

# Iteration with



**Open 06-Iteration.Rmd**

# Your Turn 1

```
mod <- lm(price ~ carat + cut + color + clarity,  
          data = diamonds)  
View(mod)
```

What kind of object is mod? Why are models stored as this kind of object?

# Partial output of fitting the linear model

```
lm(price ~ carat + cut + color + clarity, data = diamonds)
```

```
$rank  
[1] 19
```

```
$assign  
[1] 0 1 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 4 4
```

```
$contrasts  
$contrasts$cut  
[1] "contr.poly"
```

```
$contrasts$color  
[1] "contr.poly"
```

```
$contrasts$clarity  
[1] "contr.poly"
```

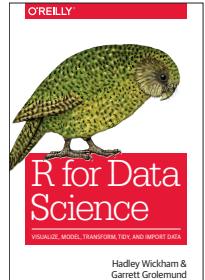
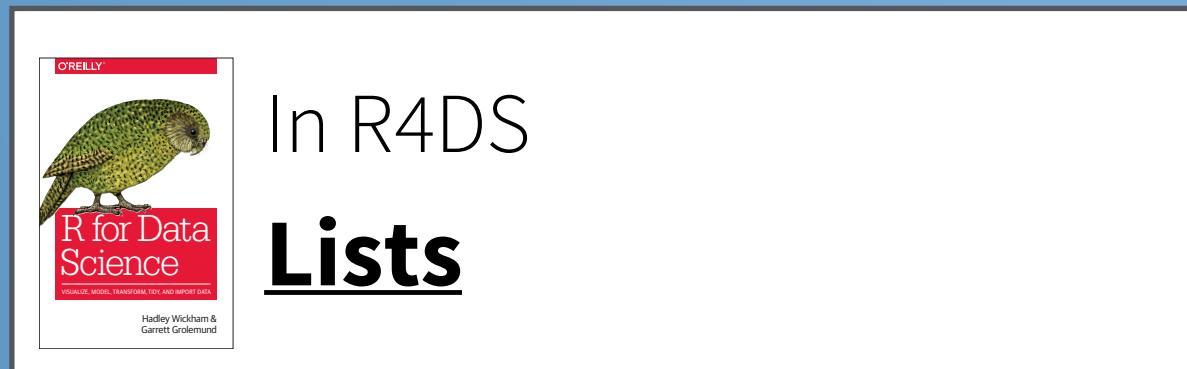
```
$call  
lm(formula = price ~ carat + cut + color + clarity, data = diamonds)
```



```
$rank  
[1] 19  
  
$assign  
[1] 0 1 2 2 2 2 3 3 3 3 3 3 4 4 4 4 4 4 4  
  
$contrasts  
$contrasts$cut  
[1] "contr.poly"  
  
$contrasts$color  
[1] "contr.poly"  
  
$contrasts$clarity  
[1] "contr.poly"  
  
$call  
lm(formula = price ~ carat + cut + color + clarity, data = diamonds)
```



# Lists



In R4DS

**Lists**

# Consider

What is the difference between an atomic vector and a list?

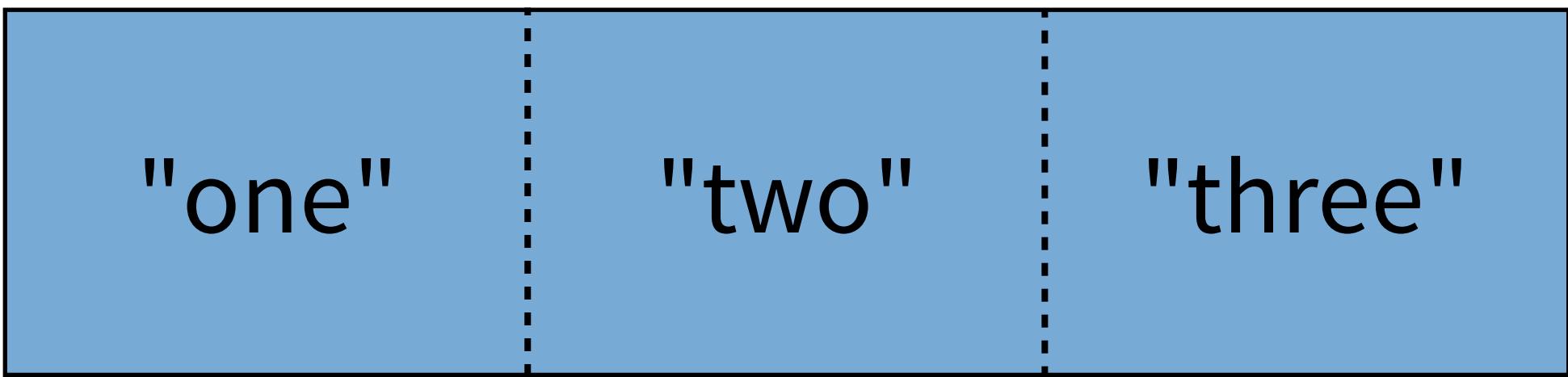
Atomic Vector



type



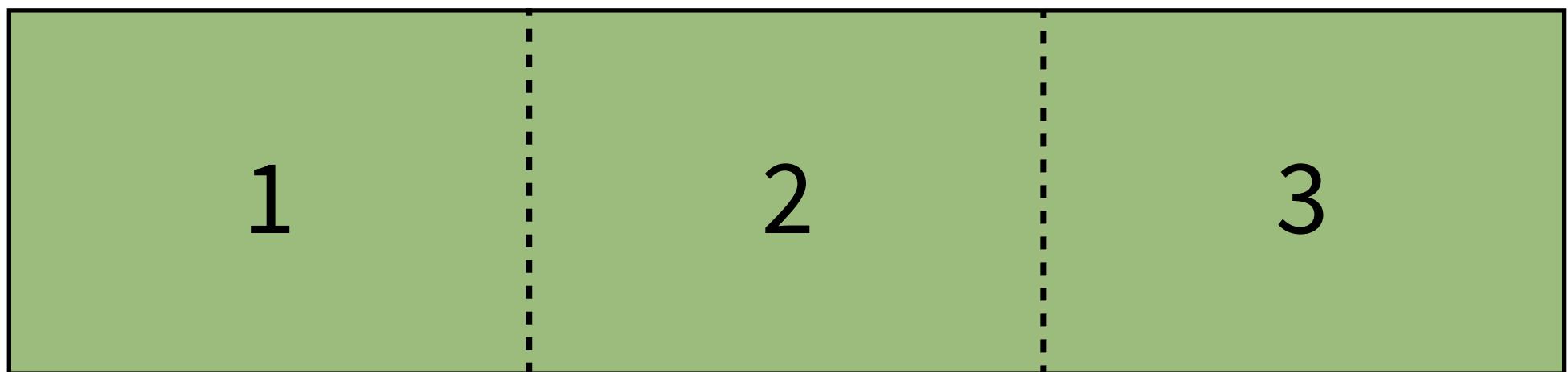
Atomic Vector



character



Atomic Vector



double



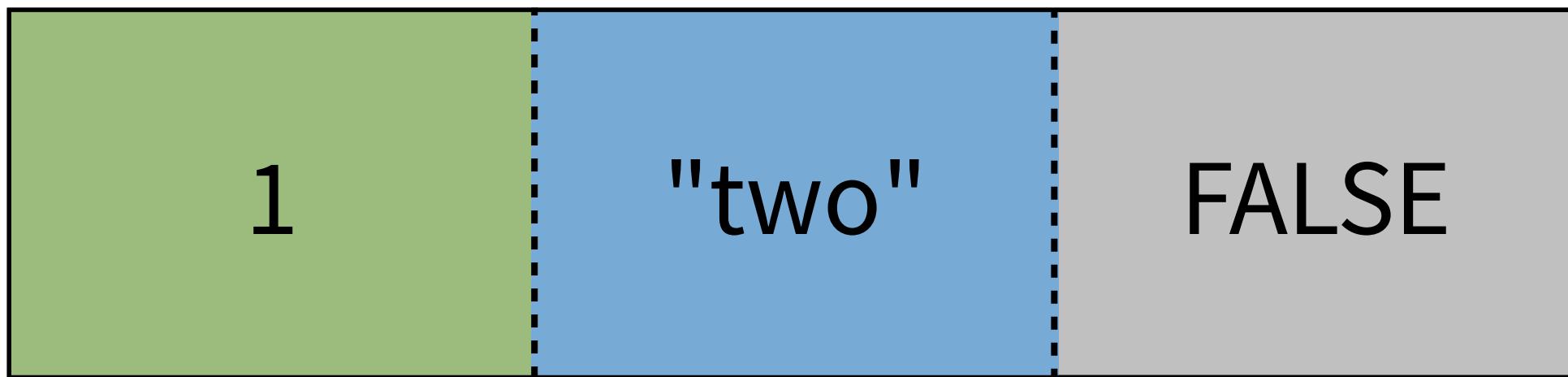
Atomic Vector

TRUE	FALSE	FALSE
------	-------	-------

logical



Atomic Vector



?



