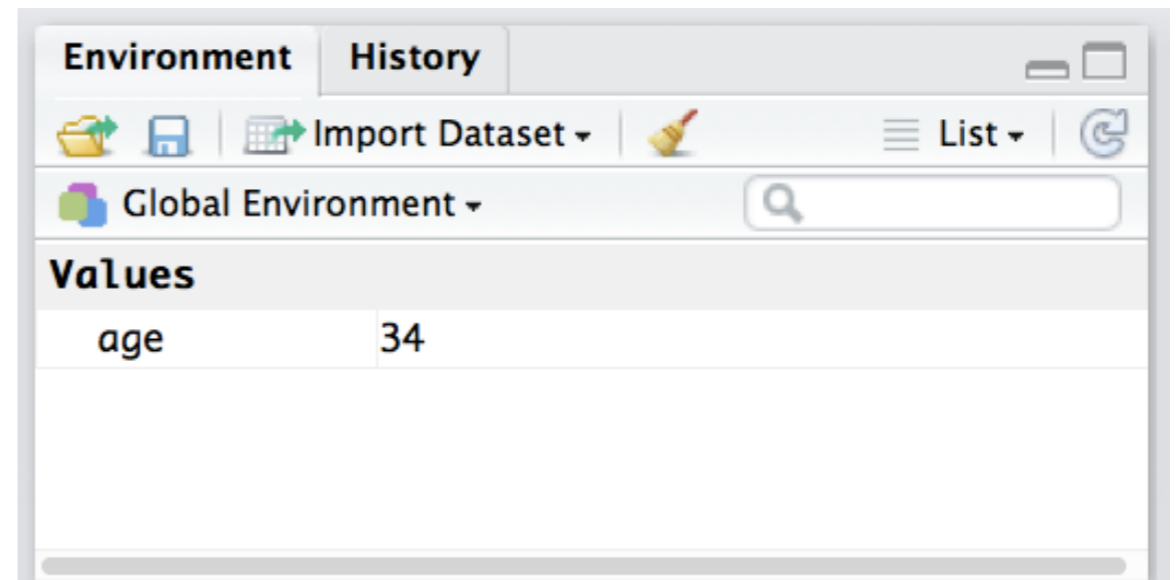
```
> age <- 34
```



Creating variables

- Variables are used to store information
- They provide a way of labelling information
- They refer to the contents of a block of computer memory
- Use the “assignment operator” `<-` to create one

```
> age <- 34
```



No output appears in the console, but the variable shows up in the Rstudio “environment” panel

Working with variables

- Variables in R behave exactly the same way as their values do
- `34 * 2` is meaningful, `"yellow" * 2` is not
- So...

```
> 34 * 2  
[1] 68
```

```
> "yellow" * 2  
Error in "yellow" * 2 : BLAH BLAH BLAH
```

```
> age <- 34  
> age * 2  
[1] 68
```

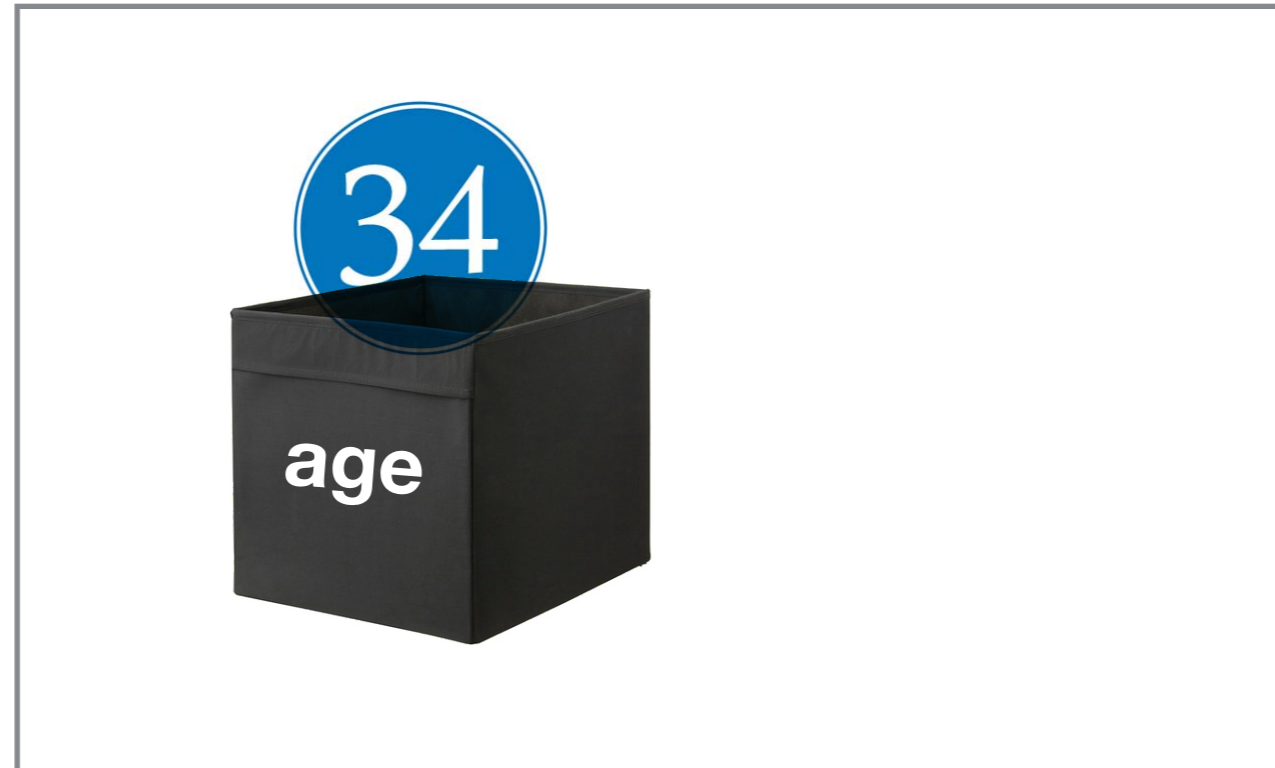
```
> myColour <- "yellow"  
> myColour * 2  
Error in myColour * 2 : BLAH BLAH BLAH
```

Using variables doesn't change the value

```
> # R ignores anything after the #  
> # this is used to make comments  
  
> # define variable  
> age <- 34  
> age  
[1] 34  
  
> # get R to print age+10  
> age + 10  
[1] 44  
  
> # age is still 34  
> age  
[1] 34
```

