HW 05 - R Package `dieroller`

Stat 133, Spring 2018, Prof. Sanchez

Due date: Fri Apr-27 (before midnight)

The purpose of this assignment is to create an R package that implements functions for simulating rolling a die. And then use the package to approximate probabilities to the famous *Chevalier De Mere's* dice problems.

Here's a list of resources that will help you complete this (very cool) HW:

- **Pack YouR Code**: gastonsanchez.com/packyourcode
- **Example package `"cointoss"`**: github.com/gastonstat/cointoss
- **Package development cheat-sheet**: packages-cheatsheet.pdf
- **R Packages**: r-pkgs.had.co.nz

The goal is to program two classes of objects: a regular `"die"` with six sides, and an object `"roll"` (i.e. the rolls of a `"die"`).

1) Object `"die"`

Because your package will be used to simulate rolling a die, the first thing to do is writing code that allows you to create a *die* object, that is, an R object of class `"die"`.

The idea is to write a function `die()` that takes two arguments, `sides` and `prob`, in order to return a `"die"` object. Here's an example of the default call to `die()`:

```r
# default call: creates a standard fair die
fair_die <- die()
fair_die
```

```
## object "die"
##
##   side      prob
## 1    1 0.1666667
## 2    2 0.1666667
## 3    3 0.1666667
## 4    4 0.1666667
## 5    5 0.1666667
## 6    6 0.1666667
```

In order to create a `"die"` object write:

- a constructor function `die()` that creates a *fair* die by default. This function should have two arguments:
  - `sides`: vector of six elements, by default numbers 1, 2, 3, 4, 5, 6.
  - `prob`: vector of probabilities for each side (all equal to 1/6 by default)
- an auxiliary function `check_sides()`, called by `die()`, that checks the validity of the argument `sides`.
- an auxiliary function `check_prob()`, called by `die()`, that checks the validity of the argument `prob`.
- a `"print"` method for `"die"` objects, that displays the class of the object, and a tabular display of the sides and the associated probabilities (see examples below).

Here are a couple of examples of various ways to call `die()`:

```
# die with non-standard sides
weird_die <- die(sides = c('i', 'ii', 'iii', 'iv', 'v', 'vi'))
weird_die
```

```
## object "die"
##
##   side      prob
## 1    i 0.1666667
## 2   ii 0.1666667
## 3  iii 0.1666667
## 4   iv 0.1666667
## 5    v 0.1666667
## 6   vi 0.1666667
```

```
# create a loaded die
loaded_die <- die(prob = c(0.075, 0.1, 0.125, 0.15, 0.20, 0.35))
loaded_die
```

```
## object "die"
##
##   side  prob
## 1    1 0.075
## 2    2 0.100
## 3    3 0.125
## 4    4 0.150
## 5    5 0.200
## 6    6 0.350
```

```
# bad sides
bad_die <- die(sides = c('a', 'b', 'c', 'd', 'e'))
```

```
## Error in check_sides(sides):
## 'sides' must be a vector of length 6
```

```r
# bad prob
bad_die <- die(
  sides = c('a', 'b', 'c', 'd', 'e', 'f'),
  prob = c(0.2, 0.1, 0.1, 0.1, 0.5, 0.1))
```

```
## Error in check_prob(prob):
## elements in 'prob' must add up to 1
```

---

2) Object `"roll"`

To *roll* an object `"die"` you will have to create a `roll()` function that takes a `die` and a number `times` of rolls, and that returns an object of class `"rolls"`. Here's a basic example for `roll()`:

```r
# roll fair die 50 times
fair_die <- die()

set.seed(123)

fair50 <- roll(fair_die, times = 50)
fair50
```

```
## object "roll"
##
## $rolls
##  [1] 3 6 4 1 1 2 5 1 5 4 1 4 6 5 2 1 3 2 3 1 1 6 5 1 5 6 5 5 3 2 1 1 6 6 2
## [36] 4 6 3 3 3 2 4 4 4 2 2 3 4 3 1
```

