Warm-Up 06 - Control Flow Structures

Stat 133, Fall 2018, Prof. Sanchez

Due date: Oct-23 (before midnight)

The purpose of this assignment is to keep working on the programming concepts that are covered in weeks 7 and 8:

- writing functions
- documenting functions with Roxygen comments
- using conditionals
- using loops

General Instructions

- Write your narrative and code in an `Rmd` (R markdown) file.
- Name this file as `warmup06-first-last.Rmd`, where `first` and `last` are your first and last names (e.g. `warmup06-gaston-sanchez.Rmd`).
- Include a code chunk at the top of your file like the one in the following screen capture:

```
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE, error = TRUE)
```
```

Binomial Probability Functions

The Binomial distribution is perhaps the most famous probability distribution among discrete random variables. This is the theoretical probability model that we use when calculating probabilities about the number of successes in a fixed number of random trials performed under identical conditions (aassuming a constant probability of success).

A classic example of a binomial random variable X involves the number of Heads (or Tails) that you get when tossing a coin n times. Say you are interested in finding the probability of getting three heads in four tosses of a fair coin: P(X = 3 heads in 4 tosses). To find the answer, we use the formula of the binomial probability:

$$Pr(X = k) = \binom{n}{k} p^k (1 - p)^{n-k}$$

where:

- n is the number of (fixed) trials
- p is the probability of success on each trial
- $1 - p$ is the probability of failure on each trial
- k is a variable that represents the number of successes out of n trials
- the first term in parenthesis is not a fraction, it is the number of combinations in which k success can occur in n trials

So, what is the probability of three heads in four tosses? Assuming that we flip a fair coin (50% chance of heads), P(X = 3) is:

$$Pr(X = 3) = \binom{4}{3} 0.5^3 (1 - 0.5)^{4-3} = 0.25$$

Your Mission

You will have to write the functions listed below. All the functions must have Roxygen comments (e.g. `@title`, `@description`, `@param`, `@return`).

- `is_integer()`
- `is_positive()`
- `is_nonnegative()`
- `is_positive_integer()`
- `is_nonneg_integer()`
- `is_probability()`
- `bin_factorial()`
- `bin_combinations()`
- `bin_probability()`
- `bin_distribution()`

Assume that `bin_probability()` and `bin_distribution()` are the "high-level" functions that a user will be invoking. You can think of the rest of the functions as "auxiliary" functions not intended to be called by the user.

In future assignments (not in this one) you will also have to write formal tests to make sure that your code works as expected in a programmatic way.

Important Restrictions

In order to practice writing loops (e..g. `for` loops), you will have to assume that R does not provide *vectorized* operations. For example, if you have a numeric vector `x <- c(1, 2, 3, 4, 5)` and you need to add 2 to each element in `x`, you will need to write a `for` loop:

```
# ===============================================
# Assume that R is not vectorized
# ===============================================
```

```r
# input vector
x <- c(1, 2, 3, 4, 5)

# output vector
y <- rep(0, 5)

# iterations
for (i in 1:length(x)) {
  y[i] <- x[i] + 2
}
y
```

```
## [1] 3 4 5 6 7
```

In addition, you are NOT allowed to use base R functions such as: `prod()`, `sum()`, `choose()`, `factorial()`, `dbinom()`, or `pbinom()`. Likewise, you CANNOT use functions from external R packages. Last but not least, do NOT use `print()` as a *return* statement of your functions. If you want to explicitly make a return statement use `return()`—although this is not mandatory, especially if you understand how R expressions work (i.e. the value of an expression if the last statement that gets executed).

Function `is_integer()`

Write a function `is_integer()` that tests if a numeric value can be considered to be an integer number (e.g. `2L` or `2`). This function should return `TRUE` if the input can be an integer, `FALSE` otherwise. *Hint:* the modulo operator `%%` is your friend (see `?'%%'` for more info). Assume that the input is always a single number. Include the code below in your Rmd file.

```r
# TRUE's
is_integer(-1)
is_integer(0)
is_integer(2L)
is_integer(2)

# FALSE's
is_integer(2.1)
is_integer(pi)
is_integer(0.01)
```

Function `is_positive()`

Write a function `is_positive()` that tests if a numeric value is a positive number. This function should return `TRUE` if the input is positive, `FALSE` otherwise. Assume that the input is always a single number. Include the code below in your Rmd file. Include the code below in your Rmd file.

```
# TRUE's
is_positive(0.01)
is_positive(2)

# FALSE's
is_positive(-2)
is_positive(0)
```

Function `is_nonnegative()`

Write a function `is_nonnegative()` that tests if a numeric value is a non-negative number. This function should return `TRUE` if the input is non-negative, `FALSE` otherwise. Assume that the input is always a single number. Include the code below in your Rmd file.

