Lab/HW 2: Basic Data Structure - Solutions

Your lab/homework must be submitted in with two files: (1) R Markdown format file; (2) a pdf or html file. Other formats will not be accepted. Your responses must be supported by both textual explanations and the code you generate to produce your result.

Part I

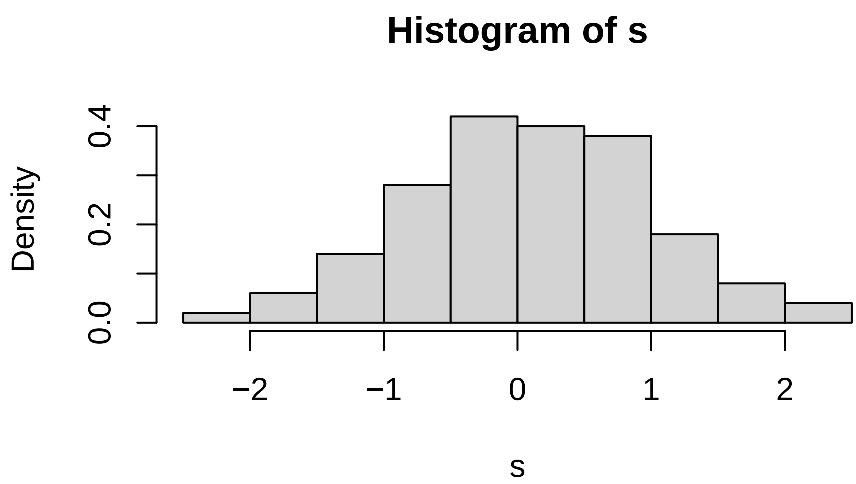
1. The normal distribution:

Explore the normal distribution in R by reading about related functions by `?rnorm`.

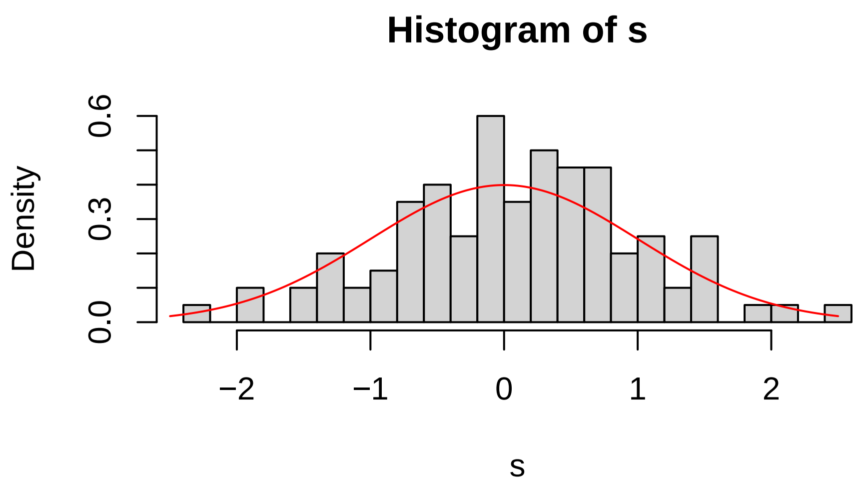
1. Generate $n = 100$ random draws from the standard normal distribution and assign those to the variable `s`.
2. What are the type and dimensions of the data that the function takes as input here?
3. What are the type and dimensions of the data that the function returns?
4. Use `hist()` to plot a histogram of `s`, normalized to be a density (so that the total area of the bars is 1, check `?hist`), make sure that the number of `breaks` (a parameter in the `hist()` function) does justice to your data, as the default value is not good enough at times. Justify your choice (you can provide several figures, prior knowledge or any other convincing argument).
5. Add a line to the histogram that shows the theoretical normal density:
   - First, determine the range you are interested in (the limits of the horizontal axis).
   - Generate a sequence that covers the above range and is dense enough to generate a smooth curve, assign it to a variable, say `x`. What is the variable type?
   - Find the appropriate function that will calculate for you the values of the density of the normal random variable. Use that to plot the additional line, in red please. What is the input data type for that density function? What is the data type of its output? Does the function operate element by element?
6. Plot another histogram of the standard normal CDF when applied to `s` (a transformation of a random variable is another random variable). Do you identify this distribution? Hint: start by identifying the range of values that are generated. If you are not sure, try taking a larger `n`.