In order to create a `"roll"` object write:

- a constructor function `roll()`. This function should have two arguments:
  - `die`: object of class `"die"`.
  - `times`: number of times to roll the die (default value of 1).
- an auxiliary function `check_times()`, called by `roll()`, that checks the validity of the argument `times`.
- the output of `roll()` will be a list containing:
  - `rolls`: vector with outputs of the rolls
  - `sides`: vector with the sides of the die
  - `prob`: vector with probabilities for each side of the die

  - `total`: total number of rolls (i.e. `times`)

- a `"print"` method for `"roll"` objects, that displays the class of the object, and the generated `rolls`.

Here are a couple of examples of various ways to call `roll()`:

```
# roll fair die 50 times
fair_die <- die()

# roll 50 times
set.seed(123)
fair_50rolls <- roll(fair_die, times = 50)

# print
fair_50rolls
```

```
## object "roll"
##
## $rolls
##  [1] 3 6 4 1 1 2 5 1 5 4 1 4 6 5 2 1 3 2 3 1 1 6 5 1 5 6 5 5 3 2 1 1 6 6 2
## [36] 4 6 3 3 3 2 4 4 4 2 2 3 4 3 1
```

```
# what's in fair50?
names(fair50)
```

```
## [1] "rolls" "sides" "prob"  "total"
```

```
fair50$rolls
```

```
##  [1] 3 6 4 1 1 2 5 1 5 4 1 4 6 5 2 1 3 2 3 1 1 6 5 1 5 6 5 5 3 2 1 1 6 6 2
## [36] 4 6 3 3 3 2 4 4 4 2 2 3 4 3 1
```

```
fair50$sides
```

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

```
fair50$prob
```

```
## [1] 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667 0.1666667
```

```
fair50$total
```

```
## [1] 50
```

A less basic example:

```
# string die
str_die <- die(
```

```r
  sides = c('a', 'b', 'c', 'd', 'e', 'f'),
  prob = c(0.075, 0.1, 0.125, 0.15, 0.20, 0.35))

# roll 20 times
set.seed(123)
str_rolls <- roll(str_die, times = 20)
names(str_rolls)
```

```
## [1] "rolls" "sides" "prob"  "total"
```

```r
str_rolls
```

```
## object "roll"
##
## $rolls
##  [1] "f" "c" "e" "b" "a" "f" "e" "b" "d" "e" "a" "e" "d" "d" "f" "b" "f"
## [18] "f" "f" "a"
```

3) Summary method for "roll" objects

Write a *summary* method—i.e. `summary.roll()`—for `"roll"` objects that returns an object `"summary.roll"` object. The output of `summary.roll()` will be a data frame called `freqs` with 3 columns:

- `side`: the sides of the rolled `die`.
- `count`: the frequency (count) of each side of the rolled `die`.
- `prop`: the relative frequency (proportion) of each side of the rolled `die`.

You will also have to write a *print* method for the summary—i.e. `print.summary.roll()`—such that when a `summary.roll` is printed, you get an output like the following example:

```r
set.seed(123)
fair_50rolls <- roll(fair_die, times = 50)
fair50_sum <- summary(fair_50rolls)

fair50_sum
```

```
## summary "roll"
##
##   side count prop
## 1    1    11 0.22
## 2    2     8 0.16
## 3    3     9 0.18
## 4    4     8 0.16
## 5    5     7 0.14
```

```
## 6    6     7 0.14
```

```r
# what's in the summary
class(fair50_sum)
```

```
## [1] "summary.roll"
```

```r
names(fair50_sum)
```

```
## [1] "freqs"
```