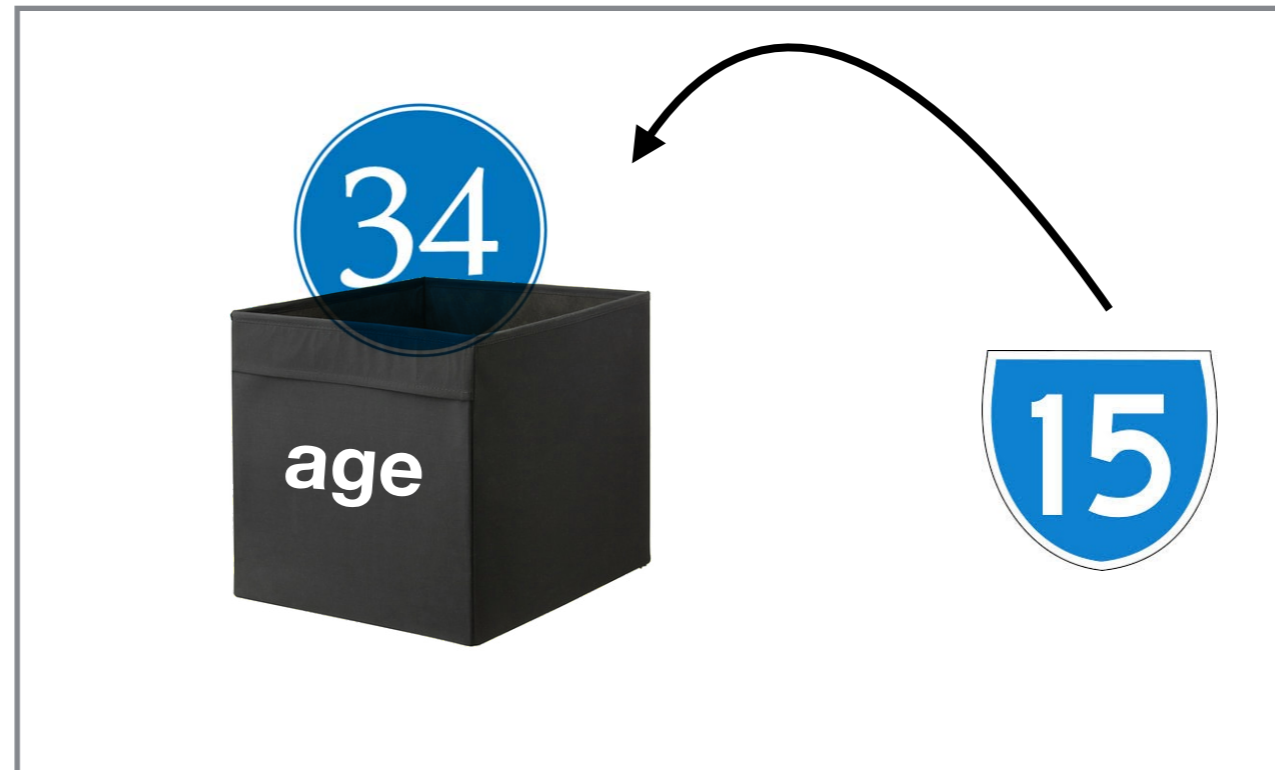


Reassigning values



The variable `age` currently stores a value of `34`

Reassigning values



Assigning a new value....

```
age <- 15
```

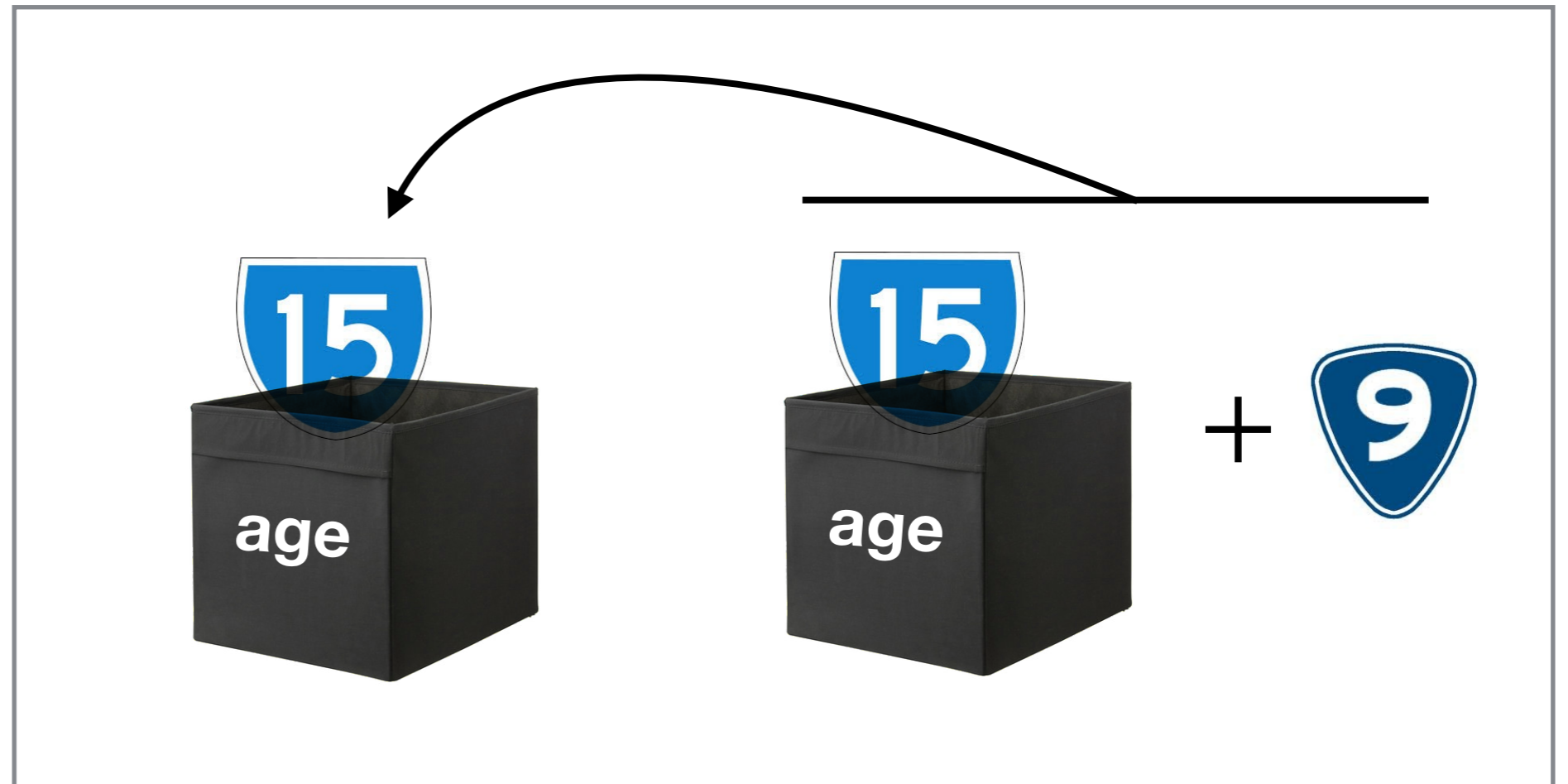
Reassigning values



... makes the old value vanish

```
age <- 15
```

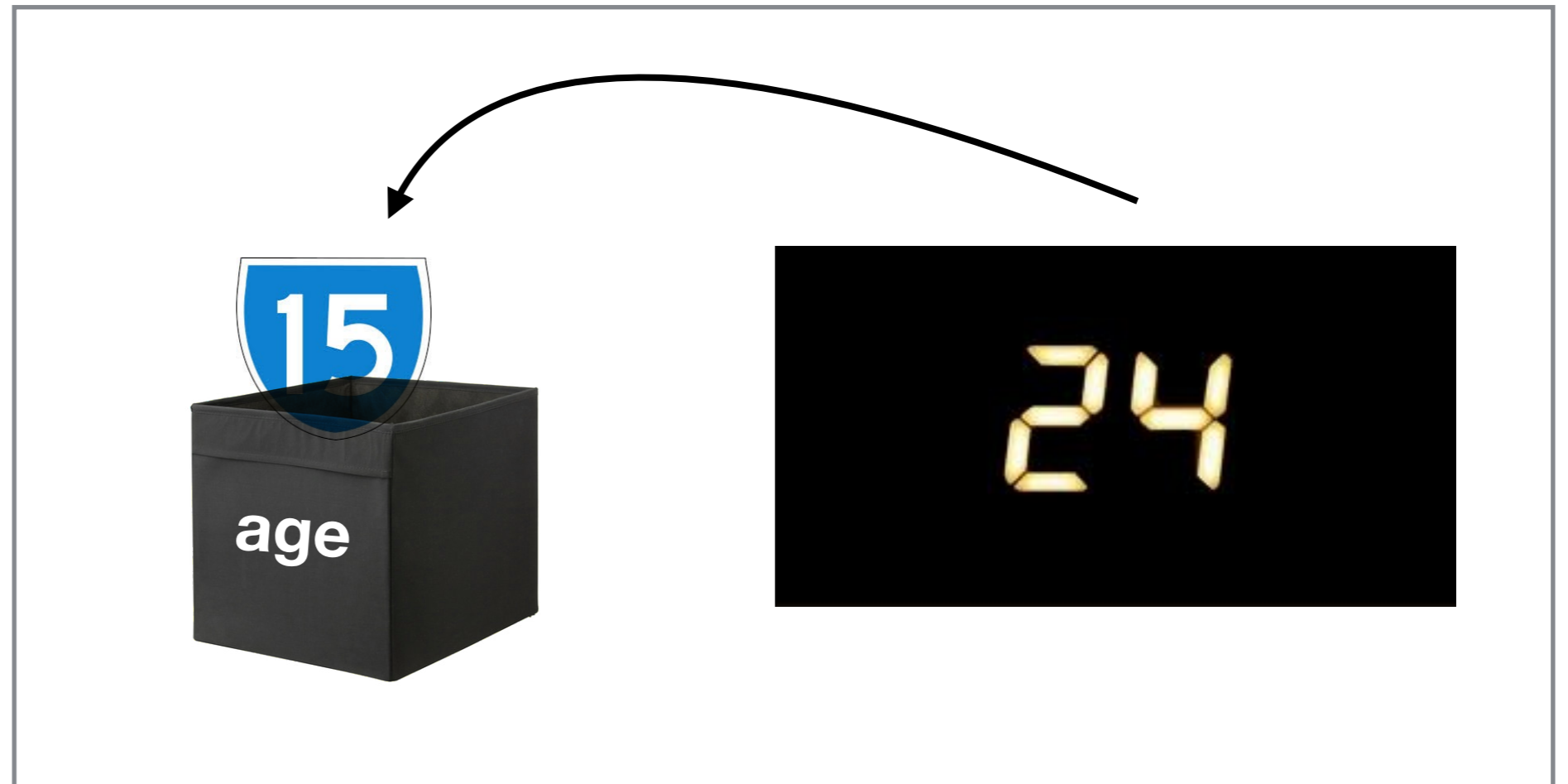

Reassigning values



You can assign a new value based on the old one...

```
age <- age + 9
```

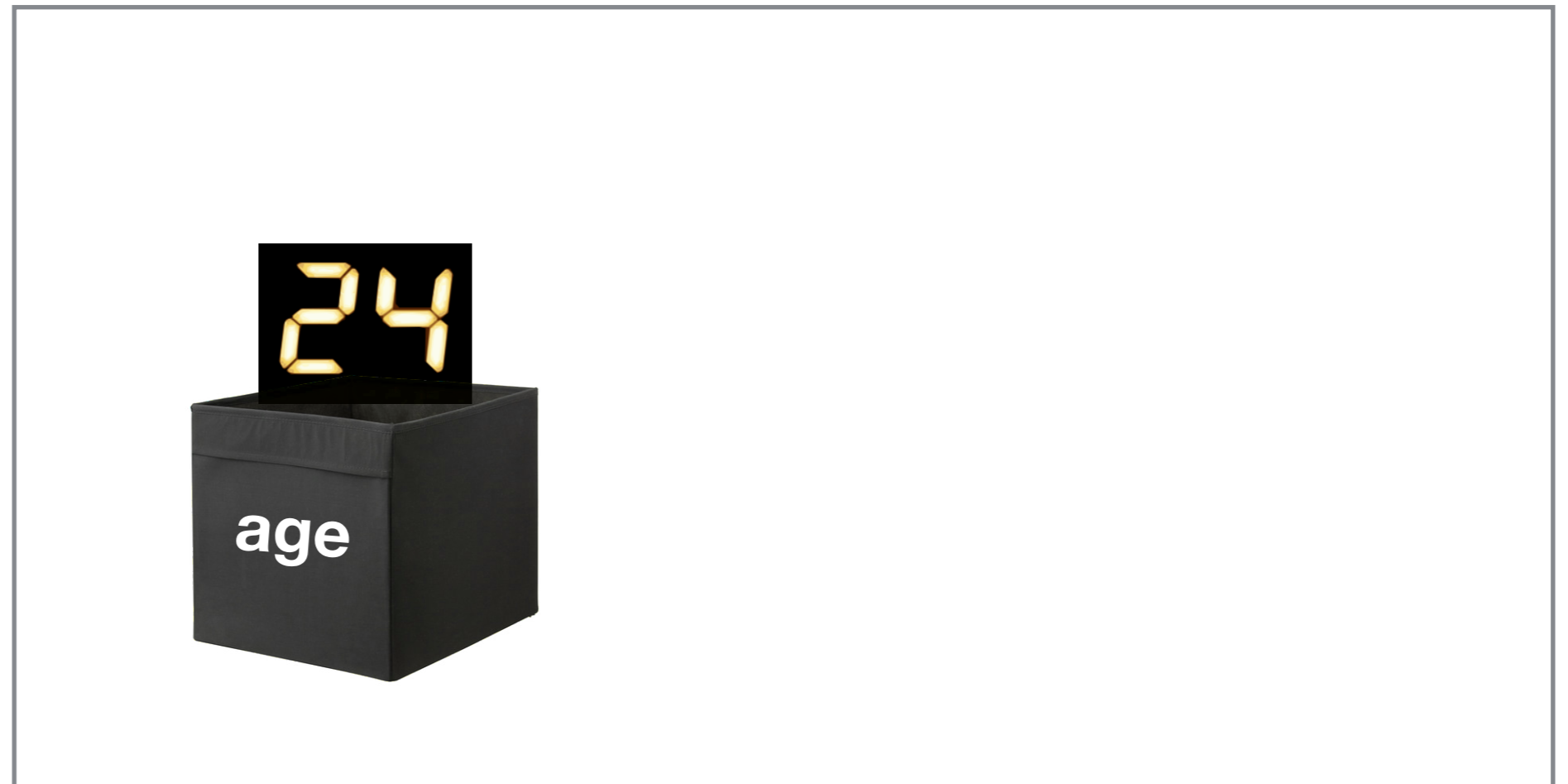
Reassigning values



You can assign a new value based on the old one...

```
age <- age + 9
```

Reassigning values

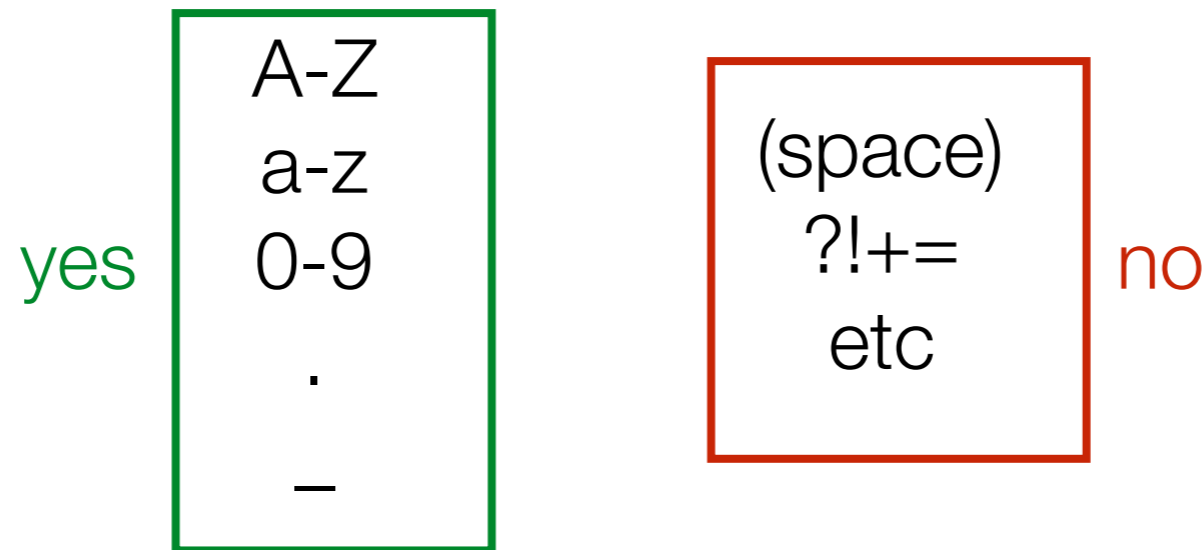


You can assign a new value based on the old one...

```
age <- age + 9
```

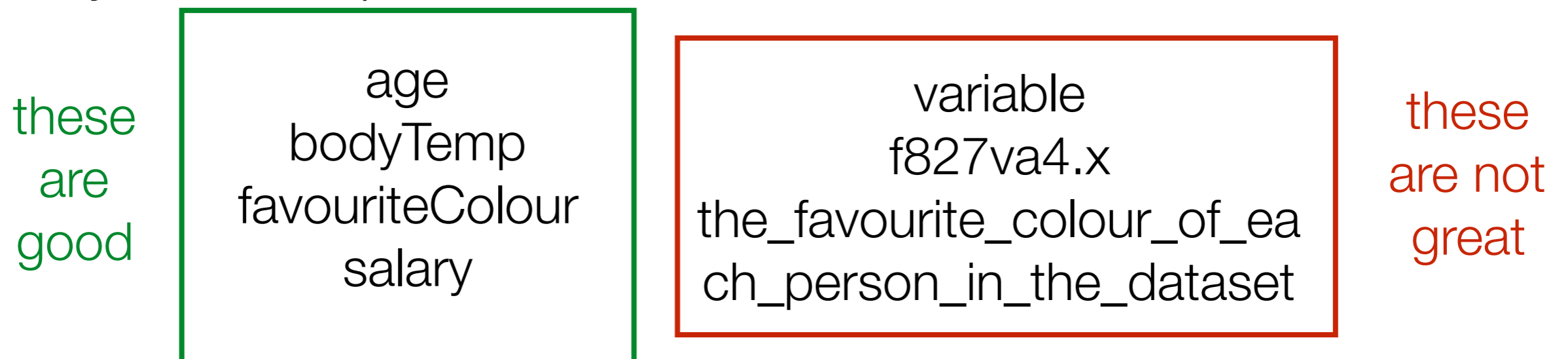
Note on variable names

