HW03 - Pipes and Programming Basics

Stat 133, Spring 2018, Prof. Sanchez

Due date: Fri Mar-23 (before midnight)

The purpose of this assignment is twofold. On one hand, you will work with shell **pipelines** and **redirection** commands. On the other hand, you will begin to put in practice some of the programming concepts that we'll cover in the course:

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

1) File Structure (10 pts)

After completing this assignment, the file structure of your project should look like this. To keep things simple with the bash pipelines, we've decided to place the `README`, the report files, and data files at the same level. Remember that `first` and `last` correspond to your first and last names.

```
hw03/
  README.md
  # report files
  hw03-first-last.Rmd
  hw03-first-last.md
  # data files
  nba2017-roster.csv
  gsw-height-weight.csv
  LAC.csv
  top10-salaries.csv
  code/
    binomial-functions.R
  images/
    ... # image files
```

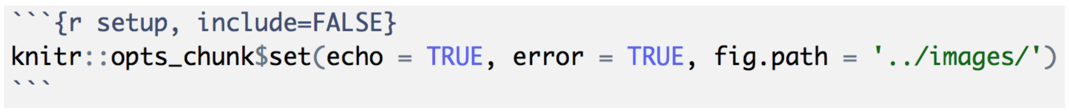
General Instructions

- Create a folder (i.e. subdirectory) `hw03` in your github classroom repository.
- Create a `README.md` file and include a description of what the HW is about.
- Create a folder `data` which will contain the data file.

- Create a folder `code` which will contain an `R` script file.
- Create a folder `images` which will contain some plot images.
- Create a folder `report` which will contain the files for your dynamic document (e.g. `Rmd` and derived files).
- In the yaml header of the `Rmd` file, set the `output` field as `output: github_document` (Do NOT use the default `"output: html_document"`).
- Name your `Rmd` file as `hw03-first-last.Rmd`, where `first` and `last` are your first and last names (e.g. `hw03-gaston-sanchez.Rmd`).
- Please do not use code chunk options such as: `echo = FALSE`, `eval = FALSE`, `results = 'hide'`. All chunks must be visible and evaluated.
- Use Git to *add* and *commit* the changes as you progress with your HW.
- And don't forget to *push* your commits to your github repository; you should push the `Rmd` and `md` files, as well as the generated folder and files containing the plot images and other outputs.
- Submit the link of your repository to bCourses. Do NOT submit any files (we will actually turn off the uploading files option).
- No html files will be taken into account (no exceptions).
- If you have questions/problems, don't hesitate to ask us for help in OH or in Piazza.

Before you start working on your `Rmd` ...

In your `Rmd` file 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, fig.path = '../images/')
```
~~~

By setting the global option `error = TRUE` you avoid the knitting process to be stopped in case a code chunk generates an error.

Since you willl be writing a couple of functions with `stop()` statements, it is essential that you set up `error = TRUE`, otherwise `"knitr"` will stop knitting your `Rmd` if it encounters an error.

Data file `nba2017-roster.csv`

You will need to get your own copy of the CSV file `nba2017-roster.csv` for this assignment. The file is in the github repo, inside the `data/` folder.

---

2) Pipelines and Redirection

For this part of the assignment, you may need to review tutorials 7 and 8, as well as the command-line cheat-sheet (files available in the course github repository)

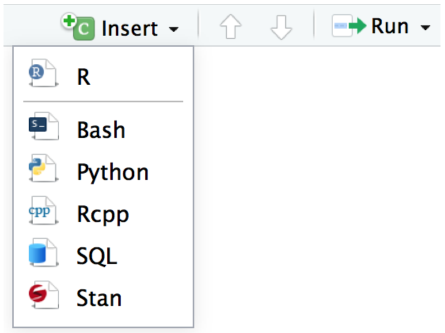
- 07-shell-redirections.md
- 08-shell-filters.md
- command-line-cheatsheet.pdf

Until now you've been working with code-chunks that execute R commands:

~~~
```{r example1}
# this is an R command
a <- 1
b <- 2
2*a + 3*b
```
~~~

```
[1] 8
```

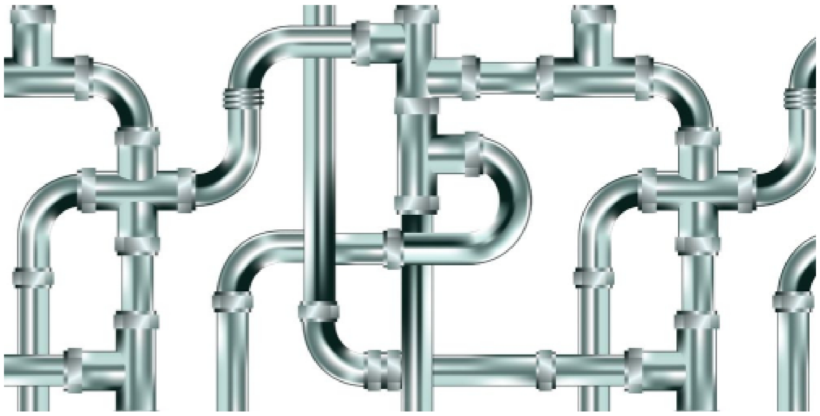
But you can also choose a different chunk flavor. If you click on the code-chunk icon, you'll see the list of available syntaxes that RStudio recognizes:



As you can tell from the image above, one of the code flavors is **Bash**. Which means that you could include a code chunk with bash-shell commands that get executed after you knit your `Rmd` file. Here's an example of a code-chunk with some bash-shell commands. Notice the use of the `bash` keywaord right after the opening brace `{`.

~~~
```{bash example2}
# these are shell commands
mkdir hw03
cd hw03
touch README.md
```
~~~

Piping



Use the `nba2017-roster.csv` data to perform the following tasks. All the commands have to be bash-shell commands (not R commands).

