Introduction to R

`[ECON 4753]` — *University of Arkansas*

```r
library(gapminder) # install.packages("gapminder")
library(palmerpenguins) # install.packages("palmerpenguins")
```

1 — R as a Calculator

The first thing we will learn is how to use R as a calculator. You can use any of the math operators you want:

- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `^` Exponentiation

Let's experiment with some arithmetic expressions:

```r
1 + 1
```

```
[1] 2
```

```r
2 / 4
```

```
[1] 0.5
```

```r
3^2
```

```
[1] 9
```

Order of operations (via PEMDAS) apply here too:

```r
5 + 2 * 3
```

```
[1] 11
```

This does 2 * 3 first and then adds 5 to get 11. If we want to do 5 + 2 first, then we can wrap it in parenthesis (the P in PEMDAS):

```r
(5 + 2) * 3
```

```
[1] 21
```

There are even some operators that you might not know that have to do with remainders:

```r
# Get the remainder
13 %% 2
```

```
[1] 1
```

```r
# Divide and round down to the nearest decimal
13 %/% 2
```

```
[1] 6
```

Exercise Compute the sample average of the following sample of baby weights (in lbs.):

(7.7, 8.2, 8.3, 7.6, 9.2, 7.4, 11.1)

```r
## (7.7 + 8.2 + 8.3 + 7.6 + 9.2 + 7.4 + 11.1) / 7


## For variance, we can do:
## ((7.7 - 8.5)^2 + ......) / (7 - 1)
```

1.1. R as a functional programming language

R is based around functions. A function takes an input (or multiple inputs) and produces an output. There are many many functions in R, but first lets learn some calculator type functions. For example, if I want to take the square root, I can use the function `sqrt`. Here are some example of math functions:

The form of a function call is `function_name(arguments)`

1. The function name, `sqrt`, `abs`, `factorial`
2. Opening parenthesis `(`
3. The argument (in the future arguments)
4. Closing parenthesis `)`

For example `sqrt(16)` says to take the argument `16` and apply the function `sqrt` of it.

Exercise

1. Calculate the square root of 147
2. Try finding the natural log of 10, using `log()`
3. Practical usage: say the $Var(x) = 12$ and we have a sample size of 55. What is the standard deviation of the sample distribution of the sample mean?

1.2. Giving Things Names (i.e. Creating Variables)

Variables are immensely helpful in R. It lets you store values by giving them a `name` and then lets you access the variables later by name. I can assign variables using either `<-` or `=`.

Create variable `x` with value 5 and a variable `y` with value 20.

```
x <- 5
y <- 20
```

What is the sum of `x` and `y`?

```
x + y
```

```
[1] 25
```

Note the form of creating the variable:

1. The variable name, `x` and `y`
2. Assignment operator `<-` or `=`
3. The value we want to store.

The reason behind the left arrow is that the arrow points to variable name where we want to put the value into.

You can create a variable containing *text* by using `""`

```
instructor_name <- "Kyle Butts"
print(instructor_name)
```

```
[1] "Kyle Butts"
```

Exercise Use quotation marks to create a string and call it `my_name`.

```
my_name <- "Kyle Butts"
print(my_name)
```

```
[1] "Kyle Butts"
```

1.3. Glueing together strings

Often time we might want to combine text from multiple sources and/or add data to our strings. We can use the `paste`/`paste0` functions to combine together strings. `paste0` will append the strings as written, while `paste` will automatically add a space between each thing it is concatenating. Dealer's choice for which you prefer

For example,

```
paste0("Kyle", "Butts")
```

```
[1] "KyleButts"
```

```r
paste("Kyle", "Butts")
```

```
[1] "Kyle Butts"
```

What is cool about these functions is they take any number of arguments and append them together. For example, we can write a paste function that takes your height in inches and prints a human-readable string.

```r
height_in_inches <- 70
paste0("My height is ", height_in_inches %/% 12, "'", height_in_inches %% 12, "\"")
```

```
[1] "My height is 5'10\""
```

Note that numbers get automatically converted to a string. One subtle point you might have missed. If we start and end strings with double quotes, how can we include one in the text itself? Above, we did this with the escape key `\"`. You probably won't need to do this, but it's worth mentioning nevertheless.