You can name your variable most things, but not *anything*.



Must start with a letter or a period. Can't be a reserved keyword (like TRUE). Don't worry too much about this, R will yell if you do it.

Try to use simple, informative names that follow a convention.



Digression: What happens if you try quit R after creating some variables?

Hm. What's all this then?

```
> q()
```

```
Save workspace image to ~/.RData? [y/n/c]: |
```

Save “workspace image”? What's this about????

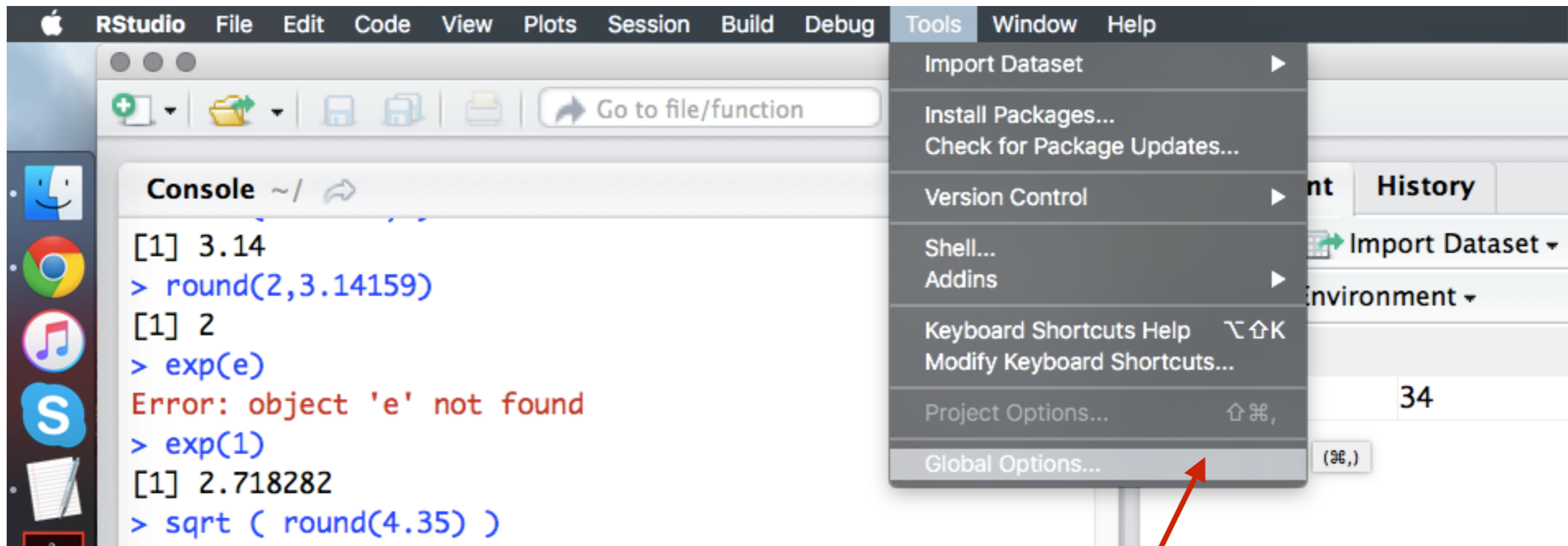
The “workspace”

- All the variables you currently have are called a workspace
- We'll talk about what this means later

- What R is asking is if you want to keep your variables for later
- It stores them in a “special” file.
- Right now, the answer is “no”.