Atomic Vector

"1"	"two"	"FALSE"
-----	-------	---------

character

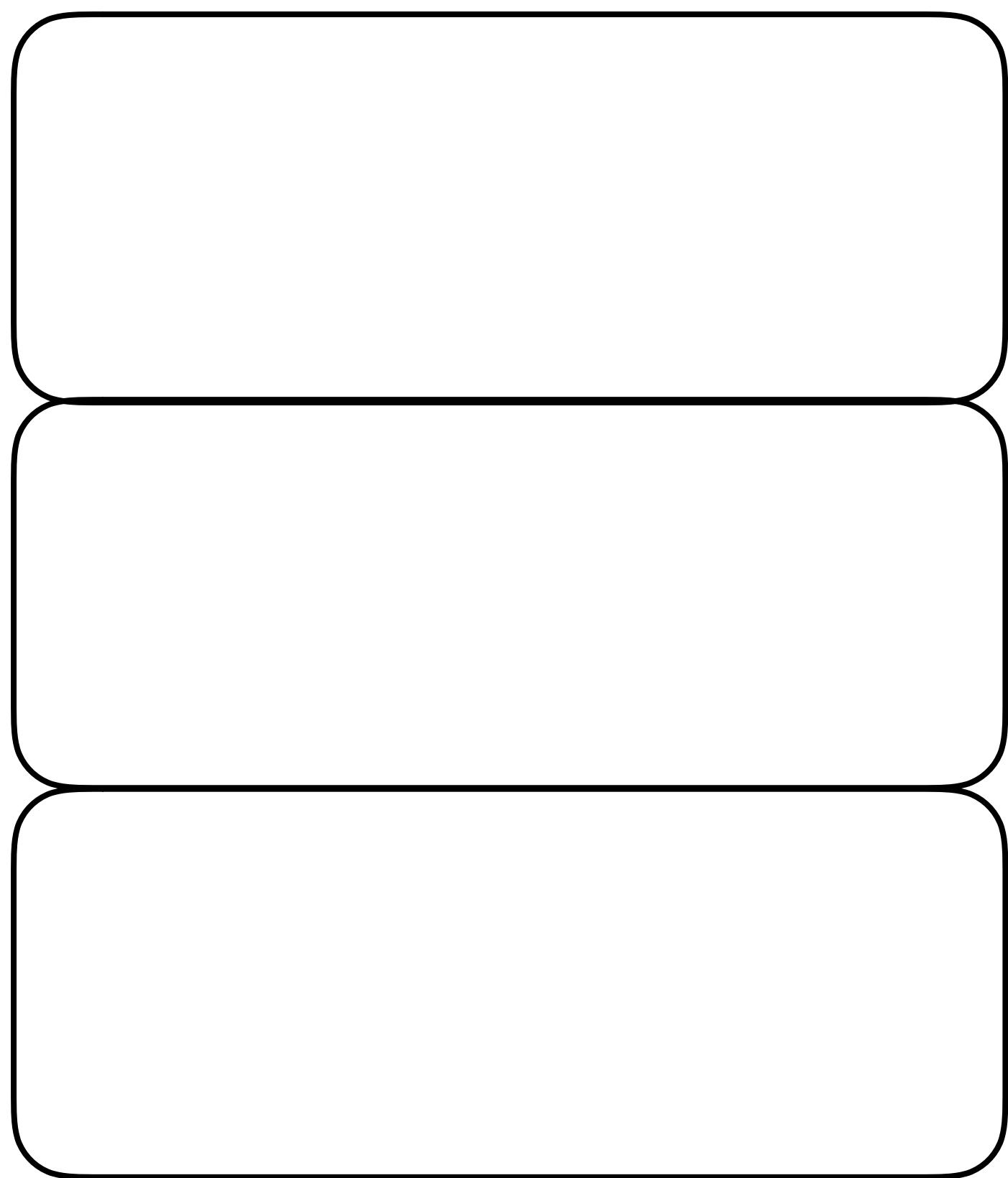


Atomic Vector

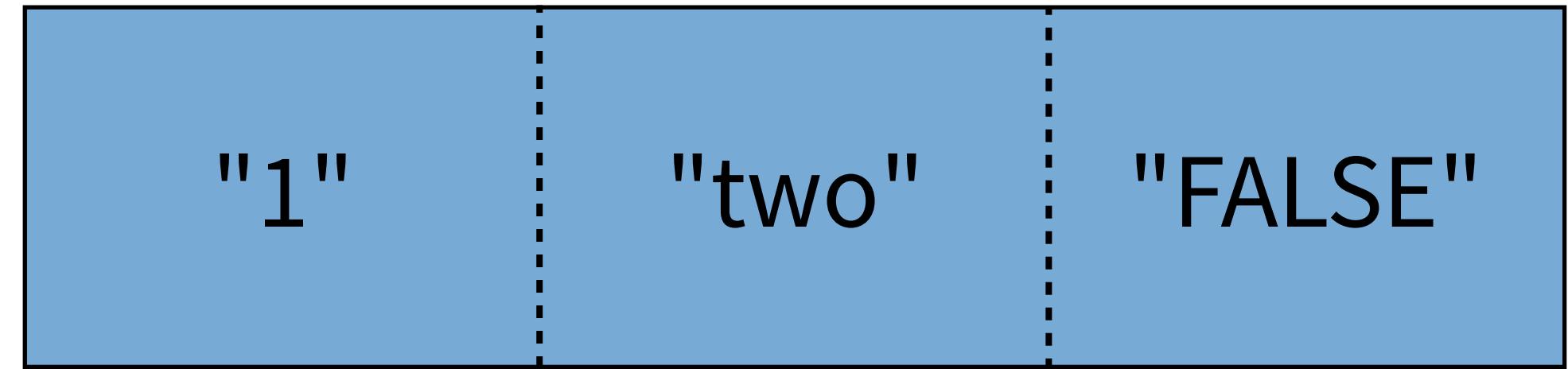


type

List

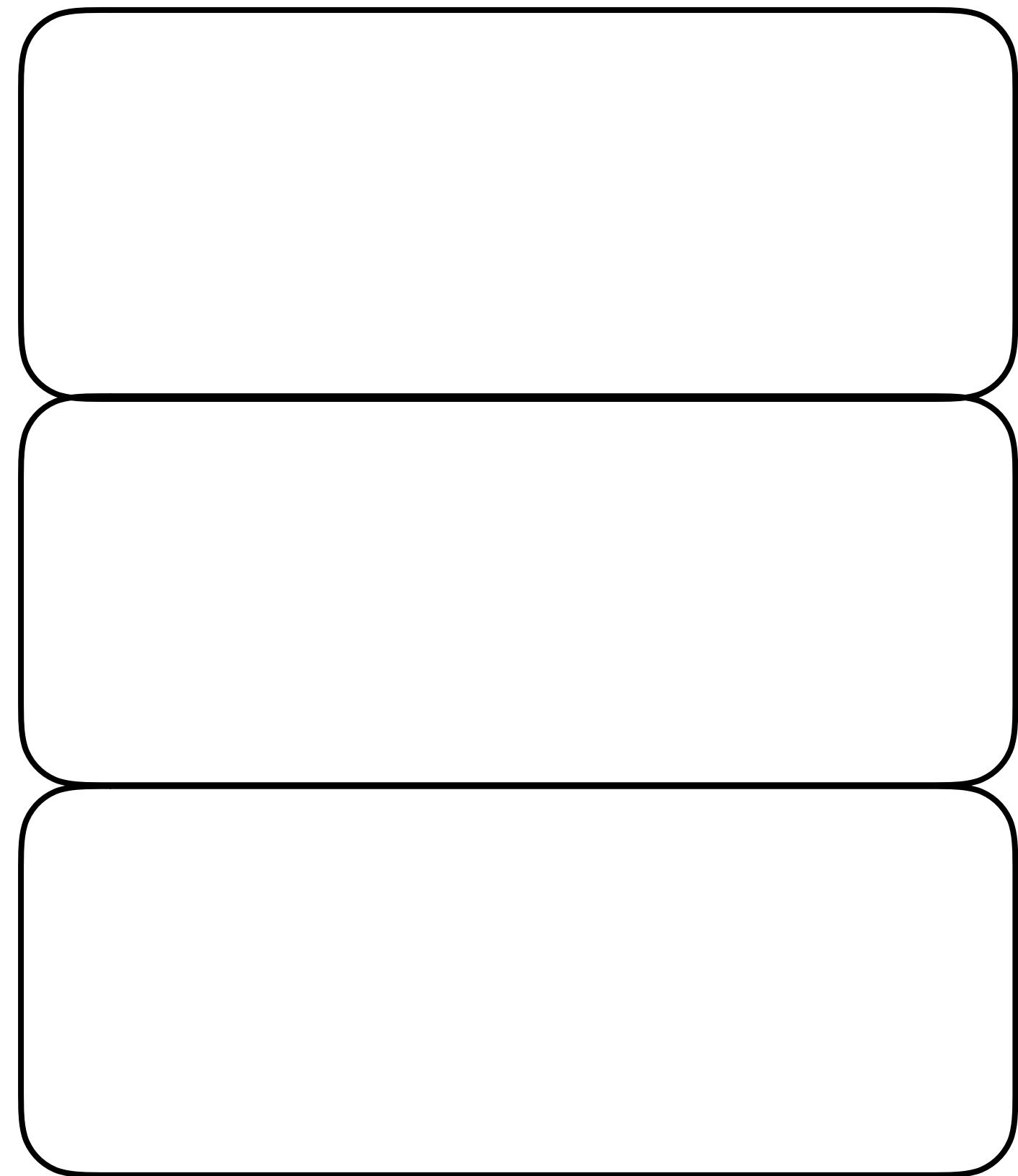


Atomic Vector

