**Table of Contents**

The Art of Benchmarking: p. 1-7

Fine Tuning: Efficient Base R: p. 9-15

Diagnosing Problems: Code Profiling: p. 17-22

Turbo-charged Code: Parallel Programming: p. 24-30

Writing efficient r code

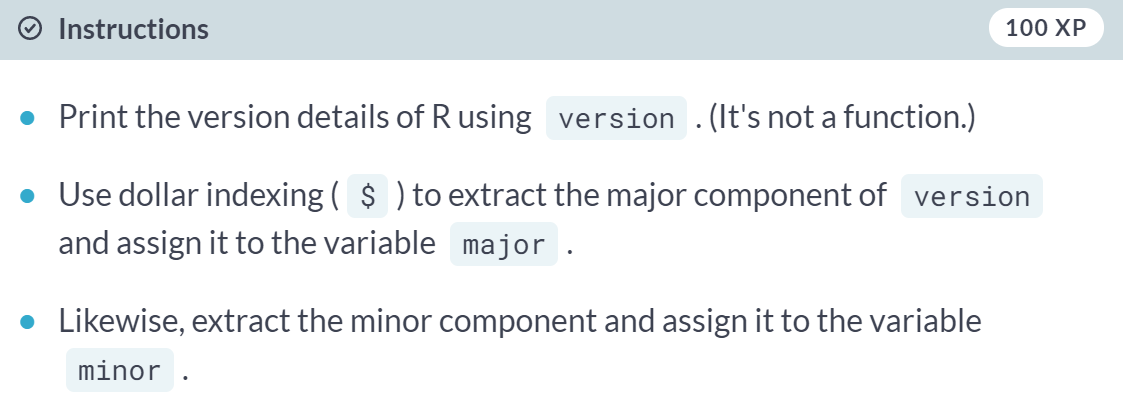
R Programming

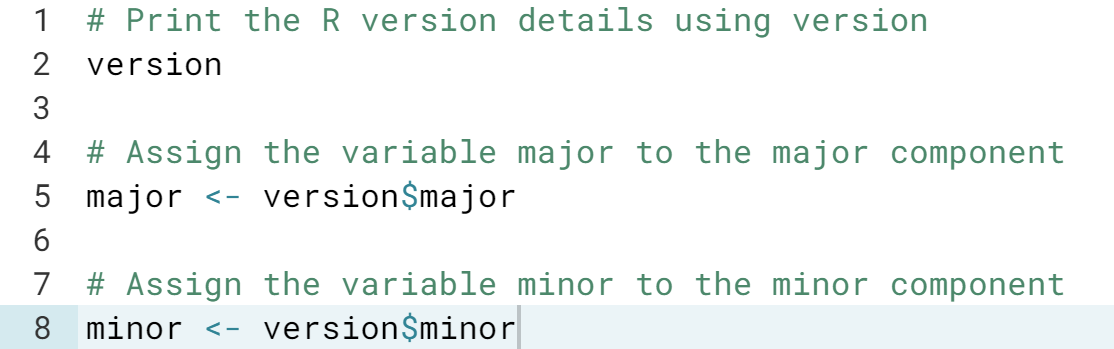
**The Art of Benchmarking**

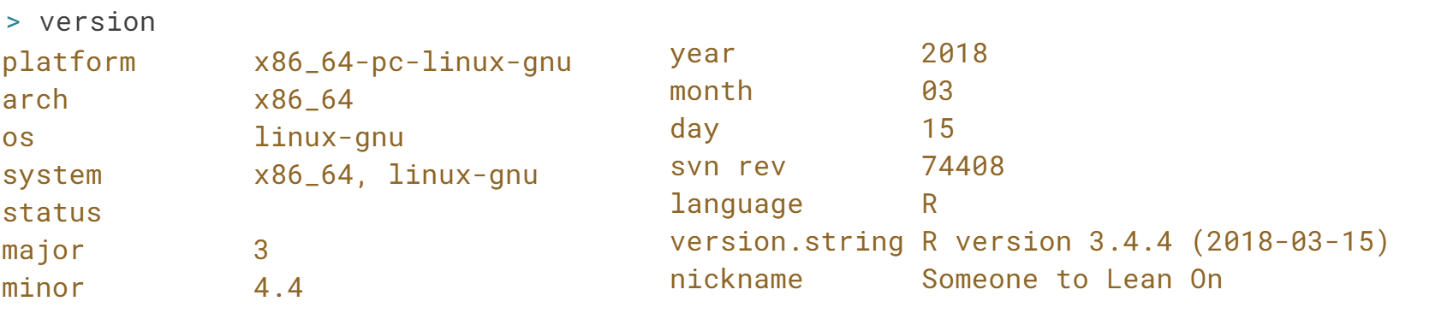
# R version

One of the relatively easy optimizations available is to use an up-to-date version of R. In general, R is very conservative, so upgrading doesn't break existing code. However, a new version will often provide free speed boosts for key functions.

The [**version**](https://www.rdocumentation.org/packages/base/topics/version) command returns a list that contains (among other things) the major and minor version of R currently being used.

****

****

****

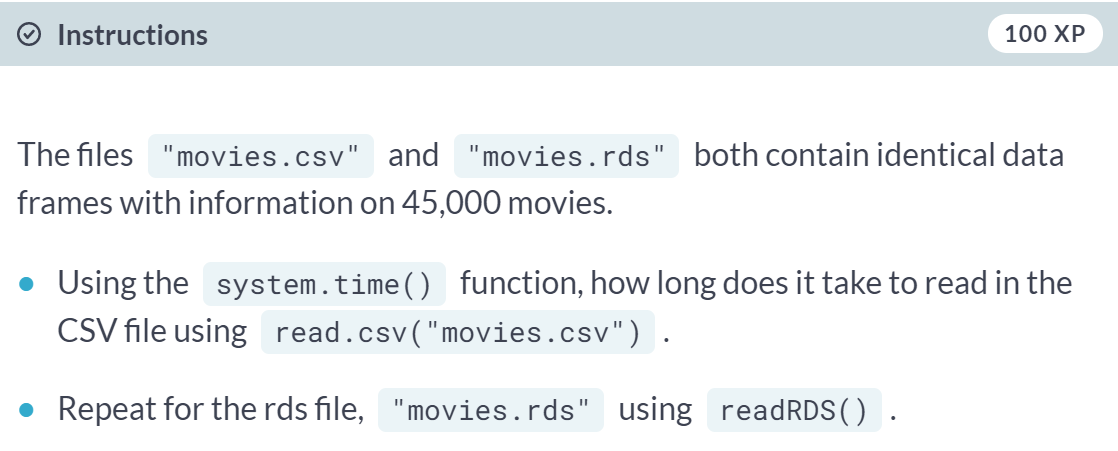
# Comparing read times of CSV and RDS files