- Extra credit: provide a proof in LaTeX to your assertion about how the above random variable is distributed.

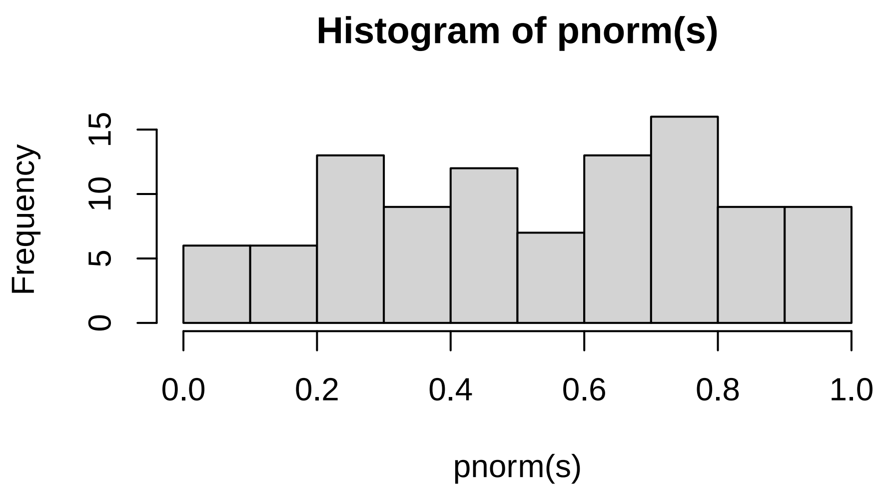
```r
set.seed(1)
# 1.
s = rnorm(100)
# 2. the function takes an integer scalar value
# (aka atomic integer vector of length 1) as input.
# 3. the function returns an atomic numeric vector of length 100.
# 4.
hist(s, freq = FALSE)
```



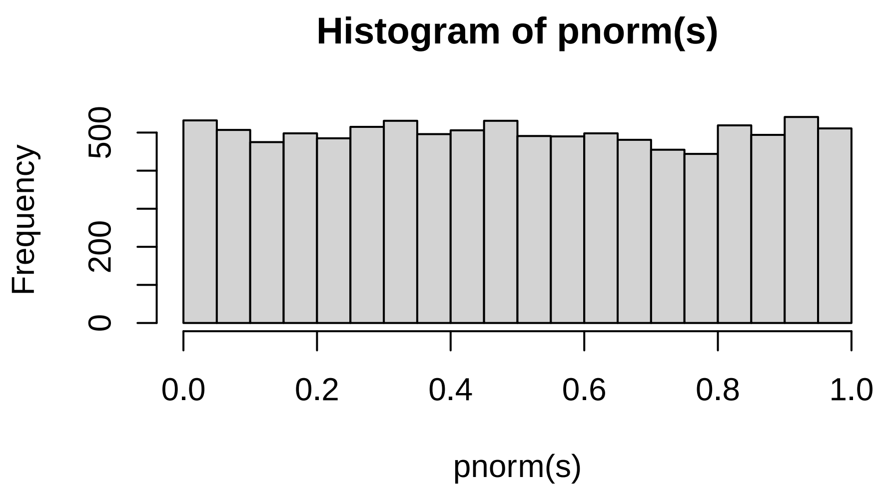

```r
hist(s, freq = FALSE, breaks = 20)
# 5.
x = seq(-2.5, 2.5, by = 0.01)
lines(x, dnorm(x), col = "red")
```




```r
# dnorm takes a numeric vector and returns a numeric vector,
# applies element by element.
# 6.
hist(pnorm(s))
```




```r
s = rnorm(10000)
hist(pnorm(s))
```




```r
# This looks like a U(0,1) random variable.
```

Claim: if $y \sim N(0,1)$, $\Phi(y) \sim U(0,1)$

Proof:

For $0 \leq C \leq 1$:

$$P(\Phi(y) \leq C) = P(y \leq \Phi^{-1}(C)) = \Phi(\Phi^{-1}(C)) = C$$

For $C < 0$:

$$P(\Phi(y) \leq C) = P(y \leq \Phi^{-1}(C)) = \Phi(\Phi^{-1}(C)) = \Phi(-\infty) = 0$$

For $C > 1$:

$$P(\Phi(y) \leq C) = P(y \leq \Phi^{-1}(C)) = \Phi(\Phi^{-1}(C)) = \Phi(\infty) = 1$$

So, the CDF of $\Phi(y)$ is given by:

$$F_{\Phi(y)}(C) = \begin{cases} 0 & \text{if } C < 0 \\ C & \text{if } 0 \leq C \leq 1 \\ 1 & \text{if } C > 1 \end{cases}$$

Which is indeed the CDF of the continuous uniform random variable supported on (0,1).

2. Sampling from the truncated normal distribution

"Sampling" is the process of generating draws from a random variable. Even though there are good methods to generate draws from distributions like the uniform, the normal and some others, it is sometimes hard (or very hard) to generate draws from many other random variables, even if their density and their probabilistic properties are known. The truncated normal is not so easy to sample from. In this homework we use a certain trick of sampling, say 10000 observations from the normal distribution and only taking observations in a certain range as samples for the truncated normal. Although the sample is indeed a sample of truncated normal draws (as is exemplified by comparing the histogram to the theoretical density) there still is a problem with this method: we do not control the exact number of samples that we generate.

The truncated normal distribution is the probability distribution derived from that of a normally distributed random variable by bounding the random variable from either below or above (or both) (See Wikipedia).

In this exercise we will generate samples from a truncated normal random variable in a very simple way.

Our goal is to sample from a truncated normal distribution that is derived from a standard normal variable that is truncated from below at 0.5.

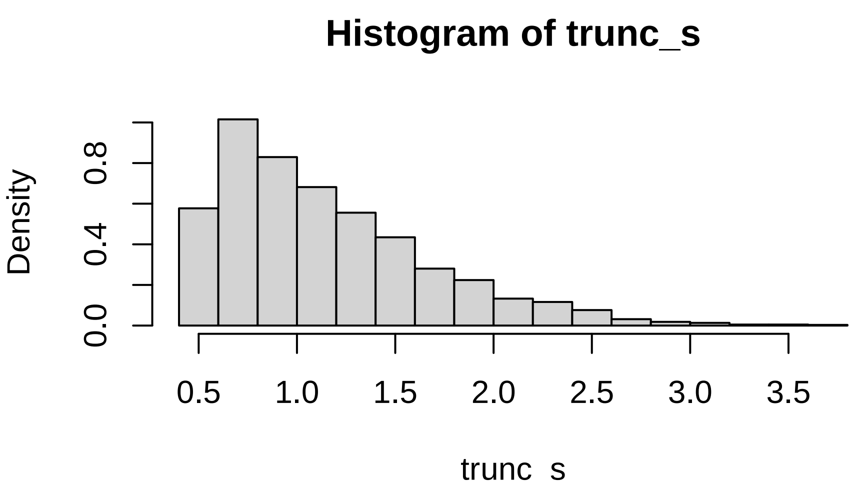
1. First, generate `n=10000` samples from a standard normal random variable, assign those to, say, a variable named `s`.
2. Then, using a single command, assign to a new variable, say, `trunc_s`, all the values of `s` that are greater than or equal to 0.5. **Explain in words exactly each stage of the computation, which functions are used, what are their input and output data types, etc.**

That is it! we generated a sample of truncated normal values.

