Studio

The R language (2 of 2)

Retrieve and use information in precise, efficient ways

Garrett Grolemund

Master Instructor, RStudio

August 2014

1. Subsetting
2. R Packages
3. Logical tests
4. Missing values

Question

```
x <- c(0, 0, 0, 0, 1, 0 ,0)
y <- x
y
# 0 0 0 0 1 0 0
```

How can you save just the fifth element of `x` to `y`?

How can you change the fifth element of `x` to a `0`?

Subsetting

Your turn

```
vec <- c(6, 1, 3, 6, 10, 5)

df <- data.frame(
  name = c("John", "Paul", "George", "Ringo"),
  birth = c(1940, 1942, 1943, 1940),
  instrument = c("guitar", "bass", "guitar", "drums")
)
```

With your neighbor, run the code on the following slide **IN YOUR HEADS**

vec

6	1	3	6	10	5

df

name	birth	instrument
John	1940	guitar
Paul	1942	bass
George	1943	guitar
Ringo	1940	drums

```
# Predict what the following code will do
# DON'T RUN IT!
```

```
vec[2]
vec[c(5, 6)]
vec[-c(5,6)]
vec[vec > 5]
```

```
df[c(2, 4), 3]
df[ , 1]
df[ , "instrument"]
df$instrument
```

Subset notation

name of object
to subset

vec

Subset notation

name of object to subset

brackets
(brackets always mean subset)

```
vec[ ]
```

Subset notation

name of object to subset

brackets
(brackets always mean subset)

```
vec[?]
```

an index
(that tells R which elements to include)

Each dimension needs its own index!

`vec[?]`

6	1	3	6	10	5

Each dimension needs its own index!

```
vec[?]
```

6	1	3	6	10	5

Each dimension needs its own index!

```
vec[?]
 df[?,?]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Each dimension needs its own index!

```
vec[?]
df[?,?]
```

which **rows** to include

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Each dimension needs its own index!

```
vec[?]
df[?,?]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

which **rows** to include

which **columns** to include

Each dimension needs its own index!

```
vec[?]
df[?,?]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

which **rows** to include

separate dimensions with a **comma**

which **columns** to include

Each dimension needs its own index!

```
vec[?]
df[?,?]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

But what should go in the indexes?

Four ways to subset

1. Integers
2. Blank spaces
3. Names
4. Logical vectors (TRUE and FALSE)

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[?,?]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[2,?]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[2,3]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[2,3]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[   ?   ,   ?   ]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[c(2,4),    ?   ]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[c(2,4),c(2,3)]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[c(2,4),c(2,3)]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[c(2,4),3]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

?

Integers (positive)

Positive integers behave just like *ij* notation in linear algebra

```
df[c(2,4),3]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

1. Colons are a useful way to create vectors

```
1:4
# 1 2 3 4
df[1:4, 1:2]
```

2. Repeating input repeats output

```
df[c(1,1,1,2,2), 1:3]
```

Integers (zero)

As an index, **zero will return nothing** from a dimension. This creates an empty object.

```
vec[0]
# numeric(0)

df[1:2, 0]
# data frame with 0 columns and 2 rows
```

Integers (negative)

Negative integers return **everything but** the elements at the specified locations.

You cannot use both negative and positive integers in the **same** dimension

Integers (negative)

Negative integers return **everything but** the elements at the specified locations.

You cannot use both negative and positive integers in the **same** dimension

```
vec[c(5,6)]
```

6	1	3	6	10	5

Integers (negative)

Negative integers return **everything but** the elements at the specified locations.

You cannot use both negative and positive integers in the **same** dimension

```
vec[-c(5,6)]
```

6	1	3	6	10	5

Integers (negative)

Negative integers return **everything but** the elements at the specified locations.

You cannot use both negative and positive integers in the **same** dimension

```
df[c(2:4), 2:3]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (negative)

Negative integers return **everything but** the elements at the specified locations.

You cannot use both negative and positive integers in the **same** dimension

```
df[-c(2:4), 2:3]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Integers (negative)

Negative integers return **everything but** the elements at the specified locations.

You cannot use both negative and positive integers in the **same** dimension

```
df[-c(2:4),-(2:3)]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Your Turn

1. Fix these poorly written subset commands

```
vec(1:4)
vec[-1:4]
vec[3, 4, 5]
```

() for functions, [] for subsetting

```
vec[1:4]
# 6 1 3 6
```

Don't mix positive and negative integers; distribute the negative sign (e.g., -1:4 = -1 0 1 2 3 4).

```
vec[-(1:4)]
# 10 5
```

Pass multiple values for the same dimension as a vector

```
vec[c(3, 4, 5)]
# 3  6  10
```

Your Turn

What is wrong with these subsetting commands?
What will they do?

```
mat[2]
df[1]
```

1	4	7
2	5	8
3	6	9

`mat[2]`

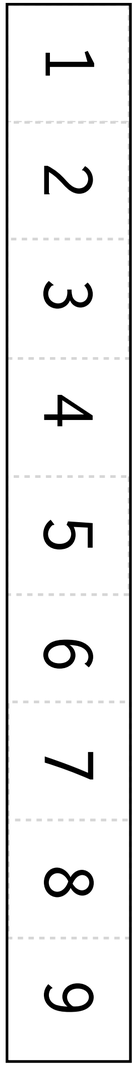
How R makes a matrix

vec

1	2	3	4	5	6	7	8	9

How R makes a matrix

vec



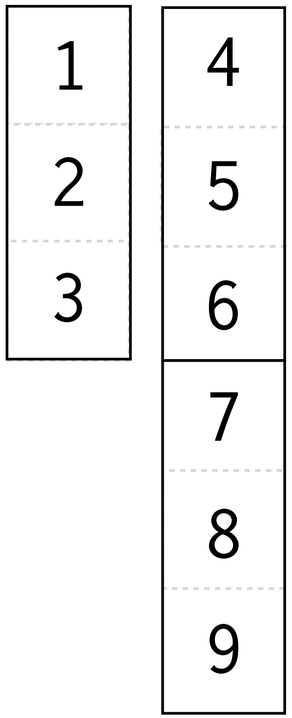
How R makes a matrix

vec

1
2
3
4
5
6
7
8
9

How R makes a matrix

vec



How R makes a matrix

vec

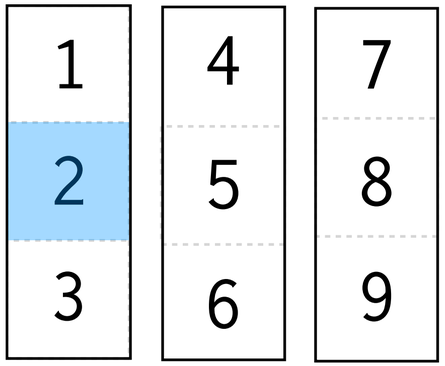
1	4	7
2	5	8
3	6	9

How R makes a matrix

~~vec~~

matrix

1	4	7
2	5	8
3	6	9



`mat[2]`

John	1940	guitar
Paul	1942	bass
George	1943	guitar
Ringo	1940	drums

df[2]

How R makes a data frame

List

`c("a","b","c","d")`	`c(1, 2, 3, 4)`	`c(T, F, T, F)`

List

```
c(
"a",
"b",
"c",
"d")
```

```
c(
1,
2,
3,
4)
```

```
c(
T,
F,
T,
F)
```

List

```
c(
"a",
"b",
"c",
"d")
```

```
c(
1,
2,
3,
4)
```

```
c(
T,
F,
T,
F)
```

~~List~~

data frame

c("a", "b", "c", "d")	c(1, 2, 3, 4)	c(T, F, T, F)

c("John","Paul", "George","Ringo")	c(1940, 1942, 1943, 1940)	c("guitar", "bass", "guitar","drums")

John	1940	guitar
Paul	1942	bass
George	1943	guitar
Ringo	1940	drums

df[2]

Blank spaces

Blank spaces return **everything**
(i.e., no subsetting occurs on that dimension)

```
vec[ ]
```

6	1	3	6	10	5

Blank spaces

Blank spaces return **everything**
(i.e., no subsetting occurs on that dimension)

```
df[1, ]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Blank spaces

Blank spaces return **everything**
(i.e., no subsetting occurs on that dimension)

```
df[ ,2]
```

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Names

If your object has names, you can ask for elements or columns back by name.

```
vec[                    ]
```

6	1	3	6	10	5

Names

If your object has names, you can ask for elements or columns back by name.

```
names(vec) <- c("a","b","c","d","e","f")
```

vec[ ]

a	b	c	d	e	f
6	1	3	6	10	5

Names

If your object has names, you can ask for elements or columns back by name.