Exercise Construct a string that reports on the average baby weight in your sample. It would be nice to create a variable that stores `mean_baby_weight` to make the code nicer to read. These were the weights: (7.7, 8.2, 8.3, 7.6, 9.2, 7.4, 11.1)

2 — Vectors

So far, we have dealth with scalar numbers and texts. But, when working with data, we will observe many units and need a way to store all their values together. This is where the bread and butter of data-science comes in: Vectors.

Vectors are a list of elements like integers, numbers, or strings. This is really useful for storing data! You use `c` to create a vector (pneumonically, `c` stands for combine).

```r
## Rebounds from 2023 NBA Season
rebounds <- c(260, 114, 252, 310, 165, 236, 148, 336, 941, 127, 384, 278, 300, 6, 136,
```

It is kind of a pain in the neck to write all these out; and worse, prone to errors! Later, we will learn how to load data from a file, making this much easier.

You can access elements of a vector by using `[#]`, where # is the i-th element you want

```
rebounds[1]
```

```
[1] 260
```

```
rebounds[2]
```

```
[1] 114
```

If you want to access more than one element, we can subset using *a vector!* (how meta):

```
rebounds[c(1, 2)]
```

```
[1] 260 114
```

One special syntax is ther `a:b` which generates $a, a+1, \ldots, b$. This makes it easy to grab the first 5 values:

```
rebounds[1:5]
```

```
[1] 260 114 252 310 165
```

Standard math operators work on vectors element by element:

```
rebounds[1:5] + 1
```

```
[1] 261 115 253 311 166
```

```
rebounds[1:5] / 12 # dozens of rebounds
```

```
[1] 21.66667  9.50000 21.00000 25.83333 13.75000
```

If we want to know how many elements are in a vector, we can use the `length` function:

```r
length(rebounds)
```

```
[1] 386
```

2.1. Summarizing vectors

The natural next step is to start trying to summarize the data. There are a set of built-in functions that provide statistical summaries of the data.

```r
## Mean, standard deviation, and variance
mean(rebounds)
```

```
[1] 250.4663
```

```r
sd(rebounds)
```

```
[1] 232.6362
```

```r
var(rebounds)
```

```
[1] 54119.62
```

```r
## Extremes
max(rebounds)
```

```
[1] 1530
```

```r
min(rebounds)
```

```
[1] 0
```

```r
## chaining functions
sqrt(var(rebounds))
```

```
[1] 232.6362
```

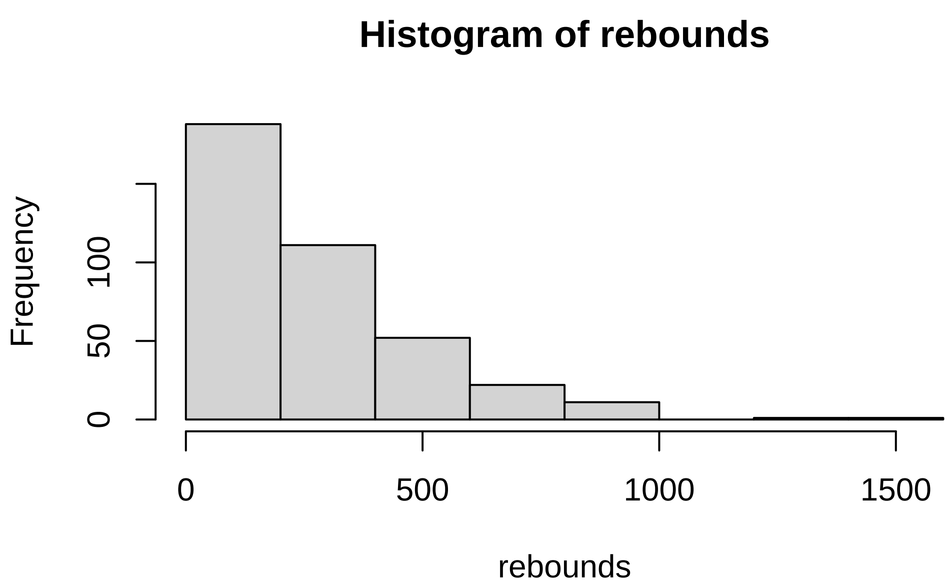
Some functions will take extra **arguments** that give more instructions. For example, the `quantile` function returns information about percentiles of the distribution. You can add extra arguments, separated by commas. For example, `quantile`'s first argument is a vector and the second argument is what percentiles you want to find.

```r
## Percentiles of the data
quantile(rebounds, c(0.1, 0.5, 0.9))
```

```
  10%   50%   90% 
 19.0 205.5 575.5 
```