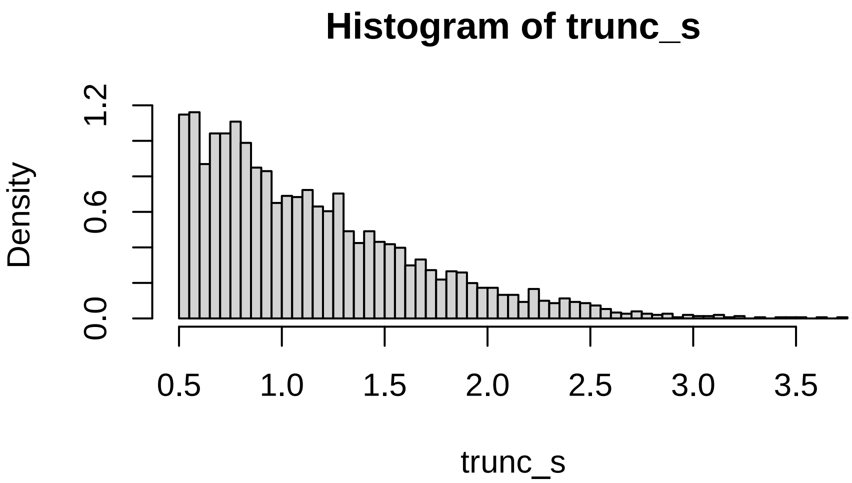
```
# 1.
s = rnorm(10000)
# 2.
trunc_s = s[s > 0.5]
# s > 0.5
# The ">" function takes as input the numeric vector s and the scalar 0.5.
# Compares each element of s to 0.5 and assigns TRUE/FALSE to the parallel
# location in a logical vector.
# Then, subsetting s by this logical vector s[logical vector] we only
# get the values of s what s is greater than to 0.5.
```

3. Use `hist()` to plot a histogram of `trunc_s`, normalized to be a density, make sure that the number of `breaks` does justice to your data. Justify your choice.

```
# 3.
hist(trunc_s, freq = FALSE)
```



```r
hist(trunc_s, freq = FALSE, breaks = 100)
```



4. Let us compare the results to the theoretical density. Choose a range for which you want to compute the density, and assign it to a vector. Using only functions for the standard normal density (and the formula for the density of the truncated normal random variable) that operate on the range vector or other constants (note, we are using $b = \infty$ here, what are the other constants?), compute the density of the truncated normal and add a red line to the histogram that depicts it. For legibility, you may separate the computations to, say, `numerator` and `denominator`. **Carefully describe each part of your computation: which functions operate on which data types? Is recycling used? Is the computation element by element?**

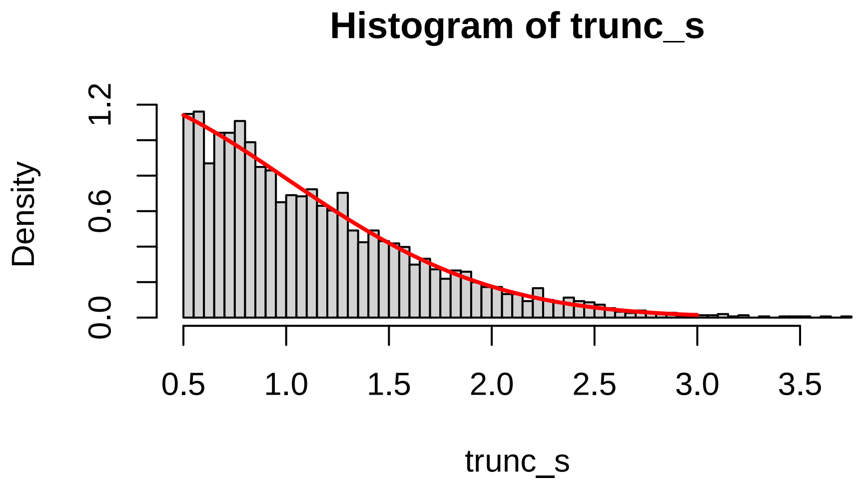
```r
# 4.
mu = 0
sigma = 1
a = 0.5
b = Inf
x = seq(0.5, 3, by = 0.01)

numerator = (1 / sigma) * dnorm( (x - mu) / sigma )
# x - mu: vector minus scalar (recycle)
#(x - mu) / sigma: vector divided by scalar (recycle)
# dnorm(...): vector input vector output (element by element)
# 1/sigma * dnorm(...): scalar * vector (recycling)

denominator = ( pnorm( (b - mu) / sigma) - pnorm( (a - mu) / sigma ) )
# all scalars, pnorm: scalar in scalar out

density_x = numerator / denominator

hist(trunc_s, freq = FALSE, breaks = 100)
lines(x, density_x, col = "red", lwd = 2)
```