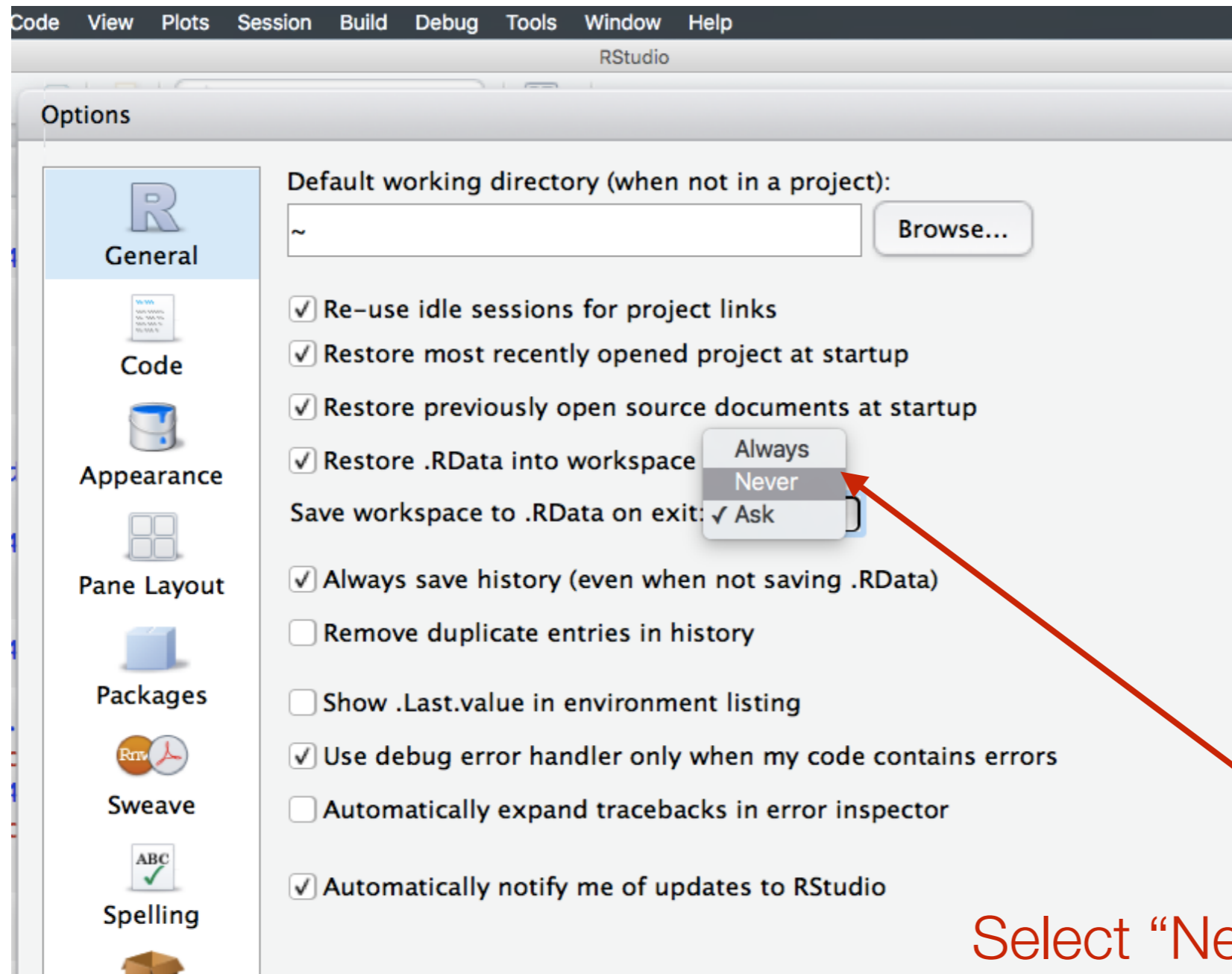
- In general, I think it's a bad idea to let R do this.
- Personally, I prefer to choose where my variables get stored
- My suggestion is that you tell R to stop whining about this...

The options menu



Choose “Global Options” from the “Tools” menu

The options panel



Select “Never” where it asks
“Save workspace to .RData on exit?”

(Then click “Apply” and then “OK” at the bottom)

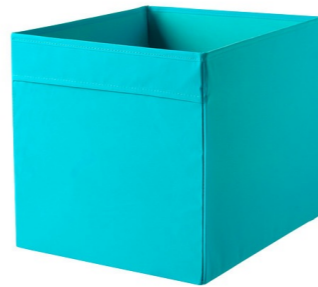
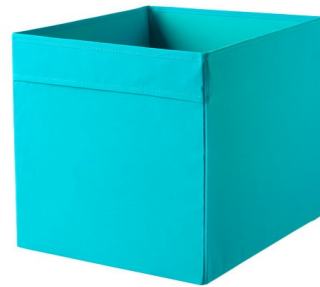
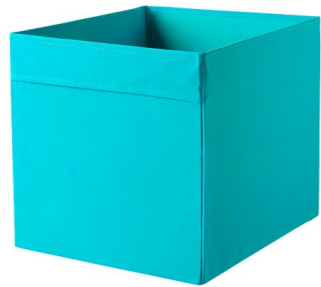
Exercises

1. Make a variable called `name` with your first name in it. Now make a variable called `name` with your last name. What has happened? How do you make one with your complete name?
2. Make a variable called `x` and set it equal to 2. Then set it equal to itself plus 2. What is it now?
3. Make a variable called `y` and set it to `TRUE`. Then add three to it. What happens? How is this different if you set it to `FALSE` instead? What do you think is going on?

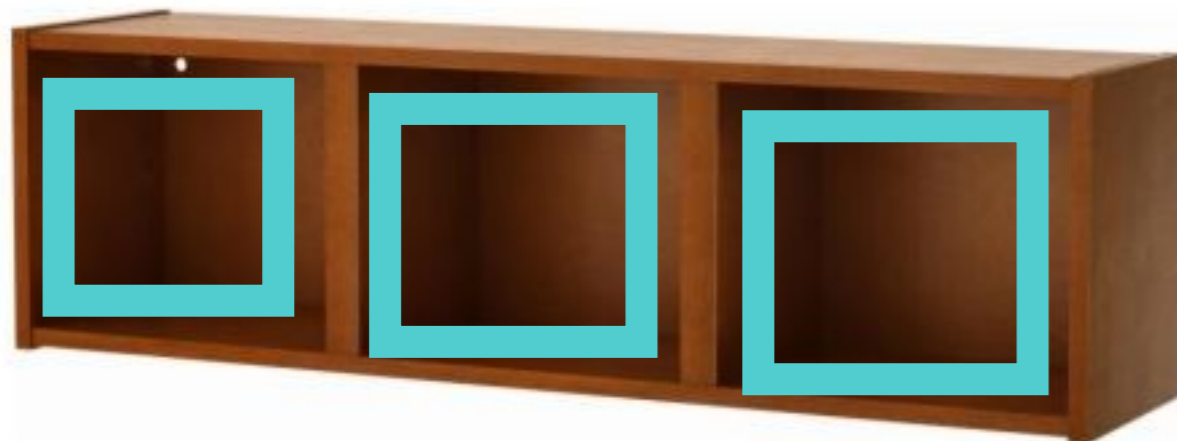
Storing multiple values
using variables



And each of those boxes
can store things



Each of those slots
is basically just a box



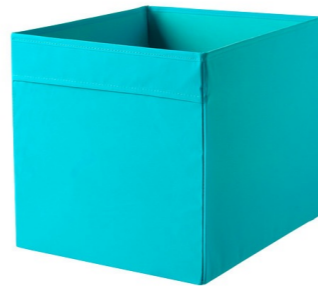
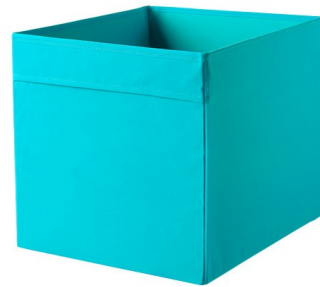
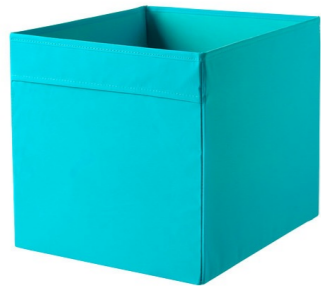
It's build from three
slots side by side



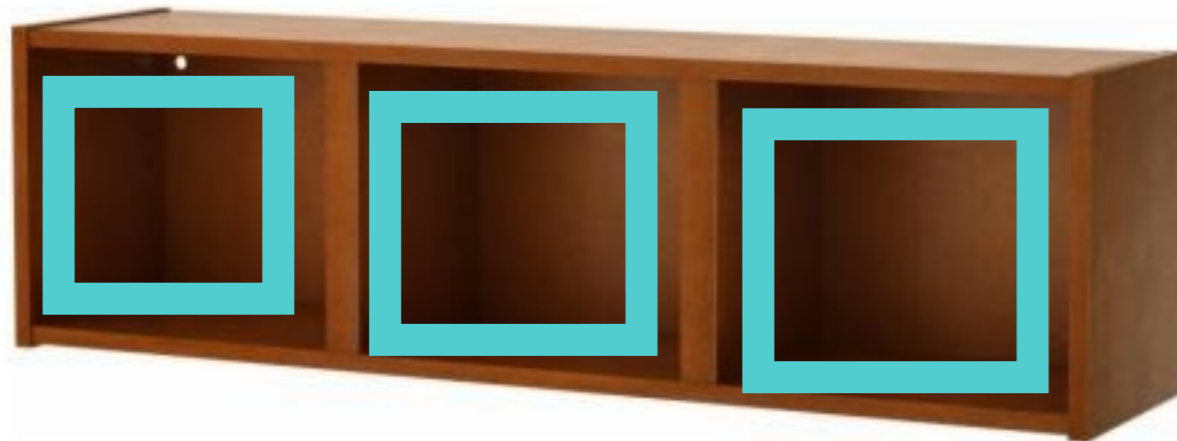
This is a big-box



And each of those elements can store **values**



Each of those elements is basically just a **variable**



It's build from three **elements** side by side



This is a **vector**

Vectors

- Vectors are variables that store multiple pieces of information
- Conceptually, a vector is just an ordered list of values...