We will talk in more details about plotting below, but for now we can use the `hist()` function to give us a view of the distribution of the vector

```r
hist(rebounds)
```



2.1.1 Help menu

How did I know the order of arguments for `quantile`? Are there ways to customize the histogram created by `hist`? In general, if you have a question about a function, then you need to address that function's **documentation**. To do so, click in to your console and type `?func_name` where `func_name` is the name of the function, e.g. `quantile`. At first, the information may be very overwhelming. The documentation in base R functions are very detailed, but are not great at introducing the functions. You should focus on the **Arguments** section and perhaps the **Examples** section, in my opinion. In particular, the order of the arguments is probably helpful in order of importance

Exercise Similar to `c`, the `seq` function creates a vector: a **seq**uence of numbers.

1. Create a sequence of all multiples of four from 4 to 100. Look at '`?seq` for help. Hint: The arguments you need here are `from`, `to`, and `by`. Store your vector in a varaible
2. Find the 19th element of this sequence
3. What is the sum of the 10th and 11th element of this sequence.

2.1.2 NAs

In the real world, some times we will not have a value for a variable for an individual (e.g. people don't fill answer a survey question). In `R`, this is represented as an `NA`.

```
reviews <- c(5, NA, 4, 4, 3, 5,  NA, 4, 5, 2)
```

What is the average (mean and median) review?

```
mean(reviews)
```

```
[1] NA
```

```
median(reviews)
```

```
[1] NA
```

By default, the statistical summary functions will all produce `NA` when they are present in the data. `R` wants you to *opt-in* to ignoring the missings. To do this, functions will take an extra **argument** called `na.rm`:

```
mean(reviews, na.rm = FALSE)
```

```
[1] NA
```

```
max(reviews, na.rm = FALSE)
```

```
[1] NA
```

A lot of this information will be presented to you by the `summary` function. For numberic vectors, `summary` produces the five-number summary, the mean, and the number of `NA`s (if any)

```
summary(reviews)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
   2.00    3.75    4.00    4.00    5.00    5.00       2 
```

2.1.3 Logical Vectors

So far, we have seen two kinds of vectors: numeric and character vectors. A third, common, vector is a **logical** vector:

```
ordered_takeout <- c(TRUE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, FALSE)
```

Logical vectors can take only two values: `TRUE` and `FALSE`. They can be treated as numbers by using `as.numeric()` with `TRUE` becoming `1` and `FALSE` becoming `0`.

```
as.numeric(ordered_takeout)
```

```
[1] 1 1 0 1 0 1 0 0
```

One common trick is to use `sum()` on a logical vector and it will return the number of `TRUE`s:

```r
sum(ordered_takeout)
```

```
[1] 4
```

Logical vectors are often returned by other operations. For example, we can check whether elements of a vector equal some value with ==. Other operators that produce a logical vector include >, <, >=, and <=.

```r
## Find five-star reviews
reviews == 5
```

```
 [1]  TRUE    NA FALSE FALSE FALSE  TRUE    NA FALSE  TRUE FALSE
```

```r
## Find three-star or lower
reviews <= 3
```

```
 [1] FALSE    NA FALSE FALSE  TRUE FALSE    NA FALSE FALSE  TRUE
```

Logical vectors can be **negated** using the ! operator:

```r
## Find non five-star reviews
!(reviews == 5)
```

```
 [1] FALSE    NA  TRUE  TRUE  TRUE FALSE    NA  TRUE FALSE  TRUE
```

You can also use the following operators to supply multiple criteria:

- & And operator. Both vector 1 **and** vector 2 must be true for the observation
- | Or operator. **Either** vector 1 **or** vector 2 must be true for the observation

```r
(reviews == 5) & (reviews == 4)
```

```
 [1] FALSE    NA FALSE FALSE FALSE FALSE    NA FALSE FALSE FALSE
```

```r
(reviews == 5) | (reviews == 4)
```

```
 [1]  TRUE    NA  TRUE  TRUE FALSE  TRUE    NA  TRUE  TRUE FALSE
```

```r
## Equivalent to
reviews >= 4
```

```
 [1]  TRUE    NA  TRUE  TRUE FALSE  TRUE    NA  TRUE  TRUE FALSE
```

2.1.4 Subsetting of vectors by logical

We have already discussed one way of subsetting vectors via a vector of integers. This is called "subsetting by index".

The other common way is by using a logical vector the same length of the vector you wish to subset. For example, let's look at the reviews for takeout orders and dine-in orders separately

```r
## Takeout orders
reviews[ordered_takeout]
```

```
[1]  5 NA  4  5  5  2
```

```r
## Dine-in orders
reviews[!ordered_takeout]
```

```
[1]  4  3 NA  4
```

Exercise

1. What is the average review for take-out orders?
2. The `is.na()` function takes a vector as an argument and returns a logical vector that equals `TRUE` if the element is `NA`. Using this and the `!`, subset the reviews to only non-`NA` values. What is the average review? Comapre this to `mean(reviews, na.rm = TRUE)`.

```
mean(reviews[!is.na(reviews)])
```

```
[1] 4
```

```
mean(reviews, na.rm = TRUE)
```

```
[1] 4
```

2.1.5 Vectorized operations

Often times we want to use multiple vectors for some calculation. Like single numbers, arithmetic can be done element-by-element with vectors. This means +, -, *, / and ^ all work on a vector.

```
x <- c(1, 2, 3)
y <- c(5, 5, 5)

x + y
```

```
[1] 6 7 8
```

```
x^2
```

```
[1] 1 4 9
```

There are two main features to remember: 1. The vectors you use should be of the same length (if they are not, some weird *recycling* rules occur that we will not discuss in this introduction). 2. Scalars are treated as a vector of the same length with that single number repeated for each element.

```
## Equivalent:
x + 2
```

```
[1] 3 4 5
```

```
x + rep(2, 3)
```

```
[1] 3 4 5
```

More, many functions are designed to be used on vectors element by element. All of the functions we used in the "Calculator" section do this:

```
exp(x)
```

```
[1]  2.718282  7.389056 20.085537
```

```
log(x)
```

```
[1] 0.0000000 0.6931472 1.0986123
```

```
sqrt(x^2)
```

```
[1] 1 2 3
```

Exercise

1. Try to guess the output of the following expression `2*x + y + 1`.

2.1.6 Sorting data

The final vector operation we will discuss is how to sort data; either ascending or descending in value. There are two ways to do this.

First, we can use the `sort()` function. The function takes a function and an optional `decreasing` argument. `decreasing` takes a logical `TRUE`/`FALSE` option. By default, it increases in values

```
sort(rebounds, decreasing = TRUE)
```

```
  [1] 1530 1258  941  934  921  909  899  870  862  845  830  829  807  770  762
 [16]  760  744  740  739  728  705  704  700  691  672  670  665  660  639  634
 [31]  633  631  630  618  607  593  580  580  578  573  564  564  556  551  549
 [46]  546  536  513  512  511  500  497  494  491  485  485  483  476  473  472
 [61]  469  468  460  454  453  451  450  449  448  447  439  435  434  432  429
 [76]  426  423  420  417  416  415  410  407  405  402  402  401  394  394  393
 [91]  391  384  381  372  350  346  344  342  336  336  330  324  319  318  317
[106]  317  314  312  312  310  310  310  308  307  305  305  301  301  300  298
[121]  296  296  296  295  295  292  292  289  286  286  282  281  278  275  272
[136]  271  270  269  269  268  265  265  261  260  260  260  259  258  257  257
[151]  257  257  256  255  253  253  252  252  249  247  247  247  246  245  244
[166]  243  240  239  238  236  236  235  233  233  233  231  227  227  227  223
[181]  223  222  220  220  219  213  211  210  209  208  206  206  206  205  204
[196]  202  202  201  191  190  189  188  188  187  185  184  184  184  183  182
[211]  182  179  178  177  173  171  170  168  168  165  162  162  161  161  152
[226]  150  149  148  147  145  145  145  145  144  142  136  132  129  127  124
[241]  118  118  118  116  114  112  112  111  110  106  105  102  101  101   99
[256]   99   98   98   97   96   96   96   95   95   95   94   92   90   88   85
[271]   85   84   82   82   81   81   81   80   79   78   78   78   77   76   75
[286]   74   71   70   69   69   69   66   66   63   63   63   61   58   56   55
[301]   54   54   54   52   51   51   49   47   47   46   45   43   42   42   41
[316]   41   39   39   37   36   35   35   34   34   34   33   33   32   30   30
[331]   30   30   28   28   27   26   26   26   26   25   24   24   24   22   22
[346]   21   19   19   19   16   16   15   15   14   12   11   11   11   11   10
[361]   10    9    8    8    7    6    5    5    5    5    3    2    2    2    1
[376]    1    1    1    1    1    0    0    0    0    0    0
```