5. Using the same careful descriptions as in the above, compute the expected value and standard deviation for this truncated normal distribution as well as the average and empirical standard deviation of the sample. Absurdly as it may sound, if you do this correctly, something should go wrong for the theoretical standard deviation. What went wrong? Fix it! Are the final values near?

```
# Expected value:
Z = pnorm(b) - pnorm(a)
mu + sigma * ( dnorm(a) - dnorm(b) ) / Z
```

```
## [1] 1.141078
```

```
# All scalars!
# Average:
mean(trunc_s)
```

```
## [1] 1.14858
```

```
# vector input scalar output, quite close!

# SD:
sqrt( sigma ^ 2 * (1 + ( a * dnorm(a) - b * dnorm(b) ) / Z - ( ( dnorm(a) - dnorm(b) ) / Z ) ) ^ 2 )
```

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

```
# Note: we get here NaN because we have b * dnorm(b) = 0 * Inf.
# So time to use some discretion:
sqrt( sigma ^ 2 * (1 + ( a * dnorm(a) ) / Z - ( dnorm(a) / Z ) ^ 2  ) )
```

```
## [1] 0.518151
```

```
sd(trunc_s)
```

```
## [1] 0.5302763
```

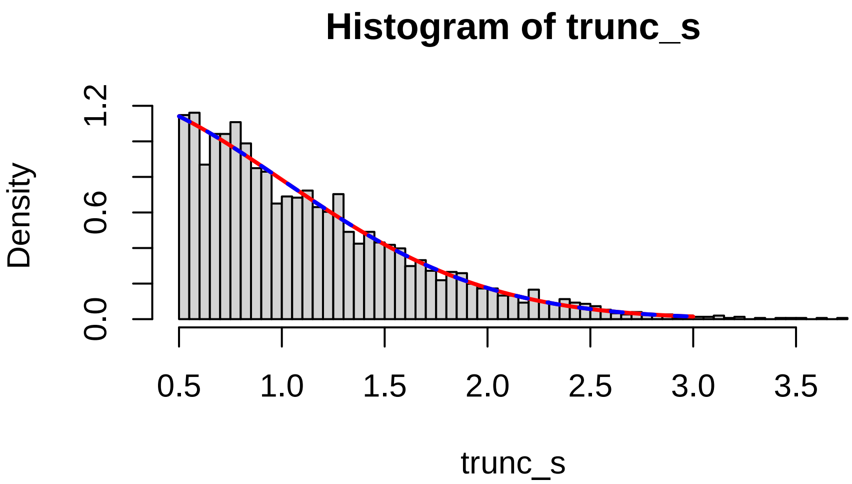
```
# Works!
```

6. Let us compare the results to the `truncnorm` package. First, install the package and load it. Then, apply `dtruncnorm` (with the correct parameters) to the same range you used before. Add another, blue dashed line to the histogram and check whether or not the lines are similar.

```
# 6.
library(truncnorm)

hist(trunc_s, freq = FALSE, breaks = 100)
```

```r
lines(x, density_x, col = "red", lwd = 2)
lines(x, truncnorm::dtruncnorm(x, a = 0.5), col = "blue", lty = 2, lwd = 2)
```



7. Are the values you computed for the truncated normal density and the ones from the package "the same" (check in at least 3 ways)?

```r
# 7.
all(density_x == dtruncnorm(x, a = 0.5))
```

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

```r
# Notice that:
max(abs(density_x - dtruncnorm(x, a = 0.5)))
```

```
## [1] 3.330669e-16
```

```r
identical(density_x, dtruncnorm(x, a = 0.5))
```

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

```r
all.equal(density_x, dtruncnorm(x, a = 0.5))
```

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

7. What is the problem/s with this sampling method? Exemplify with code.

(1) We cannot control exactly the length of the sample. (2) If `a` were larger, we'd get very few samples, so we'd need an enormous original sample size to generate a reasonable sample size of truncated normals.

```r
s = rnorm(10000)
trunc_s = s[s > 0.5]
length(trunc_s)
```

```
## [1] 3157
```

```r
s = rnorm(10000)
trunc_s = s[s > 0.5]
length(trunc_s)
```

```
## [1] 3035
```

```r
# (1) Different lengths!

# and (2):
```

```r
s = rnorm(10000)
trunc_s = s[s > 3.5]
length(trunc_s)
```