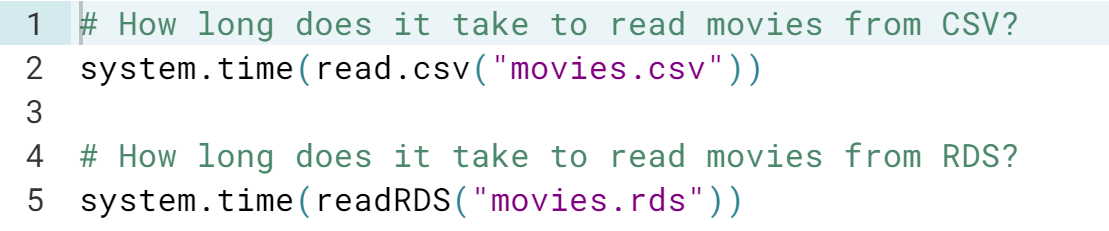
One of the most common tasks we perform is reading in data from CSV files. However, for large CSV files this can be slow. One neat trick is to read in the data and save as an R binary file (rds) using [**saveRDS()**](https://www.rdocumentation.org/packages/base/topics/readRDS). To read in the rds file, we use [**readRDS()**](https://www.rdocumentation.org/packages/base/topics/readRDS).

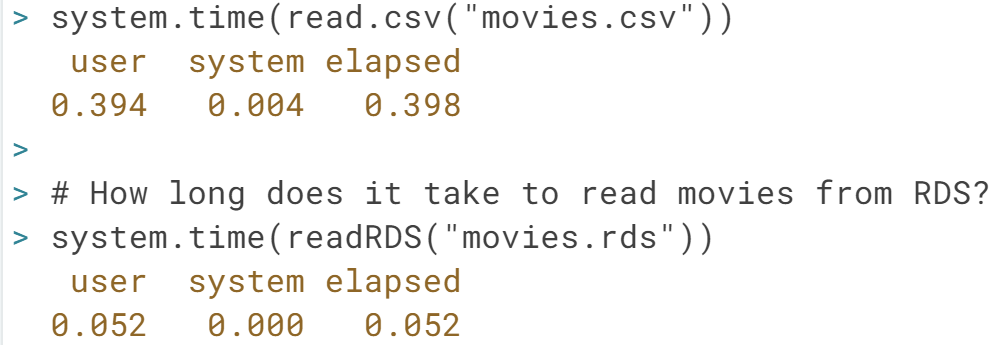
**Note**: Since rds is R's native format for storing single objects, you have not introduced any third-party dependencies that may change in the future.

To benchmark the two approaches, you can use [**system.time()**](https://www.rdocumentation.org/packages/base/topics/system.time). This function returns the time taken to evaluate any R expression. For example, to time how long it takes to calculate the square root of the numbers from one to ten million, you would write the following:

system.time(sqrt(1:1e7))

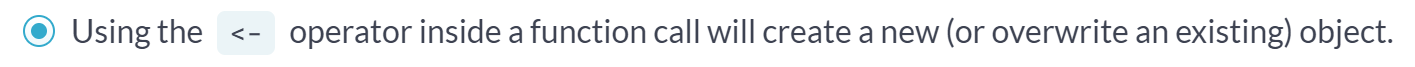
****

****

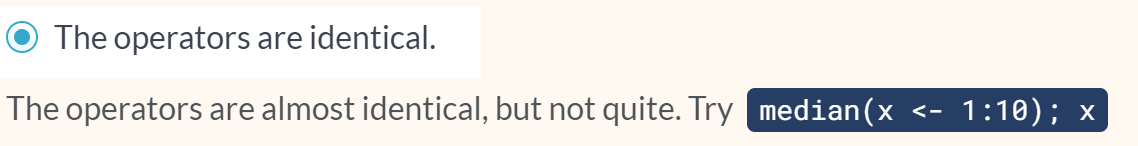
****

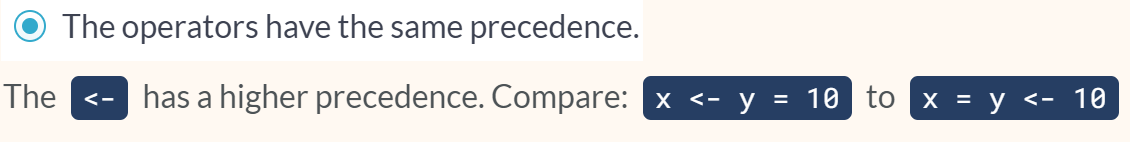
# Operational differences: "<-" and "="

There are a number of ways to assign variables to objects. The two standard ways are to use the = or <- operators. Which of the following statements is true?

****

Spot on! This makes '<-' useful inside system.time() since we can store the result.