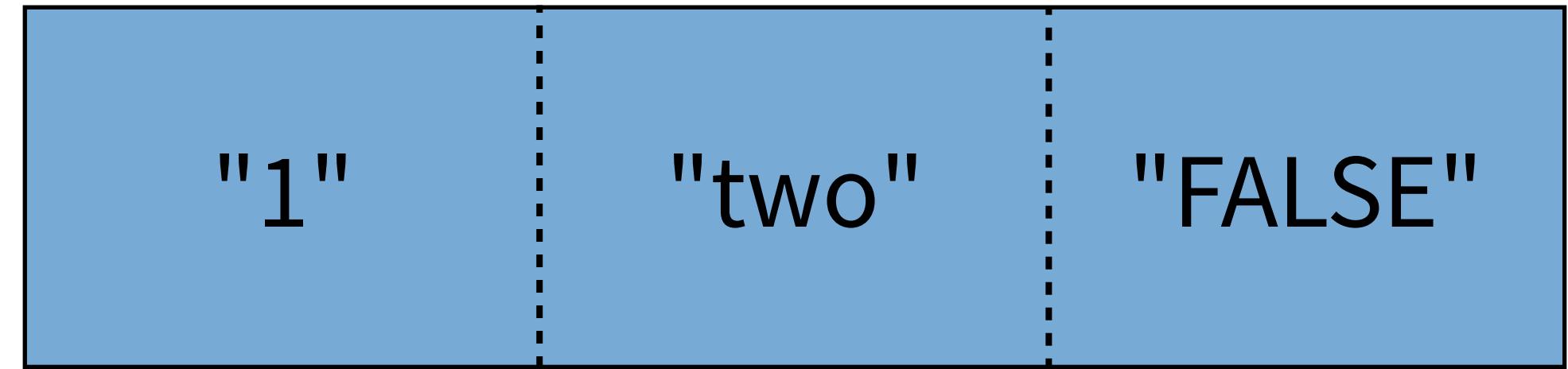


character

List

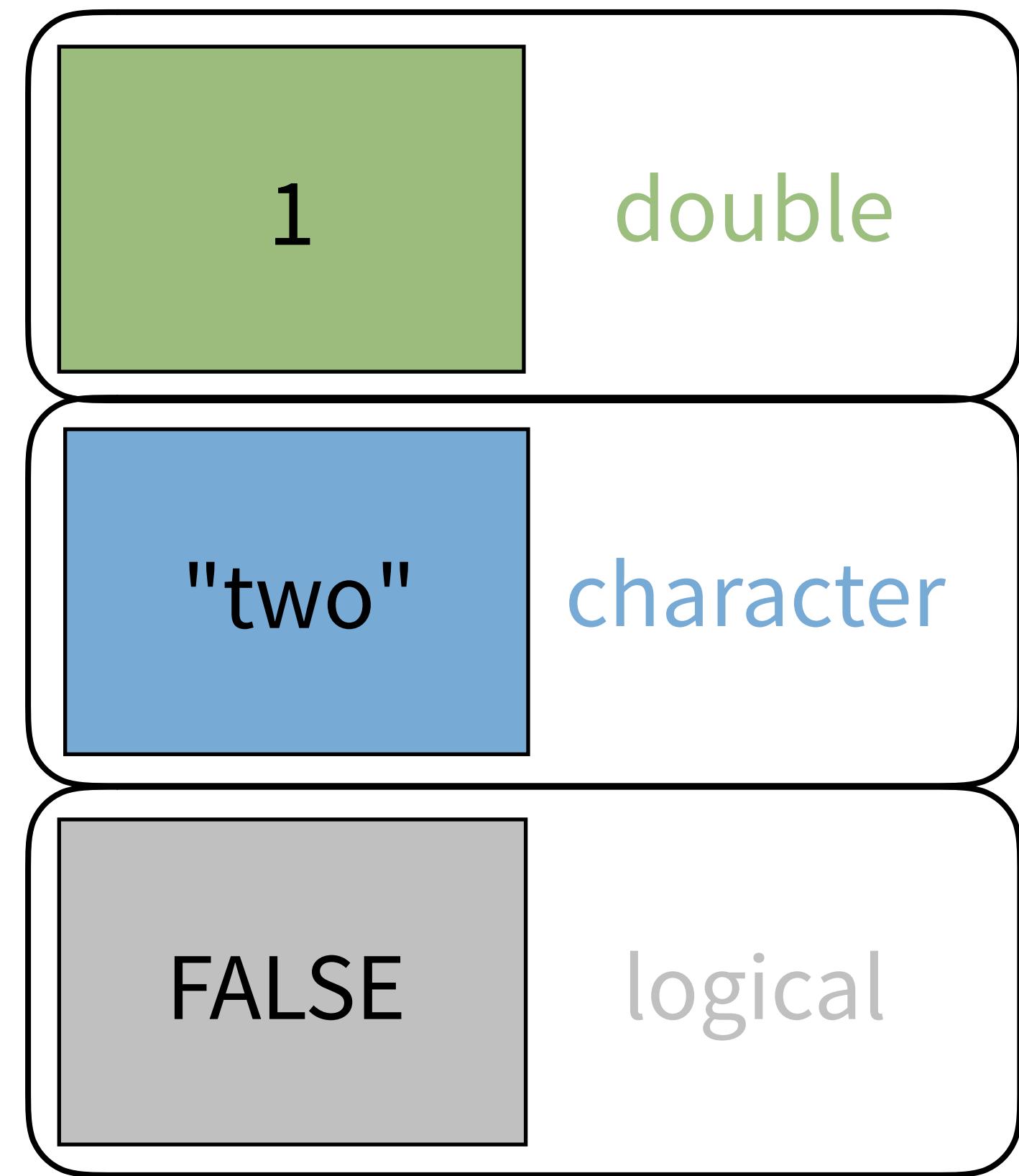


Atomic Vector

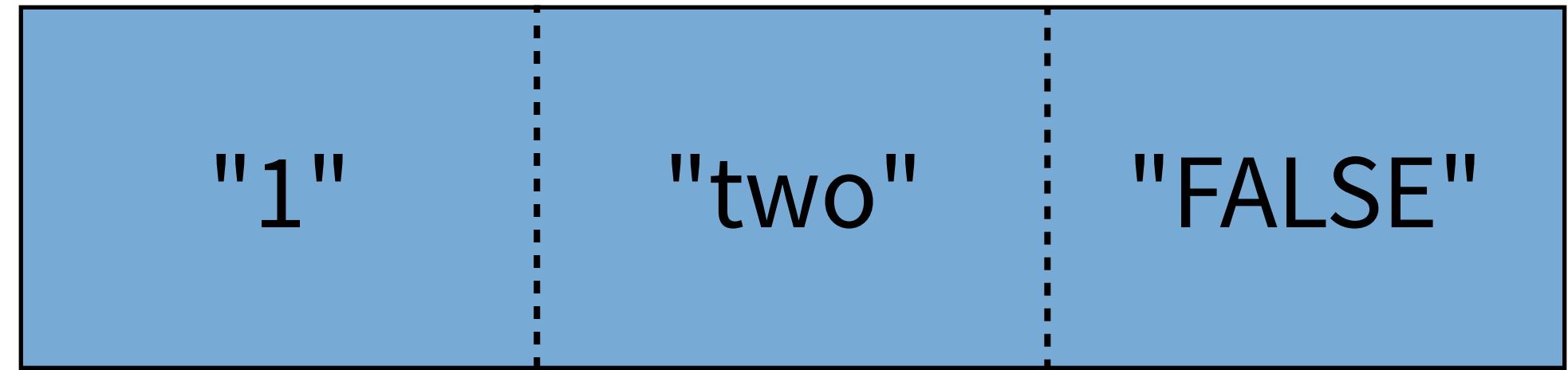


character

List

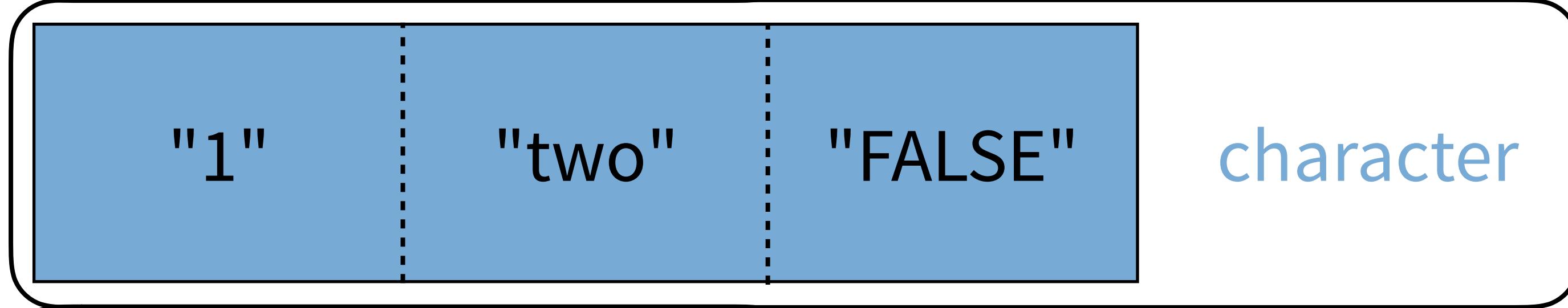
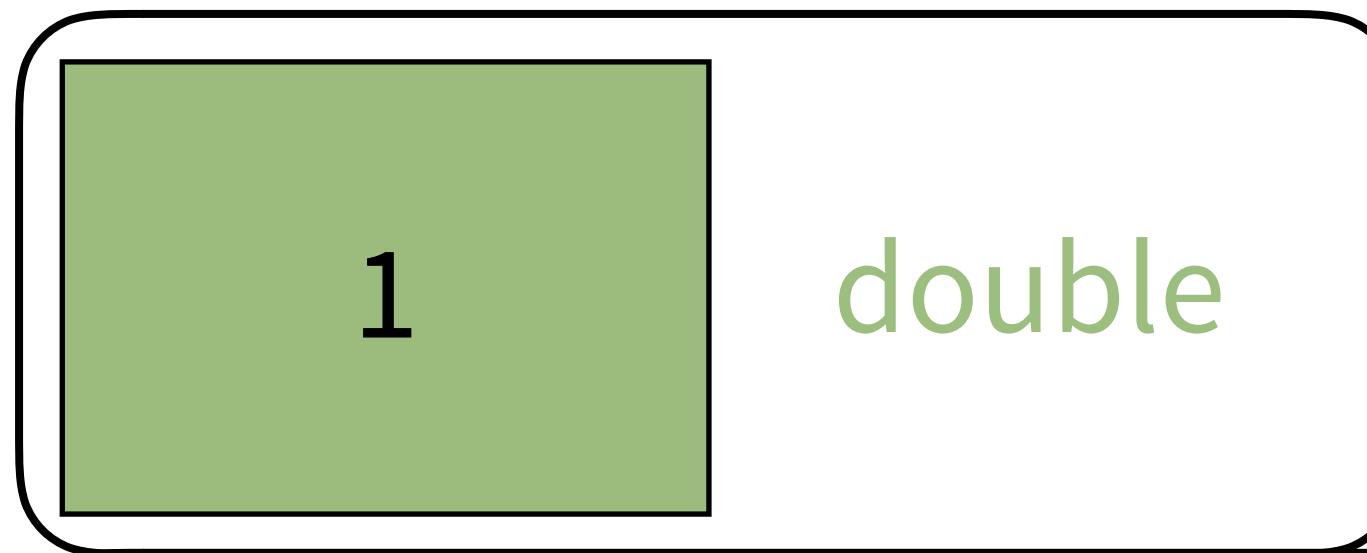


