Basic Computing 2 — Introduction to R, Part 2*

Peter Carbonetto and Aarti Venkat *University of Chicago*

Like Part 1, in Part 2 we will continue learning R by analyzing tabular data—that is, a *data frame*—but in Part 2 the table is much larger. Some of the skills we learned in Part 1 will transfer to this large-scale data setting. But we will also have to learn some new skills and practices to be able to cope with larger and more complex data. In Part 2, we will also sharpen our practices to prepare ourselves for analyzing data in the Real World.

The Tornado Super Outbreak of 1974

From *University of Chicago Magazine*, Fall, 2020:

> Fujita published his proposed tornado scale in 1971, but it needed a high-profile event to take root. On April 3, 1974, a tornado touched down in Morris, Illinois, around noon. Over the next 17 hours, 148 confirmed tornadoes tore through 13 states and Ontario, Canada. Following the 1974 Super Outbreak—one of the worst tornado outbreaks on record—Fujita and his team took a whirlwind airplane tour of more than 10,000 miles, surveying the ruins.

Fujita's scale is now known the "F-scale", and it scores tornadoes from F0 to F5 based on wind speeds and ensuing damage.

Analysis aims

Our main analysis aim is to uncover evidence for the 1974 Tornado Super Outbreak in data from NOAA's Severe Weather Data Inventory (SWDI). We will also perform a *comparative analysis* of these data to better understand the importance of this Super Outbreak in tornado climatology.

Do I have what I need?

From GitHub, download the tutorial materials to your computer, and make sure you know where to find them.

Obviously, you will need R, and, optionally, RStudio. Also, you will need the **ggplot2** and **cowplot** packages. If you haven't already done so, go ahead and install these two packages.

Where am I?

Make sure your R working directory in is the same directory containing the tutorial materials; you can run `getwd()` and `list.files()` to check this.

*This document is included as part of the Basic Computing 2—Introduction to R tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2022. **Current version**: August 11, 2022; **Corresponding author**: pcarbo@uchicago.edu. Thanks to Stefano Allesina, John Novembre, Stephanie Palmer and Matthew Stephens for their guidance.

Quit applications that are not needed and other "clutter" to reduce distractions. *Remember, the computer is device of distraction.*

Also, it is best if you start with a fresh workspace; you can refresh your environment by selecting **Session > Clear Workspace** from the RStudio menu.

Initial steps to explore the SWDI data

Here we will reuse some of the skills we learned in Part 1 to get a basic understanding of the SWDI data.

Since the SWDI data are stored as a CSV file, you should now know what to do to import the data into R:

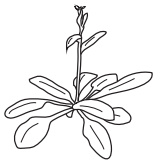
```
storms <- read.csv("StormEvents_details-ftp_v1.0_d1974_c20220425.csv.gz",
                   stringsAsFactors = FALSE)
```

Let's run a few lines of code to get an overview of the data frame:

Write your code here.

In our search for the Tornado Super Outbreak, we will use these seven columns: EVENT_TYPE, BEGIN_DAY, MONTH_NAME, STATE, BEGIN_LON, BEGIN_LAT and TOR_F_SCALE. Let's look more closely at these columns:

Write your code here.



A. thaliana

Preparing the tornado data

Our initial examinations suggest a few improvements to the data frame. *What are the improvements should we make?*

Write your notes here.

Now let's make these improvements:

Write your code here.

Since our focus is a particular type of storm event—tornadoes—let's extract the rows of the table that are relevant to our analysis.

Write your code here.

M. musculus

Keeping track of what we have done

Although we haven't made any breakthroughs yet, we have nonetheless made some progress: we have gained some basic insight into the data, and we have carefully prepared the data for our subsequent analyses. This is a good time to create a *script*—that is, the lines of code that reproduce the steps of our analysis. A script acts as both *a record of what we have done* and should, ideally, be *self-contained* so that it allow us to quickly *automate the analysis*.

Let's write a *minimal script* that reproduces the two most essential steps: (1) importing the data, and (2) prepping the data.

Write your code here.

Exercise: Check that your script is *reproducible* by running your script after clearing your workspace.

What are some reasons for writing a script?

Write your notes here.

Now that we have extracted and prepared the data, we are now ready to dive more deeply into these data. We will proceed by writing code to answer some questions.

