# R Tutorial 2

*William Bell*

*2019-03-04*

Alright, now we've dipped our toe in, and it is time to do something more advanced.

Sometimes you don't just want to do arithmetic, and often what you want to do is actually highly repetitive. For these occasions, I give you, loops!

Suppose for example that I wanted to add consecutive numbers together, say, from 1 to 10. Then I could do the following:

```
x <- 0
i <- 1

while (i <= 10) {
  x <- x+i
  i <- i+1
}

x
```

```
## [1] 55
```

This is what's called a `while` loop. How it works is that first I initiated the variable I wanted to keep (what will be the sum), then I initated an *iterator* at the first value I wanted to add. Then it gets to the loop. The first thing it does is checks the condition `i` indeed starts less than or equal to 10, so it runs the things inside the loop. Then it checks the condition again, if it is still satisfied, then it runs the loop again. Eventually we see that our iterator will be 11, and then the condition will be no longer satisfied and the loop will stop and we'll get a number back.

The `while` loop has very simple syntax. First you put the word `while`, then you put the condition in round brackets. Finally, you put what you want to be done if the condition is met in curly brackets.

The `while` loop is good for cases when the condition might take a variable amount of time before it is met. This is often true with very advanced computational algorithms, but not with the sorts of work we're going to be doing. For our cases, it often makes more sense to use `for` loops. Let's consider how we'd do the above problem with a `for` loop:

```
x <- 0

for (i in 1:10) {
  x <- x+i
}

x
```

```
## [1] 55
```

Here we see we didn't need to initiate the iterator ourselves, and we don't need to manually update it after each time step. Instead, the iterator runs through all of the values of a vector (`1:10`) given to it. Once it has gotten to the end of that vector, it stops.

There's a third kind of loop called a `repeat` loop, that doesn't have a condition, it just repeats itself until you break it manually. This is the least common kind of loop and isn't generally recommended.

The most valuable role for loops, which is not demonstrated in the above example, is in indexing along a vector, list, or whatever. Suppose for example we wanted to take the cumulative sum of a vector up to some point. Then what we want to do is add all the values from before that point to that point. Here is how we could do that using a loop:

```
a <- c(0.190525195964261, 0.0268993464682746, 0.106119740210273, 0.0297077813524987,
0.161392441554454, 0.14469647980909, 0.0974859843608399, 0.114553780381864,
0.0811194913893519, 0.0474997585090919)

length(a)
```

```
## [1] 10
```

```
for (i in 2:10) {
  a[i] <- a[i] + a[i-1]
}

a
```

```
##  [1] 0.1905252 0.2174245 0.3235443 0.3532521 0.5146445 0.6593410 0.7568270
##  [8] 0.8713808 0.9525002 1.0000000
```

Fortunately we don't have to do this because there's a function for it called `cumsum`.

Loops are one of the more difficult concepts we'll be covering so we'll stop here, and try to look at a few examples.

## Examples

1. Finding if a number is prime. Write a function that determines if a number is prime. **(Recall: A prime number is a number larger than one divisible only by one and itself, e.g. 2, 3, 5, 7, 11...)**

```
isPrime <- function(n) {
  ...
}
```

2. Now that you have a function to check if a number is prime, write a loop that checks for each of 1 to 1000 whether they're prime.

## Back to Control Flow

Loops are an essential concept, but they are just one kind of **control flow**. These are generally functions that control whether some line or lines of code actually run, and how often they run. The other main examples are `if`, `else`, and `else if` (`break` is the other main control flow function).

If does basically what it sounds like it does, it checks whether some condition is true and then it runs some code if it is true.

```
x <- "This variable is a character vector"
y <- 0

if (x == "This variable is a character vector") {
  print("Oh golly")
} else {
  print("Oh gee")
}
```

```
## [1] "Oh golly"
```

```
if (y == "This variable is a character vector") {
  print("Oh golly")
} else {
  print("Oh gee")
}
```

```
## [1] "Oh gee"
```

This can be useful if you want to pick out or do something to only some of your data, or you want to do one thing to some of your data and something else to the rest. For instance, sometimes a dataset will list missing data by including the code -999. In that case, we might want to change those to 'Not Applicable' (NA), we can do that like so:

```
nnnToNA <- function(x) { ## This is not the best
  ## implementation of this, in fact you can do
  ## it without loops or if statements by using
  ## logical subscripting
  for (i in 1:length(x)) {
    if (x[i] == -999) {
      x[i] <- NA
    }
  }
  x
}

y <- c(1:6, -999, 4:12)

y
```

```
##  [1]    1    2    3    4    5    6 -999    4    5    6    7    8    9   10
## [15]   11   12
```

```
nnnToNA(y)
```

```
##  [1]  1  2  3  4  5  6 NA  4  5  6  7  8  9 10 11 12
```

## Some Useful Functions

Some functions simply turn out to be useful really often, and so it is in your best interest to have them in the back of your head when you do anything. So we will cover a few of these.