```
# TRUE's
is_nonnegative(0)
is_nonnegative(2)

# FALSE's
is_nonnegative(-0.00001)
is_nonnegative(-2)
```

Function `is_positive_integer()`

Use `is_positive()` and `is_integer()` to write a function `is_positive_integer()` that tests if a numeric value can be considered to be a positive integer. This function should return `TRUE` if the input is positive integer, `FALSE` otherwise. Assume that the input is always a single number. Include the code below in your Rmd file.

```
# TRUE
is_positive_integer(2)
is_positive_integer(2L)

# FALSE
is_positive_integer(0)
is_positive_integer(-2)
```

Function `is_nonneg_integer()`

Use `is_nonnegative()` and `is_integer()` to write a function `is_nonneg_integer()` that tests if a numeric value can be considered to be a non-negative integer. This function should return `TRUE` if the input is non-negative integer, `FALSE` otherwise. Assume that the input is always a single number. Include the code below in your Rmd file.

```
# TRUE's
is_nonneg_integer(0)
is_nonneg_integer(1)

# FALSE
is_nonneg_integer(-1)
is_nonneg_integer(-2.5)
```

Function `is_probability()`

Write a function `is_probability()` that tests if a given number p is a valid probability value: $0 \leq p \leq 1$. This function should return `TRUE` if the input is a valid probability, `FALSE` otherwise. Assume that the input is always a single number. Include the code below in your Rmd file.

```
# TRUE's
is_probability(0)
is_probability(0.5)
is_probability(1)

# FALSE's
is_probability(-1)
is_probability(1.0000001)
```

Function `bin_factorial()`

Use a `for` loop to write a function `bin_factorial()` that calculates the factorial of a non-negative integer n. You don't need to use your function `is_nonneg_integer()` to write `bin_factorial()`, since both functions are supposed to be auxiliary functions. Assume that the input is always a single number.

Recall that the factorial, denoted by $n!$, is the product of all positive integers less than or equal to n. For example,

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Recall that the value of 0! is 1

```
# valid
bin_factorial(5)
```

```
## [1] 120
```

```
bin_factorial(0)
```

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

Function `bin_combinations()`

Use `bin_factorial()` to write a function `bin_combinations()` that calculates the number of combinations in which k successes can occur in n trials. Your function should have arguments `n` and `k`. You don't need to use your function `is_nonneg_integer()` to write `bin_combinations()`, since both functions are supposed to be auxiliary functions.

Recall that the number of combinations "n choose k" is given by:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

For instance, the number of combinations in which $k = 2$ success can occur in $n = 5$ trials is:

$$\binom{n=5}{k=2} = \frac{5!}{2!(5-2)!} = 10$$

Here's how you should be able to invoke `bin_combinations()`

```
bin_combinations(n = 5, k = 2)
bin_combinations(10, 3)
bin_combinations(4, 4)
```

Function `bin_probability()`

Use your functions `is_nonneg_integer()`, `is_probability()`, and `bin_combinations()` to create a `bin_probability()` function. Your function should have arguments `trials`, `success`, and `prob`. Here's how you should be able to invoke `bin_probability()`:

```
# probability of getting 2 successes in 5 trials
# (assuming prob of success = 0.5)
bin_probability(trials = 5, success = 2, prob = 0.5)
```

```
## [1] 0.3125
```

Use `is_nonneg_integer()` to check that `trials` and `success` are valid non-integer numbers. If any of `trials` or `success` is invalid, then `bin_probability()` should raise an error—triggered by `stop()`—e.g. something like `'invalid trials value'` or `'invalid success value'`. Likewise, use `is_probability()` to test that `prob` is a valid probability value. If `prob` is invalid, then `bin_probability()` should `stop()` execution with an error—e.g. something like `'invalid prob value'`. You should also check that `success` does not exceed the number of `trials`, otherwise `stop()` the execution of `bin_probability()`.

Use `bin_probability()` to compute:

```
# bad trials
bin_probability(trials = 0, success = 2, prob = 0.5)

# bad success
bin_probability(trials = 5, success = 2.5, prob = 0.5)

# success > trials
bin_probability(trials = 5, success = 6, prob = 0.5)

# bad prob
bin_probability(trials = 5, success = 2, prob = -0.5)

# 55 heads in 100 tosses of a loaded coin with 45% chance of heads
bin_probability(trials = 100, success = 55, prob = 0.45)
```

Function `bin_distribution()`

Use `bin_probability()` to create a `bin_distribution()` function. Your function should have arguments `trials`, and `prob`. This function should return a data frame with the probability distribution:

```
# binomial probability distribution
bin_distribution(trials = 5, prob = 0.5)
```

```
##   success probability
## 1       0     0.03125
## 2       1     0.15625
## 3       2     0.31250
## 4       3     0.31250
## 5       4     0.15625
## 6       5     0.03125
```