```
names(vec) <- c("a","b","c","d","e","f")
```

```
vec[c("a","b","d")]
```

a	b	c	d	e	f
6	1	3	6	10	5

Names

If your object has names, you can ask for elements or columns back by name.

```
names(vec) <- c("a","b","c","d","e","f")
```

```
vec[c("a","c","f")]
```

a	b	c	d	e	f
6	1	3	6	10	5

Names

If your object has names, you can ask for elements or columns back by name.

```
df[ ,"birth"]
```

name	birth	instrument
John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Names

If your object has names, you can ask for elements or columns back by name.

```
df[ ,c("name","birth")]
```

name	birth	instrument
John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Logical

You can subset with a logical vector of the same length as the dimension you are subsetting. Each element that corresponds to a TRUE will be returned.

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

6	1	3	6	10	5

Logical

You can subset with a logical vector of the same length as the dimension you are subsetting. Each element that corresponds to a TRUE will be returned.

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

6	1	3	6	10	5

Logical

You can subset with a logical vector of the same length as the dimension you are subsetting. Each element that corresponds to a TRUE will be returned.

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

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Logical

You can subset with a logical vector of the same length as the dimension you are subsetting. Each element that corresponds to a TRUE will be returned.

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

John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Subset notation

	effect
integers	positive: returns specified elements
	0: returns nothing
	negative: returns everything but the specified elements
blank spaces	returns everything
names	returns elements or columns with the specified names
logicals	returns elements that correspond to TRUE

Your Turn

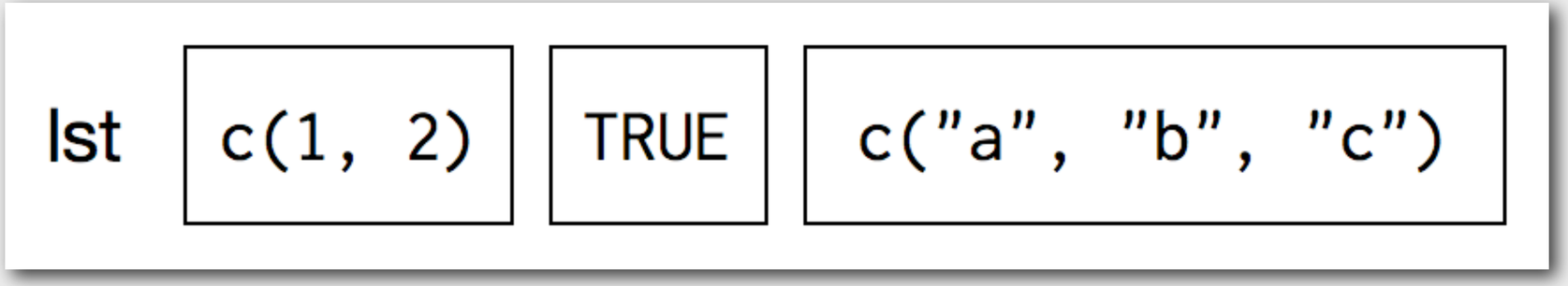
Write down as many ways to extract the name `"John"` from `df` as you can. Make sure each works. You have two minutes.

name	birth	instrument
John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

Answers

```
df[1, 1]
df[1, "name"]
df[1, -(2:3)]
df[1, c(TRUE, FALSE, FALSE)]
df[-(2:4), 1]
df[-(2:4), "name"]
df[-(2:4), -(2:3)]
df[-(2:4), c(TRUE, FALSE, FALSE)]
df[c(TRUE, FALSE, FALSE, FALSE), 1]
df[c(TRUE, FALSE, FALSE, FALSE), "name"]
df[c(TRUE, FALSE, FALSE, FALSE), -(2:3)]
df[c(T, F, F, F), c(T, F, F)]
```

Your Turn




```
lst <- list(c(1, 2), TRUE, c("a", "b", "c"))
```

Can you extract the vector `c(1,2)` from `lst` and run `sum` on it? Note that sum calculates the sum of a vector:

```
sum(c(1,2,3)) # 6
```

Subsetting lists

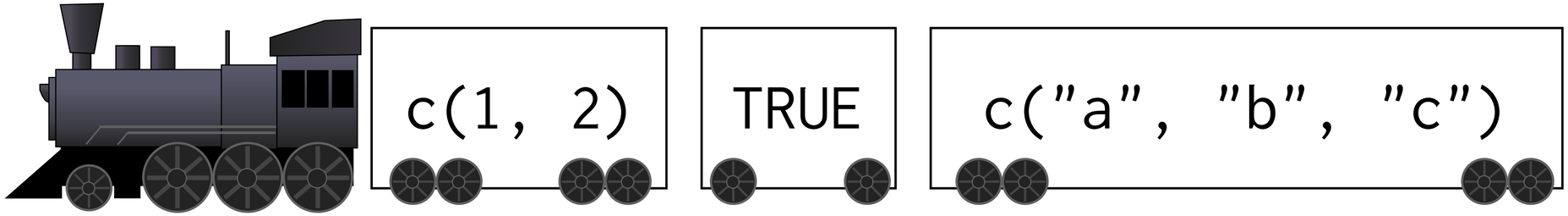
lst	c(1, 2)	TRUE	c("a", "b", "c")

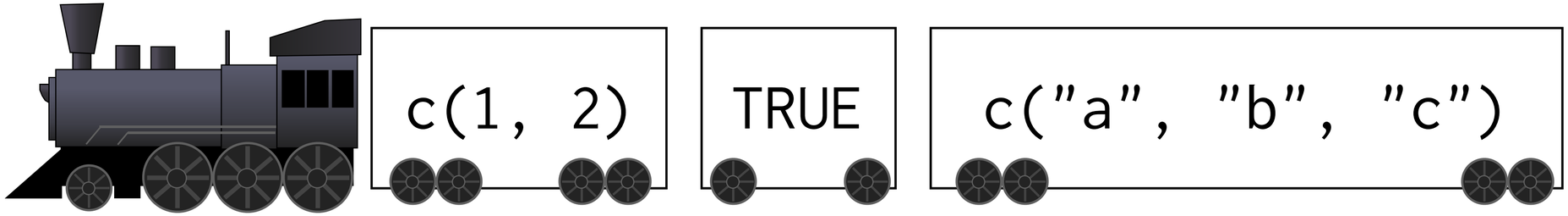
```
sum(lst[1]) # Error!

# What is the difference?
lst[c(1,2)]
lst[1]
lst[[1]]
```

If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[4:6]` is a train of cars 4-6.

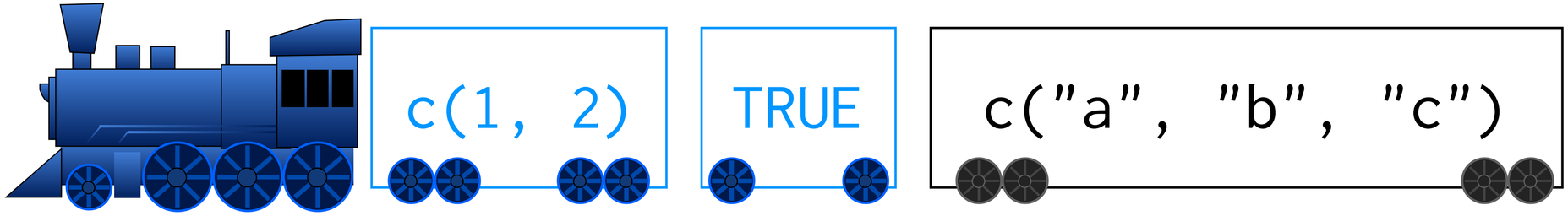
http://twitter.com/#!/RLangTip/status/118339256388304896

lst	c(1, 2)	TRUE	c("a", "b", "c")

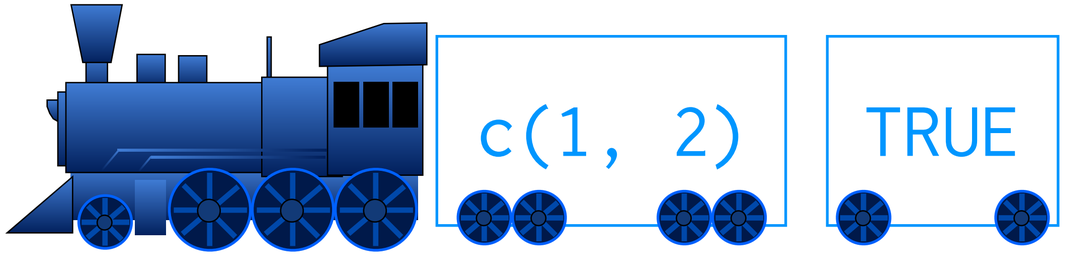




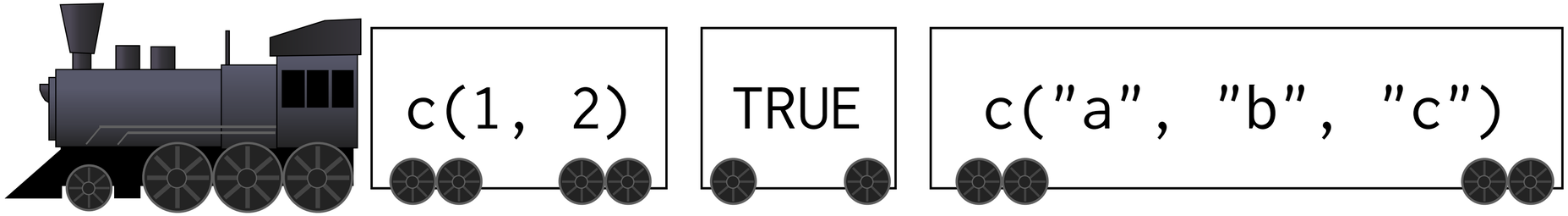