There are a few basic tasks you might want to do in R, one is creating some vector that follows some general pattern, reading and writing data to/from files, another is manipulating data, and finally you might want to summarize data. We will cover functions that perform these three activities.

### 1. Creating Things that Follow Patterns

For this, there are a couple functions you should know, but almost anything you want can be done with four functions: `rep` takes in a vector and repeats it a particular number of times.

e.g.

```
rep(1, 10)
```

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

```
rep(0:3, 3)
```

```
##  [1] 0 1 2 3 0 1 2 3 0 1 2 3
```

The `seq` function can produce a sequence with any step size, unlike the `:` operator which only operates in steps of 1.

e.g.

```
seq(1, 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

```
seq(10, 100, by = 5)
```

```
##  [1]  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90
## [18]  95 100
```

In fact the `:` operator calls the `seq` function underneath the surface.

### 2. Input/Output (Reading and Writing to Files)

For the work we'll be doing, you first get a dataset into R from twitter. You might want to save it for later, or to look at in Excel. Behind the scenes I've collected a dog dataset like the ones we've already collected:

```
head(dogdf[ , 2:9])
```

```
##   favorited favoriteCount replyToSN             created truncated
## 1     FALSE             0      <NA> 2019-03-07 14:43:05     FALSE
## 2     FALSE             0      <NA> 2019-03-07 14:43:01      TRUE
## 3     FALSE             0      <NA> 2019-03-07 14:42:41      TRUE
## 4     FALSE             0      <NA> 2019-03-07 14:42:32      TRUE
## 5     FALSE             0      <NA> 2019-03-07 14:42:26     FALSE
## 6     FALSE             0      <NA> 2019-03-07 14:42:11     FALSE
##   replyToSID                  id replyToUID
## 1       <NA> 1103667388581474304       <NA>
## 2       <NA> 1103667373008134144       <NA>
## 3       <NA> 1103667290820800518       <NA>
## 4       <NA> 1103667249183932416       <NA>
## 5       <NA> 1103667225997778945       <NA>
## 6       <NA> 1103667163250991104       <NA>
```

This can be saved as a csv file (which can be opened in Excel) as follows:

```
write.csv(dogdf, "dogTweets.csv", row.names = F)
```

I set the `row.names` argument to FALSE so that it doesn't add numbering in front of the tweets, which I find annoying. You can find where the csv file is saved by checking `getwd()`, which is where your R session is looking for and saving files.

Eventually you will need to load the data back into R, and when you need to do that, you can use the `read.csv` function.

```
dogdfRead <- read.csv("./dogTweets.csv", stringsAsFactors = F)
```

For our data I suggest *always* set the `stringsAsFactors` argument to FALSE. The details don't matter, suffice to say that you aren't interested in using factors for this dataset. If you have your working directory the same as when you wrote the data to a file, then the line above should work exactly as written. Otherwise you might need to set your working directory. This can be done by navigating in the bottom right window in

RStudio and using the little gear, or with the `setwd` function in R, look into the way file paths are written on your kind of computer (PC vs Mac/Linux). In short, for our purposes you can use absolute file paths which start from the main drive on your computer, e.g. ":C\Desktop..." or ":C\Documents" on a PC, and "~/Desktop/..." or "~/Downloads/..." on a Mac/Linux computer. If you're already in the correct general working directory, then you can use *relative* file paths like I used in the example code above where the period basically means, "starting from our current working directory".

There are faster ways and ways with better defaults for how to transfer data into and out of R, for instance `read_csv` in the `readr` package and `fread` in the `data.table` package. However the basic functions above are good enough for our purposes, and are generally good enough unless you're working with ~100k rows of data.

### 3. Manipulating Data

We've already explored subsetting in a previous tutorial, which explains how to get a portion of your data that we're interested in. In addition to these operations, let's consider three more: `rbind`, and `dplyr::inner_join`.

These functions are primarily for the purpose of collecting together different datasets which are tied by a common feature.

`rbind` combines datasets that have the same columns by row. So for instance, suppose I have tweets on dogs from today, and tweets on dogs from yesterday, and I want to consider them all together. Then what I must do is attach the two datasets by column.

e.g. these two datasets are the same kind, but taken from different survey locations at two different sites. We can consider them together:

```
##   monarchButterfly turtle
## 1             TRUE  FALSE
## 2             TRUE  FALSE
## 3            FALSE   TRUE
## 4            FALSE   TRUE
```

```
##   monarchButterfly turtle
## 1             TRUE   TRUE
## 2            FALSE   TRUE
## 3             TRUE   TRUE
## 4             TRUE   TRUE
```

```
combinedData <- rbind(foundAtSite1, foundAtSite2)

combinedData
```

```
##   monarchButterfly turtle
## 1             TRUE  FALSE
## 2             TRUE  FALSE
## 3            FALSE   TRUE
## 4            FALSE   TRUE
## 5             TRUE   TRUE
## 6            FALSE   TRUE
## 7             TRUE   TRUE
## 8             TRUE   TRUE
```

... and `rbind` allows to do that in a simple manner.

For the next operation which allows for combining datasets, we're going to use a package that adds to R. This package is called `dplyr`, and is useful if you want to really develop programming skills for data manipulation. However for our purposes we will only focus on one function from it, called `inner_join`.

Consider the following two datasets (mercilessly stolen from this introduction to joins):

```
library(readr)
suppressPackageStartupMessages(library(dplyr))

superheroes <- "
    name, alignment, gender,         publisher
 Magneto,       bad,   male,            Marvel
   Storm,      good, female,            Marvel
Mystique,       bad, female,            Marvel
  Batman,      good,   male,                DC
   Joker,       bad,   male,                DC
Catwoman,       bad, female,                DC
 Hellboy,      good,   male, Dark Horse Comics
"
superheroes <- read_csv(superheroes, skip = 1)

publishers <- "
  publisher, yr_founded
         DC,       1934
     Marvel,       1939
      Image,       1992
"
publishers <- read_csv(publishers, skip = 1)
```

These datasets are clearly related, but we wouldn't be able to combine them straightforwardly by row. What we want to do in order to get the information of both datasets is match up the publisher column in both datasets with each other. We can do this with a join.

The inner join combines them by a specified column, and only keeps rows that appear in both datasets based on the column used to combine them. So in this case:

```
newSuperheroes <- inner_join(superheroes, publishers, by = "publisher")

newSuperheroes
```

```
## # A tibble: 6 x 5
##   name     alignment gender publisher yr_founded
##   <chr>    <chr>     <chr>  <chr>          <dbl>
## 1 Magneto  bad       male   Marvel          1939
## 2 Storm    good      female Marvel          1939
## 3 Mystique bad       female Marvel          1939
## 4 Batman   good      male   DC              1934
## 5 Joker    bad       male   DC              1934
## 6 Catwoman bad       female DC              1934
```

We see that since Dark Horse Comics was not in the publishers dataset, Hellboy was removed, and since no superhero from Image comics was in the superheroes dataset, Image was removed. What remains is the information from both however, including the year that the publisher was founded with each surviving row regarding the superheroes.

We will use the `inner_join` for exactly one operation, which will be introduced in the next tutorial. However the generally idea is very useful if you're going to do more work with datasets that are connected but not of the same kind as in the publisher/superhero case above (I've had cause to use them in my own work, with relational datasets between papers found in a systematic review, and rows of summary statistics from those various papers kept in separate files).

### 4. Summary (Descriptive) Statistics

Finally, once we have information like the above, we need to sort through it and get the important information. I'm neither prepared nor qualified to give you a lesson in statistics, but I can teach you various descriptive statistics or figures that you can calculate which, while they aren't enough to make an inference, are enough to get you started. These include mean, median, standard deviation, correlations, sums, proportions, and your sample size.

Consider the geyser time dataset provided on Avenue to Learn or here. We will first determine the sample size. Then we will calculate the mean and median time between eruptions, and length of eruptions for this geyser. Then we will calculate the standard deviations for both of these variables, and the Pearson correlation between the two variables. We will calculate the number of eruptions and proportion that are above the mean and below it for the length of the eruption.

First, let's read the dataset into R:

```
geyserTime <- read.csv("./geyserTime.csv")

head(geyserTime)
```

```
##   Eruption.Length..min. Wait.Time.since.Previous.Eruption..min.
## 1              4.016667                                       77
## 2              4.216667                                       89
## 3              3.833333                                       88
## 4              4.166667                                       80
## 5              4.200000                                       89
## 6              4.350000                                       89
```

We see that the dataset has two columns, both numerical. The dataset is for Old Faithful, a geyser in Yellowstone National Park, over the year 2001. There are two columns with informative names, the first is Eruption Length in minutes, and the second is the time in-between eruptions, also in minutes. The first column is double, meaning that it includes decimal places, and the second is integer, meaning that it does not.

The data is in R as a dataframe, which is just a bunch of vectors right beside each other of the same length. You can subset it in a similar way to how you subset vectors, except you have to do it by row and column. So for instance:

```
## If I want the 2nd row, 2nd column:

geyserTime[2, 2]
```

```
## [1] 89
```

```
## If I want everything in the 2nd row:

geyserTime[2, ]
```

```
##   Eruption.Length..min. Wait.Time.since.Previous.Eruption..min.
## 2              4.216667                                       89
```

```
## If I want everything in the 2nd column:

geyserTime[ , 2]
```

```
## [1] 77 89 88 80 89
##  [ reached getOption("max.print") -- omitted 1501 entries ]
```

```
## If I want the column named Eruption.Length.min:

geyserTime$Eruption.Length..min.
```

```
## [1] 4.016667 4.216667 3.833333 4.166667 4.200000
##  [ reached getOption("max.print") -- omitted 1501 entries ]
```

The first feature we want to determine is the sample size. Since every row represents a distinct observation, the sample size is simply the number of rows.

```
nrow(geyserTime)
```

```
## [1] 1506
```

The `nrow` function calculates the number of rows, and is complemented by the `ncol` function which calculates the number of columns.

There are two columns, so let's calculate the mean and median for both, there are functions for both with highly self-evident names:

```
mean(geyserTime$Eruption.Length..min.)
```

```
## [1] 4.15726
```

```
median(geyserTime$Eruption.Length..min.)
```

```
## [1] 4.166667
```

```
mean(geyserTime$Wait.Time.since.Previous.Eruption..min.)
```

```
## [1] 90.83798
```

```
median(geyserTime$Wait.Time.since.Previous.Eruption..min.)
```

```
## [1] 91
```

The mean is the average, and the median is the 'middle' of the data, so we expect them to be similar. The main difference is we expect the mean to be more sensitive to outliers (points that lay far away from the main group of points). However for this dataset, the mean and median seem to be largely in agreement.

We can also get a greater range of information from the summary function:

```
summary(geyserTime)
```

```
##  Eruption.Length..min. Wait.Time.since.Previous.Eruption..min.
##  Min.   : 1.067        Min.   : 47.00
##  1st Qu.: 4.067        1st Qu.: 86.00
##  Median : 4.167        Median : 91.00
##  Mean   : 4.157        Mean   : 90.84
##  3rd Qu.: 4.300        3rd Qu.: 95.75
##  Max.   :18.967        Max.   :127.00
```

However we're not just interested in where the central tendency of the data, we're also interested in the spread - or how widely distributed the data is. The most popular measure for this is the standard deviation, which purports, more-or-less, to be a measure of the distance of the average point from the mean. We can calculate it with the `sd` function:

```
sd(geyserTime$Eruption.Length..min.)
```

```
## [1] 0.8721003
```

```r
sd(geyserTime$Wait.Time.since.Previous.Eruption..min.)
```

```
## [1] 8.388747
```

They're very different, but we should expect this, since the magnitudes of the time between eruptions is much larger than the length of eruptions, so we expect there to be more variation in the former.

Another measure we might want to calculate is the Pearson correlation between the two variables. The Pearson correlation is roughly speaking, the amount of variation in one parameter that is explained by variation in the other parameter. The Pearson correlation needs to be tested for significance, but I will suggest a basic procedure for that to save you guys the trouble of developing a model for your data:

```r
cor(geyserTime$Eruption.Length..min.,
    geyserTime$Wait.Time.since.Previous.Eruption..min.,
    method = "pearson")
```

```
## [1] 0.03870653
```

```r
permuteCor <- function(n, x, y, method = "pearson") {
  cors <- rep(NA, n)
  for (i in seq_len(n)) {
    cors[i] <- cor(x, sample(y), method = method, use = "complete.obs")
  }
  actual <- cor(x, y, method = method, use = "complete.obs")
  pval <- mean(actual <= cors)
  list(pValue = pval, nullCor = mean(cors))
}