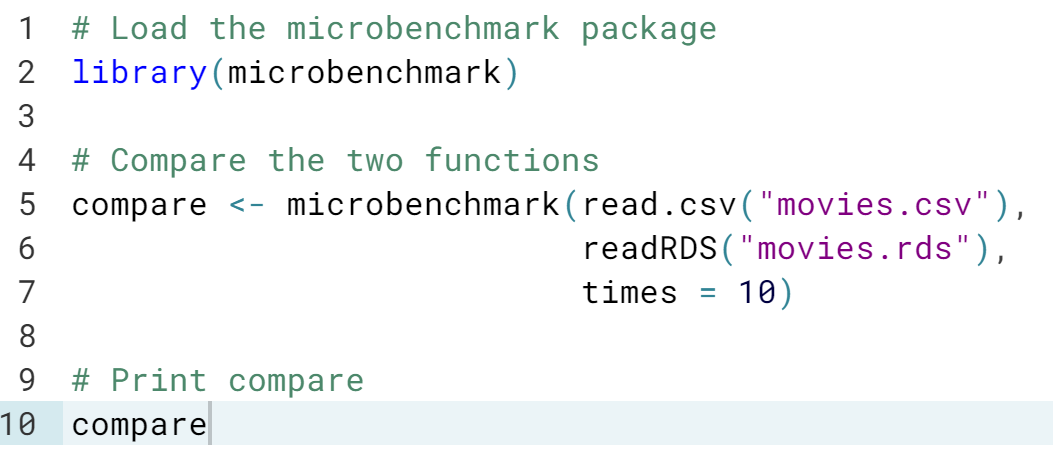
****

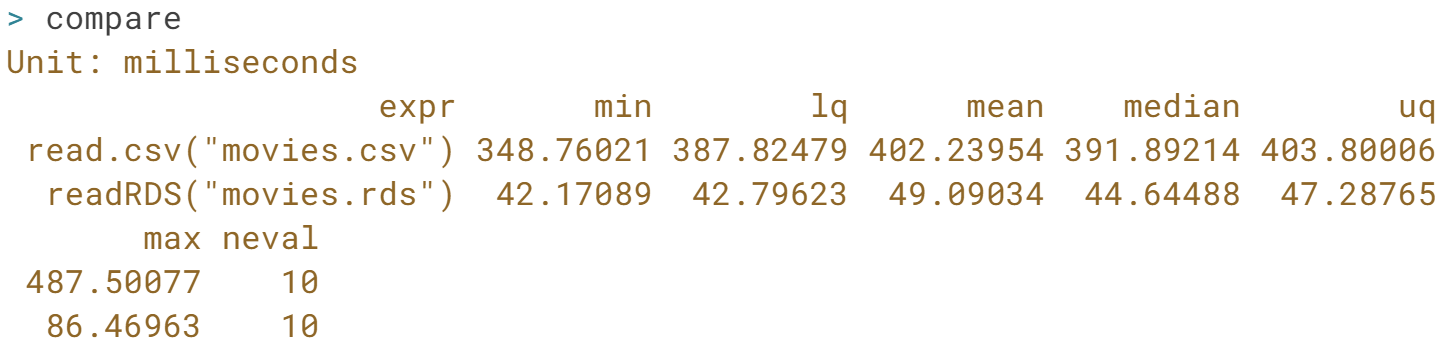
****

# Elapsed time

Using system.time() is convenient, but it does have its drawbacks when comparing multiple function calls. The [**microbenchmark**](https://www.rdocumentation.org/packages/microbenchmark) package solves this problem with the [**microbenchmark()**](https://www.rdocumentation.org/packages/microbenchmark/versions/1.4-2.1/topics/microbenchmark) function.

****

****

****

# Relative time

When benchmarking it's important to consider both absolute and relative times. Using the timings below, on average a single call to read.csv() is 720 - 80 = 640 milliseconds slower than that of readRDS().

> microbenchmark(read.csv("movies.csv"),

readRDS("movies.rds"),

times = 10)

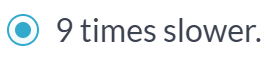
Unit: milliseconds

expr min lq mean median uq neval cld

read.csv("movies.csv") 540 660 680 720 720 740 10 b

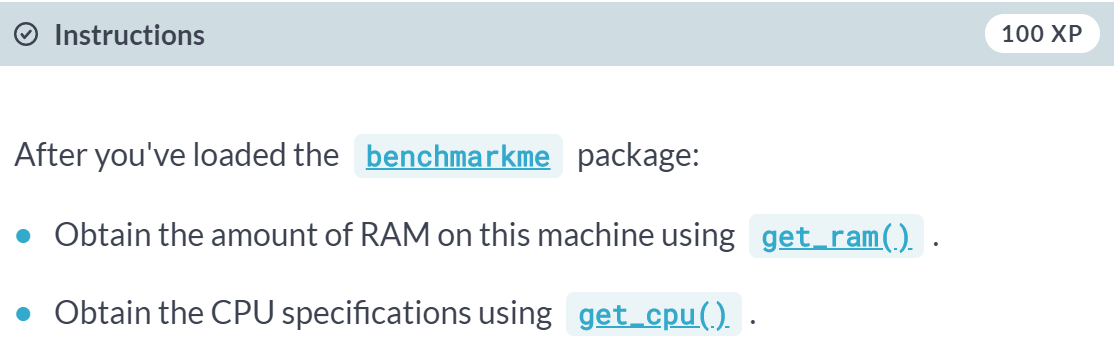
readRDS("movies.rds") 70 80 90 80 100 110 10 a

Approximately, how much slower is the mean time of read.csv() compared to readRDS()?

****

# DataCamp hardware

For many problems **your** time is the expensive part. If having a faster computer makes you more productive, it can be cost effective to buy one. However, before you splash out on new toys for yourself, your boss/partner may want to see some numbers to justify the expense. Measuring the performance of your computer is called benchmarking, and you can do that with the [**benchmarkme**](https://www.rdocumentation.org/packages/benchmarkme) package.