```
lst[c(1,2)]
```



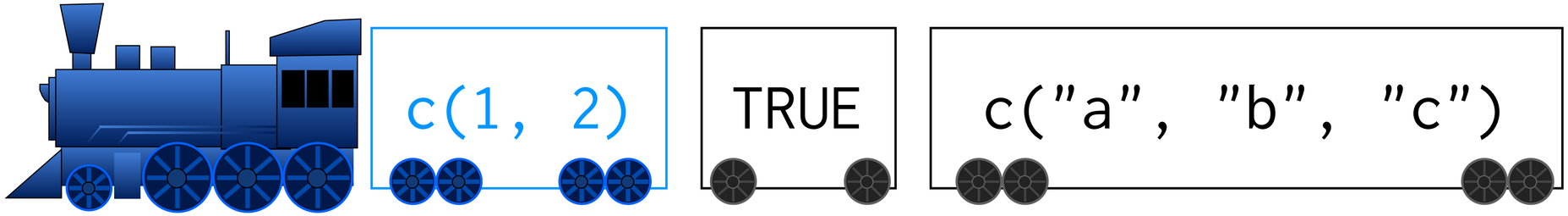

```
lst[c(1,2)]
```



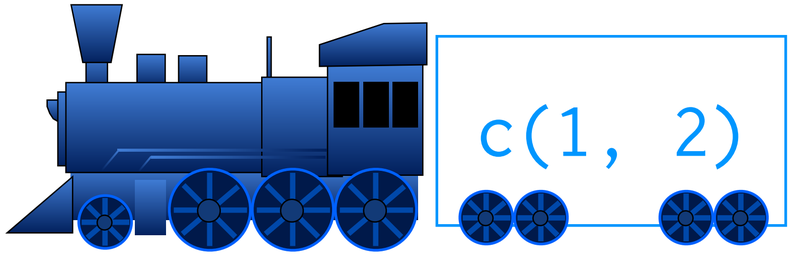

```
# [[1]]
# [1] 1 2
#
# [[2]]
# [1] TRUE
```



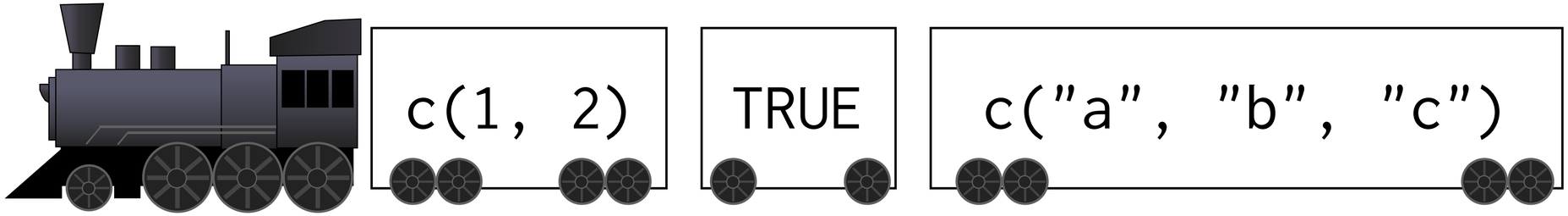