- `cut` allows you to *select* columns
- `grep` allows you to *filter* rows
- `sort` can be used to *arrange* lines
- `sort` could be used to *group by* lines
- `sort` and `uniq` can be used to *count* occurrences

2.1) Write a pipeline to obtain the unique team names, and redirect the output to a text file `team-names.txt`. Use `head` to display the first five lines of the created file (output shown belown).

```
"ATL"
"BOS"
"BRK"
"CHI"
"CHO"
```

2.2) Write a pipeline to obtain the unique positions, and redirect the output to a text file `position-names.txt`. Use `head` to display the first five lines of the created file (output shown belown).

```
"C"
"PF"
"PG"
"SF"
"SG"
```

2.3) Write a pipeline to obtain the counts (i.e. frequencies) of the different `experience` values, displayed from largest to smallest (i.e. descending order). Redirect the output to a text file `experience-counts.txt`. Use `head` to display the first five lines of the created file (output shown belown). The first column corresponds to count, the second column corresponds to experience.

```
  80 0
  52 1
  46 2
  36 3
  35 4
```

2.4) Use output redirection commands to create a CSV file `LAC.csv` containing data for the `LAC` team (Los Angeles Clippers). Your CSV file should include column names. Use `cat` to display the content of the created file (output shown belown).

```
"player","team","position","height","weight","age","experience","salary"
"Alan Anderson","LAC","SF",78,220,34,7,1315448
"Austin Rivers","LAC","SG",76,200,24,4,1.1e+07
"Blake Griffin","LAC","PF",82,251,27,6,20140838
"Brandon Bass","LAC","PF",80,250,31,11,1551659
"Brice Johnson","LAC","PF",82,230,22,0,1273920
"Chris Paul","LAC","PG",72,175,31,11,22868828
"DeAndre Jordan","LAC","C",83,265,28,8,21165675
"Diamond Stone","LAC","C",83,255,19,0,543471
"J.J. Redick","LAC","SG",76,190,32,10,7377500
"Jamal Crawford","LAC","SG",77,200,36,16,13253012
"Luc Mbah a Moute","LAC","SF",80,230,30,8,2203000
"Marreese Speights","LAC","C",82,255,29,8,1403611
"Paul Pierce","LAC","SF",79,235,39,18,3500000
"Raymond Felton","LAC","PG",73,205,32,11,1551659
"Wesley Johnson","LAC","SF",79,215,29,6,5628000
```

2.5) Write a pipeline to display the age frequencies of `LAL` players. The first column corresponds to count, the second column corresponds to age.

```
   2 19
   1 20
   2 22
   3 24
   2 25
   2 30
   2 31
   1 37
```

2.6) Write a pipeline to find the number of players in `CLE` (Cleveland) team; the output should be just the number of players.

```
      15
```

2.7) Write pipelines to create a CSV file `gsw-height-weight.csv` that contains the `player`, `height` and `weight` of `GSW` players. Your CSV file should include column names. Use `cat` to display the file contents:

```
"player","height","weight"
"Andre Iguodala",78,215
"Damian Jones",84,245
"David West",81,250
"Draymond Green",79,230
"Ian Clark",75,175
"James Michael McAdoo",81,230
"JaVale McGee",84,270
"Kevin Durant",81,240
"Kevon Looney",81,220
"Klay Thompson",79,215
"Matt Barnes",79,226
"Patrick McCaw",79,185
"Shaun Livingston",79,192
"Stephen Curry",75,190
"Zaza Pachulia",83,270
```

2.8) Write pipelines to create a file `top10-salaries.csv` containing the top10 player salaries, arranged by `salary` from largest to smallest. Your CSV file should include column names. Use `cat` to display the file contents:

```
"player","salary"
"LeBron James",30963450
"Russell Westbrook",26540100
"Mike Conley",26540100
"Kevin Durant",26540100
"James Harden",26540100
"DeMar DeRozan",26540100
"Al Horford",26540100
"Carmelo Anthony",24559380
"Damian Lillard",24328425
"Dwyane Wade",23200000
```

---

3) Binomial Probability Functions

Consider 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

R script `binomial-functions.R`

You will have to write code implementing the functions listed below. Write the functions in an `R` script `binomial-functions.R`, and save it inside the `code/` folder. The script file should have a header with fields like title, and description. In addition, all the functions must have Roxygen comments. For more information, refer to lab07.

- `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:

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

```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.

```r
# 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.

```r
# 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.

```r
# 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.

```
# 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.

```
# 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

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

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

```r
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()`

```r
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()`:

```r
# 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()`). 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.

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:

```r
# 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
```

`Rmd` file

In addition to including the bash-shell commands, your Rmd file should source your `binomial-functions.R` script: use `source()` to import your `R` script in your `Rmd` file. **Use a relative path!** (don't set absolute directories).

In your `Rmd` file report, write code to carry out the following computations

- Assume that the "successful" event is getting a "six" when rolling a die. Consider rolling a fair die 10 times. Use `bin_probability()` to find the probability of getting exactly 3 sixes.
- Use `bin_distribution()` to obtain the distribution of the number of "sixes" when rolling a loaded die 10 times, in which the number "six" has probability of 0.25. Make a plot of this distribution.
- Use `bin_probability()`, and a `for` loop, to obtain the probability of getting more than 3 heads in 5 tosses with a biased coin of 35% chance of heads.
- Use `bin_distribution()` to obtain the probability distribution of the number of heads when tossing a loaded coin 15 times, with 35% chance of heads. Make a plot of this distribution.