When did the tornadoes occur?

Write your code to answer this question:

Write your code here.

Where did the tornadoes occur?

Write your code to answer this question. To help you with this, I have written a *custom function* that takes a data frame "latlongs" as input and outputs a map of the US with the geographic locations projected onto it:

```
# Creates a map of the United States overlaid with points given by
# their geographic co-ordinates (lats and longs).
# The inputs are two numeric vectors of the same length.
# The output is a ggplot object.
map_usa_latlongs <- function (lats, longs) {
  dat <- data.frame(lat = lats,long = longs)
  return(ggplot(dat,aes(x = long,y = lat)) +
         geom_path(data = map_data("state"),
                   aes(x = long,y = lat,group = group),
                   color = "gray") +
         geom_point(shape = 20,size = 1) +
         theme_classic())
}
```

Write your code here.

What are the outliers?

Plotting the tornadoes by geographic location revealed some "outliers". Write some code to track down these outliers in the table, then remove them from the table:

Write your code here.

Exercise: Mapping the Super Outbreak

Create a map of the tornadoes again, this time focussed on the tornadoes that occurred on the single day of the tornado Super Outbreak. *You may be able to reuse the* `map_usa_latlongs` *custom function to answer this question.*

Write your code here.

Compare your map to https://en.wikipedia.org/wiki/1974_Super_Outbreak.

Activity: How unusual was the 1974 Tornado Super Outbreak?

Using our R skills, and the data provided by the NOAA, we recreated a detailed picture of the 1974 Tornado Super Outbreak—the same events that were carefully studied by Fujita and his colleagues.

Uncovering an exciting result—whether it is expected or surprising—is often only the beginning of your research. Typically, more analyses will need be done to confirm your result. This may involve collecting more data, or refining your analyses, or both.

As mentioned, creating *scripts* is an important element of your data analysis practice.

Here, we expand on this idea and develop a more flexible *generic script* that can be quickly reused to perform multiple analyses. As we develop more complex analyses, another important skill is knowing *when*—and *how*—to *reuse code*.

A natural question is whether there was something special about 1974. Are these "bursts" of tornadoes common, or exceedingly rare? To answer this question, we would need to download and analyze the storm event data from many years, not just 1974. Suddenly our large-scale data analysis has gotten even larger.

This is your task. Create a more flexible script that automates all the steps of the analysis: (1) load the data, (2) prepare the data, (3) extract the tornado data, and (4) create a plot showing number of tornadoes by date. This is mostly the same as what we did before, *but with a twist*: this new script is more flexible because it can analyze tornado event data from *any year*, not just 1974. To do this, define a new variable "year" and use this variable in your script to modify the analysis according to the target year.

Hint: You should be able to write this script by adapting the code we have written above. The function `paste` or `paste0` may also be useful.

The git repository includes SWDI data from 1974, 1975 and 1976 which you can use to test your script.

Optional: Download more storm event data from the NOAA SWDI database and try running your script on those data.

Discuss: Does this script achieve our goal of comparing tornado patterns across different years? What are ways in which this comparison could be improved?

Challenge problem: Use a "for loop" to automate your analysis—including the step to download the CSV files from the SWDI database—for all years from 1974 to 2020.

A few closing remarks

You will often find that it isn't long into a data analysis—sometimes after writing as little as 10–20 lines of code—when you run into problems. Some of these problems can be avoided with Good Practices. Examples of Good Practices include:

1. Keeping track of what you have done in *self-contained scripts.*
2. Giving names to variables that help you remember what they are for.
3. Including short, high-level comments explaining what the code does.

(How well did follow these practices in the examples above?)

One powerful idea is to identify a useful bit of code, then package this bit of code into a *function* that can be quickly reused.

E. coli

Programming challenge

Instructions

You will work with your group to solve the exercises below. When you have found the solutions, go to https://jnovembre.github.io/BSD-QBio8 and follow the link "Submit solution to challenge 2" to submit your answer.