```
lst[1]
```



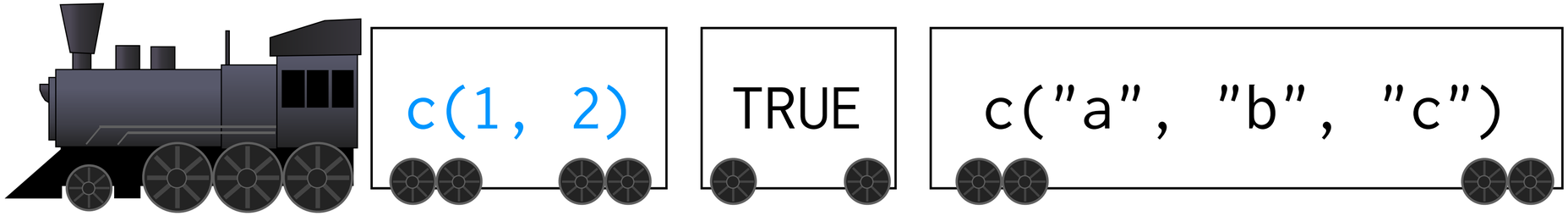

```
lst[1]
```




```
# [[1]]
# [1] 1 2
```

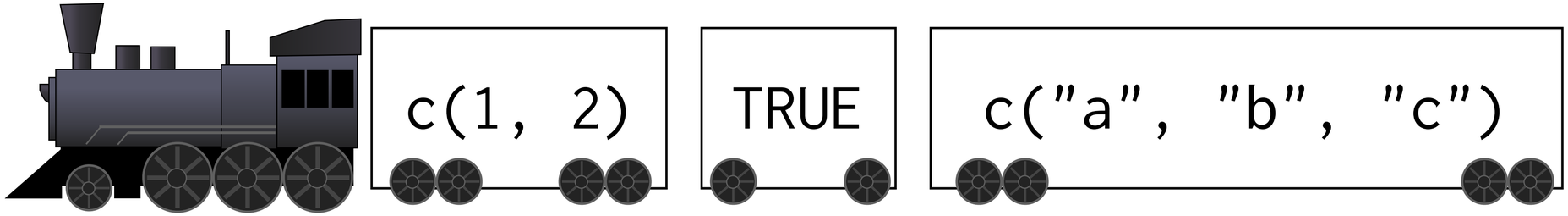



```
lst[[1]]
```



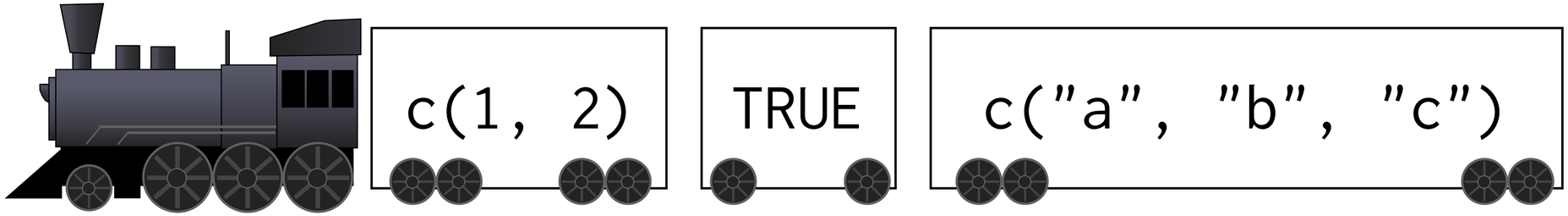

```
lst[[1]]
```

```
c(1, 2)
```

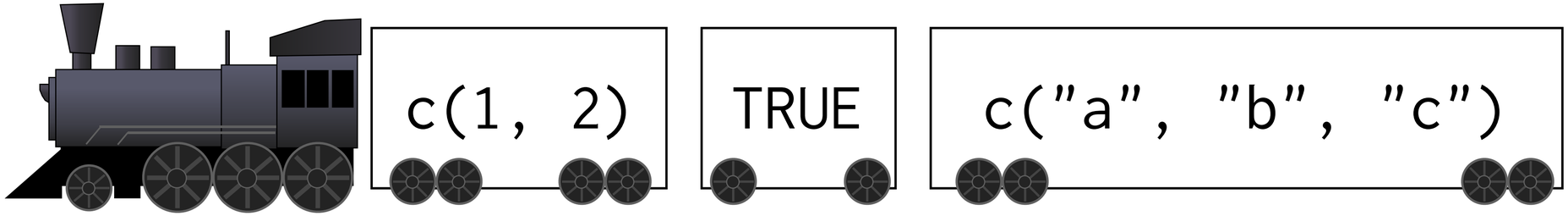



```
lst[[1]][2]
```

What will this return?

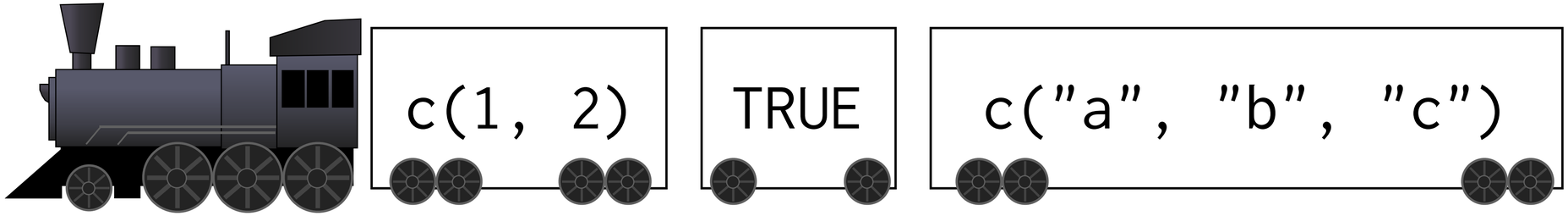



```
lst[[1]][2]
```




```
lst[[1]][2]
```

```
c(1, 2)[2]
```




```
lst[[1]][2]
```

```
c(1, 2)[2]
```

```
2
```

$

The most common syntax for subsetting lists and data frames

```
names(lst) <- c("alpha", "beta", "gamma")
```

	alpha	beta	gamma
lst	c(1, 2)	TRUE	c("a", "b", "c")

```
lst$alpha
```

	alpha	beta	gamma
lst	`c(1, 2)`	TRUE	`c("a", "b", "c")`

```
lst$alpha
```

name of list

	alpha	beta	gamma
lst	`c(1, 2)`	TRUE	`c("a", "b", "c")`

```
lst$alpha
```

name of list

$

	alpha	beta	gamma
lst	`c(1, 2)`	TRUE	`c("a", "b", "c")`

```
lst$alpha
```

name of list

$

name of element
(no quotes)

	alpha	beta	gamma
lst	`c(1, 2)`	TRUE	`c("a", "b", "c")`

`c(1, 2)`

`lst$alpha`

name of list

$

name of element
(no quotes)

	alpha	beta	gamma
lst	c(1, 2)	TRUE	c("a", "b", "c")

c(1, 2)

```
lst$alpha
```

Same as lst[["alpha"]]

name	birth	instrument
John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

```
df$birth
```

name	birth	instrument
John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

```
df$birth
```

name of data frame

name	birth	instrument
John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

```
df$birth
```

name of data frame

$

name	birth	instrument
John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

```
df$birth
```

name of data frame

$

name of column
(no quotes)

name	birth	instrument
John	1940	guitar
Paul	1941	bass
George	1943	guitar
Ringo	1940	drums

```
c(1940, 1941, 1943, 1940)
```

```
df$birth
```

name of data frame

$

name of column
(no quotes)

R Packages

R Packages

A collection of code and functions written for the R language.

Usually focuses on a specific task or problem.

Most of the useful R applications appear in packages.

Start RStudio

Internet

Your hard drive

Your R session

Install your package with
`install.packages("ggplot2")`

(ONE TIME)

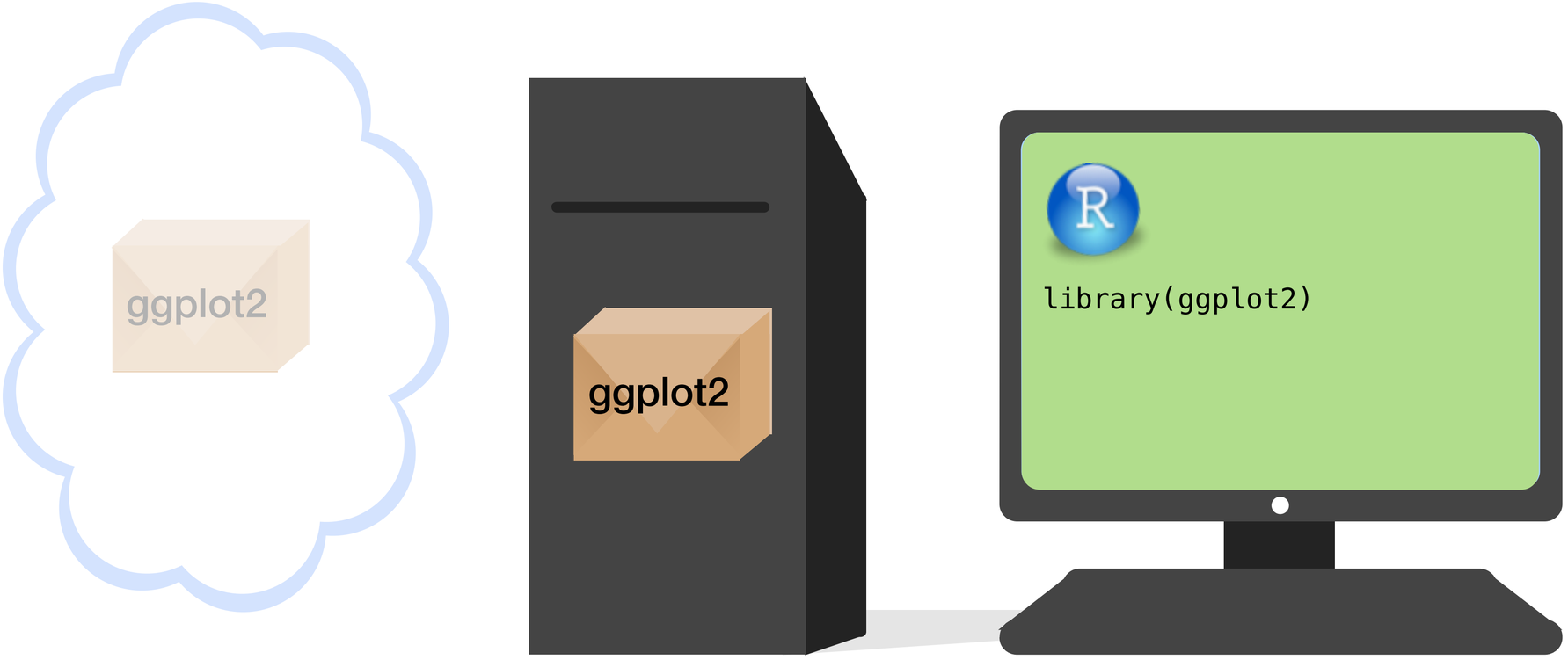
Internet

Your hard drive

Your R session

Load your package with `library(ggplot2)`

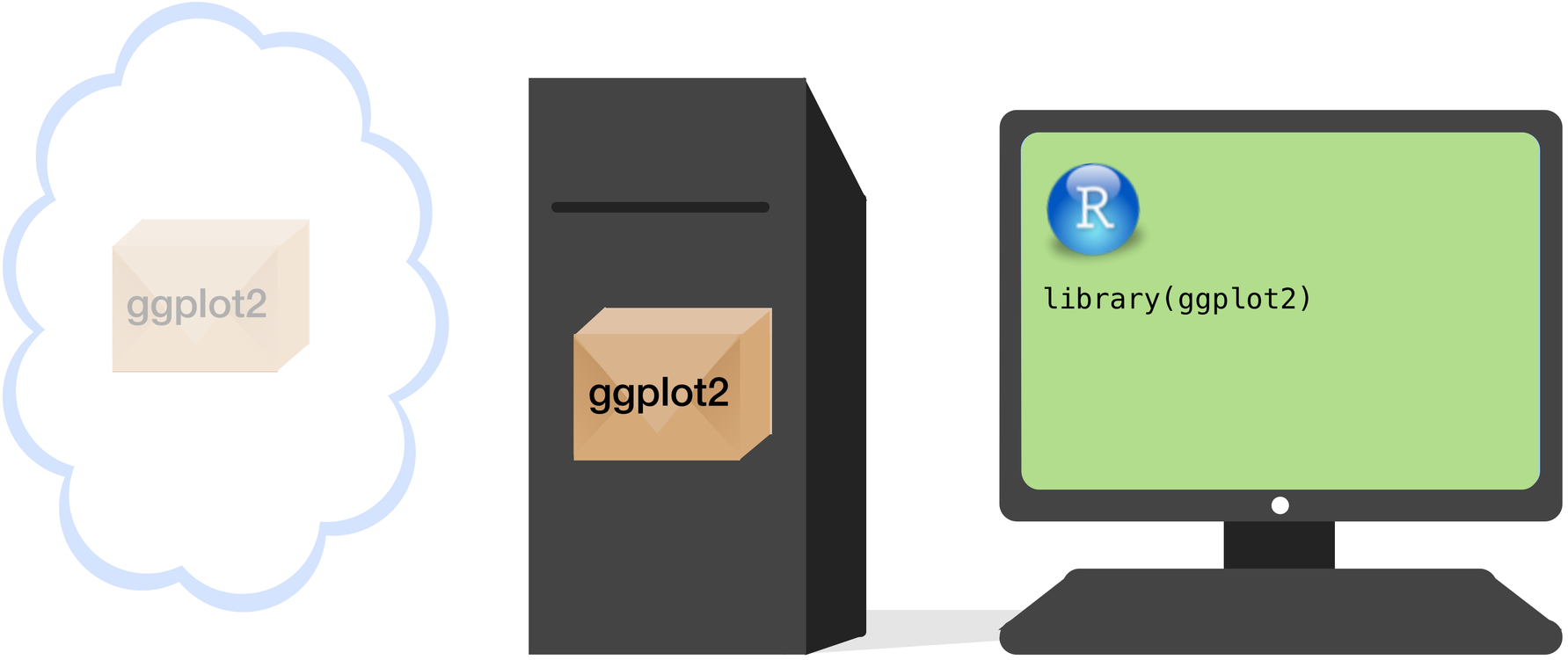
(EVERY TIME)



Internet

Your hard drive

Your R session



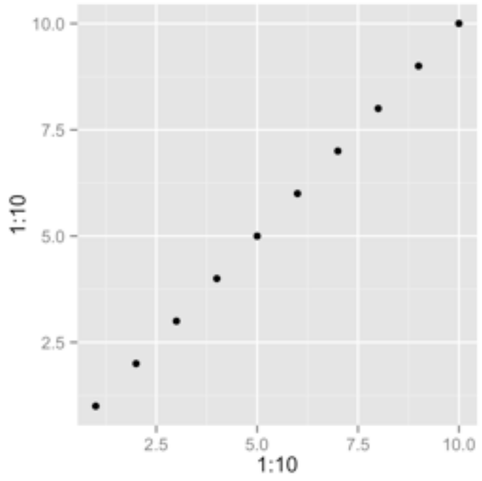
Internet

Your hard drive

Your R session