****

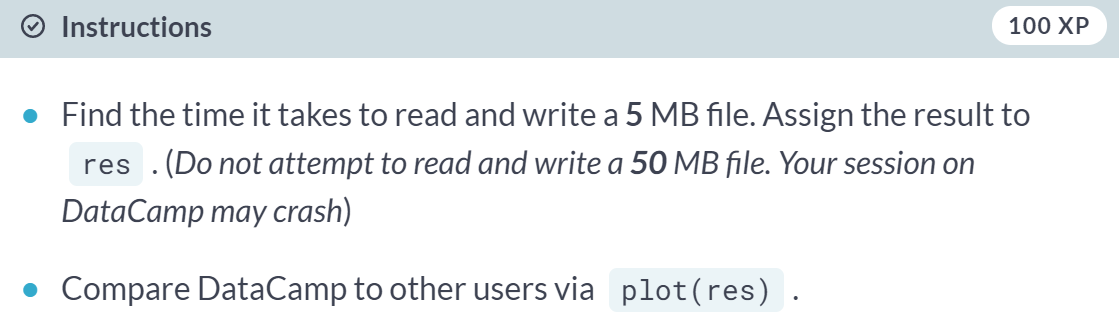
****

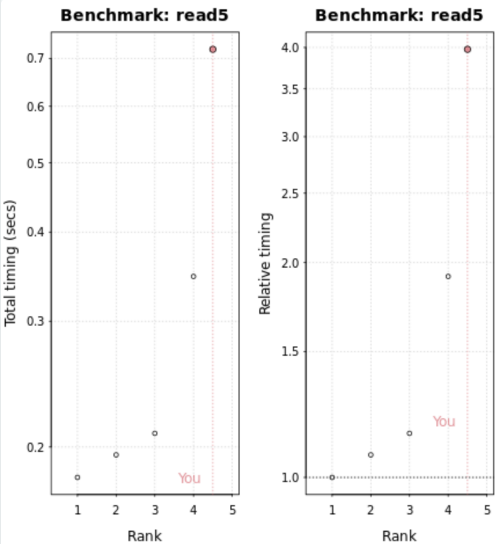
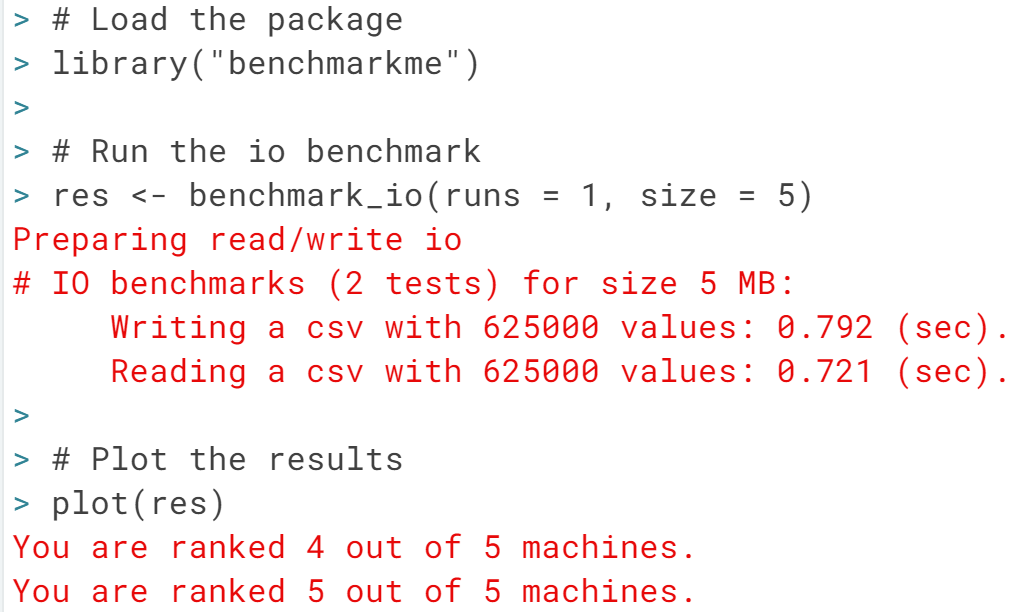
# Benchmark DataCamp's machine

The [**benchmarkme**](https://www.rdocumentation.org/packages/benchmarkme) package allows you to run a set of standardized benchmarks and compare your results to other users. One set of benchmarks tests is reading and writing speeds.

The function call records the length of time it takes to read and write a 5MB file.

res = benchmark\_io(runs = 1, size = 5)

****

****

#### Fine Tuning: Efficient Base R

# Why is this piece of code slow?

Consider the following piece of code:

x <- NULL

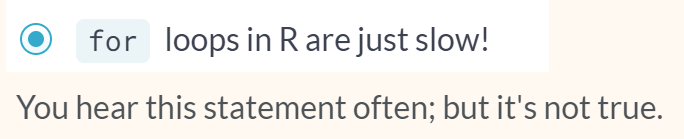
n <- 30000

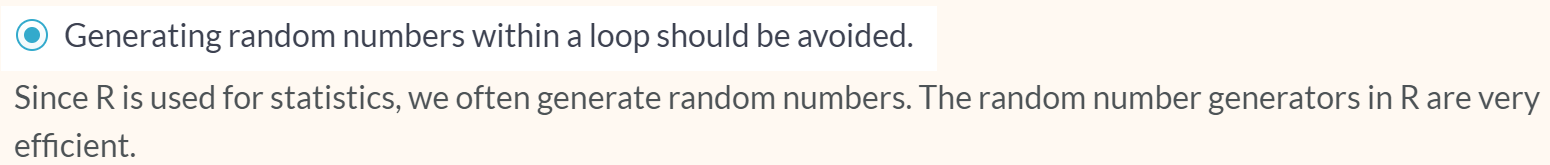
for(i in 1:n) {

x <- c(x, rnorm(1))

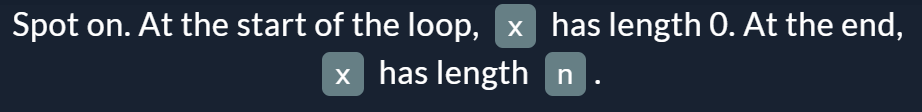
}

Which of the following statements is correct?

****

****

****

****

# Timings - growing a vector

Growing a vector is one of the deadly sins in R; you should always avoid it.

The growing() function defined below generates n random standard normal numbers, but grows the size of the vector each time an element is added!

Note: Standard normal numbers are numbers drawn from a normal distribution with mean 0 and standard deviation 1.

n <- 30000

# Slow code

growing <- function(n) {

x <- NULL

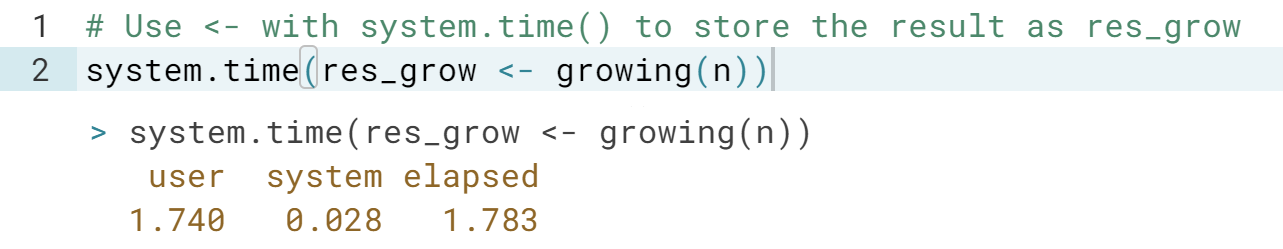
for(i in 1:n)

x <- c(x, rnorm(1))

x

}

****

****

# Timings - pre-allocation

In the previous exercise, growing the vector took around 2 seconds. How long does it take when we pre-allocate the vector? The pre\_allocate() function is defined below.