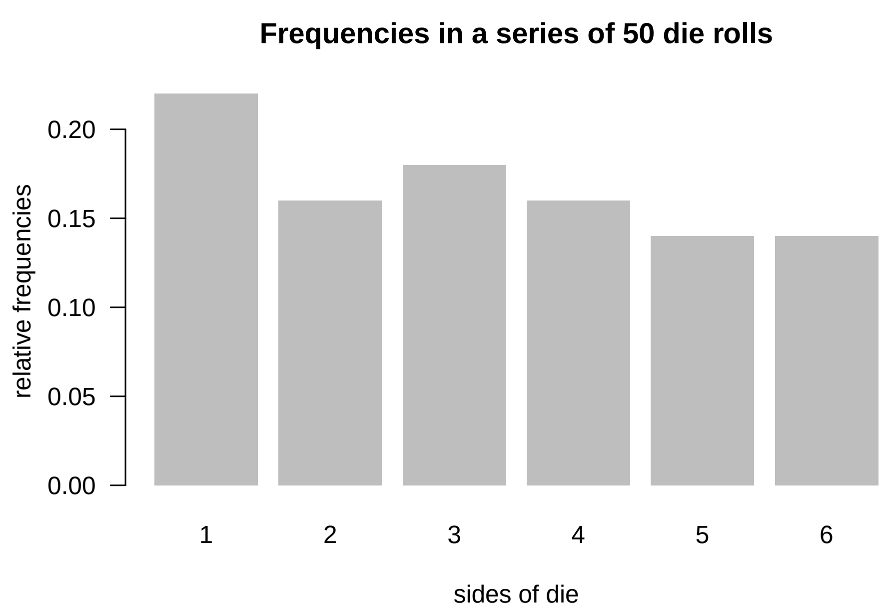
```r
fair50_sum$freqs
```

```
##   side count prop
## 1    1    11 0.22
## 2    2     8 0.16
## 3    3     9 0.18
## 4    4     8 0.16
## 5    5     7 0.14
## 6    6     7 0.14
```

4) Plot methd for `"roll"` objects

Write a *plot* method for `"roll"` objects—i.e. `plot.roll()`. You need to graph a barchart of frequencies (relative frequencies of `1`'s, `2`'s, `3`'s, `4`'s, `5`'s, and `6`'s). Use `barplot()` to implement this method.

```r
# plot method
plot(fair_50rolls)
```



5) Additional Methods

Also, write functions for the following methods:

- an extraction method `"["` to extract the value of a given roll.
- a replacement method `"[<-"` to replace the value of a given roll.
- an addition `"+"` method to add more rolls.

Example

Here's a comprehensive example in which a `"die"` object is created, and then it gets rolled 500 times to obtain an object `"roll"` on which we apply the various programmed methods:

```r
# roll fair die
set.seed(123)
fair_die <- die()
fair500 <- roll(fair_die, times = 500)

# summary method
summary(fair500)
```

```
## summary "roll"
##
##   side count  prop
## 1    1    80 0.160
## 2    2    81 0.162
## 3    3    92 0.184
## 4    4    92 0.184
## 5    5    72 0.144
## 6    6    83 0.166
```

```r
# extracting roll in position 500
fair500[500]
```

```
## [1] 6
```

```r
# replacing last roll
fair500[500] <- 1
fair500[500]
```