Atomic Vector



character

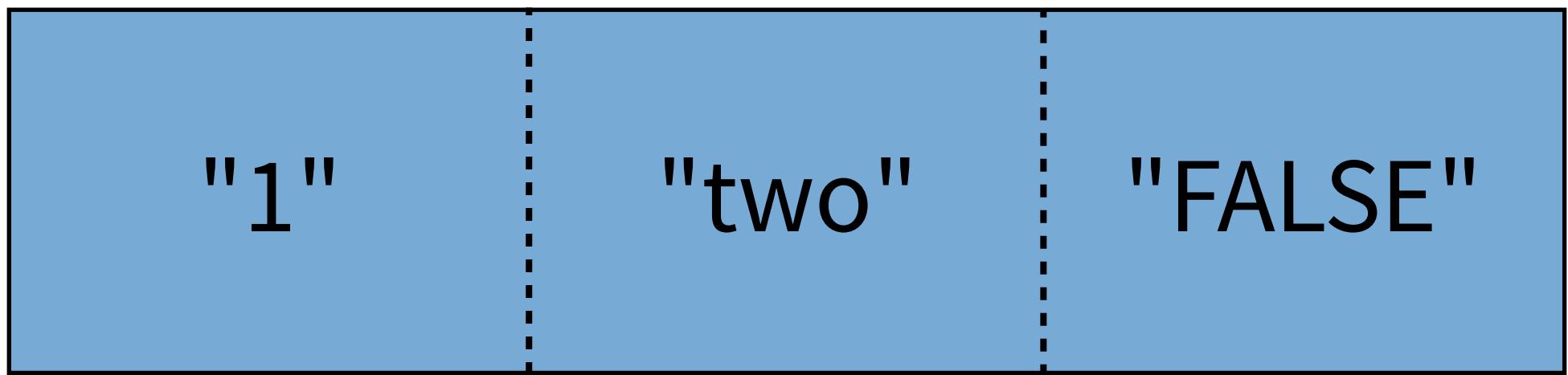
List



character

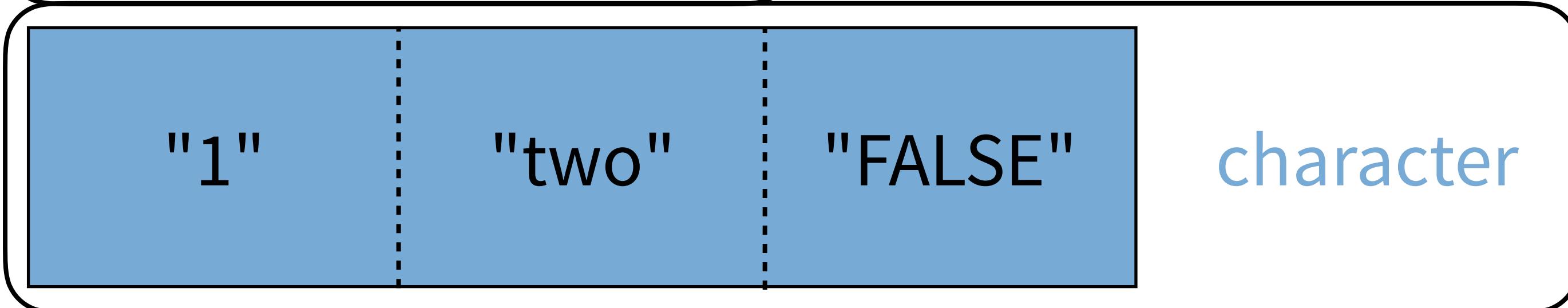
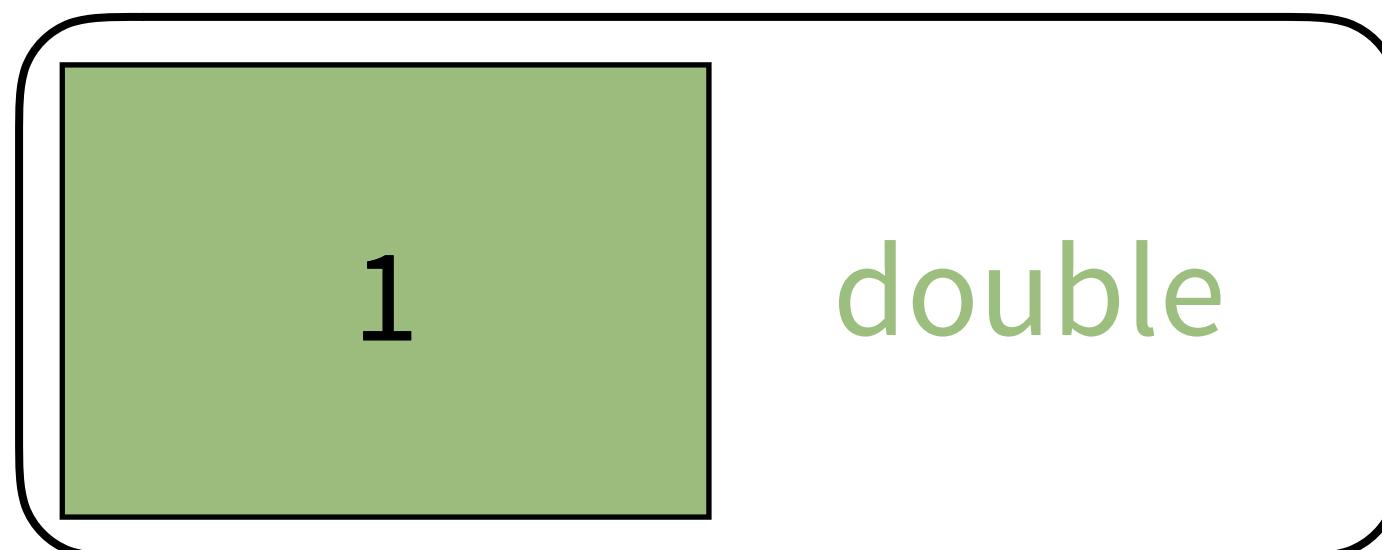


Atomic Vector

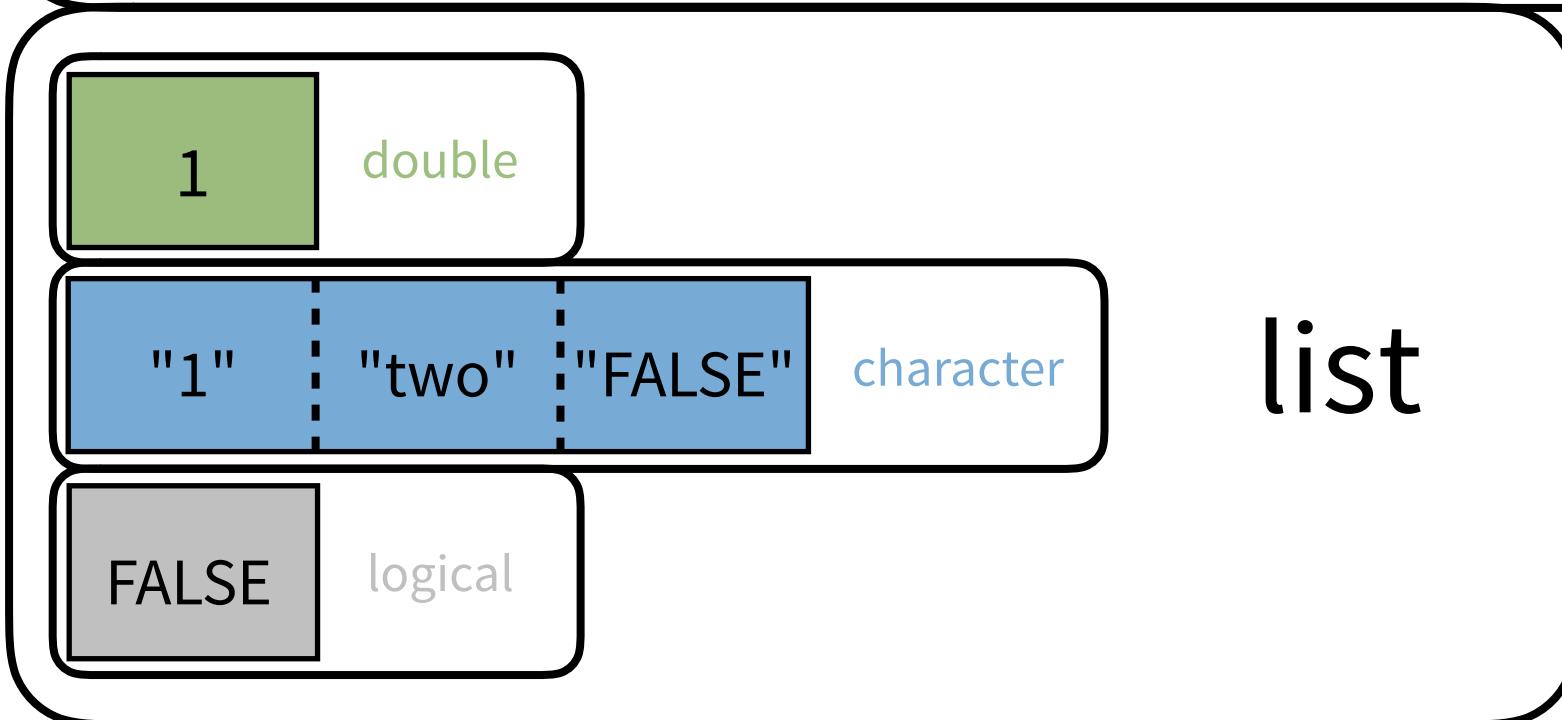


character

List



character



# Your Turn 2

Here is a list:

```
a_list <- list(num = c(8, 9),  
                 log = TRUE,  
                 cha = c("a", "b", "c"))
```

Here are two subsetting commands. Do they return the same values? Run the code chunks to confirm

```
a_list["num"]
```

```
a_list$num
```

```
a_list["num"]
```

```
$num  
[1] 8 9
```