```
qplot(1:10, 1:10)
## Error: could not find function "qplot"
```

```
library(ggplot2)
qplot(1:10, 1:10)
##
```



You cannot use a function in a package until you load the package

Package summary

1. Download the package with
   `install.packages("name")`
   - You only have to do this once
   - You should be connected to the internet

2. Load the package with
   `library("name")`
   - You have to do this each time you start an R session.

There are over 5100 R packages

Your Turn

We're going to use the ggplot2, maps, RColorBrewer, and scales packages today.

Load them with

```
library("ggplot2")
library("maps")
library("RColorBrewer")
```

Note: If you have not yet installed them, you'll need to run `install.packages(c("ggplot2", “maps", "RColorBrewer"))` first.

Diamonds

Diamonds data

- ~**54,000** round diamonds from http://www.diamondse.info/
- comes in the ggplot2 package
- Carat, colour, clarity, cut
- Total depth, table, depth, width, height
- Price

Your turn

`diamonds` is huge!

Use subsetting to look at just the first six rows of `diamonds`

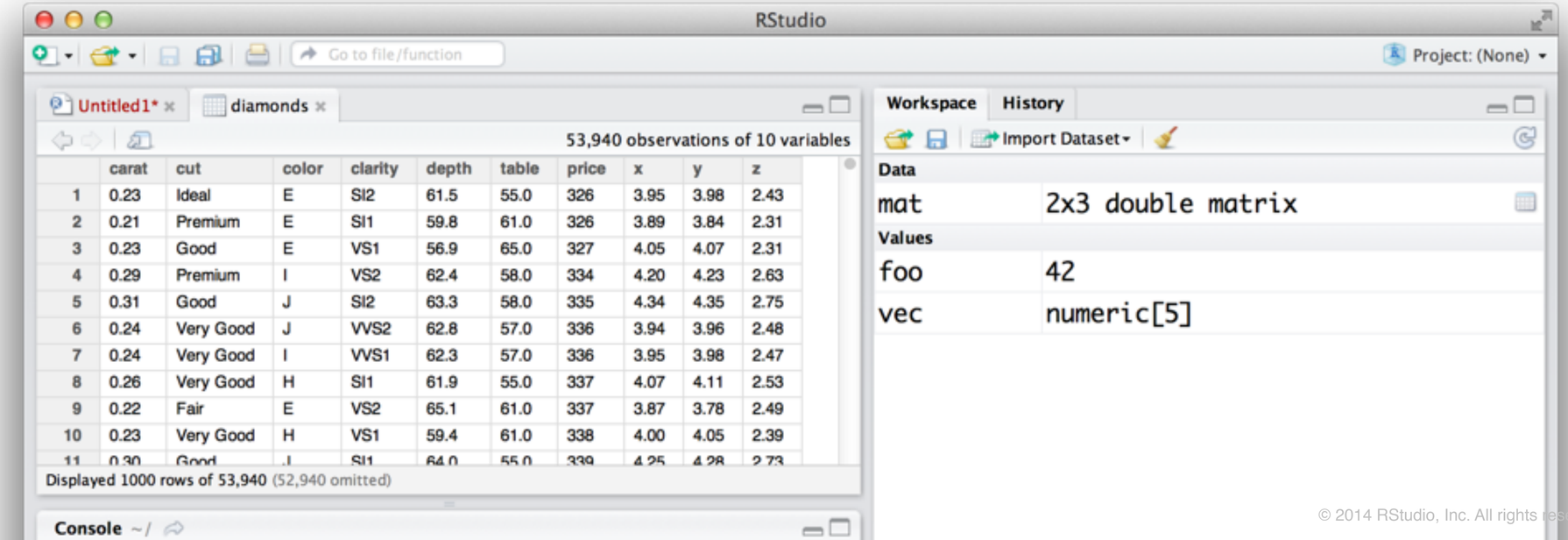
Challenge: use subsetting to look at just the **last** six rows

```
diamonds[1:6, ]
nrow(diamonds)
# 53940
diamonds[53935:53940, ]