Alternatively, we can use the `order()` function to get the *row indices* of the ordering:

```r
order(rebounds, decreasing = TRUE)
```

```
  [1] 298 379   9 313 113 215 177 244 347 252  25 295 139 310 141  85 216 111
 [19]  98  58 258 371 250 209  88 245 181 130 255 317 183  82 272  71 373 158
 [37]  65 127  41 219 217 271 263 265  64 197 273  30 356 180  96  97 238 101
 [55]  23  39 213  49 235  79 349 227 228 372 234 195 240 332 249 150 108 330
 [73]  33 138 194 266  60  19 327 294 315  87 163 377 129 189  67 204 285 282
 [91] 320  11 344 364 292  62  44 144   8 291 229  63 287 233 179 283 184 174
[109] 262   4 148 311 205 106 169 385 286 319  13 103  32 190 303 261 369 175
[127] 176  93  61 123 376  43  12 159  91 152 333  36 381 230 334 359 193   1
[145] 131 384  83 268 140 210 290 363 323 275 109 348   3 232 208  92 342 378
[163] 353  40  56 221 383  34 361   6 366  37  17 214 270 134  90 107 203  55
[181] 362  38 120 212 297 116 105 115 117 218 162 198 370 188 316 104 318  53
[199] 331 124  72  99 325  70 146  74 136 165 339 126 178 248 100 321 277 299
[217] 149 112 307   5  31  42  22 202  66 118 284   7 289  16 119 243 374 122
[235]  18  15 225 375  10 300 132 206 309 324   2  27 171 246 242 280 128 157
[253]  76 358  46 155 151 167 257  95 259 306  69 145 276 367 207 352 355 156
[271] 211  52  47 237  26 322 368 173  48  80 182 354 147 326 386 251 281  89
[289]  51  73 137 196 312  29  94 201 114  68 125 350 241 260 301  54 164 279
[307] 187 302 338 191 267 170 143 314  81 154  45 264 351  77  20 199  57  75
[325] 226  86 236 185 172 224 253 304 329 360 254 121 192 239 305 220 110 142
[343] 160  35 341 365 288 293 345  28 222 256 380 308  84  21 135 186 278 161
[361] 168 382 153 269  78  14  59 200 274 335  50 223 296 346  24 102 231 247
[379] 328 336 133 166 337 340 343 357
```

The first element of the resulting vector is the index of the maximum number (1530)

```r
order(rebounds, decreasing = TRUE)[1]
```

```
[1] 298
```

```r
which.max(rebounds)
```

```
[1] 298
```