**A list**

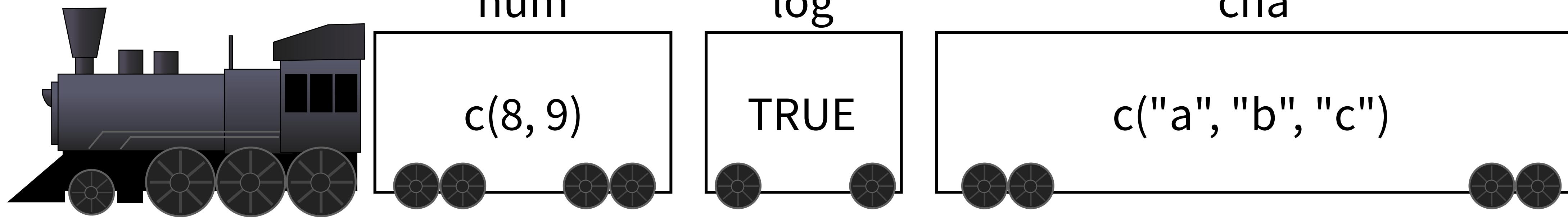
(with one element named num that contains an atomic vector)

```
a_list$num
```

```
[1] 8 9
```

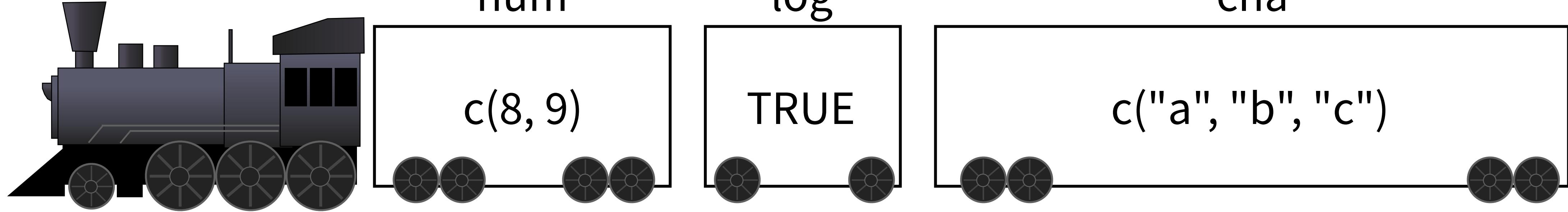
An atomic vector



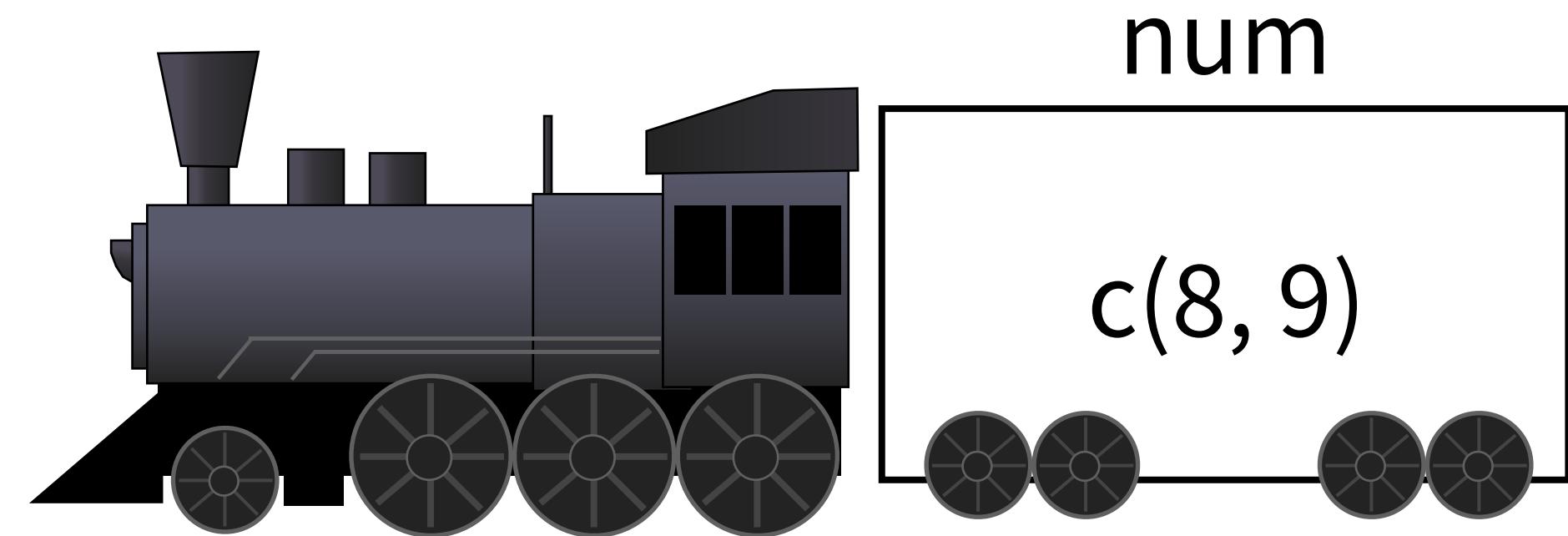


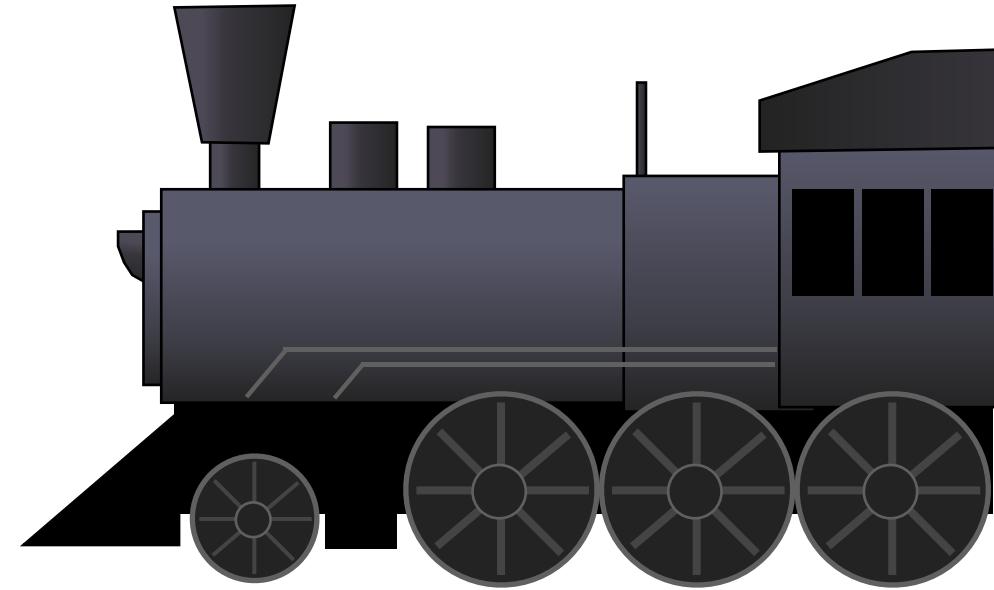
```
lst <- list(num = c(8,9), log = TRUE, cha = c("a", "b", "c"))
```





lst["num"]





num

c(8, 9)

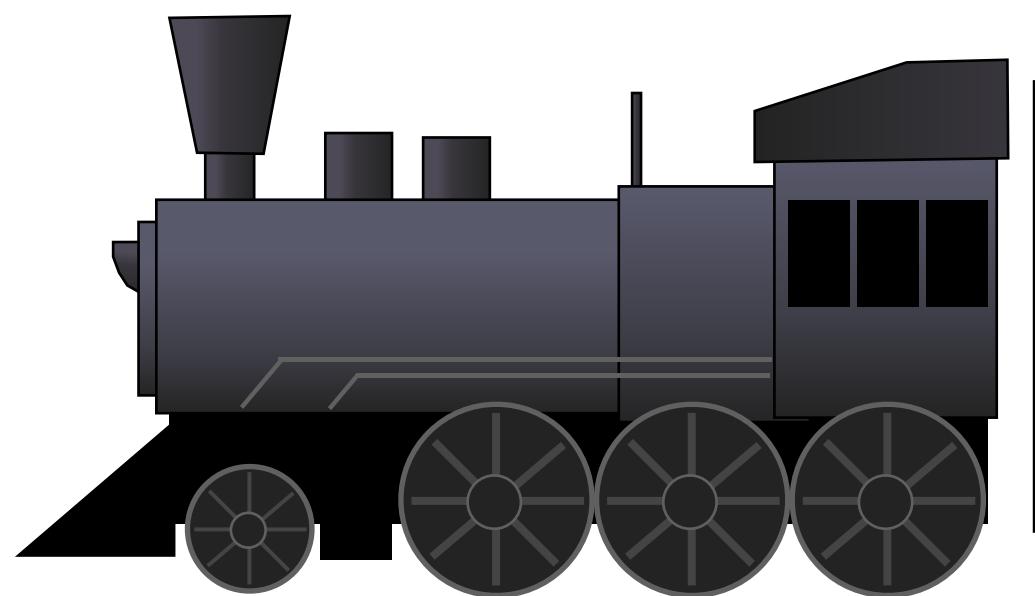
log

TRUE

cha

c("a", "b", "c")

lst["num"]