# Same as
head(diamonds)
tail(diamonds)
```

View

The View function can also help you examine a data set, it opens a spreadsheet like data viewer.

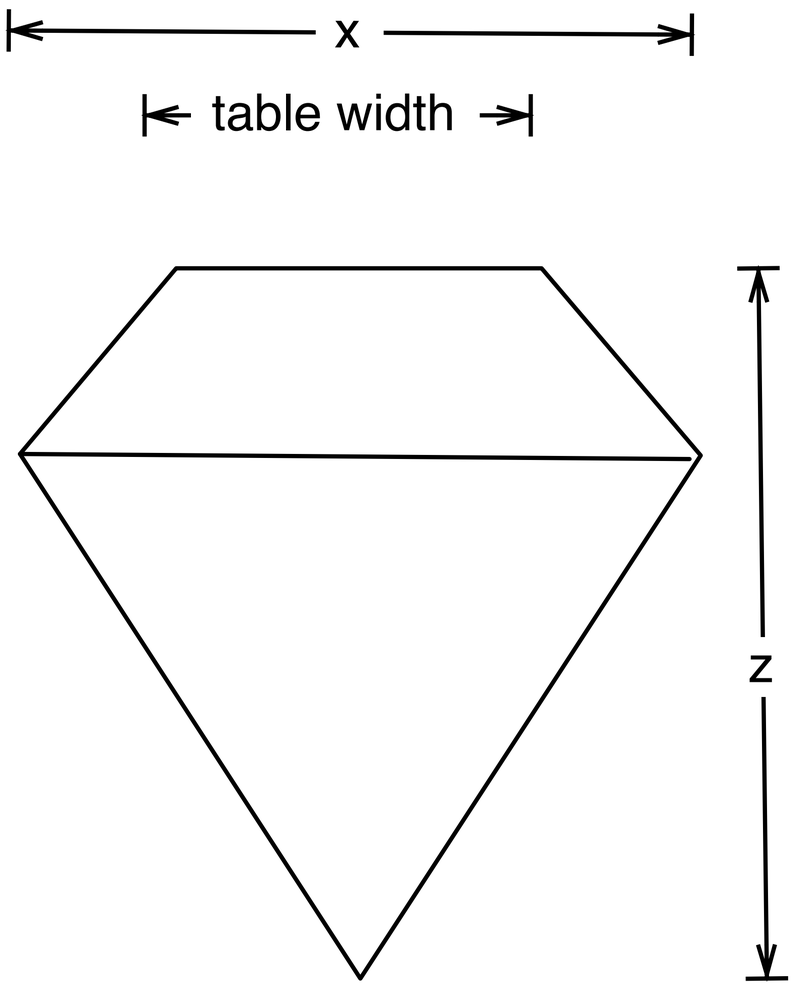
```
View(diamonds) # notice: Capital V
```



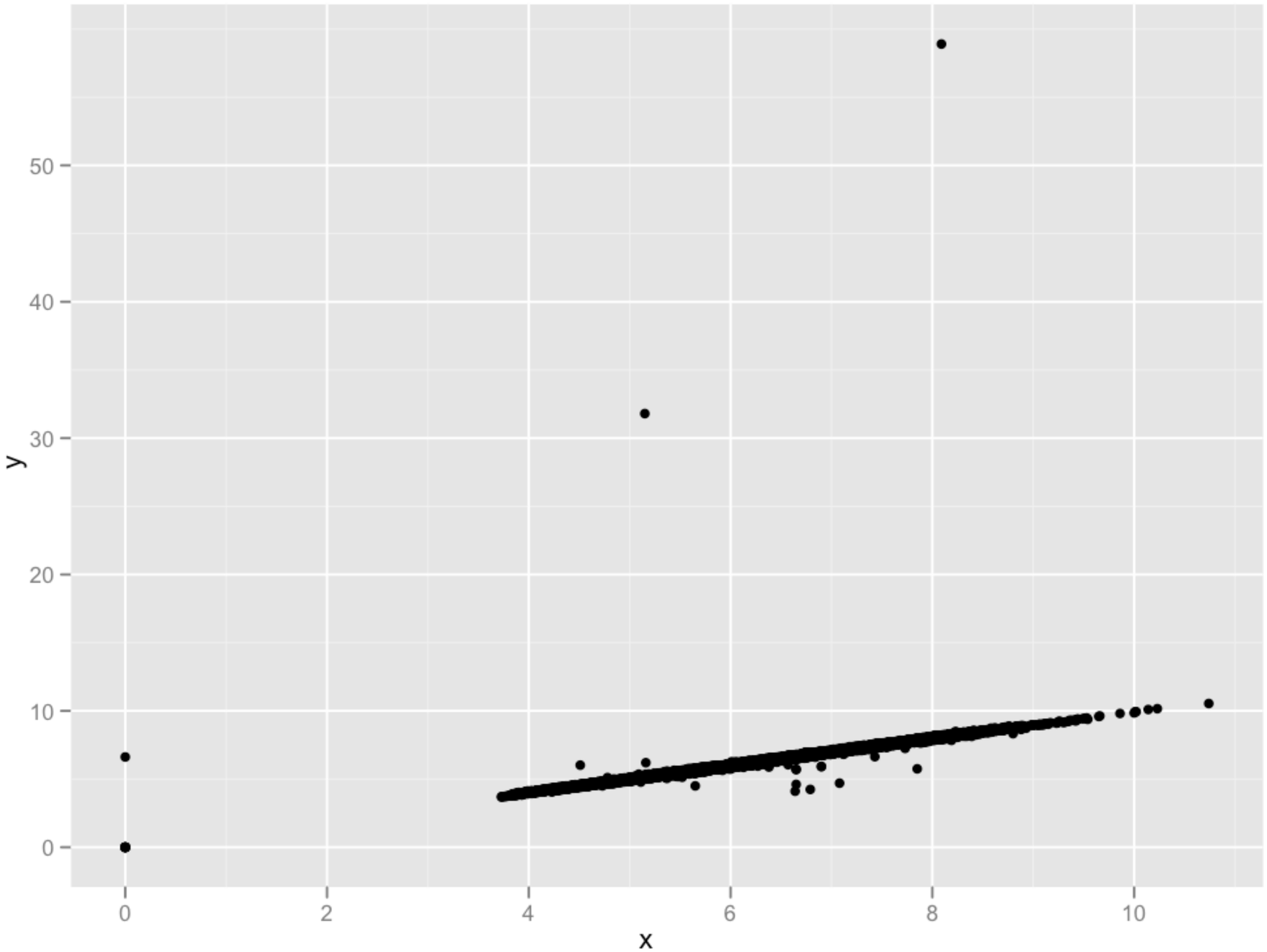
Help pages

You can open the help page for any R object (including functions) by typing ? followed by the object's name

```
?diamonds
```

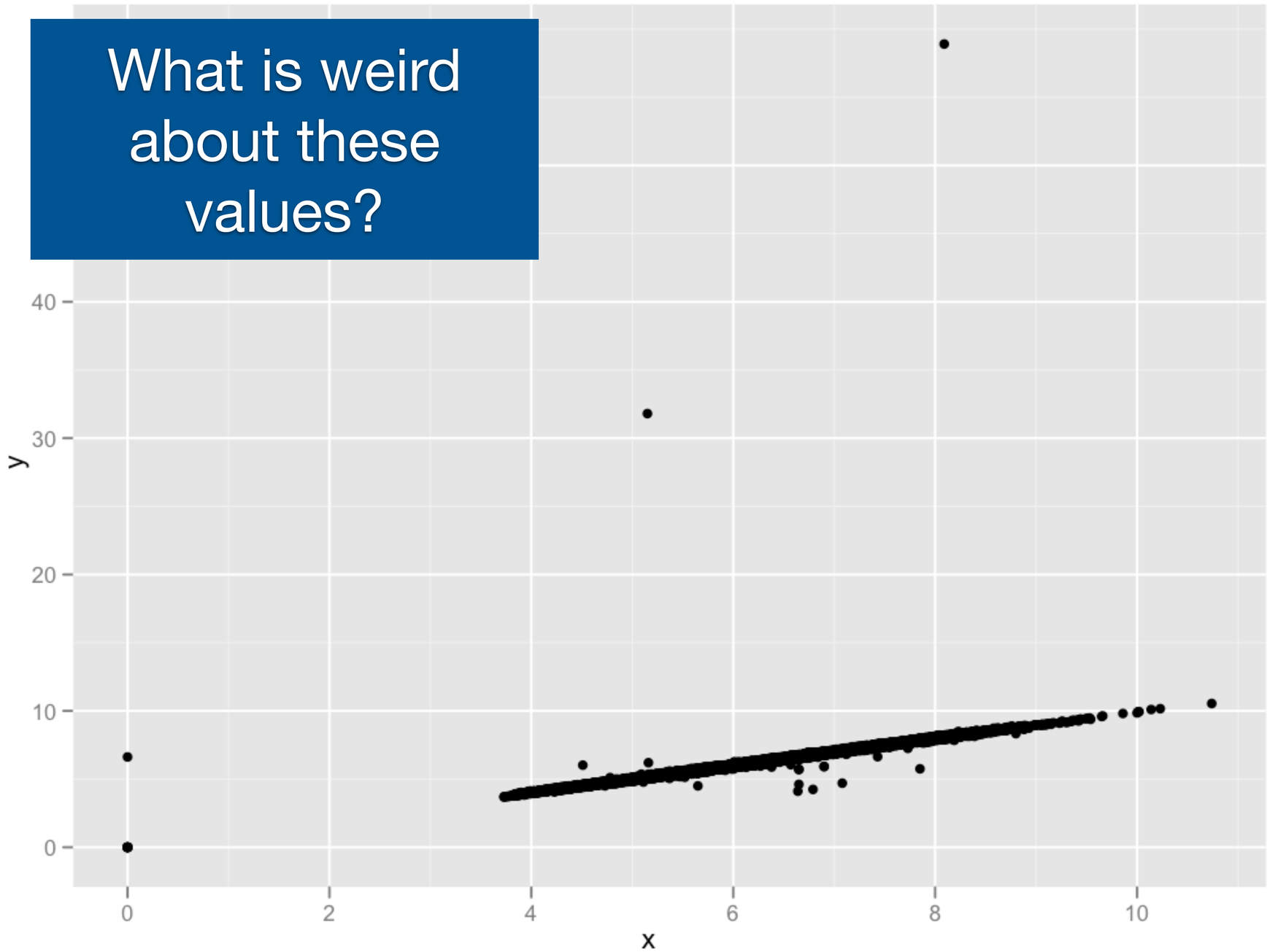


depth = z / diameter
table = table width / x * 100
x, y, z in mm



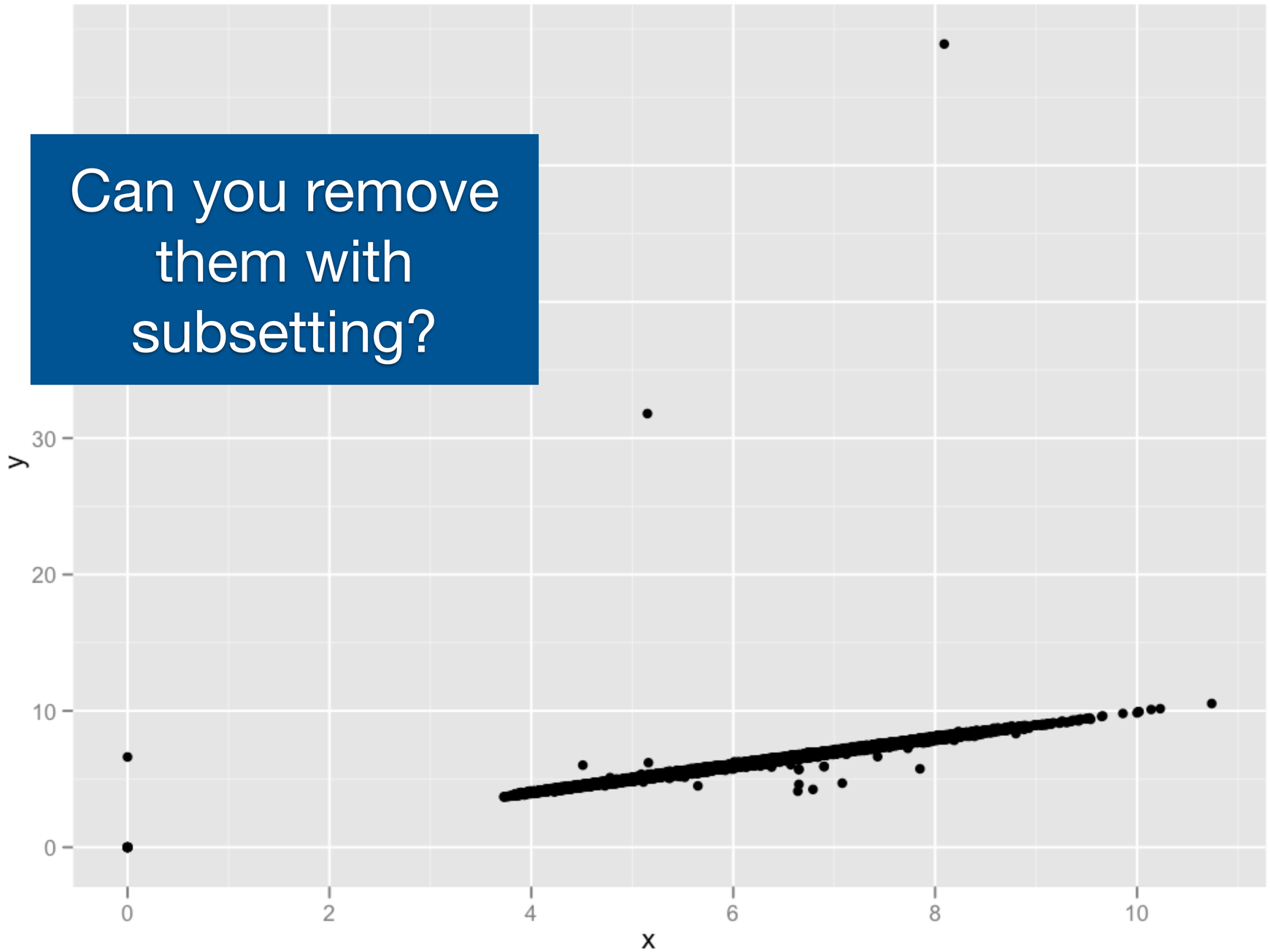