## A p-value:
set.seed(1)

cat("The p-value is ", permuteCor(10000, geyserTime$Eruption.Length..min.,
    geyserTime$Wait.Time.since.Previous.Eruption..min.,
    method = "pearson")[[1]])
```

```
## The p-value is  0.057
```

Generally speaking, anything below ~0.4 is low and unlikely to be significant except for very large datasets, and indeed that is the case here based off of the function I just ran to test statistical significance. It turns out that there is not really much of a systematic relationship between the variables of interest here.

Finally I wish to calculate the number of eruptions above and below the mean for the length of eruption. We'll make that our cut off.

```r
cutoff <- mean(geyserTime$Eruption.Length..min.)

geyserTimeBelow <- geyserTime[cutoff > geyserTime$Eruption.Length..min., ]

nrow(geyserTimeBelow)
```

```
## [1] 694
```

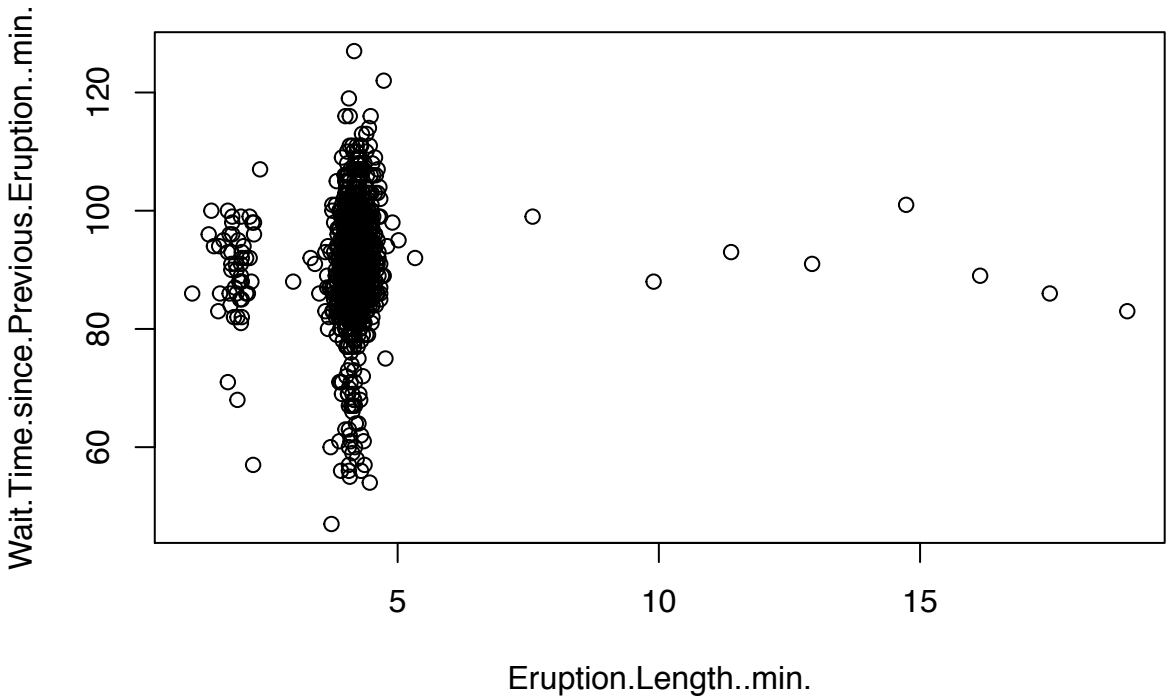
```r
geyserTimeAbove <- geyserTime[cutoff <= geyserTime$Eruption.Length..min., ]

nrow(geyserTimeAbove)
```

```
## [1] 812
```

Intriguingly, there are far fewer datapoints below than above, suggesting that they're further away in order to balance out the ones above.

In preparation for our next tutorial, I am going to make a scatterplot so that you can look at the data.

```
plot(geyserTime)
```




We see there are a few outliers in terms of eruption length, but that for the most part they're all very consistent. We see that there are two clusters in the data, which is another interesting phenomenon we could choose to investigate.

## Conclusion

In this tutorial we've covered loops, joins, and some basic useful functions for working with data. Next we will cover methods specifically for text-based data, including string manipulation methods and basic sentiment analysis.