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

```r
summary(fair500)
```

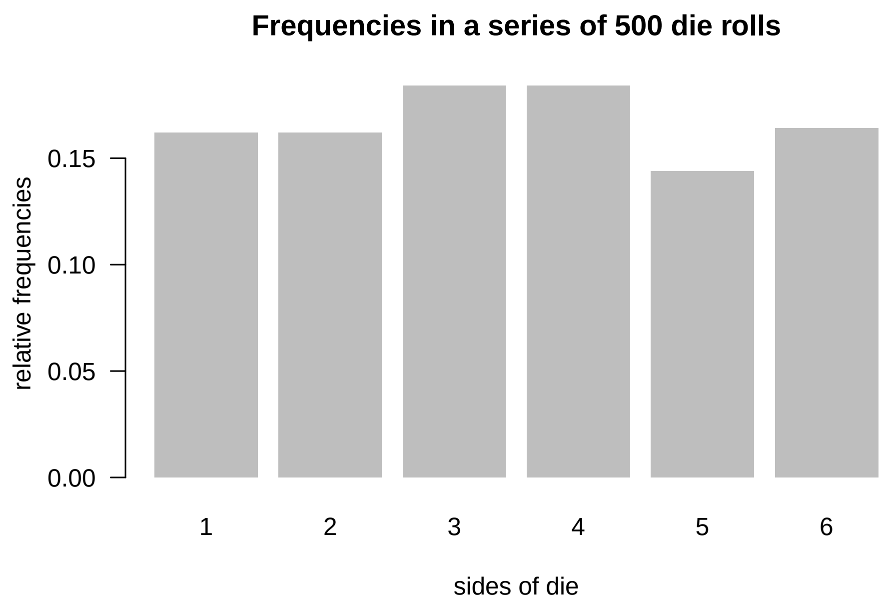
```
## summary "roll"
##
##   side count  prop
## 1    1    81 0.162
```

```
## 2     2    81 0.162
## 3     3    92 0.184
## 4     4    92 0.184
## 5     5    72 0.144
## 6     6    82 0.164
```

```r
# adding 100 rolls
fair600 <- fair500 + 100
summary(fair600)
```

```
## summary "roll"
##
##   side count      prop
## 1    1   100 0.1666667
## 2    2    97 0.1616667
## 3    3   104 0.1733333
## 4    4   109 0.1816667
## 5    5    91 0.1516667
## 6    6    99 0.1650000
```

```r
# plot method
plot(fair500, 500)
```



Package Creation

Carefully check the example package "`cointoss`" to get some hints and inspiration:

https://github.com/gastonstat/cointoss

Tests: Your package should include tests for your functions `die()` and `roll()`, as well as for the auxiliary functions called by them: e.g. `check_sides()`, `check_prob()`, `check_times()`.

Vignette: Your package should include an introductory vignette that shows the user how to utilize the various functionalities of your package.

Package Structure

After completion, your package `"dieroller"` should have the following filestructure:

```
dieroller/
   .Rbuildignore
   dieroller.Rproj
   devtools-flow.R
   DESCRIPTION
   NAMESPACE
   README.md
   R/
   man/
   tests/
   vignettes/
   inst/
```

Submission

- Create a folder (i.e. subdirectory) `dieroller/` in your github classroom repository. This will be the folder of your package.
- Create another folder (i.e. subdirectory) `hw05` in your github classroom repository.
- Use `hw05` to write a report (github document) in which you use your package `"dieroller"` to show us how to use your package. Specifically, use your package to find approximate solutions to the famous *Chevalier De Mere's problem* (read below).

De Mere's problem

As part of your `hw05` report, use your objects `"die"` and `"roll()"` to simulate a series of 1000 games for the famous *Chevalier De Mere's* dice problems:

- One gambling problem that launched modern probability theory (by Dan Ma).

Problem I: The first problem involves computing the probability of getting at least one "6" in four rolls of a die. This probability can be computed analytically as:

$$1 - (5/6)^4$$

- simulate one series of 1000 games of this game-I.
- each game involves rolling a die four times.
- this means that you will end up generating a total of $1000 \times 4 = 4000$ rolls.
- count the number of games in which there is at least one 6.
- then compute the relative frequency of getting at least one "6" (this will be the approximate probability)

Problem II: The other problem involves computing the probability of getting at least two "6" in 24 rolls of a pair of dice. This probability can be computed analytically as:

$$1 - (35/36)^{24}$$

- simulate one series of 1000 games of this game-II.
- each game involves rolling a pair of dice 24 times.
- this means that you will end up generating a total of $1000 \times 24 \times 2 = 48000$ rolls.
- count the number of games in which there is at least one double 6.
- then compute the relative frequency of getting at least two "6" (this will be the approximate probability).