Google Flu Trends

Google Flu started strong, with a paper in *Nature* (Ginsberg *et al*, 2009, doi:10.1038/nature07634) showing that, using data on Web search engine queries, one could predict the number of physician visits for influenza-like symptoms. Over time, the quality of predictions degraded considerably, requiring many adjustments to the model. Now defunct, Google Flu Trends has been proposed as a poster child of "Big Data hubris" (Lanzer *et al*, *Science*, 2014, doi:10.1126/science.1248506). In the folder containing the Basic Computing 2 tutorial materials, you will find the data used by Preis and Moat in their 2014 paper (doi:10.1098/rsos.140095) to show that, after accounting for some additional historical data, Google Flu Trends are correlated with outpatient visits due to influenza-like illnesses.

1. Read the data using function `read.csv` and plot the number of weekly outpatient visits versus the Google Flu Trends estimates.
2. Calculate the (Pearson's) correlation using the `cor` function.
3. The data span 2010–2013. In August 2013, Google Flu changed their algorithm. Did this lead to improvements? Compare the data from weeks starting in August and September 2013 with the data from weeks starting in August and September 2010, 2011 and 2012. For each, calculate the correlation, and see whether the correlation is higher for 2013.

Hint: You will need to extract the year from a string for each row. There are several ways to do this, but one simple way is using `substr(gf$WeekCommencing,1,4)`, in which `gf` is the data frame containing the Google Flu data.

P. polytes

Other topics

Packages

R is the most popular statistical computing software among biologists. One reason for its popularity is the availability of many packages for tackling specialized research problems. These packages are often written by biologists for biologists. You can contribute a package, too! The RStudio website (goo.gl/harVqF) provides guidance on how to start developing R packages. See also Hadley Wickham's free online book (r-pkgs.had.co.nz).

You can often find highly specialized packages to address your research questions. Here are some suggestions for finding an appropriate package. The Comprehensive R Archive Network (CRAN) offers several ways to find specific packages for your task. You can browse the full list of CRAN packages (goo.gl/7oVyKC). Or you can go to the CRAN Task Views (goo.gl/0WdIcu) and browse a compilation of packages related to a topic or discipline.

From within R or RStudio, you can also call the function `RSiteSearch("keyword")`, which submits a search query to the website search.r-project.org. The website rseek.org casts an even wider net, as it not only includes package names and their documentation, but also blogs and mailing lists related to R. If your research interests are to high-throughput genomic data or other topics in bioinformatics or computational biology, you should also search the packages provided by Bioconductor (goo.gl/7dwQlq).

Installing a package

Suppose you want to install the rsvd package. The **rsvd** package provides functions to quickly perform singular value decompositions (SVD) and principal components analysis (PCA) on large data sets. To install the package, run:

```
install.packages("rsvd")
```

Or, in RStudio, select the **Packages** panel, and click **Install**.

Loading packages

Once it is successfully installed, to load the **rsvd** package into your R environment, run:

```
library(rsvd)
```

Once you have loaded the package, you can use, for example, the rpca function for running PCA. To access the documentation that explains what rpca does, and how to use it, type

```
help(rpca)
```

Now suppose you would like to access the "bacteria" data set, which reports the incidence of *H. influenzae* in Australian children. The data set is included with **MASS** package. If you try to access these data before loading the package, you will get an error:

```
data(bacteria)
```

First, you need to load the package:

```
library(MASS)
```

Now the data set is available, and you can load it:

```
data(bacteria)
head(bacteria)
```

C. elegans

Random numbers

Your will sometimes need to generate random numbers. (They are actually "pseudorandom" numbers because they are not perfectly random.) In fact, random numbers are needed in the "case study" below.

R has many functions to sample random numbers from different statistical distributions. For example, use `runif` to create a vector containing 10 random numbers:

```
runif(10)
```

Question: What kind of random numbers are generated by `runif`? How could you check this?

To sample from a set of values, use `sample`:

```
v <- c("a", "b", "c", "d")
sample(v, 2)                    # Sample without replacement.
sample(v, 6, replace = TRUE)    # Sample with replacement.
sample(v)                       # Shuffle the elements.
```

The normal distribution is one of the most commonly used distributions, so naturally there is a function in R for simulating from the normal distribution:

```
rnorm(3)                    # Three draws from the standard normal.
rnorm(3, mean = 5, sd = 4) # Change the mean and standard deviation.
```

Exercise: The normal distribution has a familiar shape. Use `rnorm` to generate a large number of values from the standard normal, then use `hist` to draw a histogram of these values (you can adjust the number of bins in the histogram with the `n` argument). Is the histogram "bell shaped"? Use `mean`, `median` and `sd` to verify that the random numbers recover the expected properties of the normal distribution. Here is some code to get you started:

```
set.seed(1)
x <- rnorm(10000)
hist(x, n = 64)
```

Why is `set.seed` useful? What happens if we remove the call to `set.seed`?