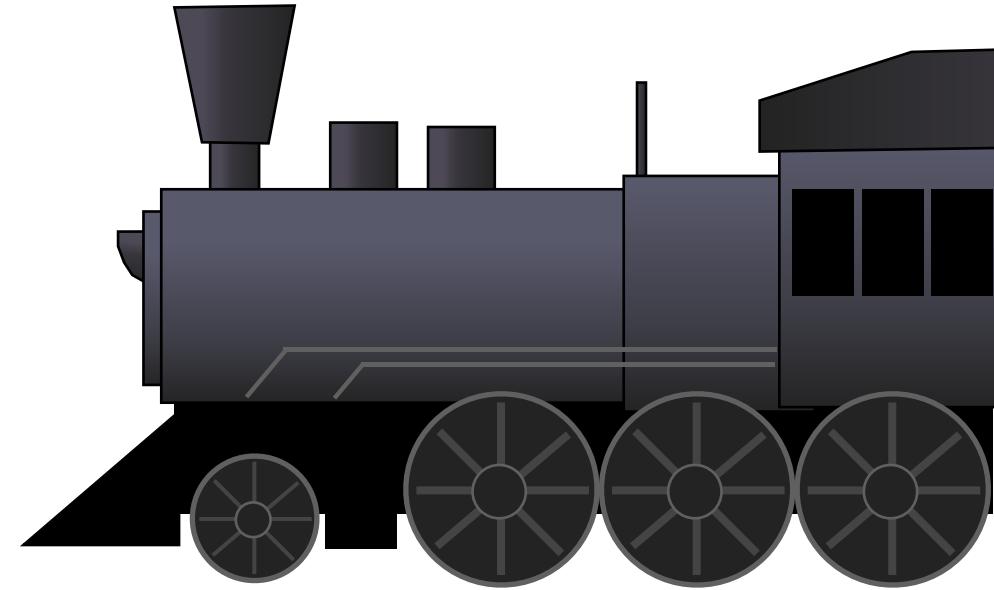
num

c(8, 9)

lst[["num"]]

c(8, 9)





num

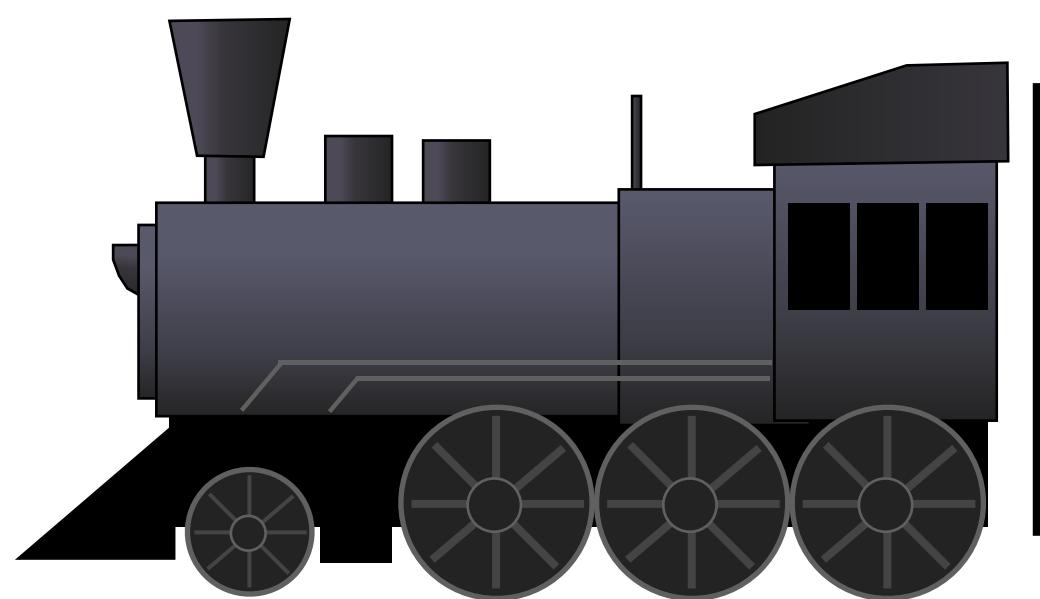
c(8, 9)

log

TRUE

cha

c("a", "b", "c")



num

c(8, 9)

lst["num"]

lst[["num"]]

lst\$num

c(8, 9)

c(8, 9)



# Your Turn 3

What will each of these return? Run the code chunks to confirm.

```
vec <- c(-2, -1, 0, 1, 2)  
abs(vec)
```

```
lst <- list(-2, -1, 0, 1, 2)  
abs(lst)
```

```
vec <- c(-2, -1, 0, 1, 2)  
abs(vec)
```

```
# [1] 2 1 0 1 2
```

```
lst <- list(-2, -1, 0, 1, 2)  
abs(lst)
```

```
# Error in abs(lst) : non-numeric argument to mathematical  
function
```

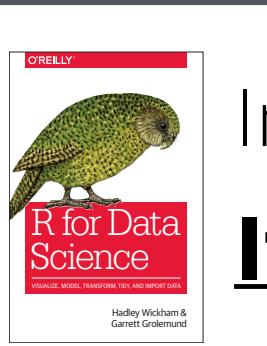
# Take aways

Lists are a useful way to organize data.

But you need to arrange manually for functions to iterate over the elements of a list.



# Iteration

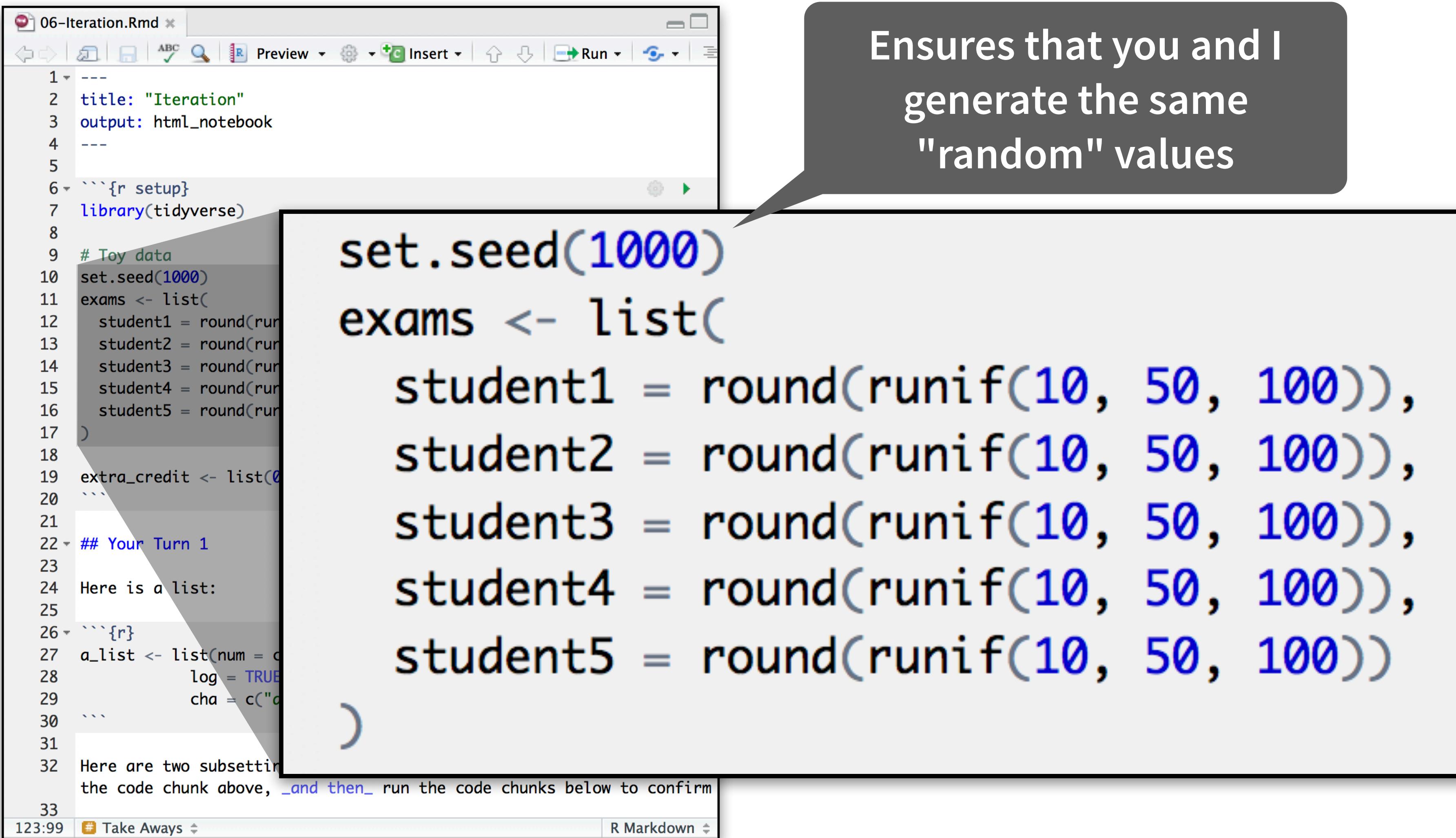


In R4DS

**Iteration**

# Toy data

Suppose we have the exam scores of five students...



Ensures that you and I generate the same "random" values

```
1 ---  
2 title: "Iteration"  
3 output: html_notebook  
4 ---  
5  
6 ```{r setup}  
7 library(tidyverse)  
8  
9 # Toy data  
10 set.seed(1000)  
11 exams <- list(  
12   student1 = round(runif(10, 50, 100)),  
13   student2 = round(runif(10, 50, 100)),  
14   student3 = round(runif(10, 50, 100)),  
15   student4 = round(runif(10, 50, 100)),  
16   student5 = round(runif(10, 50, 100))  
17 )  
18  
19 extra_credit <- list()  
20 ...  
21  
22 ## Your Turn 1  
23  
24 Here is a list:  
25  
26 ```{r}  
27 a_list <- list(num = c(1, 2, 3),  
28                 log = TRUE,  
29                 cha = c("cat", "dog"))  
30 ...  
31  
32 Here are two subsetting  
the code chunk above, and then run the code chunks below to confirm  
33
```

123:99 | # Take Aways | R Markdown



Suppose we have the exam scores of five students...

exams

\$student1

```
[1] 66 88 56 85 76 53 87 79 61 63
```

\$student2

```
[1] 67 88 66 93 88 54 75 82 54 79
```

\$student3

```
[1] 58 90 64 54 77 84 73 91 55 56
```

\$student4

```
[1] 78 52 78 98 75 85 51 89 79 66
```

\$student5

```
[1] 100 77 55 82 90 86 85 78 63 75
```

How can we compute the mean grade for each student?



# How could we compute the average grade?

```
mean(exams)
```



```
argument is not numeric or logical: returning NA[1] NA
```



How could we compute the average grade?

```
list(student1 = mean(exams$student1),  
     student2 = mean(exams$student2),  
     student3 = mean(exams$student3),  
     student4 = mean(exams$student4),  
     student5 = mean(exams$student5))
```

\$student1  
[1] 71.4

\$student2  
[1] 74.6

\$student3  
[1] 70.2

\$student4  
[1] 75.1

\$student5  
[1] 79.1

Is there a better way?  


# purrr



# purrr



Functions for iteration.

```
library(purrr)
```



# Your Turn 4

Run the code in the chunk. What does it do?

```
map(exams, mean)
```



```
exams %>% map(mean)
```

```
$student1
```

```
[1] 71.4
```

```
$student2
```

```
[1] 74.6
```

```
$student3
```

```
[1] 70.2
```

```
$student4
```

```
[1] 75.1
```

```
$student5
```

```
[1] 79.1
```



# map()

Applies a function to every element of a list.  
Returns the results as a list.