```
qplot(x, y, data = diamonds)
```

What is weird about these values?




```
qplot(x, y, data = diamonds)
```

Can you remove them with subsetting?




```
qplot(x, y, data = diamonds)
```

Logical tests

Logical comparisons

What will these return?

```
1 < 3
1 > 3
c(1, 2, 3, 4, 5) > 3
```

Your turn

x <- c(1, 2, 3, 4, 5)

Operator	Result	Comparison
x > 3	c(F, F, F, T, T)	greater than
x >= 3		
x < 3		
x <= 3		
x == 3		
x != 3		
x = 3		

Your turn

x <- c(1, 2, 3, 4, 5)

Operator	Result	Comparison
x > 3	c(F, F, F, T, T)	greater than
x >= 3	c(F, F, T, T, T)	greater than or equal to
x < 3		
x <= 3		
x == 3		
x != 3		
x = 3		

Your turn

x <- c(1, 2, 3, 4, 5)

Operator	Result	Comparison
x > 3	c(F, F, F, T, T)	greater than
x >= 3	c(F, F, T, T, T)	greater than or equal to
x < 3	c(T, T, F, F, F)	less than
x <= 3		
x == 3		
x != 3		
x = 3		

Your turn

x <- c(1, 2, 3, 4, 5)

Operator	Result	Comparison
x > 3	c(F, F, F, T, T)	greater than
x >= 3	c(F, F, T, T, T)	greater than or equal to
x < 3	c(T, T, F, F, F)	less than
x <= 3	c(T, T, T, F, F)	less than or equal to
x == 3		
x != 3		
x = 3		

Your turn

x <- c(1, 2, 3, 4, 5)

Operator	Result	Comparison
x > 3	c(F, F, F, T, T)	greater than
x >= 3	c(F, F, T, T, T)	greater than or equal to
x < 3	c(T, T, F, F, F)	less than
x <= 3	c(T, T, T, F, F)	less than or equal to
x == 3	c(F, F, T, F, F)	equal to
x != 3		
x = 3		

Your turn

x <- c(1, 2, 3, 4, 5)

Operator	Result	Comparison
x > 3	c(F, F, F, T, T)	greater than
x >= 3	c(F, F, T, T, T)	greater than or equal to
x < 3	c(T, T, F, F, F)	less than
x <= 3	c(T, T, T, F, F)	less than or equal to
x == 3	c(F, F, T, F, F)	equal to
x != 3	c(T, T, F, T, T)	not equal to
x = 3		

Your turn

x <- c(1, 2, 3, 4, 5)

Operator	Result	Comparison
x > 3	c(F, F, F, T, T)	greater than
x >= 3	c(F, F, T, T, T)	greater than or equal to
x < 3	c(T, T, F, F, F)	less than
x <= 3	c(T, T, T, F, F)	less than or equal to
x == 3	c(F, F, T, F, F)	equal to
x != 3	c(T, T, F, T, T)	not equal to
x = 3		same as <-

%in%

```
# What does this do?
1 %in% c(1, 2, 3, 4)
1 %in% c(2, 3, 4)
c(3,4,5,6) %in% c(2, 3, 4)
```

%in%

`%in%` tests whether the object on the left is a member of the group on the right.

```
1 %in% c(1, 2, 3, 4)
# TRUE
1 %in% c(2, 3, 4)
# FALSE
c(3,4,5,6) %in% c(2, 3, 4)
# TRUE TRUE FALSE FALSE
```

Boolean operators

You can combine logical tests with `&`, `|`, `xor`, `!`, `any`, and `all`

```
x > 2 & x < 9
```

Boolean operators

You can combine logical tests with `&`, `|`, `xor`, `!`, `any`, and `all`

```
x > 2 & x < 9
```

```
TRUE &
```

Boolean operators

You can combine logical tests with `&`, `|`, `xor`, `!`, `any`, and `all`

```
x > 2 & x < 9
```

↓

```
TRUE & TRUE
```

Boolean operators

You can combine logical tests with `&`, `|`, `xor`, `!`, `any`, and `all`