Writing functions

It is good practice to subdivide your analysis into functions, and then write a short "master" program that calls the functions and performs the analysis. In this way, the code will be more legible, easier to debug, and you will be able to recycle the functions for your other projects.

In `R`, every function has this form:

```
my_function_name <- function (arg1, arg2, arg3) {
  #
  # Body of the function.
  #
  return(return_value) # Not required, but most functions output something.
}
```

Here is a very simple example:

```
sum_two_numbers <- function (a, b) {
  s <- a + b
  return(s)
}
sum_two_numbers(5, 7.2)
```

In R, a function can return only one object. If you need to return multiple values, organize them into a vector, matrix or list, and return that; e.g.,

```
sum_and_prod <- function (a, b) {
  s <- a + b
  p <- a * b
  return(c(s,p))
}
sum_and_prod(5, 7.2)
```

Here is a more interesting function. It accepts two arguments, `x` and `s`, and returns the density of the normal distribution with zero mean and standard deviation `s` at `x`. This is the mathematical formula for the normal density with mean zero and standard deviation `s`:

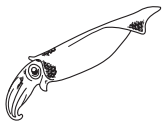
$$\frac{e^{-(x/s)^2/2}}{\sigma\sqrt{2\pi}}$$

Let's call this function `normpdf`:

```
normpdf <- function (x, s) {
  y <- exp(-(x/s)^2/2)/(sqrt(2*pi)*s)
  return(y)
}
```

Exercise: Check that this function gives the correct answers by comparing to the built-in function `dnorm`.

When developing and testing your function, remember this rule: *Whatever happens in the function stays in the function.* Inside `normpdf`, a new object, `y`, is created. But you will not see `y` in your environment (unless you happen to already have an object named `y`).



D. pealeii

Vectorization

`R` has a feature called *vectorization*—many operations in `R`, including most of the basic mathematical operations, are automatically applied to all values in a vector. (We briefly learned about vectorization in Basic Computing 1.) Let's check whether `R` automatically vectorizes the `normpdf` function by running this code:

```
n <- 1000
x <- seq(-3, 3, length.out = n)
y <- normpdf(x, 1)
print(y)
plot(x, y, type = "l")
```

Activity: Which operations in `normpdf` were applied 1,000 times, and which were applied just once? It may not be immediately obvious just by looking at the code, so to investigate this question, try running parts of the code in the console, e.g., `exp(-(x/s)^2/2)`, and see what happens.

Vectorization is very powerful, but it may take time to get comfortable with it, and know when it will work.

Conditional branching

When we want a block of code to be executed only when a certain condition is met, we can write a conditional branching point. The syntax is as follows:

```
if (condition is met) {
  # Execute this block of code.
} else {
  # Execute this other block of code.
}
```

For example, try running these lines of code (you might want to try running them a few times):

```
x <- rnorm(1)
if (x < 0) {
  msg <- paste(x, "is less than zero")
} else if (x > 0) {
  msg <- paste(x, "is greater than zero")
} else {
  msg <- paste(x, "is equal to zero")
}
print(msg)
```

We have created a conditional branching point, so that the value of `msg` changes depending on whether `x` is less than zero, greater than zero, or equal to zero.

 *D. rerio*

Activity: Improved normal probability density function

The probability density function of the normal distribution is not defined if the standard devation is less than zero. When the standard deviation is exactly zero, the density is a "spike" at zero; it is `Inf` exactly at $x = 0$, and zero everywhere else. The pseudocode for this improved normal probability density function might look something like this:

```
normpdf(x, s)
  if s < 0
    return NaN
  else if s = 0
    if x = 0
      return Inf
    else
      return 0
  else
    evaluate normal pdf with s.d. s at x
```

Using this pseudocode as a guide, write an improved `normpdf` function:

Activity: The quadratic formula

There is a famous formula used to solve for x in the quadratic equation $ax^2 + bx + c = 0$. It is called the *quadratic formula.* Write down the quadratic formula here:

Write a function, `solvequad`, that takes three numbers as input (a, b and c) and returns the solution(s) x that are real (*i.e.*, not complex). **Hint:** Recall that a quadratic equation may have more than one real solution—or it may have none! You will need to use `if` and `else` to handle all possible cases.