A function called `c()`



`pets <- c("cat", "dog", "lizard")`

Creating vectors

`c()` combines a set of values, and stores them as a vector...

numeric vectors:

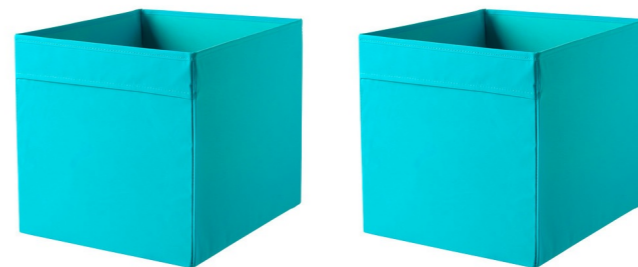
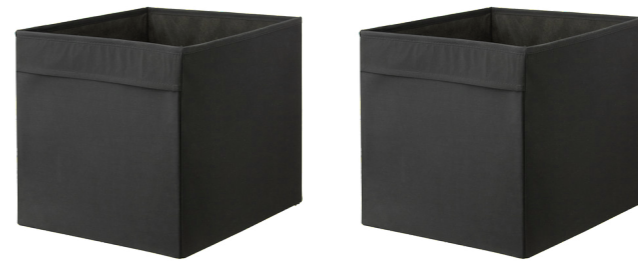
```
> age <- c( 34, 2 )  
> age  
[1] 34  2
```

character vectors:

```
> name <- c( "dan", "alex" )  
> name  
[1] "dan" "alex"
```

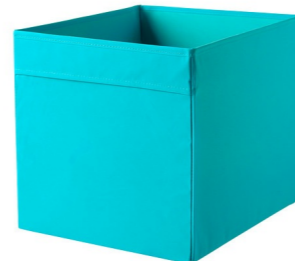
logical vectors:

```
> nerd <- c( TRUE, FALSE )  
> nerd  
[1] TRUE FALSE
```



Creating vectors

Note that all variables in a vector have to be of the same class. If they aren't, R will force them to be (another "silent fail").



```
> myVector <- c(TRUE,3,3.2)
> myVector
[1] 1.0 3.0 3.2
```

```
> myVector <- c("cat",3,TRUE)
> myVector
[1] "cat" "3" "TRUE"
```

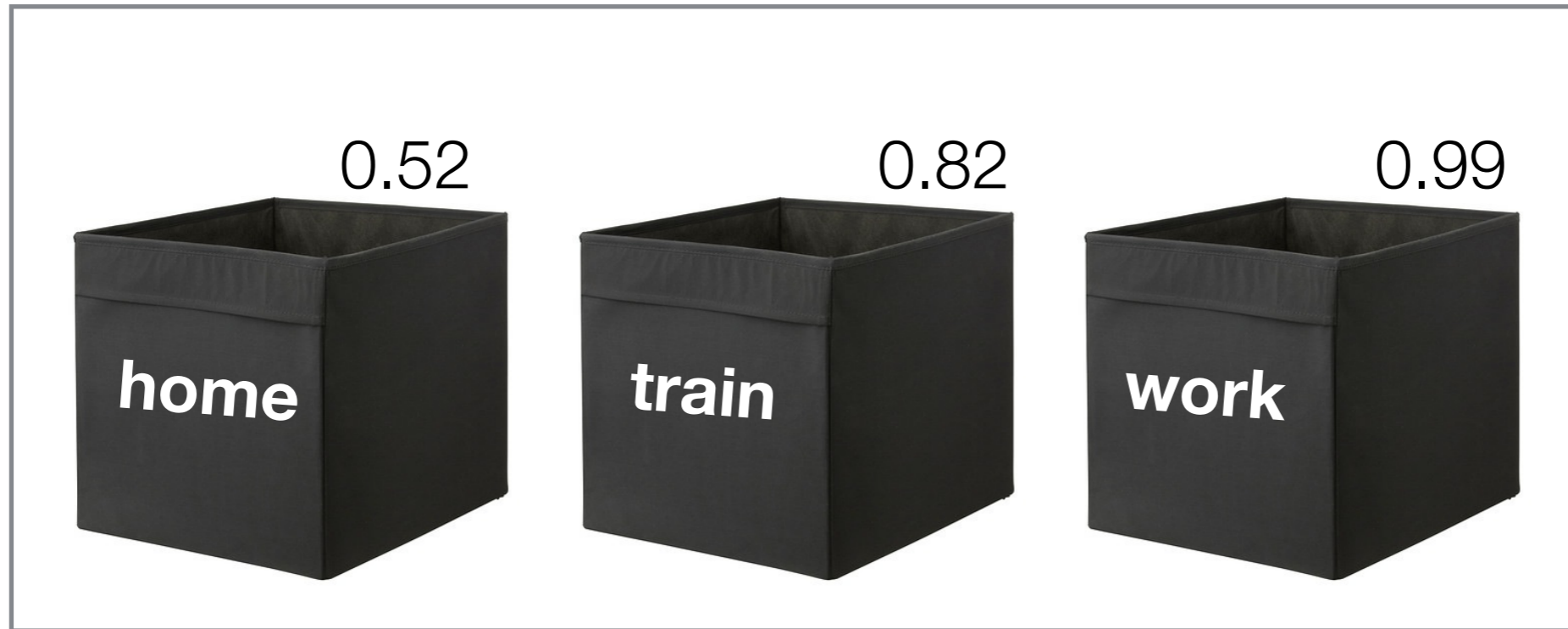

You can give **names** to the elements

boredom



```
> boredom <- c( home = 0.52, train = 0.82, work = 0.99 )  
> boredom  
home train work  
0.52  0.82  0.99
```