What this means is that we can use the result of order to subset the vector to get the sorted vector:

```r
rebounds[order(rebounds, decreasing = TRUE)]
```

```
  [1] 1530 1258  941  934  921  909  899  870  862  845  830  829  807  770  762
 [16]  760  744  740  739  728  705  704  700  691  672  670  665  660  639  634
 [31]  633  631  630  618  607  593  580  580  578  573  564  564  556  551  549
 [46]  546  536  513  512  511  500  497  494  491  485  485  483  476  473  472
 [61]  469  468  460  454  453  451  450  449  448  447  439  435  434  432  429
 [76]  426  423  420  417  416  415  410  407  405  402  402  401  394  394  393
 [91]  391  384  381  372  350  346  344  342  336  336  330  324  319  318  317
[106]  317  314  312  312  310  310  310  308  307  305  305  301  301  300  298
[121]  296  296  296  295  295  292  292  289  286  286  282  281  278  275  272
[136]  271  270  269  269  268  265  265  261  260  260  260  259  258  257  257
[151]  257  257  256  255  253  253  252  252  249  247  247  247  246  245  244
[166]  243  240  239  238  236  236  235  233  233  233  231  227  227  227  223
[181]  223  222  220  220  219  213  211  210  209  208  206  206  206  205  204
[196]  202  202  201  191  190  189  188  188  187  185  184  184  184  183  182
[211]  182  179  178  177  173  171  170  168  168  165  162  162  161  161  152
[226]  150  149  148  147  145  145  145  145  144  142  136  132  129  127  124
[241]  118  118  118  116  114  112  112  111  110  106  105  102  101  101   99
[256]   99   98   98   97   96   96   96   95   95   95   94   92   90   88   85
[271]   85   84   82   82   81   81   81   80   79   78   78   78   77   76   75
[286]   74   71   70   69   69   69   66   66   63   63   63   61   58   56   55
[301]   54   54   54   52   51   51   49   47   47   46   45   43   42   42   41
[316]   41   39   39   37   36   35   35   34   34   34   33   33   32   30   30
```

```
[331]    30    30    28    28    27    26    26    26    26    25    24    24    24    22    22
[346]    21    19    19    19    16    16    15    15    14    12    11    11    11    11    10
[361]    10     9     8     8     7     6     5     5     5     5     3     2     2     2     1
[376]     1     1     1     1     1     0     0     0     0     0     0
```

This might seem like a bit silly, but it will prove useful when we want to sort multiple vectors at the *same time* based on one of the vectors.

3 — Dataframes (or, a group of vectors)

Dataframes are a special object in R. A dataframe is simply a collection of **vectors** and looks like a typical excel spreadsheet. The columns of a dataframe are each **vectors** that contain variables and a row contains an **observation**. This is the coding equivalent of an excel spreadsheet. If you are using Positron, clicking the dataframe in the `Variables` tab or typing `View(df_name)` into the console will let you interactively scroll though the data.

First, we will load some data.frames that come with a **package** in R. We can do that using the `data` function. Let's load the `penguins` data set which contain a census conducted on multiple species of penguins on a set of islands.

```
data(penguins, package = "palmerpenguins")
```

We can use the `head()` function to view the first few rows. It prints out the first 6 rows of the dataset so you can see the variables. BTW, this function works on vectors too!

```
head(penguins)
```

```
# A tibble: 6 x 8
  species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
1 Adelie  Torgersen           39.1          18.7               181        3750
2 Adelie  Torgersen           39.5          17.4               186        3800
3 Adelie  Torgersen           40.3          18                 195        3250
```

```
4 Adelie  Torgersen           NA            NA                NA          NA
5 Adelie  Torgersen           36.7          19.3             193        3450
6 Adelie  Torgersen           39.3          20.6             190        3650
# i 2 more variables: sex <fct>, year <int>
```

Another helpful function is `str()` which prints a similar format, but is a little easier to read, especially when there are a lot of variables in the dataset.

```
str(penguins)
```

```
tibble [344 x 8] (S3: tbl_df/tbl/data.frame)
 $ species          : Factor w/ 3 levels "Adelie","Chinstrap",..: 1 1 1 1 1 1 1 1 1 1
 $ island           : Factor w/ 3 levels "Biscoe","Dream",..: 3 3 3 3 3 3 3 3 3 3 ...
 $ bill_length_mm   : num [1:344] 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
 $ bill_depth_mm    : num [1:344] 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
 $ flipper_length_mm: int [1:344] 181 186 195 NA 193 190 181 195 193 190 ...
 $ body_mass_g      : int [1:344] 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
 $ sex              : Factor w/ 2 levels "female","male": 2 1 1 NA 1 2 1 2 NA NA ...
 $ year             : int [1:344] 2007 2007 2007 2007 2007 2007 2007 2007 2007 2007 ..
```