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

A list

A function to apply to  
each element of the list  
(element become first  
argument)

Other  
arguments to  
pass to the  
function



# map()

map( exams , mean, ...)

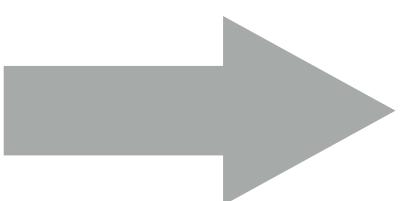
\$student1

\$student2

\$student3

\$student4

\$student5



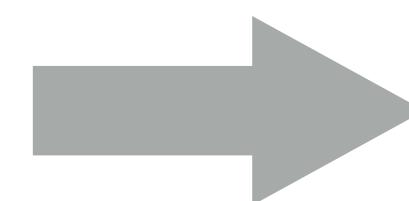
mean(\$student1, ...)

mean(\$student2, ...)

mean(\$student3, ...)

mean(\$student4, ...)

mean(\$student5, ...)



list

result1

result2

result3

result4

result5



# Your Turn 5

Calculate the variance (`var()`) of each student's exam grades.



```
exams %>% map(var)
```

\$student1

[1] 174.0444

\$student2

[1] 194.7111

\$student3

[1] 216.8444

\$student4

[1] 227.2111

\$student5

[1] 167.6556



# map functions

<b>function</b>	<b>returns results as</b>
map()	list
map_chr()	character vector
map_dbl()	double vector (numeric)
map_int()	integer vector
map_lgl()	logical vector
map_df()	data frame



# map\_dbl()

If we want the output as a vector:

```
exams %>%  
  map_dbl(mean)
```

```
## student1   student2   student3   student4   student5  
## 71.34850   74.60950   70.21575   75.30758   79.06386
```

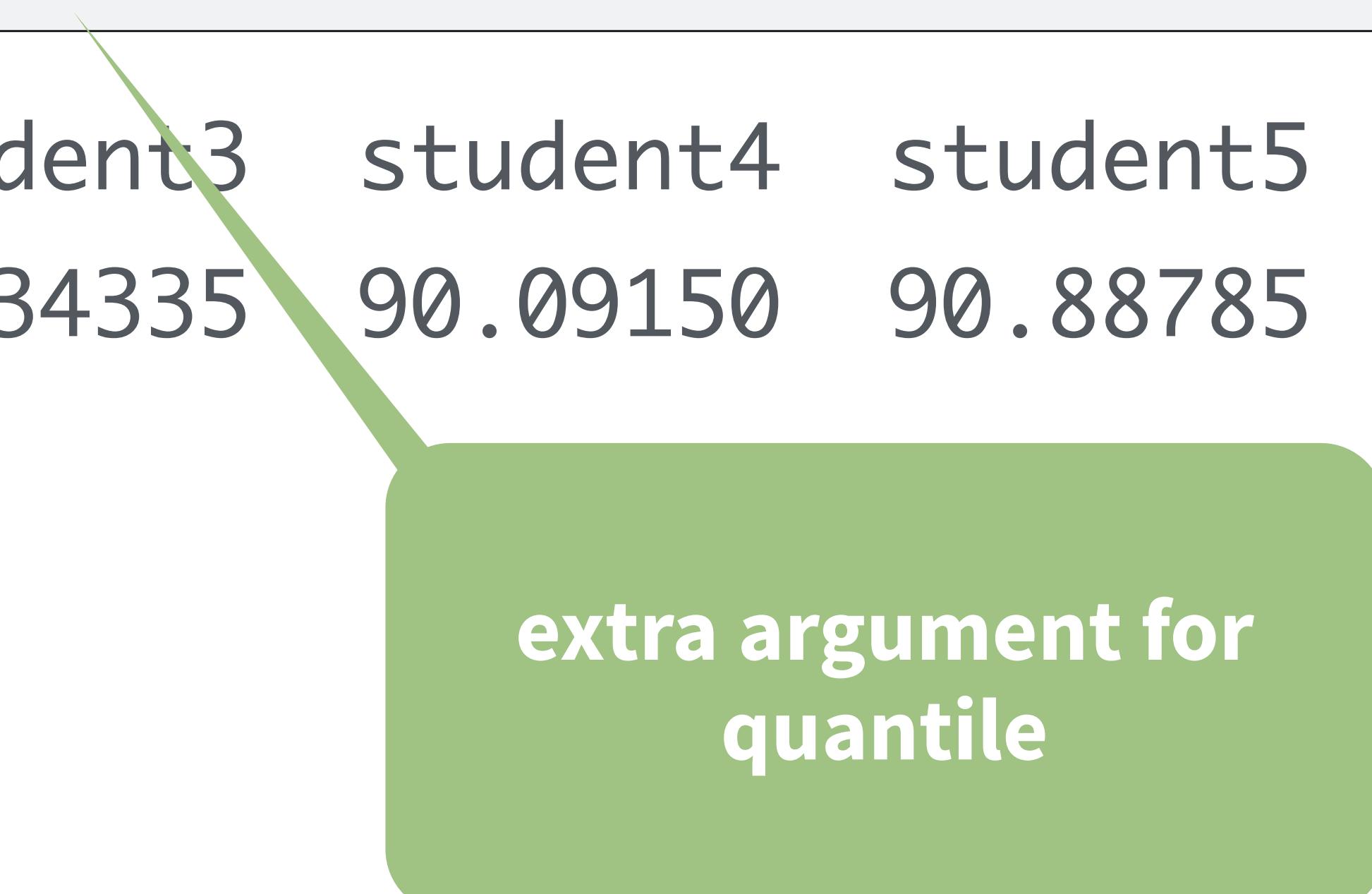


# extra arguments

What if the grade was the 90th percentile score?

```
exams %>%  
  map_db1(quantile, prob = 0.9)
```

```
## student1   student2   student3   student4   student5  
## 87.03640   88.71630   90.34335   90.09150   90.88785
```



extra argument for  
quantile



# map\_lgl()

How about a participation grade?

```
exams %>%  
  map(length) %>%  
  map_lgl(all.equal, 10)
```

```
## student1  student2  student3  student4  student5  
##      TRUE      TRUE      TRUE      TRUE      TRUE
```



# Your Turn 6

Calculate the max grade (**max()**)for each student. Return the result as a vector.



```
exams %>%  
  map_db1(max)
```

```
## student1  student2  student3  student4  student5  
##      88       93       91       98      100
```



# Consider

What if what we want to do is not a function?

For example, what if the final grade is the mean exam score **after we drop the lowest score?**

# Consider

What if what we want to do is not a function?

For example, what if the final grade is the mean exam score **after we drop the lowest score?**

A: Write a function.

# Functions



# Functions (very basics)

1. Write code that solves the problem for a real object

```
vec <- exams$student1
```



# To write a function (very basics)

1. Write code that solves the problem for a real object

```
vec <- exams$student1  
(sum(vec) - min(vec)) / (length(vec) - 1)  
# 73.34424
```



Note: this code does the same thing no matter what vec is.  
But it is a bother to redefine vec each time we use the code.

```
vec <- exams$student1  
  (sum(vec) - min(vec)) / (length(vec) - 1)  
vec <- exams$student2  
  (sum(vec) - min(vec)) / (length(vec) - 1)  
vec <- exams$student3  
  (sum(vec) - min(vec)) / (length(vec) - 1)  
vec <- exams$student4  
  (sum(vec) - min(vec)) / (length(vec) - 1)  
vec <- exams$student5  
  (sum(vec) - min(vec)) / (length(vec) - 1)
```



# To write a function (very basics)

1. Write code that solves the problem for a real object
2. Wrap the code in **function(){}**  to save it

```
vec <- exams[[1]]  
grade <- function() {  
  (sum(vec) - min(vec)) / (length(vec) - 1)  
}
```



# To write a function (very basics)

1. Write code that solves the problem for a real object
2. Wrap the code in `function(){}` to save it
3. Add the name of the real object as the function argument

```
vec <- exams[[1]]  
grade <- function(vec) {  
  (sum(vec) - min(vec)) / (length(vec) - 1)  
}
```



# To write a function (very basics)

1. Write code that solves the problem for a real object
2. Wrap the code in `function(){}` to save it
3. Add the name of the real object as the function argument
4. To run the function, call the object followed by parentheses.  
Supply new values to use for each of the arguments.

```
vec <- exams[[1]]  
  
grade <- function(vec) {  
  (sum(vec) - min(vec)) / (length(vec) - 1)  
}  
  
grade(exams[[2]]) # 76.93898
```



```
grade <- function(vec) {  
  (sum(vec) - min(vec)) / (length(vec) - 1)  
}  
  
exams %>%  
  map_dbl(grade)  
## student1  student2  student3  student4  student5  
## 73.34424  76.93898  72.06320  78.00649  81.68257
```



```
grade <- function(x) {  
  (sum(x) - min(x)) / (length(x) - 1)  
}  
  
exams %>%  
  map_dbl(grade)  
## student1  student2  student3  student4  student5  
## 73.34424  76.93898  72.06320  78.00649  81.68257
```



```
grade <- function(x) (sum(x) - min(x)) / (length(x) - 1)  
exams %>%  
  map_dbl(grade)
```

```
## student1  student2  student3  student4  student5  
## 73.34424  76.93898  72.06320  78.00649  81.68257
```



```
grade <- function(x) (sum(x) - min(x)) / (length(x) - 1)

exams %>%
  map_dbl(function(x) (sum(x) - min(x)) / (length(x) - 1))
## student1  student2  student3  student4  student5
## 73.34424  76.93898  72.06320  78.00649  81.68257
```



# Your Turn 7

```
recall | grade <- function(x) (sum(x) - min(x)) / (length(x) - 1)  
|  
| exams %>%  
|   map_db1(function(x) (sum(x) - min(x)) / (length(x) - 1))  
| ## student1  student2  student3  student4  student5  
| ## 73.34424  76.93898  72.06320  78.00649  81.68257
```

Write a function that counts the best exam twice and then takes the average. Use it to grade all of the students.