boredom



↑
1st element

`boredom[1]`

↑
2nd element

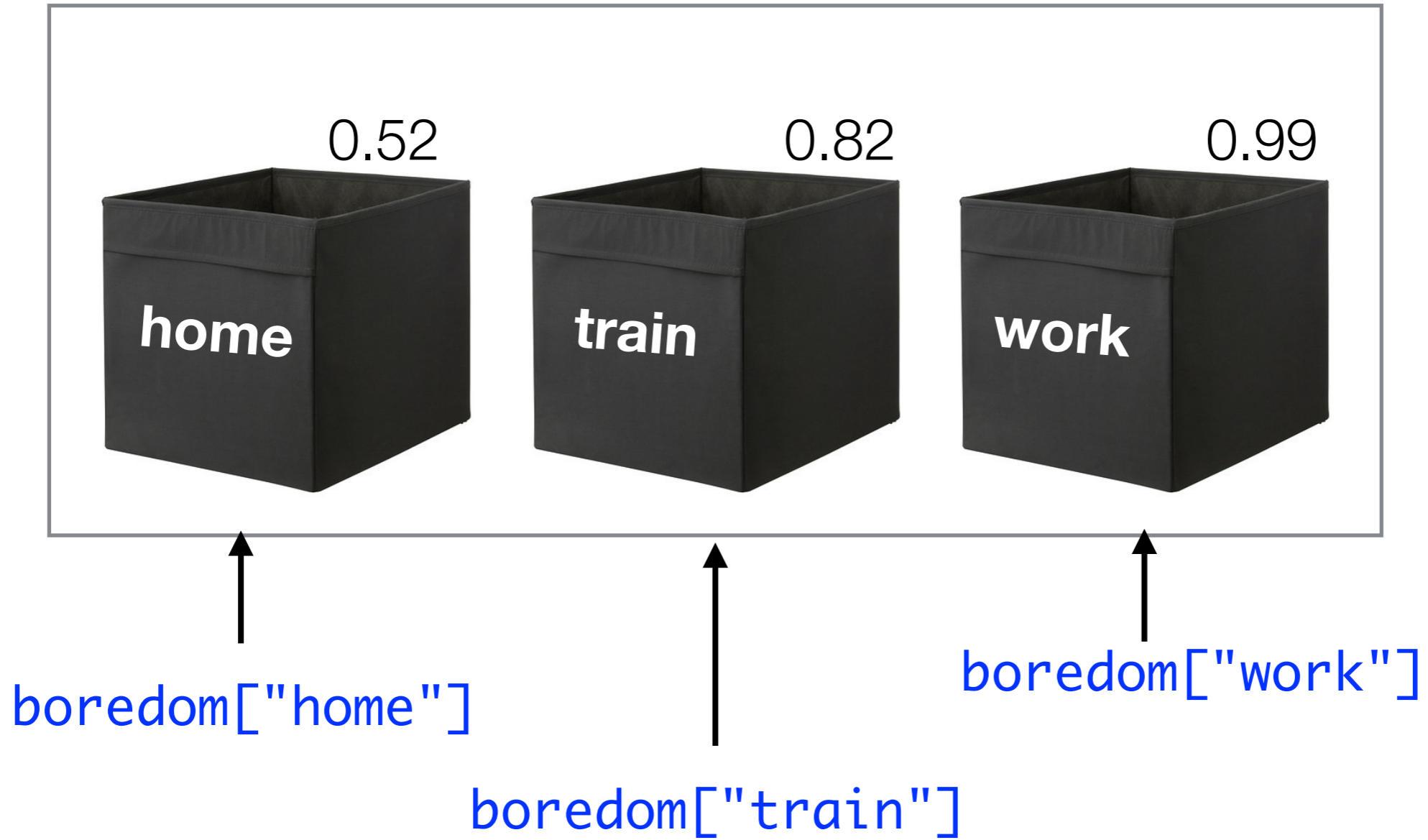
`boredom[2]`

↑
3rd element

`boredom[3]`

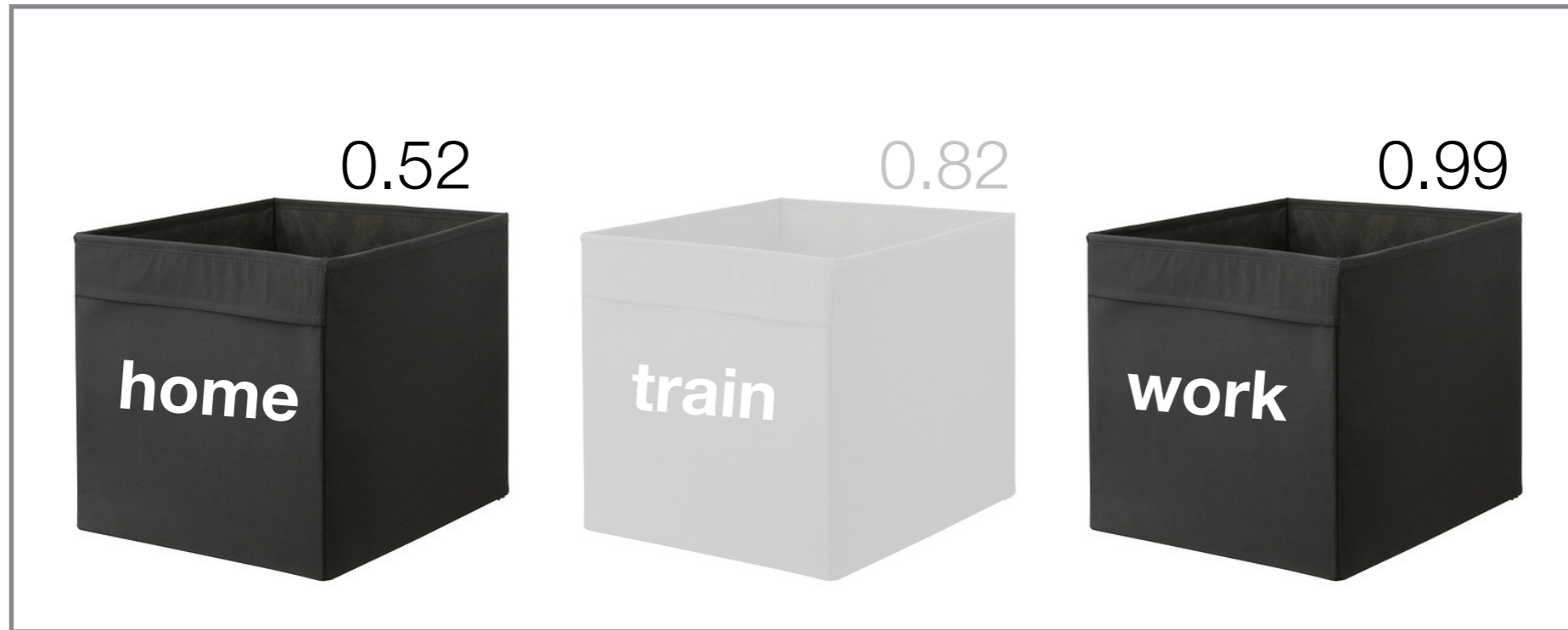
Selecting one element by **position**

boredom



Selecting one element by **name**

boredom

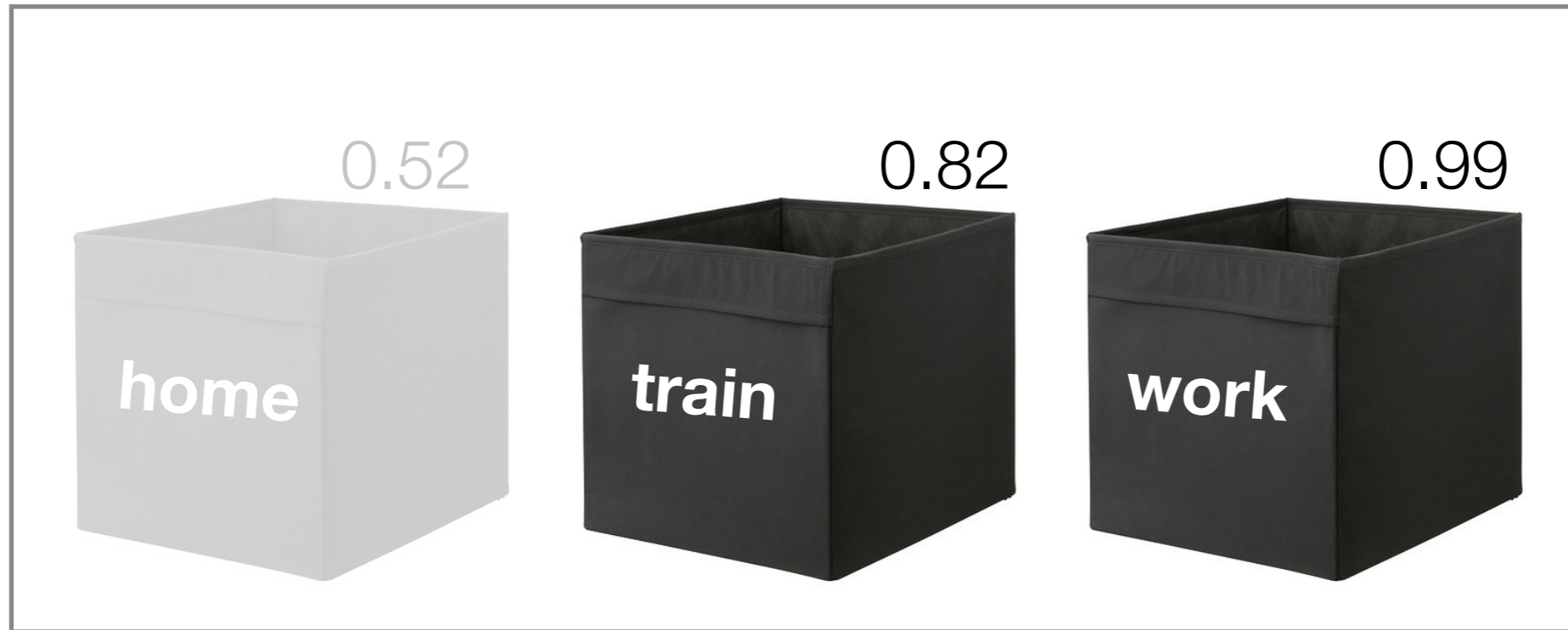


all the elements except the 2nd one

`boredom[-2]`

Dropping one element by position

boredom



all the elements except the 1st one

`boredom[-1]`

Dropping one element by position

boredom



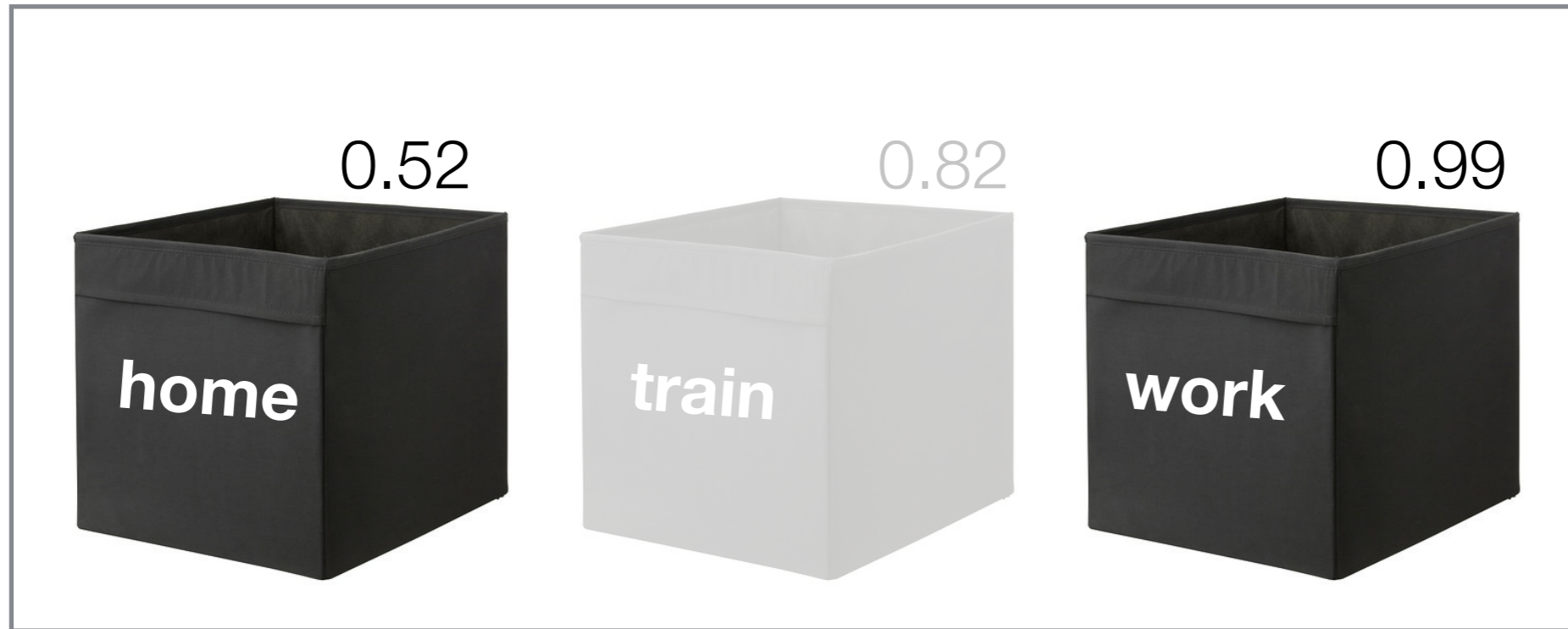
the 2nd and 3rd elements

```
boredom[ c(2,3) ]
```

Notice that the thing in the square brackets is itself a vector!