Exercise

1. What constitutes a row in the penguins dataframe? What constitutes a column?

3.1. Accessing individual observations/variables

To access an individual variable, we can use the $ operator. We use the `dataframe$varible` symbol to extract `variable` from the `dataframe`. For example, let's grab the species variable from penguins

Try grabing the `species` and the `sex` variables us

```
penguins$species
```

```
  [1] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
```

```
  [8] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [15] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [22] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [29] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [36] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [43] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [50] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [57] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [64] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [71] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [78] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [85] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [92] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
 [99] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[106] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[113] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[120] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[127] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[134] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[141] Adelie    Adelie    Adelie    Adelie    Adelie    Adelie    Adelie
[148] Adelie    Adelie    Adelie    Adelie    Adelie    Gentoo    Gentoo
[155] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[162] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[169] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[176] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[183] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[190] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[197] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[204] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
[211] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo
```

```
[218] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo   
[225] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo   
[232] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo   
[239] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo   
[246] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo   
[253] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo   
[260] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo   
[267] Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo    Gentoo   
[274] Gentoo    Gentoo    Gentoo    Chinstrap Chinstrap Chinstrap Chinstrap
[281] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[288] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[295] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[302] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[309] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[316] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[323] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[330] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[337] Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap Chinstrap
[344] Chinstrap
Levels: Adelie Chinstrap Gentoo
```

```
penguins$sex
```

```
  [1] male   female female <NA>   female male   female male   <NA>   <NA>  
 [11] <NA>   <NA>   female male   male   female female male   female male  
 [21] female male   female male   male   female male   female female male  
 [31] female male   female male   female male   male   female female male  
 [41] female male   female male   female male   male   <NA>   female male  
 [51] female male   female male   female male   female male   female male  
 [61] female male   female male   female male   female male   female male  
 [71] female male   female male   female male   female male   female male  
```

```
 [81] female male   female male   female male   male   female male   female
 [91] female male   female male   female male   female male   female male
[101] female male   female male   female male   female male   female male
[111] female male   female male   female male   female male   female male
[121] female male   female male   female male   female male   female male
[131] female male   female male   female male   female male   female male
[141] female male   female male   female male   male   female female male
[151] female male   female male   female male   male   female female male
[161] female male   female male   female male   female male   female male
[171] female male   male   female female male   female male   <NA>   male
[181] female male   male   female female male   female male   female male
[191] female male   female male   female male   male   female female male
[201] female male   female male   female male   female male   female male
[211] female male   female male   female male   female male   <NA>   male
[221] female male   female male   male   female female male   female male
[231] female male   female male   female male   female male   female male
[241] female male   female male   female male   female male   male   female
[251] female male   female male   female male   <NA>   male   female male
[261] female male   female male   female male   female male   <NA>   male
[271] female <NA>   female male   female male   female male   male   female
[281] male   female female male   female male   female male   female male
[291] female male   male   female female male   female male   female male
[301] female male   female male   female male   female male   female male
[311] male   female female male   female male   male   female male   female
[321] female male   female male   male   female female male   female male
[331] female male   female male   male   female male   female female male
[341] female male   male   female
Levels: female male
```

The `[,]` operator wil let us subset rows and columns. Before comma = rows and After comma = columns

```r
penguins[1:5, "species"]
```

```
# A tibble: 5 x 1
  species
  <fct>
1 Adelie
2 Adelie
3 Adelie
4 Adelie
5 Adelie
```

```r
penguins[1:5,]
```

```
# A tibble: 5 x 8
  species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
  <fct>   <fct>              <dbl>         <dbl>             <int>       <int>
1 Adelie  Torgersen           39.1          18.7               181        3750
2 Adelie  Torgersen           39.5          17.4               186        3800
3 Adelie  Torgersen           40.3          18                 195        3250
4 Adelie  Torgersen           NA            NA                  NA          NA
5 Adelie  Torgersen           36.7          19.3               193        3450
# i 2 more variables: sex <fct>, year <int>
```

```r
penguins[, "species"]
```

```
# A tibble: 344 x 1
   species
   <fct>
 1 Adelie
 2 Adelie
 3 Adelie
```

```
 4 Adelie
 5 Adelie
 6 Adelie
 7 Adelie
 8 Adelie
 9 Adelie
10 Adelie
# i 334 more rows
```

```
penguins[1:10, c("species", "island")]
```

```
# A tibble: 10 x 2
   species island
   <fct>   <fct>
 1 Adelie  Torgersen
 2 Adelie  Torgersen
 3 Adelie  Torgersen
 4 Adelie  Torgersen
 5 Adelie  Torgersen
 6 Adelie  Torgersen
 7 Adelie  Torgersen
 8 Adelie  Torgersen
 9 Adelie  Torgersen
10 Adelie  Torgersen
```