n <- 30000

# Fast code

pre\_allocate <- function(n) {

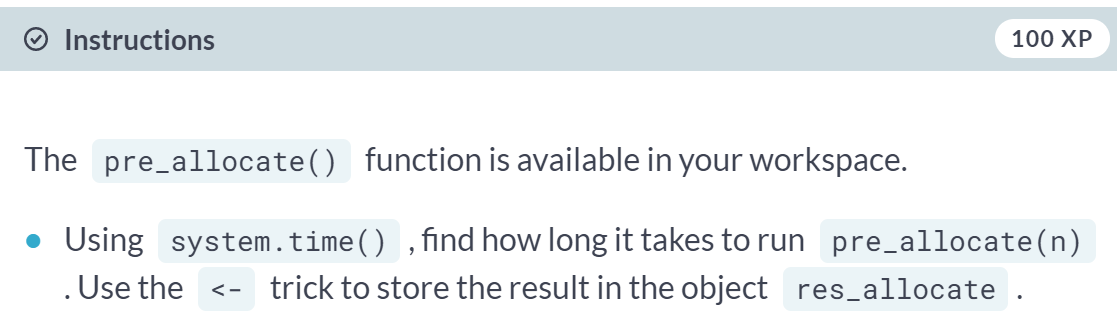
x <- numeric(n) # Pre-allocate

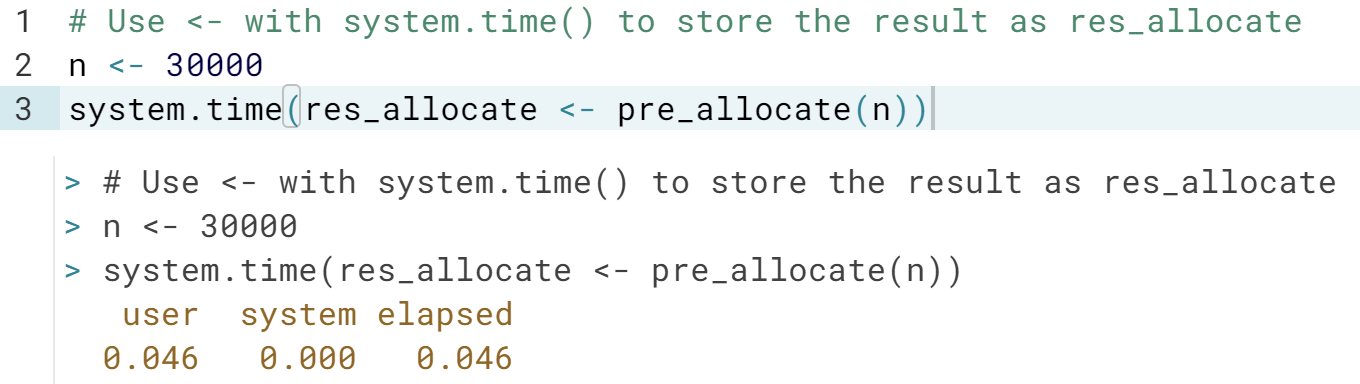
for(i in 1:n)

x[i] <- rnorm(1)

x

}

****

****

# Vectorized code: multiplication

The following piece of code is written like traditional C or Fortran code. Instead of using the vectorized version of multiplication, it uses a for loop.

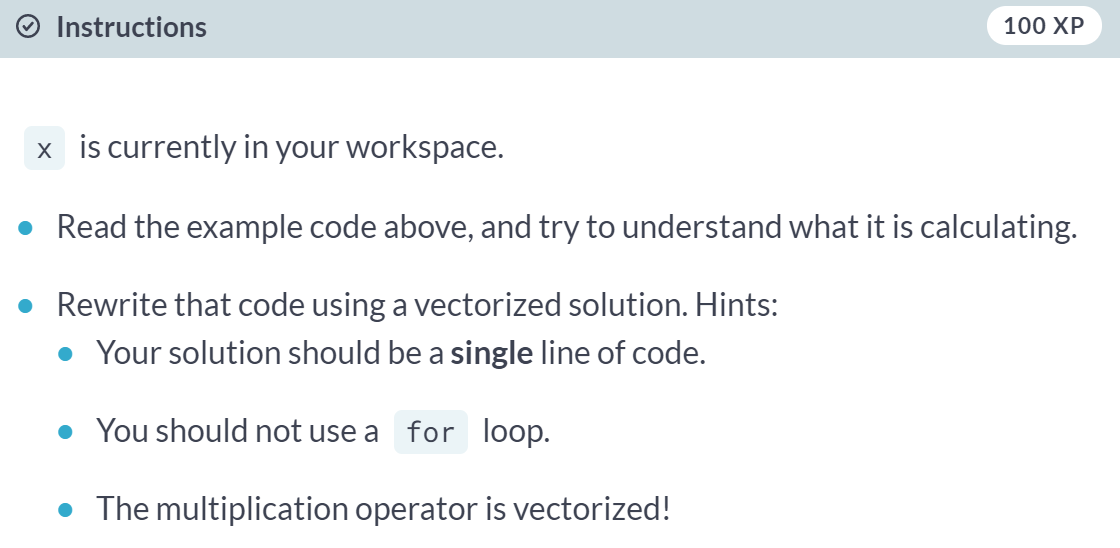
x <- rnorm(10)

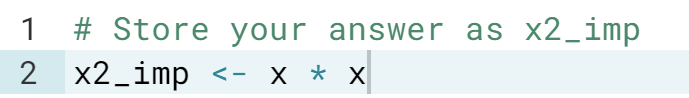
x2 <- numeric(length(x))

for(i in 1:10)

x2[i] <- x[i] \* x[i]

Your job is to make this code more "R-like" by vectorizing it.

****

****

# Vectorized code: calculating a log-sum

A common operation in statistics is to calculate the sum of log probabilities. The following code calculates the log-sum (the sum of the logs).

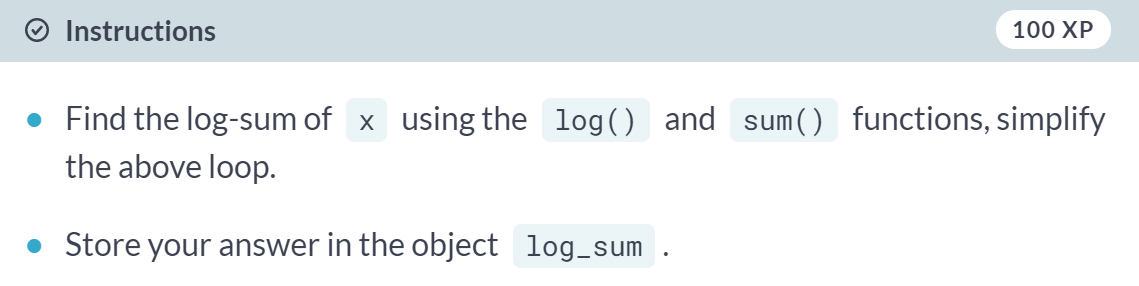
# x is a vector of probabilities

total <- 0

for(i in seq\_along(x))

total <- total + log(x[i])

However, this piece of code could be significantly improved using vectorized code.