Selecting **more than one** element (by position)

boredom

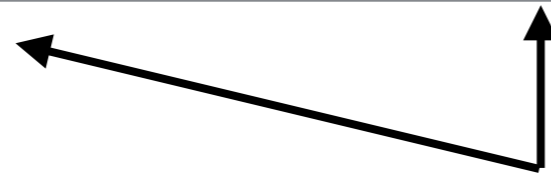
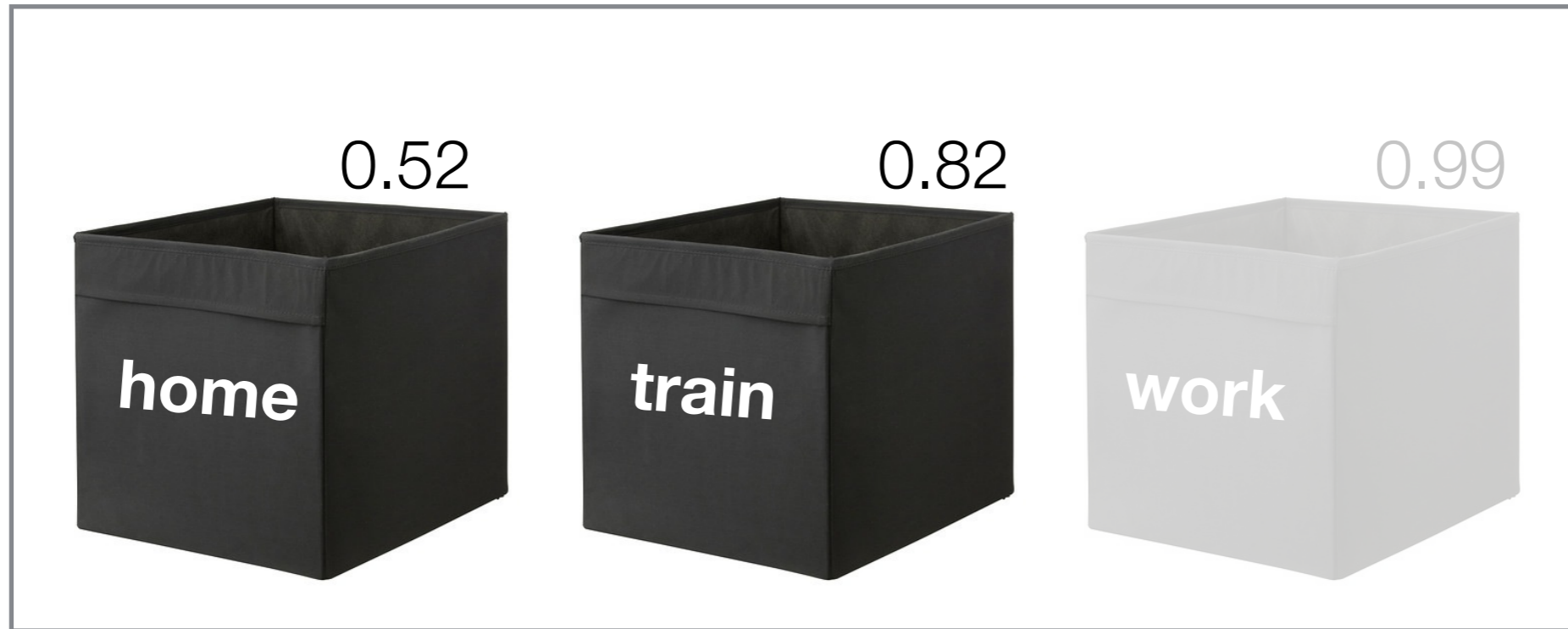


the 1st and 3rd elements

```
boredom[ c(1,3) ]
```

Selecting **more than one** element (by position)

boredom

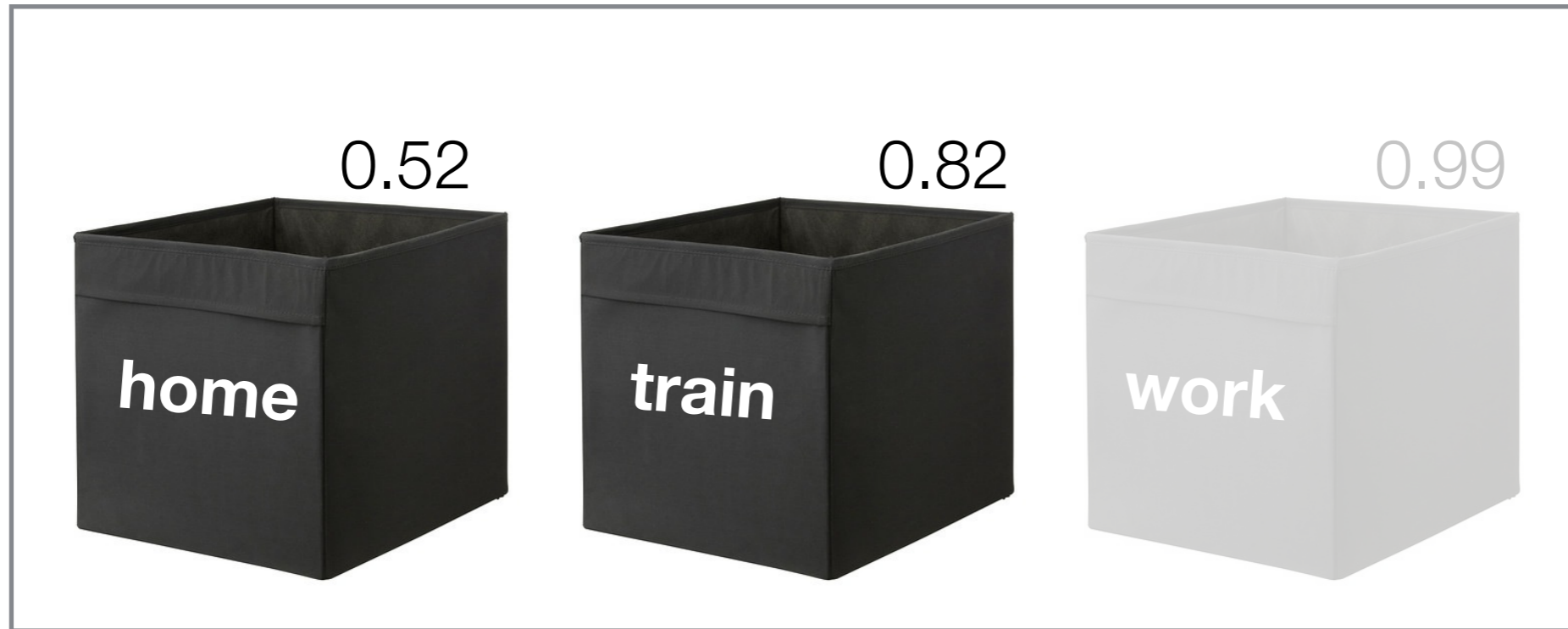


the 1st and 2nd elements

`boredom[c(1,2)]`

Selecting **more than one** element (by position)

boredom



Selecting **more than one** element (by position)

Let's check that...

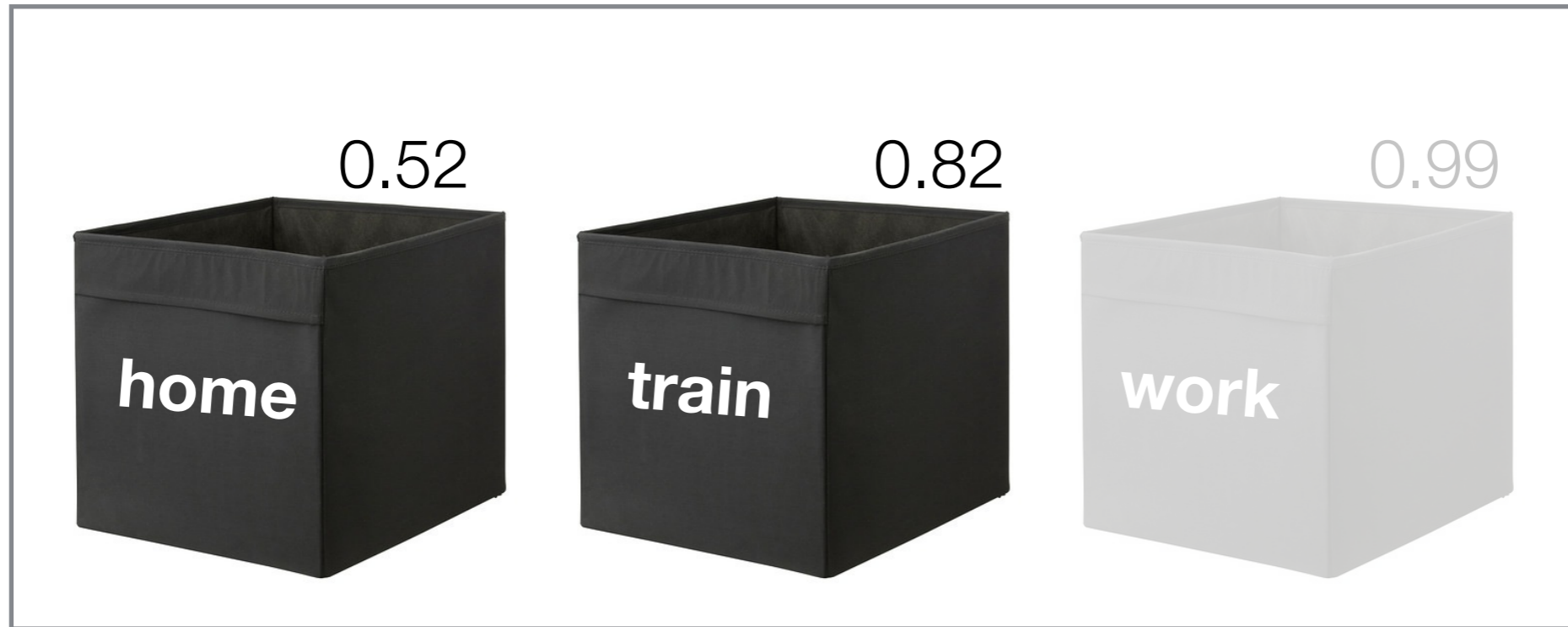
```
> c(3,4,5,6,7,8,9)
[1] 3 4 5 6 7 8 9
```