You can pair these together, for example let's say I want the variable `island` for the first 6 observations:

```
penguins[1:6, "island"]
```

```
# A tibble: 6 x 1
  island
```

```
  <fct>
1 Torgersen
2 Torgersen
3 Torgersen
4 Torgersen
5 Torgersen
6 Torgersen
```

```
penguins[1:6, ]$island
```

```
[1] Torgersen Torgersen Torgersen Torgersen Torgersen Torgersen
Levels: Biscoe Dream Torgersen
```

3.1.1 Exercise

1. Use the `unique()` function to find the unique values of the variable `species` in the penguins dataset.

```
unique(penguins$species)
```

```
[1] Adelie    Gentoo    Chinstrap
Levels: Adelie Chinstrap Gentoo
```

2. Use the `table()` function to findhow many penguins there are of each species.

3.2. Loading data into R

In R, you can either load data from a website or from a computer. Usually data is found in a .csv file, but sometimes it will be in different forms that R can read.

```
# From a website
fandago <- read.csv("https://raw.githubusercontent.com/kylebutts/UARK_4753/refs/heads/
head(fandago)
```

	FILM	RottenTomatoes	RottenTomatoes_User	Metacritic
1	Avengers: Age of Ultron (2015)	74	86	66
2	Cinderella (2015)	85	80	67
3	Ant-Man (2015)	80	90	64
4	Do You Believe? (2015)	18	84	22
5	Hot Tub Time Machine 2 (2015)	14	28	29
6	The Water Diviner (2015)	63	62	50

	Metacritic_User	IMDB	Fandango_Stars	Fandango_Ratingvalue	RT_norm	RT_user_norm
1	7.1	7.8	5.0	4.5	3.70	4.3
2	7.5	7.1	5.0	4.5	4.25	4.0
3	8.1	7.8	5.0	4.5	4.00	4.5
4	4.7	5.4	5.0	4.5	0.90	4.2
5	3.4	5.1	3.5	3.0	0.70	1.4
6	6.8	7.2	4.5	4.0	3.15	3.1

	Metacritic_norm	Metacritic_user_nom	IMDB_norm	RT_norm_round
1	3.30	3.55	3.90	3.5
2	3.35	3.75	3.55	4.5
3	3.20	4.05	3.90	4.0
4	1.10	2.35	2.70	1.0
5	1.45	1.70	2.55	0.5
6	2.50	3.40	3.60	3.0

	RT_user_norm_round	Metacritic_norm_round	Metacritic_user_norm_round
1	4.5	3.5	3.5
2	4.0	3.5	4.0
3	4.5	3.0	4.0
4	4.0	1.0	2.5
5	1.5	1.5	1.5
6	3.0	2.5	3.5

	IMDB_norm_round	Metacritic_user_vote_count	IMDB_user_vote_count
1	4.0	1330	271107

```
2           3.5                          249                65709
3           4.0                          627               103660
4           2.5                           31                 3136
5           2.5                           88                19560
6           3.5                           34                39373
  Fandango_votes Fandango_Difference
1          14846                 0.5
2          12640                 0.5
3          12055                 0.5
4           1793                 0.5
5           1021                 0.5
6            397                 0.5
```

However, most common is to download the data and put it in the folder where your `.Rmd` file is. To load data you will need to find the file location. Remember to set your working directory for this.

```
# penguins <- read.csv("penguins.csv")
```

4 — Distribution Functions

5 — Plotting

6 — Statistics