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

Extra credit: compute the expected length of the sample generated as a function of `a`. Is it "close" to the actual length that you got? You may sample several times and see how the results look like.

```r
# The expected length is computed by (normal_CDF(b) - normal_CDF(a)) * n:
(pnorm(Inf) - pnorm(0.5)) * 10000
```

```
## [1] 3085.375
```

```r
s = rnorm(10000)
trunc_s = s[s > 0.5]
length(trunc_s)
```

```
## [1] 3025
```

```r
s = rnorm(10000)
trunc_s = s[s > 0.5]
length(trunc_s)
```

```
## [1] 3095
```

```r
s = rnorm(10000)
trunc_s = s[s > 0.5]
length(trunc_s)
```

```
## [1] 3108
```

```r
s = rnorm(10000)
trunc_s = s[s > 0.5]
length(trunc_s)
```

```
## [1] 3033
```

```r
s = rnorm(10000)
trunc_s = s[s > 0.5]
length(trunc_s)
```

```
## [1] 3127
```

```r
s = rnorm(10000)
trunc_s = s[s > 0.5]
length(trunc_s)
```

```
## [1] 3042
```

```r
# Reasonably scattered around the expected value.
```

Part II

Syntax and class-typing.

```
a. For each of the following commands, either explain why they should be errors, or explain the non-erro

      vector1 <- c("5", "12", "7", "32")
      max(vector1)
      sort(vector1)
```

```
sum(vector1)
```

```
b. For the next series of commands, either explain their results, or why they should produce errors.
```

```
vector2 <- c("5",7,12)
vector2[2] + vector2[3]

list4 <- list(z1="6", z2=42, z3="49", z4=126)
list4[[2]]+list4[[4]]
list4[2]+list4[4]
```

Part II - solutions

Question 1

part a

vector1 <- c("5", "12", "7", "32")
max(vector1)
sort(vector1)
sum(vector1)

max(vector1) returns "7" because the max is comparing parts of the binary and 7 (7) is grater than 5 (5), 1 (12), and 3 (32).
sort(vector1) sorts the vector by the first digit. therefore 12 is considered the smallest because 1 is the minimum of the set. The order is "12" "32" "5" "7"
sum(vector1) does not yield a result because you cannot sum data labeled as characters.

part b

vector2 <- c("5",7,12)
vector2[2] + vector2[3]

This is not a valid operation because the addition is between two characters because the first data type is a character.

list4 <- list(z1="6", z2=42, z3="49", z4=126)
list4[[2]]+list4[[4]]
list4[2]+list4[4]

the first addition is valid because [[]] returns a numeric in the 2nd and 4th slot of the list.
the second addition is not valid because [] returns a value of type "list" and one cannot add value of lists together.

2. Some regression

The following code generates a sequence of `X` values from 1 to 100, each three times. It defines `Y` as a linear function of `X` plus normal noise.

```
n = 100
X = rep(1:n, each = 3)
Y = 0.5 + 2 * X + rnorm(100 * 3)
```

1. Carefully explain each of the above computations: which functions are in it, in which order, what are their respective inputs and outputs, how are the computations performed.

The `rep()` function admits several inputs: `1:n` is a numeric vector and `each` is an integer. `rep` then repeats each element of the input vector several times, according to the value of `each`. It thus returns a numeric vector of length `length(1:n) * 3`, in our case 300.

`rnorm()` admits as input an integer, and returns a numeric vector of that length where each of the entries is an independent draw from a standard normal random variable (here, the `"*"` function admits two numeric values, `100` and `3`, and returns their scalar product). `2 * X` admits the numeric vectors `2` and `X`, the first of length 1 the second of length 300. It recycles the shorter (i.e. `2`) and multiplies each element of `X` by it. It thus returns a vector of length 300. The right `"+"` function admits two numeric vectors of length 300 (`2 * X` and `rnorm(100 * 3)`), multiplies them element by element and returns a numeric vector of the same length. the left `"+"` function admits the numeric vectors `0.5` and `2 * X + rnorm(100 * 3)`, the first of length 1 the second of length 300. It recycles the shorter (i.e. `0.5`) and adds each element of `X` by it. It thus returns a vector of length 300.