The long way

```
> 3:9
[1] 3 4 5 6 7 8 9
```

The shortcut

boredom



↑
Keep element 1?
TRUE

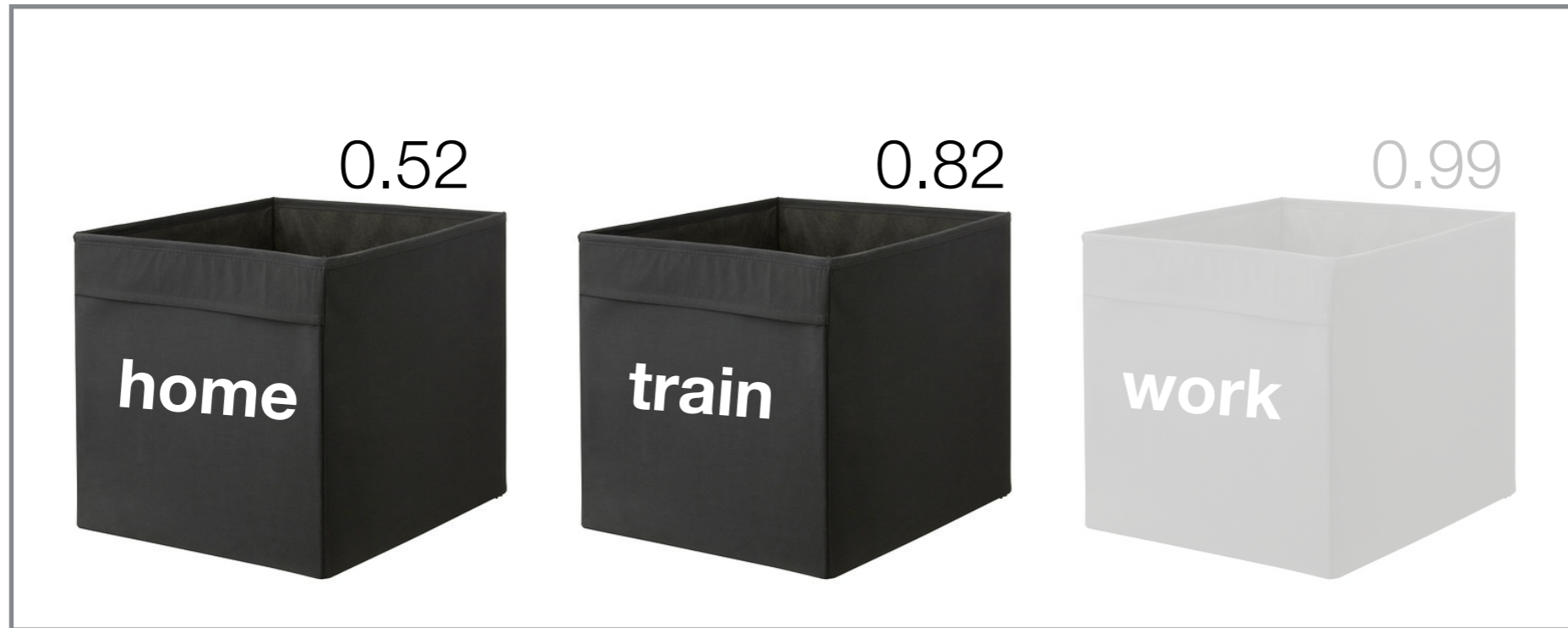
↑
Keep element 2?
TRUE

↑
Keep element 3?
FALSE

```
boredom[ c(TRUE, TRUE, FALSE) ]
```

Selecting elements “logically”

boredom



↑
Is element 1 of
boredom less than .9?
TRUE

↑
Is element 2 of
boredom less than .9?
TRUE

↑
Is element 3 of
boredom less than .9?
FALSE

`boredom[boredom < .9]`

Selecting elements “logically”

Why the heck would we ever want to
select elements logically?

An almost realistic example

```
> subject <- c( "STAT1", "STAT1", "STAT2", "STAT2" )  
> person <- c( "ann", "bec", "ann", "bec" )  
> grades <- c( 82, 71, 63, 80 )
```

Create vectors that
contain useful data

```
> data.frame( person, subject, grades )
```

	person	subject	grades
1	ann	STAT1	82
2	bec	STAT1	71
3	ann	STAT2	63
4	bec	STAT2	80

(Sneak preview... this is
what it looks like as an
actual data set...)

An almost realistic example

```
> subject <- c( "STAT1", "STAT1", "STAT2", "STAT2" )  
> person <- c( "ann", "bec", "ann", "bec" )  
> grades <- c( 82, 71, 63, 80 )
```

```
> grades[ subject == "STAT1" ]  
[1] 82 71
```

Here are the grades for STAT1

```
> subject[ grades >= 65 ]  
[1] "STAT1" "STAT1" "STAT2"
```

STAT1 has two credit or higher grades, STAT2 has one

```
> grades[ person == "ann" ]  
[1] 82 63
```

Here are the grades for Ann

An almost realistic example

```
> grades[ person == "ann" & subject == "STAT1" ]  
[1] 82
```

Find the grade where...
the person is Ann
and the subject is STAT1

An almost realistic example

```
> grades[ person == "ann" & subject == "STAT1" ]  
[1] 82
```

Find the grade where...
the person is Ann
and the subject is STAT1

```
> grades[ person == "ann" & subject == "STAT1" & grades < 50 ]  
numeric(0)
```

Try to find the grade where...
the person is Ann
and the subject is STAT1
and the grade was a fail

There aren't any grades like that!
The output is a "numeric" vector with 0 elements

Exercises

1. Make a vector called `family` with the names of everyone in your family, and another vector called `ages` with their ages (in the same order). If you don't have two or more people in your family, make some up.
2. Have R select the first item in the `ages` vector. Then Have R select the ages that are older than 90. What happens if there are none? (Use a different threshold than 90 if you need to). Do the same thing with 20 instead.
3. Have R return the names of the people in your family who are older than 20.
4. ** The modulus operator in R is `%` and will return the remainder after division (e.g. `10%%8` is 2). Create a vector called `n` with the numbers from 1 to 100, and then use the modulus operator to return only the even numbers from `n`.

Intro to R cheat sheet

1

operators and logical statements

+ addition
- subtraction
* multiplication
/ division
^ taking powers
<- assignment

& AND
| OR
! NOT

== equality
!= inequality
> greater than
>= greater than or equal to
< less than
<= less than or equal to

2

functions

`sqrt()` - Square root
`round()` - Round a number
`log()` - Logarithm
`exp()` - Exponentiation
`abs()` - Absolute value

3

functions take arguments
(order matters, unless you name them)

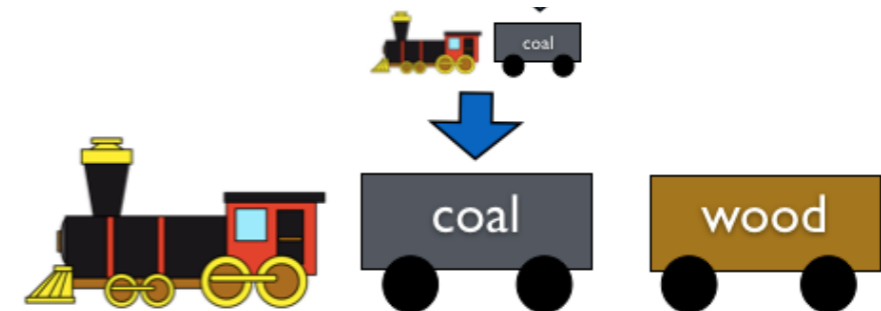


```
round (3.1415, 2)
```

```
round (x=3.1415, digits=2)
```

4

functions can take other functions
evaluated from the inside out



```
sqrt ( round (4.45) )
```




```
sqrt ( 4 )
```

2

Intro to R cheat sheet

5

variable classes

variable	example	picture
numeric	212	
logical	TRUE	
character	"a cat"	

7

vectors are lists of variables of the same class

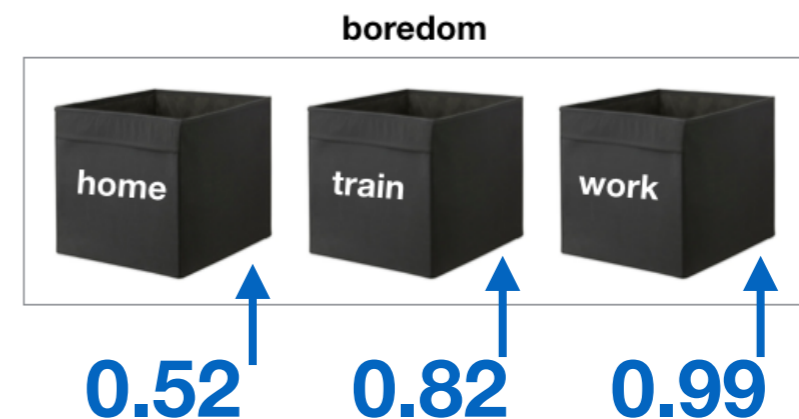


```
myVector <- c("cat", "dog", "lizard")
```

can access or assign specific variables in a vector by location, name, or logic

6

variable assignment



```
boredom <- c(home=0.52, train=0.82, work=0.99)
```

all these pick out the first item in the vector