Before writing any R code, first describe your `solvequad` function without worrying about R syntax—that is, using pseudocode:

Guided by your pseudocode, write the R code for your `solvequad` function:

After creating `solvequad`, check that it does the right thing by running these tests:

```
solvequad(4, 4, 1)     # Should return -1/2.
solvequad(1, -1, -2)   # Should return 2 and -1.
solvequad(1, 1, 1)     # Should return no solutions.
```

Run a few more tests using www.wolframalpha.com.

D. melanogaster

Looping

Another way to change the flow of your program is to write a loop. A loop is simply a series of commands that are repeated a number of times. For example, you want to run the same analysis on different data sets that you collected; you want to plot the results contained in a set of files; you want to test your simulation over a number of parameter sets; etc.

`R` provides you with two ways to loop over code blocks: the `for` loop and the `while` loop. Let's start with the `for` loop, which is used to iterate over a vector or list; for each value of the vector (or list), a series of commands will be run, as shown by the following example:

```
v <- 1:10
for (i in v) {
  a <- i ^ 2
  print(a)
}
```

In the code above, the variable `i` takes the value of each element of `v` in sequence. Inside the block within the `for` loop, you can use the variable `i` to perform operations.

The anatomy of the `for` statement:

```
for (variable in list_or_vector) {
  execute these commands
} # Automatically moves to the next value.
```

You should use a for loop when you know that you want to perform the analysis over a given set of values (e.g., files of a directory, rows in your data frames, sequences of a fasta file, *etc*).

The `while` loop is used when the commands need to be repeated while a certain condition is true, as shown by the following example:

```
i <- 1
while (i <= 10) {
  a <- i ^ 2
  i <- i + 1
  print(a)
}
```

The script gives exactly the same result as the `for` loop above. A key difference is that you need to include a step to update the value of `i`, (using `i <- i + 1`), whereas in the `for` loop it is done for you automatically. The anatomy of the `while` statement:

```
while (condition is met) {
  execute these commands
} # Beware of infinite loops... remember to update the condition!
```

You can break a loop using `break`. For example:

```
i <- 1
while (TRUE) {
  if (i > 10) {
    break
  }
  a <- i ^ 2
  i <- i + 1
  print(a)
}
```

Question: Above, we ran three different loops, we found that each of them accomplished the same thing. Is one approach better? Why?

Creating computational notebooks using R Markdown

> *Let us change our traditional attitude to the construction of programs: instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to humans what we want the computer to do.*
>
> Donald E. Knuth, *Literate Programming*, 1984

When doing experiments, it is important to develop the habit of writing down everything you do in a laboratory notebook. That way, when writing your manuscript, responding to queries or discussing progress with your advisor, you can go back to your notes to find exactly what you did, how you did it, and possibly *why* you did it. The same should be true for computational work.

`RStudio` makes it very easy to build a computational laboratory notebook. First, create a new `R Markdown` file. (Choose **File > New File > R Markdown** from the RStudio menu bar.)

An R Markdown file is simply a text file. But it is interpreted in a special way that allows the text to be transformed it into a webpage (`.html`) or PDF file. You can use special syntax to render the text in different ways. Here are a few examples of R Markdown syntax:

```
*Italic text* **Bold text**

# Very large header

## Large header

### Smaller header

Unordered and ordered lists:
```

```
+ First
+ Second
    + Second 1
    + Second 2

1. This is a
2. Numbered list
```

When rendered as a PDF, the above R Markdown looks like this:

Very large header

Large header

Smaller header

Italic text **Bold text**

Unordered and ordered lists:

- First
- Second
  - Second 1
  - Second 2

1. This is a
2. Numbered list

You can also insert inline code by enclosing it in backticks.