2. The following code regresses `Y` on `X` and presents the estimated coefficients and predicted values (expected `Y` values for the same `X` values).

a. What are the corresponding data types?
b. Do the results make sense to you (regression-wise)? Why?

```
reg_Y_X = lm(Y ~ X)
coef(reg_Y_X)
```

```
## (Intercept)           X
##    0.419664    2.001212
```

```
predict(reg_Y_X)
```

```
##         1         2         3         4         5         6         7
##  2.420876  2.420876  2.420876  4.422088  4.422088  4.422088  6.423301
##         8         9        10        11        12        13        14
##  6.423301  6.423301  8.424513  8.424513  8.424513 10.425725 10.425725
##        15        16        17        18        19        20        21
## 10.425725 12.426937 12.426937 12.426937 14.428149 14.428149 14.428149
##        22        23        24        25        26        27        28
## 16.429361 16.429361 16.429361 18.430574 18.430574 18.430574 20.431786
##        29        30        31        32        33        34        35
## 20.431786 20.431786 22.432998 22.432998 22.432998 24.434210 24.434210
##        36        37        38        39        40        41        42
## 24.434210 26.435422 26.435422 26.435422 28.436635 28.436635 28.436635
##        43        44        45        46        47        48        49
## 30.437847 30.437847 30.437847 32.439059 32.439059 32.439059 34.440271
##        50        51        52        53        54        55        56
## 34.440271 34.440271 36.441483 36.441483 36.441483 38.442696 38.442696
##        57        58        59        60        61        62        63
## 38.442696 40.443908 40.443908 40.443908 42.445120 42.445120 42.445120
##        64        65        66        67        68        69        70
## 44.446332 44.446332 44.446332 46.447544 46.447544 46.447544 48.448756
##        71        72        73        74        75        76        77
## 48.448756 48.448756 50.449969 50.449969 50.449969 52.451181 52.451181
##        78        79        80        81        82        83        84
## 52.451181 54.452393 54.452393 54.452393 56.453605 56.453605 56.453605
##        85        86        87        88        89        90        91
## 58.454817 58.454817 58.454817 60.456030 60.456030 60.456030 62.457242
##        92        93        94        95        96        97        98
## 62.457242 62.457242 64.458454 64.458454 64.458454 66.459666 66.459666
##        99       100       101       102       103       104       105
```

```
##  66.459666  68.460878  68.460878  68.460878  70.462091  70.462091  70.462091
##        106        107        108        109        110        111        112
##  72.463303  72.463303  72.463303  74.464515  74.464515  74.464515  76.465727
##        113        114        115        116        117        118        119
##  76.465727  76.465727  78.466939  78.466939  78.466939  80.468151  80.468151
##        120        121        122        123        124        125        126
##  80.468151  82.469364  82.469364  82.469364  84.470576  84.470576  84.470576
##        127        128        129        130        131        132        133
##  86.471788  86.471788  86.471788  88.473000  88.473000  88.473000  90.474212
##        134        135        136        137        138        139        140
##  90.474212  90.474212  92.475425  92.475425  92.475425  94.476637  94.476637
##        141        142        143        144        145        146        147
##  94.476637  96.477849  96.477849  96.477849  98.479061  98.479061  98.479061
##        148        149        150        151        152        153        154
## 100.480273 100.480273 100.480273 102.481485 102.481485 102.481485 104.482698
##        155        156        157        158        159        160        161
## 104.482698 104.482698 106.483910 106.483910 106.483910 108.485122 108.485122
##        162        163        164        165        166        167        168
## 108.485122 110.486334 110.486334 110.486334 112.487546 112.487546 112.487546
##        169        170        171        172        173        174        175
## 114.488759 114.488759 114.488759 116.489971 116.489971 116.489971 118.491183
##        176        177        178        179        180        181        182
## 118.491183 118.491183 120.492395 120.492395 120.492395 122.493607 122.493607
##        183        184        185        186        187        188        189
## 122.493607 124.494820 124.494820 124.494820 126.496032 126.496032 126.496032
##        190        191        192        193        194        195        196
## 128.497244 128.497244 128.497244 130.498456 130.498456 130.498456 132.499668
##        197        198        199        200        201        202        203
## 132.499668 132.499668 134.500880 134.500880 134.500880 136.502093 136.502093
##        204        205        206        207        208        209        210
## 136.502093 138.503305 138.503305 138.503305 140.504517 140.504517 140.504517
##        211        212        213        214        215        216        217
## 142.505729 142.505729 142.505729 144.506941 144.506941 144.506941 146.508154
##        218        219        220        221        222        223        224
## 146.508154 146.508154 148.509366 148.509366 148.509366 150.510578 150.510578
##        225        226        227        228        229        230        231
## 150.510578 152.511790 152.511790 152.511790 154.513002 154.513002 154.513002
##        232        233        234        235        236        237        238
## 156.514214 156.514214 156.514214 158.515427 158.515427 158.515427 160.516639
##        239        240        241        242        243        244        245
## 160.516639 160.516639 162.517851 162.517851 162.517851 164.519063 164.519063
##        246        247        248        249        250        251        252
## 164.519063 166.520275 166.520275 166.520275 168.521488 168.521488 168.521488
##        253        254        255        256        257        258        259
## 170.522700 170.522700 170.522700 172.523912 172.523912 172.523912 174.525124
##        260        261        262        263        264        265        266
## 174.525124 174.525124 176.526336 176.526336 176.526336 178.527549 178.527549
##        267        268        269        270        271        272        273
## 178.527549 180.528761 180.528761 180.528761 182.529973 182.529973 182.529973
##        274        275        276        277        278        279        280
## 184.531185 184.531185 184.531185 186.532397 186.532397 186.532397 188.533609
##        281        282        283        284        285        286        287
## 188.533609 188.533609 190.534822 190.534822 190.534822 192.536034 192.536034
##        288        289        290        291        292        293        294
```

```
## 192.536034 194.537246 194.537246 194.537246 196.538458 196.538458 196.538458
##        295        296        297        298        299        300
## 198.539670 198.539670 198.539670 200.540883 200.540883 200.540883
```