1. Write code that solves the problem for a real object
2. Wrap the code in `function(){}` to save it
3. Add the name of the real object as the function argument



Define a new function, and pass in its name

```
double_best <- function(x) {  
  (sum(x) + max(x)) / (length(x) + 1)  
}  
  
exams %>%  
  map_dbl(double_best)  
## student1      student2      student3      student4      student5  
## 72.85703     76.30779     72.12398     77.39862     80.94991
```



# Use an anonymous function

```
exams %>%  
  map_dbl(function(x) (sum(x) + max(x)) / (length(x) + 1))  
  
## student1      student2      student3      student4      student5  
## 72.85703     76.30779     72.12398     77.39862     80.94991
```



Use a purrr ~ (formula) shortcut

```
exams %>%  
  map_db1(~ (sum(.x) + max(.x)) / (length(.x) + 1))  
  
## student1      student2      student3      student4      student5  
## 72.85703     76.30779     72.12398     77.39862     80.94991
```

More on this approach at: <https://github.com/cwickham/purrr-tutorial>



# Consider

What does this return?

```
add_1 <- function(x) x + 1
```

```
add_1(1)
```

# Consider

What does this return?

```
add_1 <- function(x) x + 1
```

```
add_1(1)
```

```
# 2
```

# Consider

What does this return?

```
add_2 <- function(x, y) x + y
```

```
add_2(2, 3)
```

# Consider

What does this return?

```
add_2 <- function(x, y) x + y
```

```
add_2(2, 3)
```

```
# 5
```

If functions can take two arguments, how  
can you pass two lists as the arguments?

# map2()

Applies a function to every element of two lists.  
Returns the results as a list.

```
map2(.x, .y, .f, ...)
```

A list of elements  
to pass to the first  
argument of .f

A list of elements to  
pass to the second  
argument of .f

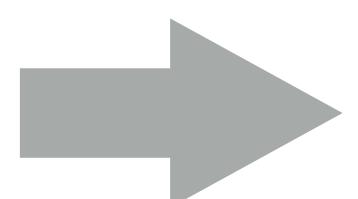


# map2()

map2(

exams	extra
\$student1	\$extra1
\$student2	\$extra2
\$student3	\$extra3
\$student4	\$extra4
\$student5	\$extra5

, fun, ...)



fun(

exams	extra
\$student1	\$extra1
\$student2	\$extra2
\$student3	\$extra3
\$student4	\$extra4
\$student5	\$extra5

, ...)  
fun(

exams	extra
\$student1	\$extra1
\$student2	\$extra2
\$student3	\$extra3
\$student4	\$extra4
\$student5	\$extra5

, ...)  
fun(

exams	extra
\$student1	\$extra1
\$student2	\$extra2
\$student3	\$extra3
\$student4	\$extra4
\$student5	\$extra5

, ...)  
fun(

exams	extra
\$student1	\$extra1
\$student2	\$extra2
\$student3	\$extra3
\$student4	\$extra4
\$student5	\$extra5

, ...)  
fun(

exams	extra
\$student1	\$extra1
\$student2	\$extra2
\$student3	\$extra3
\$student4	\$extra4
\$student5	\$extra5

, ...)

list
result1
result2
result3
result4
result5



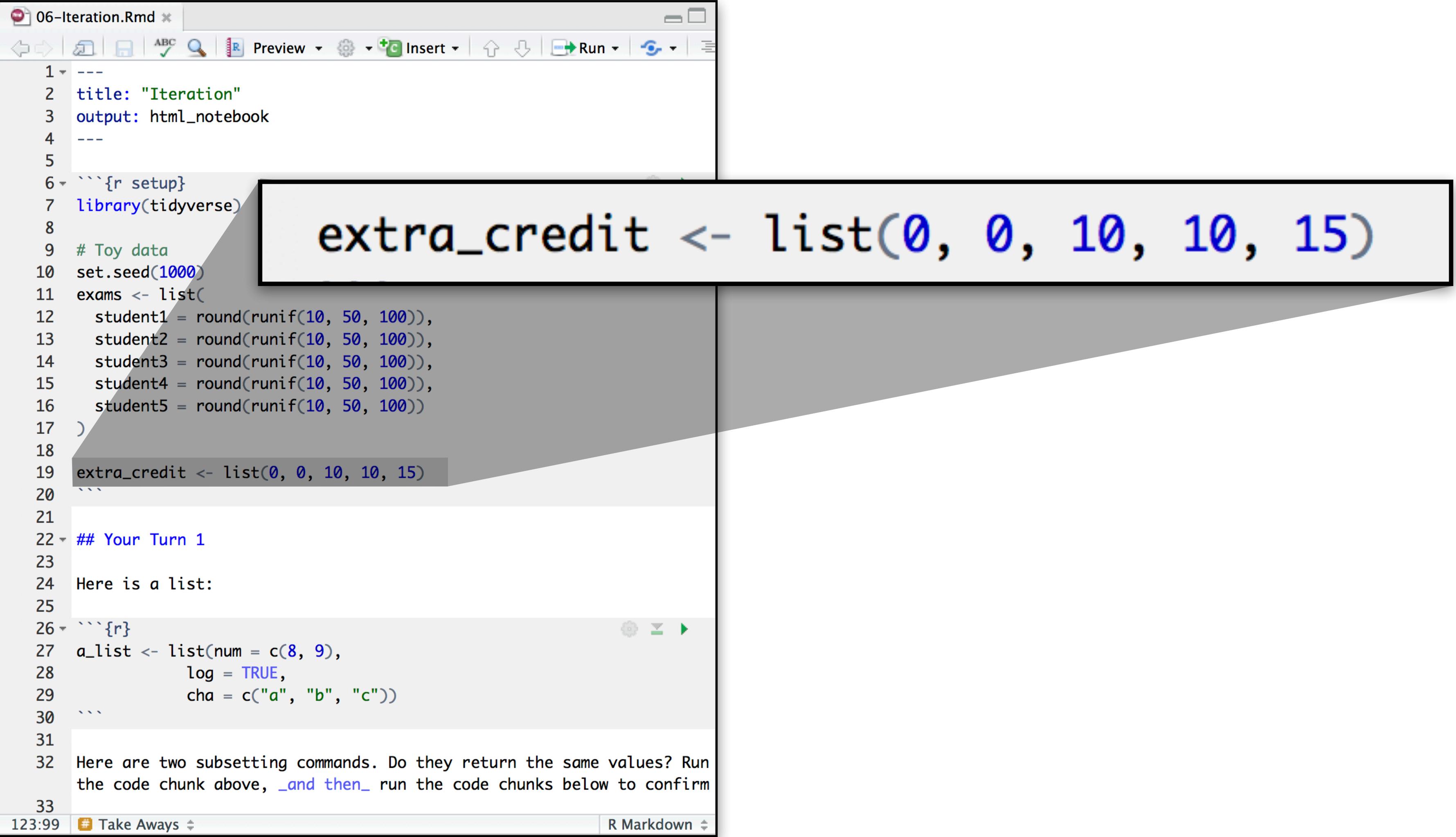
# map functions

<b>single list</b>	<b>two lists</b>	<b>returns results as</b>
map()	map2()	list
map_chr()	map2_chr()	character vector
map_dbl()	map2_dbl()	double vector
map_int()	map2_int()	integer vector
map_lgl()	map2_lgl()	logical vector
map_df()	map2_df()	data frame



# Toy data

Suppose we have extra credit for the five students...



```
1 ---  
2 title: "Iteration"  
3 output: html_notebook  
4 ---  
5  
6 ```{r setup}  
7 library(tidyverse)  
8  
9 # Toy data  
10 set.seed(1000)  
11 exams <- list(  
12   student1 = round(runif(10, 50, 100)),  
13   student2 = round(runif(10, 50, 100)),  
14   student3 = round(runif(10, 50, 100)),  
15   student4 = round(runif(10, 50, 100)),  
16   student5 = round(runif(10, 50, 100))  
17 )  
18  
19 extra_credit <- list(0, 0, 10, 10, 15)  
20 ````  
21  
22 ## Your Turn 1  
23  
24 Here is a list:  
25  
26 ```{r}  
27 a_list <- list(num = c(8, 9),  
28                 log = TRUE,  
29                 cha = c("a", "b", "c"))  
30 ````  
31  
32 Here are two subsetting commands. Do they return the same values? Run  
the code chunk above, and then run the code chunks below to confirm  
33
```



# Your Turn 8

Compute a final grade for each student, where the final grade is the average test score plus any extra credit assigned to the student.  
Return the results as a double (i.e. numeric) vector.



The grades with extra credit...

```
exams %>%  
  map2_db1(extra_credit, function(x, y) mean(x) + y)
```

```
## student1  student2  student3  student4  student5  
##      71.4      74.6      80.2      85.1      94.1
```



# Other mapping functions

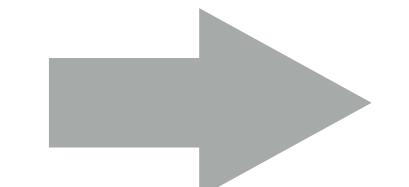


# pmap()

Map over three or more lists. Put the lists into a list of list whose names match argument names in the function.

pmap(list(

exams	extra	more
\$student1	\$extra1	\$more1
\$student2	\$extra2	\$more2
\$student3	\$extra3	\$more3
\$student4	\$extra4	\$more4
\$student5	\$extra5	\$more5



), fun, ...)

exams	extra	more
fun(\$student1,	\$extra1,	\$more1,
fun(\$student2,	\$extra2,	\$more2,
fun(\$student3,	\$extra3,	\$more3,
fun(\$student4,	\$extra4,	\$more4,
fun(\$student5,	\$extra5,	\$more5,



# walk(), walk\_2(), and pwalk()

Versions of map(), map2(), and pmap() that do not return results. These are for triggering side effects (like writing files or saving graphs).



# map and walk functions

<b>single list</b>	<b>two lists</b>	<b>n lists</b>	<b>returns results as</b>
map()	map2()	pmap()	list
map_chr()	map2_chr()	pmap_chr()	character vector
map_dbl()	map2_dbl()	pmap_dbl()	double vector
map_int()	map2_int()	pmap_int()	integer vector
map_lgl()	map2_lgl()	pmap_lgl()	logical vector
map_df()	map2_df()	pmap_df()	data frame
walk()	walk2()	pwalk()	side effect



# Iteration with