The most important feature of R Markdown is that you can include blocks of code, and they will be interpreted and executed by `R`. You can therefore combine effectively the code itself with the description of what you are doing.

For example, including a code chunk in your R Markdown file,

````
```{r hello-world}
cat("Hello world!")
```
````

will render a document containing both the results and the code that run to generate those results:

```r
cat("Hello world!")
# Hello world!
```

If you don't want to run the `R` code, but just display it, use `{r hello-world, eval = FALSE}`; if you want to show the output but not the code, use `{r hello-world, echo = FALSE}`.

You can include plots, tables, and even mathematical equations using LaTeX. In summary, when exploring your data, or describing the methods for your paper, give R Markdown a try!

You can find inspiration in this Boot Camp; the materials for Basic and Advanced Computing were written in R Markdown.

M. mulatta

Bonus case study: Do shorter titles lead to more citations?

To keep learning about `R`, we study the following question:

Is the length of a paper title related to the number of citations?

This is what Letchford *et al* claimed (doi:10.1098/rsos.150266). In 2015, they analyzed 140,000 papers, and they found that *shorter titles* were associated with a larger number of citations.

In the folder containing the Basic Computing 2 tutorial materials, you will find data on scientific articles published between 2004 and 2013 in three top disciplinary journals, *Nature Neuroscience*, *Nature Genetics* and *Ecology Letters*. These data are conatined in three CSV files. We are going to use these data to explore this question.

Load data

Start by reading in the data:

```
data.file <- file.path("citations", "nature_neuroscience.csv")
papers    <- read.csv(data.file, stringsAsFactors = FALSE)
```

Next, take a peek at the data. How large is it?

```
nrow(papers)
ncol(papers)
```

Let's see the first few rows:

```
head(papers)
```

The goal is to test whether papers with longer titles accrue fewer (or perhaps more?) citations than those with shorter titles. The first step is to add another column to the data containing the length of the title for each paper:

```
papers$TitleLength <- nchar(papers$Title)
```

Basic statistics

In the original paper, Letchford *et al* used rank-correlation: rank all the papers according to their title length and the number of citations. If Kendall's τ ("rank correlation") is positive, then longer titles are associated with *more* citations; if τ is negative, longer titles are associated with *fewer* citations. In R, you can compute rank correlation using `cor`:

```
k <- cor(papers$TitleLength, papers$Cited.by, method = "kendall")
```

To perform a significance test for correlation, use `cor.test`:

```
k.test <- cor.test(papers$TitleLength, papers$Cited.by, method = "kendall")
```

Does the output of `cor.test` show that the correlation between the ranks is positive or negative? Is this positive or negative correlation significant? You should find that the correlation is opposite of the one reported by Letchford *et al*—longer titles are associated with *more* citations!

Now we are going to examine the data in a different way to test whether this result is robust.

Basic plotting

To plot title length vs. number of citations, we need to learn about plotting in R. To produce a simple scatterplot using the base plotting functions, simply run:

```
plot(papers$TitleLength, papers$Cited.by)
```

The problem with this very simple plot is that it is hard to detect any trend—a few papers have many more citations than the rest, obscuring the data at the bottom of the plot. This suggests that plotting the data on the *logarithmic scale* is a better approach:

```
plot(papers$TitleLength, log10(papers$Cited.by))
```

Question: This is a better plot, but there is one problem with it. What is the problem, and what fix would you suggest? Write your code to fix the problem:

Again, it is hard to see any trend in here. Maybe we should plot the best fitting line and overlay it on top of the graph. To do so, we first need to learn about linear regressions in R.

Linear regression

R was born for statistics — the fact that it is very easy to fit a linear regression in is not surprising! To build a linear model comparing columns x and y in data frame, dat, use lm, the "Swiss army knife" for linear regression in R:

```
model <- lm(y ~ x, dat)  # y = a + b*x + error
```

Let's perform a linear regression of the number of citations (on the log-scale) vs. title length. To do so, first create a new column in the data frame containing the counts on the log-scale:

```
papers$LogCits <- log10(papers$Cited.by + 1)
```

Now you can perform a linear regression:

```
model_citations <- lm(LogCits ~ TitleLength, papers)
model_citations           # Gives best-fit line.
summary(model_citations)  # Gives more info.
```

And we can easily add this best-fit line to our plot:

```
plot(papers$TitleLength, papers$LogCits)
abline(model_citations, col = "red", lty = "dotted")
```

Once we add the best-fit line, the positive trend is more clear.