Both results are *named* vectors. Coefficients of length 2, predictions of length 300. The names are informative of the respective coefficient, or the observation number (though it is a character vector). It makes sense - we generated data from a true regression line with $\beta_0 = 0.5$ and $\beta_1 = 2$, the estimates are pretty near those! The predictions also make sense - for the same X values we get the same Y values (this can also be verified by direct calculation).

```r
str(reg_Y_X, max.level = 1)
```

```
## List of 12
##  $ coefficients : Named num [1:2] 0.42 2
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "X"
##  $ residuals    : Named num [1:300] 0.317 -0.192 0.651 -0.92 -0.316 ...
##   ..- attr(*, "names")= chr [1:300] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:300] -1757.7 1000.556 0.651 -0.919 -0.316 ...
##   ..- attr(*, "names")= chr [1:300] "(Intercept)" "X" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:300] 2.42 2.42 2.42 4.42 4.42 ...
##   ..- attr(*, "names")= chr [1:300] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 298
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = Y ~ X)
##  $ terms        :Classes 'terms', 'formula'  language Y ~ X
##   .. ..- attr(*, "variables")= language list(Y, X)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. ..- attr(*, "term.labels")= chr "X"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(Y, X)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "Y" "X"
##  $ model        :'data.frame':   300 obs. of  2 variables:
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language Y ~ X
##   .. .. ..- attr(*, "variables")= language list(Y, X)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..- attr(*, "term.labels")= chr "X"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(Y, X)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "Y" "X"
##  - attr(*, "class")= chr "lm"
```

All of the list's elements are named, but some of the actual values are not named: - $coefficients : Named numeric vector of length 2 - $residuals : Named numeric vector of length 300 - $effects : Named numeric vector of length 300 - $ rank : Unnamed integer vector of length 1 - $ fitted.values : Named numeric vector of length 300 - $ assign : Unnamed integer vector of length 1 - $ qr : List of 5 - $ df.residual : Unnamed integer vector of length 1 - $ xlevels : Named list() - $ call : formula - $ terms : Classes 'terms', 'formula' language Y ~ X - $ model : data frame

So the `coef` function is simply equivalent to calling `reg_Y_X$coefficients`.