```
x > 2 & x < 9
```

↓

```
TRUE & TRUE
```

↓

```
TRUE
```

&

Are both condition 1 **and** condition 2 true?

expression	outcome
TRUE & TRUE	TRUE
TRUE & FALSE	FALSE
FALSE & TRUE	FALSE
FALSE & FALSE	FALSE

|

Is either condition 1 **or** condition 2 true?

expression	outcome
TRUE | TRUE	TRUE
TRUE | FALSE	TRUE
FALSE | TRUE	TRUE
FALSE | FALSE	FALSE

xor

Is either condition 1 **or** condition 2 true, **but not both**?

expression	outcome
xor(TRUE, TRUE)	FALSE
xor(TRUE, FALSE)	TRUE
xor(FALSE, TRUE)	TRUE
xor(FALSE, FALSE)	FALSE

!

Negation

expression	outcome
!(TRUE)	FALSE
!(FALSE)	TRUE

any

Is **any** condition TRUE?

expression	outcome
any(c(TRUE, FALSE, FALSE))	TRUE
any(c(FALSE, FALSE, FALSE))	FALSE

all

Is **every** condition TRUE?

expression	outcome
all(c(TRUE, TRUE, TRUE))	TRUE
all(c(TRUE, FALSE, TRUE))	FALSE

Logical operators

operator	tests
x > y	is x greater than y?
x >= y	is x greater than or equal to y?
x < y	is x less than y?
x <= y	is x less than or equal to y?
x == y	is x equal to y?
x != y	is x not equal to y?
x %in% c(y, z)	is x in the set c(y, z)?

Boolean operators

operator	tests
a & b	both a and b are TRUE
a | b	at least one of a and b is TRUE (or)
xor(a, b)	a is TRUE or b is TRUE, but not both
!(a)	not a (TRUE goes to FALSE, FALSE goes to TRUE)
any(a, b, c)	at least one of a, b , or c is TRUE
all(a, b, c)	each of a, b, and c is TRUE

```
w <- c(-1, 0, 1)
x <- c(5, 15)
y <- "February"
z <- c("Monday", "Tuesday", "Friday")
```

Your turn

Turn these sentences into logical tests in R

Is `w` positive?

Is `x` greater than 10 and less than 20?

Is object `y` the word February?

Is every value in `z` a day of the week?

```
# Answers
w > 0
10 < x & x < 20
y == "February"
all(z %in% c("Monday", "Tuesday", "Wednesday",
  "Thursday", "Friday", "Saturday", "Sunday"))
```

```
# Common mistakes

x > 10 & < 20
y = "February"
all(z == "Monday" | "Tuesday" | "Wednesday"...)
```

```
x > 10 & < 20
```

```
x > 10 & < 20
```

```
TRUE &
```

```
x > 10 & < 20
```

```
TRUE & error!
```

```
x > 10 & < 20
```

↓

```
TRUE & error!
```

↓

```
error!
```

Logical subsetting

Logical subsetting

Combining logical tests with subsetting is a very powerful technique!

```
x_zeroes <- diamonds$x == 0
# FALSE FALSE FALSE FALSE FALSE FALSE ...

# What will this return?
diamonds[x_zeroes, ]
```

Saving results

```
# Prints to screen
diamonds[diamonds$x > 10, ]

# Saves to new data frame
big <- diamonds[diamonds$x > 10, ]

# Overwrites existing data frame. Dangerous!
diamonds <- diamonds[diamonds$x < 10,]
```

```
diamonds <- diamonds[1, 1]
diamonds

# Uh oh!

rm(diamonds)
str(diamonds)

# Phew!
```

Missing values

Data errors

Typically removing the entire row because of one error is overkill. Better to selectively replace problem values with missing values.

In R, missing values are indicated by `NA`

Expression	Guess	Actual
`5 + NA`		
`mean(c(5, NA))`		
`NA < 3`		
`NA == 3`		
`NA == NA`		

Expression	Guess	Actual
`5 + NA`		`NA`
`mean(c(5, NA))`		
`NA < 3`		
`NA == 3`		
`NA == NA`		

Expression	Guess	Actual
`5 + NA`		`NA`
`mean(c(5, NA))`		`NA`
`NA < 3`		
`NA == 3`		
`NA == NA`		

Expression	Guess	Actual
`5 + NA`		`NA`
`mean(c(5, NA))`		`NA`
`NA < 3`		`NA`
`NA == 3`		
`NA == NA`		

Expression	Guess	Actual
`5 + NA`		`NA`
`mean(c(5, NA))`		`NA`
`NA < 3`		`NA`
`NA == 3`		`NA`
`NA == NA`		

Expression	Guess	Actual
`5 + NA`		NA
`mean(c(5, NA))`		NA
`NA < 3`		NA
`NA == 3`		NA
`NA == NA`		NA

NA Behavior

Missing values propagate

Use `is.na()` to check for missing values

```
a <- c(1, NA)
a == NA
# NA NA
is.na(a)
# FALSE TRUE
```

Many functions (e.g. `sum` and `mean`) have `na.rm` argument to remove missing values prior to computation.

na.rm

Many functions (e.g. `sum` and `mean`) have `na.rm` argument to remove missing values prior to computation.

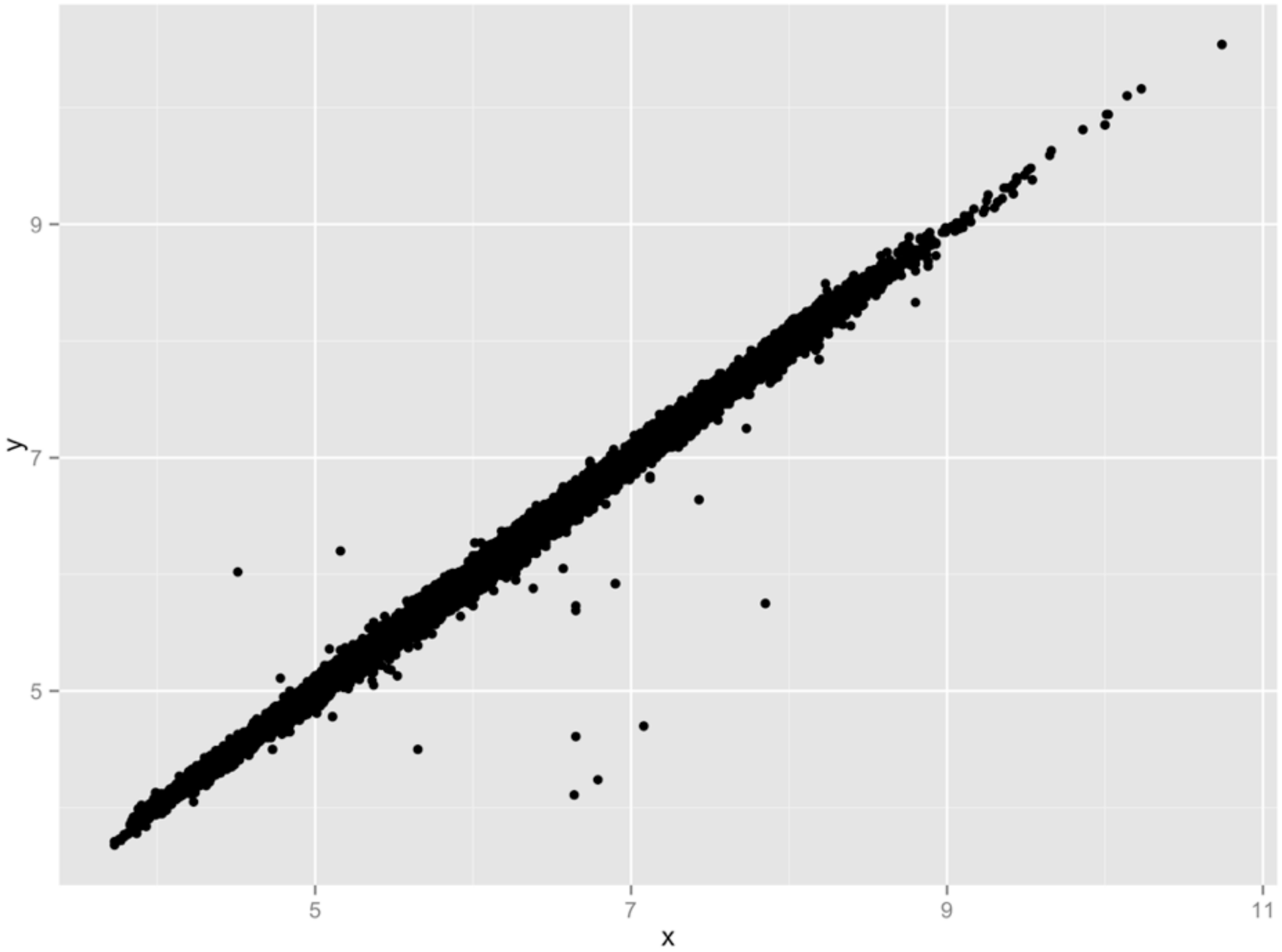
```
b <- c(1, 2, 3, 4 NA)

sum(b)
# NA

sum(b, na.rm = TRUE)
# 10
```

Assignment

You can use subset notation with <- to change individual values within an object

```
summary(diamonds$x)

diamonds$x[diamonds$x == 0]
diamonds$x[diamonds$x == 0] <- NA

summary(diamonds$x)
```

```
summary(diamonds$y)

diamonds$y[diamonds$y == 0] <- NA

y_big <- diamonds$y > 20
diamonds$y[y_big] <- NA

summary(diamonds$y)
```




```
qplot(x, y, data = diamonds)
```