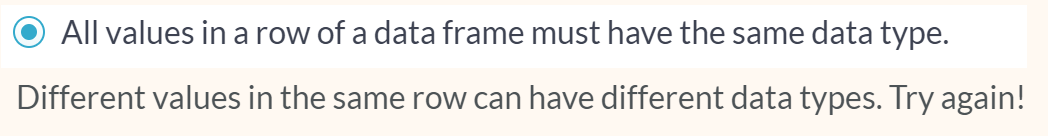


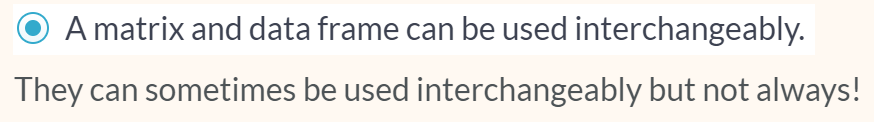


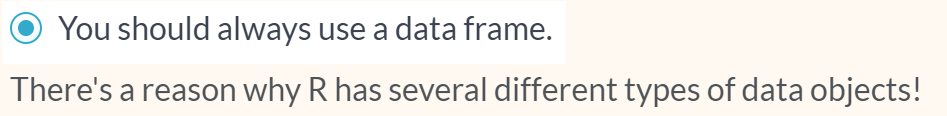
Nice one! log() can take in a vector and outputs a vector. sum() takes in a vector and returns the sum of all the values in the vector.

# Data frames vs matrices

Which of the following statements is true?









# Data frames and matrices - column selection

All values in a matrix must have the same data type, which has efficiency implications when selecting rows and columns.

Suppose we have two objects, mat (a matrix) and df (a data frame).

#### 

#### 

# Selecting a row in a data frame

What does selecting a row in a data frame return?

#### 

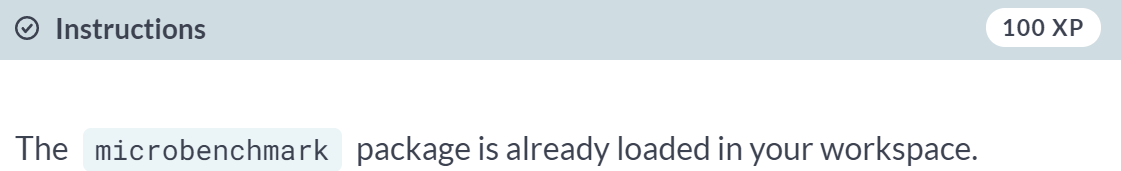
#### 

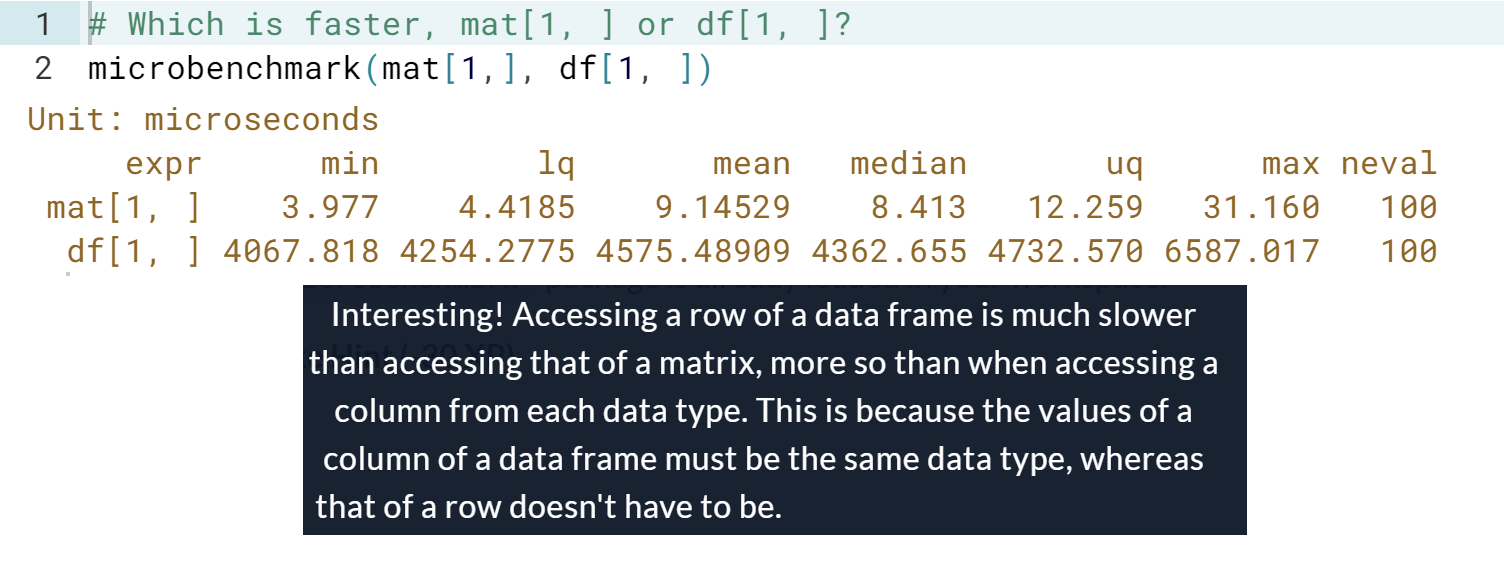
#### 

# Row timings

Similar to the previous example, the objects mat and df are a matrix and a data frame, respectively. Using microbenchmark(), how long does it take to select the first row from each of these objects?

To make the comparison fair, just use mat[1, ] and df[1, ].





#### Diagnosing Problems: Code Profiling

# Profiling a function

How does [**profvis**](https://www.rdocumentation.org/packages/profvis) profile a function?

#### 

#### 

#### 

# Where's the hold-up?

#### 

#### 

# Profvis in action

Examine the code on the right that performs a standard data analysis. It loads and selects data, plots the data of interest, and adds in a regression line.

#### 

#### 

#### 

# Change the data frame to a matrix

One of the parts of the code that profvis highlighted was the line where we generated the possible dice rolls and stored the results in a data frame:

df <- data.frame(d1 = sample(1:6, 3, replace = TRUE),

d2 = sample(1:6, 3, replace = TRUE))

We can optimize this code by making two improvements:  
1. Switching from a data frame to a matrix  
2. Generating the 6 dice rolls in a single step

This gives m <- matrix(sample(1:6, 6, replace = TRUE), ncol = 2)

#### 

#### 

# Calculating row sums

The second bottleneck identified was calculating the row sums.

total <- apply(d, 1, sum)

In the previous exercise you switched the underlying object to a matrix. This makes the above apply operation three times faster. But there's one further optimization you can use - switch apply() with [**rowSums()**](https://www.rdocumentation.org/packages/base/topics/colSums).