One thing to consider is that this data set spans a decade. Naturally, older papers have had more time to accrue citations. In our models, we should control for this effect. But first, we should explore whether this is an important factor to consider.

First, let's plot the distribution of the number of citations for all the papers:

```
hist(papers$LogCits)
```

You can control the number of histogram bins with the n argument:

```
hist(papers$LogCits, n = 50)
```

Alternatively, estimate the density using density, then plot it:

```
plot(density(papers$LogCits))
```

Next, compare the distributions for papers published in 2004, 2009 and 2013:

```
plot(density(papers$LogCits[papers$Year == 2004]), col = "black")
lines(density(papers$LogCits[papers$Year == 2009]), col = "blue")
lines(density(papers$LogCits[papers$Year == 2013]), col = "red")
```

More recent papers should have fewer citations. Does your plot support this hypothesis? You can account for this in your regression model by incorporating the year of publication into the linear regression:

```
model_citations_better <- lm(LogCits ~ TitleLength + Year, papers)
summary(model_citations_better)
```

Does the regression coefficient (slope) for year confirm that older papers have more citations?

Our new analysis is better than before, but might be even better to have a separate "baseline" for each year. This can be done by converting the "Year" column to a factor:

```
papers$Year            <- factor(papers$Year)
model_citations_better <- lm(LogCits ~ Year + TitleLength, papers)
summary(model_citations_better)
```

This model has a different baseline for each year, and then title length influences this baseline. In this new model, are longer titles still associated with more citations?

Computing *p*-values using randomization

Kendall's τ takes as input two rankings, x and y, both of the same length, n. It calculates the number of "concordant pairs" (if $x_i > x_j$, then $y_i > y_j$) and the number of "discordant pairs". The final value is

$$\tau = \frac{n_{\text{concordant}} - n_{\text{discordant}}}{\frac{n(n-1)}{2}}$$

If x and y are completely independent, we would expect τ to have a distribution centered at zero. The variance of the "null" distribution of τ depends on the data. It is typically approximated as a normal distribution. If you want to have a stronger result that does not rely on a normality assumption, you can use randomizations to calculate a p-value. Simply, compute τ for the actual data, as well as for many "fake" datasets obtained by randomizing the data. Your p-value is then the proportion of τ values for the randomized sets that exceed the τ value for the actual data.

Here, we will try to implement this randomization to calculate a p-value for papers published in 2006, and then we will compare against the p-value obtained from running `cor.test`. To do this, we will use a for-loop for the randomization.

First, subset the data:

```
dat <- papers[papers$Year == 2006, ]
```

Compute τ from these data:

```
k <- cor(dat$TitleLength, dat$Cited.by, method = "kendall")
```

Now calculate τ in "fake" data sets by randomly scrambling the citation counts. Begin by doing this for one fake data set:

```
shuffled_citation_counts <- sample(dat$Cited.by)
k.fake <- cor(dat$TitleLength, shuffled_citation_counts, method = "kendall")
```

Is the value of τ closer to zero in this "shuffled" data set?

To get an accurate p-value, we should compute τ for a large number of shuffled data sets. Let's try 1,000 of them. This and similar randomization techniques are known as "bootstrapping".

```
nr     <- 1000         # Number of fake data sets.
k.fake <- rep(0, nr)   # Storage all the "fake" taus.
```

Since this computation involves lots of repetition, a for-loop makes a lot of sense here:

```
for (i in 1:nr){
  shuffled_citation_counts <- sample(dat$Cited.by)
  k.fake[i] <- cor(dat$TitleLength, shuffled_citation_counts,
                   method = "kendall")
}
```

After running this loop, you should have 1,000 correlations calculated from 1,000 fake data sets. You have just generated a "null" distribution for τ. What does this null distribution look like? Try plotting it:

```
hist(k.fake, n = 50)
```

What proportion of the fake data sets have a correlation that exceeds the correlation in the actual data? This is the p-value.

```
pvalue <- mean(k.fake >= k)
```

How does your new p-value compare to the p-value computed by `cor.test`? Is it smaller or larger?

```
cor.test(dat$TitleLength, dat$Cited.by, method = "kendall")
```

Question: Did you get the same result as the instructor, or your neighbours? If not, why? How could you ensure that your result is more similar, or the same?