#### 

#### 

#### 

# Use && instead of &

To determine if both dice are the same, the move\_square() function uses if statements.

if (is\_double[1] & is\_double[2] & is\_double[3]) {

current <- 11 # Go To Jail - square 11 == Jail

}

The & operator will always evaluate both its arguments. That is, if you type x & y, R will always try to work out what x and y are. There are some cases where this is inefficient. For example, if x is FALSE, then x & y will always be FALSE, regardless of the value of y. Thus, you can save a little processing time by not calculating it. The && operator takes advantage of this trick, and doesn't bother to calculate y if it doesn't make a difference to the overall result.

In this code, if is\_double[1] is FALSE we **don't** need to evaluate is\_double[2] or is\_double[3], so we can get a speedup by swapping & for &&.

One thing to note is that && only works on single logical values, i.e., logical vectors of length 1 (like you would pass into an if condition), but & also works on vectors of length greater than 1.

(Exercise on the next page)

#### 

#### 

#### 

#### Turbo Charged Code: Parallel Programming

# How many cores does this machine have?

The [**parallel**](https://www.rdocumentation.org/packages/parallel) package has a function [**detectCores()**](https://www.rdocumentation.org/packages/parallel/topics/detectCores) that determines the number of cores in a machine.

How many cores does this machine have?

#### 

#### 

# Can this loop run in parallel (1)?

The following piece of code implements a simple dice game. The game is as follows:

* Initialize: total <- 0.
* Roll a single die and add it to total.
* If total is even, reset total to zero.
* If total is greater than 10, the game finishes.

This is straightforward to code in R

total <- no\_of\_rolls <- 0 # Initialise

while(total < 10) {

total <- total + sample(1:6, 1)

if(total %% 2 == 0) total <- 0 # If even. Reset to 0

no\_of\_rolls <- no\_of\_rolls + 1

}

no\_of\_rolls

Do you think this algorithm can be (easily) run in parallel?

#### 

#### 

# Can this loop run in parallel (2)?

You have enjoyed the previous dice game so much, you want to play it multiple times! You wrap the original code in a function:

play <- function() {

total <- no\_of\_rolls <- 0

while(total < 10) {

total <- total + sample(1:6, 1)

# If even. Reset to 0

if(total %% 2 == 0) total <- 0

no\_of\_rolls <- no\_of\_rolls + 1

}

no\_of\_rolls

}

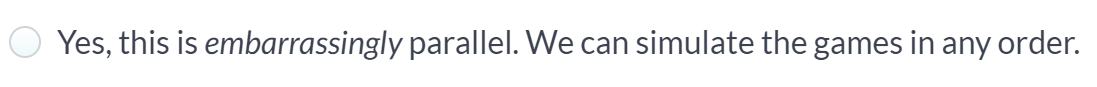
and construct a loop to play the game:

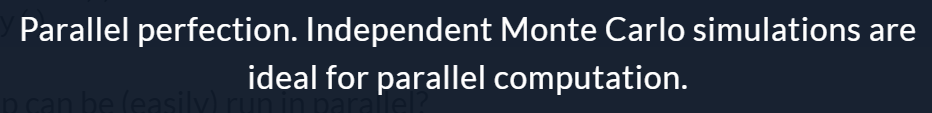
results <- numeric(100)

for(i in seq\_along(results))

results[i] <- play()

Do you think this for loop can be (easily) run in parallel?

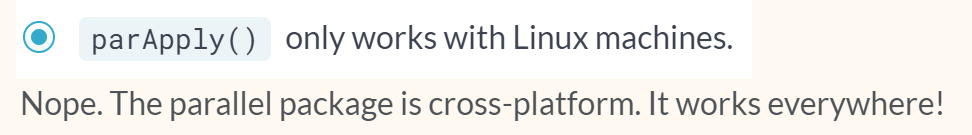




# Moving to parallel programming

Which of the following statements is true?







# Moving to parApply

To run code in parallel using the parallel package, the basic workflow has three steps.

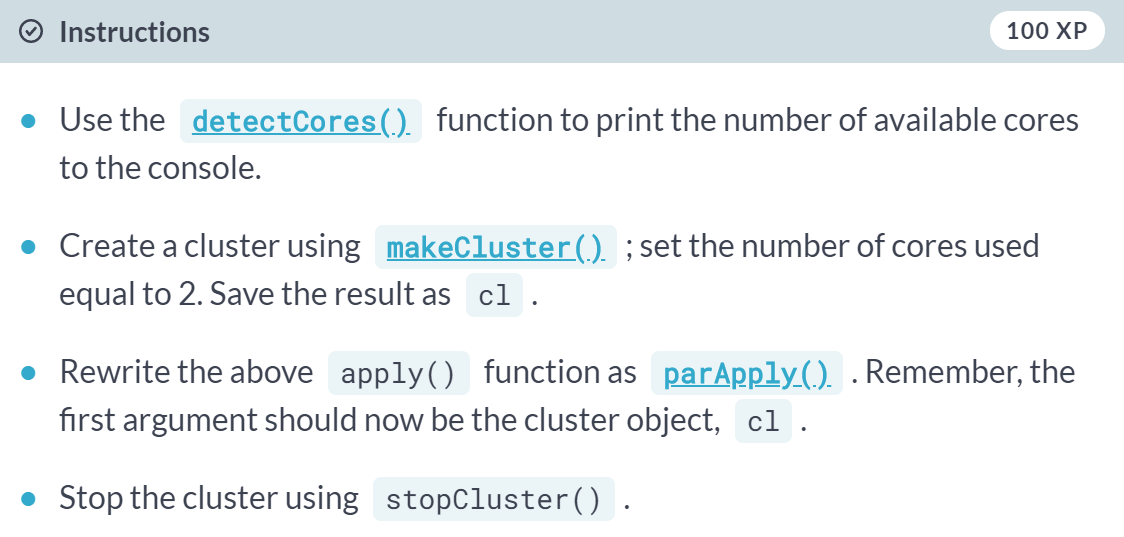
1. Create a cluster using [**makeCluster()**](https://www.rdocumentation.org/packages/parallel/topics/makeCluster).
2. Do some work.
3. Stop the cluster using [**stopCluster()**](https://www.rdocumentation.org/packages/parallel/topics/makeCluster).

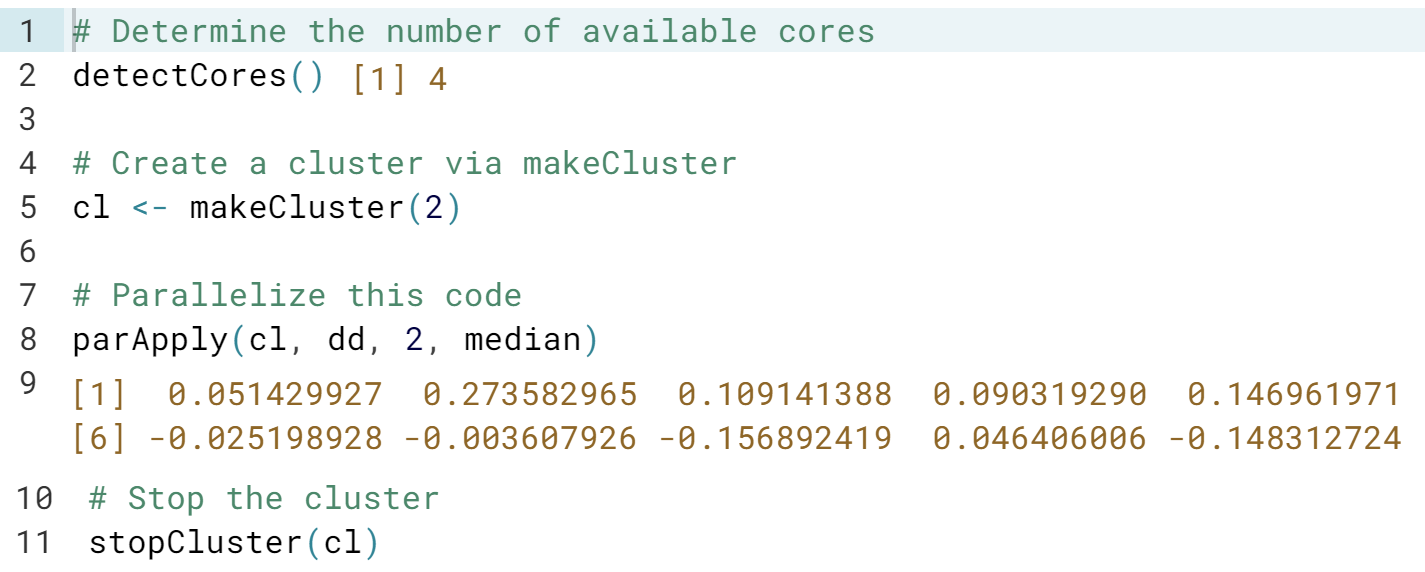
The simplest way to make a cluster is to pass a number to makeCluster(). This creates a cluster of the default type, running the code on that many cores.

The object dd is a data frame with 10 columns and 100 rows. The following code uses apply() to calculate the column medians:

apply(dd, 2, median)

To run this in parallel, you swap apply() for [**parApply()**](https://www.rdocumentation.org/packages/parallel/topics/clusterApply). The arguments to this function are the same, except that it takes a cluster argument before the usual apply() arguments.





# Using parSapply()

We previously played the following game:

* Initialize: total = 0.
* Roll a single die and add it to total.
* If total is even, reset total to zero.
* If total is greater than 10. The game finishes.

The game could be simulated using the play() function:

play <- function() {

total <- no\_of\_rolls <- 0

while(total < 10) {

total <- total + sample(1:6, 1)

# If even. Reset to 0

if(total %% 2 == 0) total <- 0

no\_of\_rolls <- no\_of\_rolls + 1

}

no\_of\_rolls

}

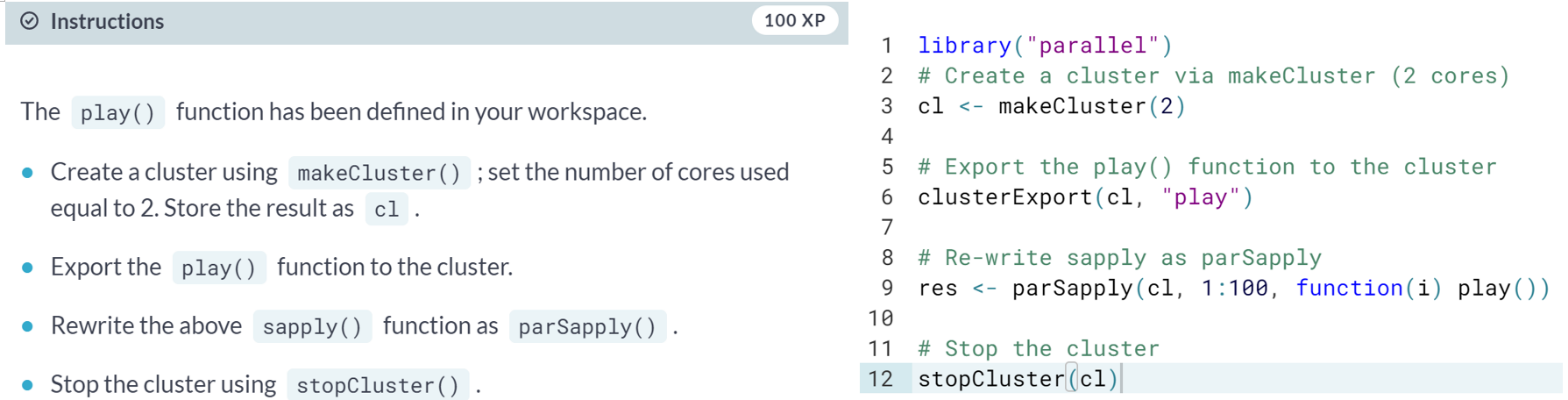
To simulate the game 100 times, we could use a for loop or sapply():

res <- sapply(1:100, function(i) play())

This is perfect for running in parallel!

To make functions available on a cluster, you use the [**clusterExport()**](https://www.rdocumentation.org/packages/parallel/topics/clusterApply) function. This takes a cluster and a string naming the function.

clusterExport(cl, "some\_function")



# Timings parSapply()

Running the dice game is embarrassingly parallel. These types of simulations usually (but not always) produce a good speed-up. As before, we can use [**microbenchmark()**](https://www.rdocumentation.org/packages/microbenchmark/topics/microbenchmark) or [**system.time()**](https://www.rdocumentation.org/packages/base/topics/system.time). For simplicity, we'll use system.time() in this exercise.