Whenever possible, use randomizations rather than relying on classical tests. They are more difficult to implement, and more computationally expensive, but they allow you to avoid making assumptions about your data.

Repeating the analysis for each year

Up until this point, we have only analyzed the citation data for 2006. Does the result we obtained for 2006 hold up in other years? Let's explore this question—we will use a for-loop to repeat the analysis for 2004 to 2013. Let's be smart about designing our code and use a *function* to decompose the problem into parts. The code for the final analysis will look like this:

```
years <- 2004:2013
for (i in years){
  dat <- papers[papers$Year == i, ]
  out <- analyze_citations(dat)
  cat("year:", i, "tau:", out$k, "pvalue:", out$pvalue, "\n")
}
```

The missing piece is the code implementing function `analyze_citations`. You can re-use your code above to write this function.

```
analyze_citations <- function (dat) {
  nr     <- 1000
  k      <- cor(dat$TitleLength, dat$Cited.by, method = "kendall")
  k.fake <- rep(0, nr)
  for (i in 1:nr) {
    k.fake[i] <- cor(dat$TitleLength, sample(dat$Cited.by), method = "kendall")
  }
  return(list(k = k, pvalue = mean(k.fake >= k)))
}
```

Activity: Organizing and running your code

Now we would like to be able to automate the analysis, such that we can repeat it for each journal. This is a good place to pause and introduce how to go about writing programs that are well-organized, easy to write, easy to debug, and easier to reuse.

1. Take the problem, and divide it into smaller tasks (these are the functions).
2. Write the code for each task (function) separately, and make sure it does what it is supposed to do.
3. Document the code so that you can later understand what you did, how you did it, and why.
4. Combine the functions into a master program.

For example, let's say we want to write a program that takes as input the name of files containing citation data. The program should first fit a linear regression model,

```
log(citations + 1) ~ as.factor(Year) + TitleLength
```

then output the coefficient associated with `TitleLength`, and its p-value.

We could split the program into the following tasks:

1. A function to load and prepare the data for a linear regression analysis.
2. A function to run the linear regression analysis.
3. A master code that puts it all together.

Let's begin with the master code—the bulk of the code is a for-loop that repeats the regression analysis for each journal:

```
files <- list.files("citations", full.names = TRUE)
for (i in files) {
  cat("Analyzing data from",i,"\n")
  papers <- load_citation_data(i)
  out    <- fit_citation_model(papers)
  cat("coefficient:", out$estimate, "p-value:", out$pvalue, "\n")
}
```

This code doesn't work yet because you haven't written the functions that are called inside the loop. (What error message to you get when you try to run the code?)

The `load_citation_data` function reads in the data from the CSV file, then prepares the data for the linear regression analysis:

```
load_citation_data <- function (filename) {
  dat <- read.csv(filename, stringsAsFactors = FALSE)
  dat$TitleLength <- nchar(dat$Title)
  dat$LogCits     <- log10(dat$Cited.by + 1)
  dat$Year        <- as.factor(dat$Year)
  return(dat)
}
```

Before continuing, check that it works by running it on one of the CSV files:

```
papers <- load_citation_data("citations/nature_neuroscience.csv")
```

The `fit_citation_model` function fits a linear regression model to the input data, then extracts the quantities from the regression analysis we are most interested in (the "best-fit" slope and the *p*-value corresponding to "TitleLength").

```
fit_citation_model <- function (papers) {
  model <- lm(LogCits ~ Year + TitleLength, papers)
  terms <- summary(model)$coefficients
  return(list(estimate = terms["TitleLength", "Estimate"],
              pvalue   = terms["TitleLength", "Pr(>|t|)"]))
}
```

Check that this function runs, and does what it is supposed to do:

```
out <- fit_citation_model(papers)
```

Now that you have defined the necessary functions, try running the master code above.

Question: Suppose you download a fourth CSV file containing data on papers from the *American Journal of Human Genetics*. Would any changes need to be made to your R code above to run it on the four citation data sets?