4

Getting Started with File Management

Careful file management is crucial for reproducible research. Remember two of the guidelines from Chapter 2:

- Explicitly tie your files together.
- Have a plan to organize, store, and make your files available.

Apart from the times when you have an email exchange (or even meet in person) with someone interested in reproducing your research, the main information independent researchers have about the procedures is what they access in files you make available: data files, analysis files, and presentation files. If these files are well organized and the way they are tied together is clear, replication will be much easier. File management is also important for you as a researcher, because if your files are well organized you will be able to more easily make changes, benefit from work you have already done, and collaborate with others.

Using tools such as R, *knitr*/*rmarkdown*, and markup languages like LaTeX requires fairly detailed knowledge of where files are stored in your computer. Handling files to enable reproducibility may require you to use command-line tools to access and organize your files. R and Unix-like shell programs allow you to control files–creating, deleting, relocating–in powerful and really reproducible ways. By typing these commands you are documenting every step you take. This is a major advantage over graphical user interface-type systems where you organize files by clicking and dragging them with the cursor. However, text commands require you to know your files' specific addresses–their file paths.

In this chapter we discuss how a reproducible research project may be organized and cover the basics of file path naming conventions in Unix-like operating systems, such as Mac OS X and Linux, as well as Windows. We then learn how to organize them with RStudio Projects. Finally, we'll cover some basic R and Unix-like shell commands for manipulating files as well as how to navigate through files in RStudio in the *Files* pane. The skills you will learn in this chapter will be heavily used in the next chapter (Chapter 5) and throughout the book.

In this chapter we work with locally stored files, i.e. files stored on your computer. In the next chapter we will discuss various ways to store and access files remotely stored in the cloud.

4.1 File Paths & Naming Conventions

All of the operating systems covered in this book organize files in hierarchical directories, also known as file trees. To a large extent, directories can be thought of as the folders you usually see on your Windows or Mac desktop.[1] They are called hierarchical because directories are located inside of other directories, as in Figure 4.1.

4.1.1 Root directories

A root directory is the first level in a disk, such as a hard drive. It is the root out of which the file tree 'grows'. All other directories are subdirectories of the root directory.

On Windows computers you can have multiple root directories, one for each storage device or partition of a storage device. The root directory is given a drive letter assignment. If you use Windows regularly you will most likely be familiar with `C:\` used to denote the C partition of the hard drive. This is a root directory. On Unix-like systems, including Macs and Linux computers, the root directory is simply denoted by a forward slash (/) with nothing before it.

4.1.2 Subdirectories & parent directories

You will probably not store all of your files in the root directory. This would get very messy. Instead you will likely store your files in subdirectories of the root directory. Inside of these subdirectories may be further subdirectories and so on. Directories inside of other directories are also referred to as child directories of a parent directory.

On Windows computers separate subdirectories are indicated with a back slash (\). For example, if we have a folder called *Data* inside of a folder called *ExampleProject* which is located in the C root directory it has the address `C:\ExampleProject\Data`.[2] When you type Windows file paths into R you need to use two backslashes rather than one: e.g. `C:\\ExampleProject\\Data`. This is because the \ is an escape character in R.[3] Escape characters tell R to interpret the next character or sequence of characters differently. For example, in Section 5.1 you'll see how `\t` can be interpreted by R as a tab rather than the letter "t". Add another escape character to neutralize the escape character so that R interprets it as a backslash. In other words, use an escape character to escape the escape character. Another option for writing Windows file names in R is to use one forward slash (/).

On Unix-like systems, including Mac computers, directories are indicated

[1] To simplify things, I use the terms 'directory' and 'folder' interchangeably in this book.

[2] For more information on Windows file path names see this helpful website: `http://msdn.microsoft.com/en-us/library/windows/desktop/aa365247(v=vs.85).aspx`

[3] As we will see in Part IV, it is also a LaTeX and Markdown escape character.

with a forward slash (/). The file path of the *Data* file on a Unix-like system would be: `/ExampleProject/Data`. Remember that a forward slash with nothing before it indicates the root directory. So `/ExampleProject/Data` has a different meaning than `ExampleProject/Data`. In the former, *ExampleProject* is a subdirectory of the root. In the latter, *ExampleProject* is a subdirectory of the current working directory (see below for details about working directories). This is also true in Windows.

In this chapter I switch between the two file system naming conventions to expose you to both. For the remainder of the book I use Unix-like file paths. When you use relative paths, these will work across operating systems in R. We'll get to relative paths in a moment.

4.1.3 Working directories

When you use R, markup languages, and many of the other tools covered in this book, it is important to keep in mind what your current working directory is. The working directory is the directory where the program automatically looks for files and other directories, unless you tell it to look elsewhere. It is also where it will save files. Later in this chapter we will cover commands for finding and changing the working directory.

4.1.4 Absolute vs. relative paths

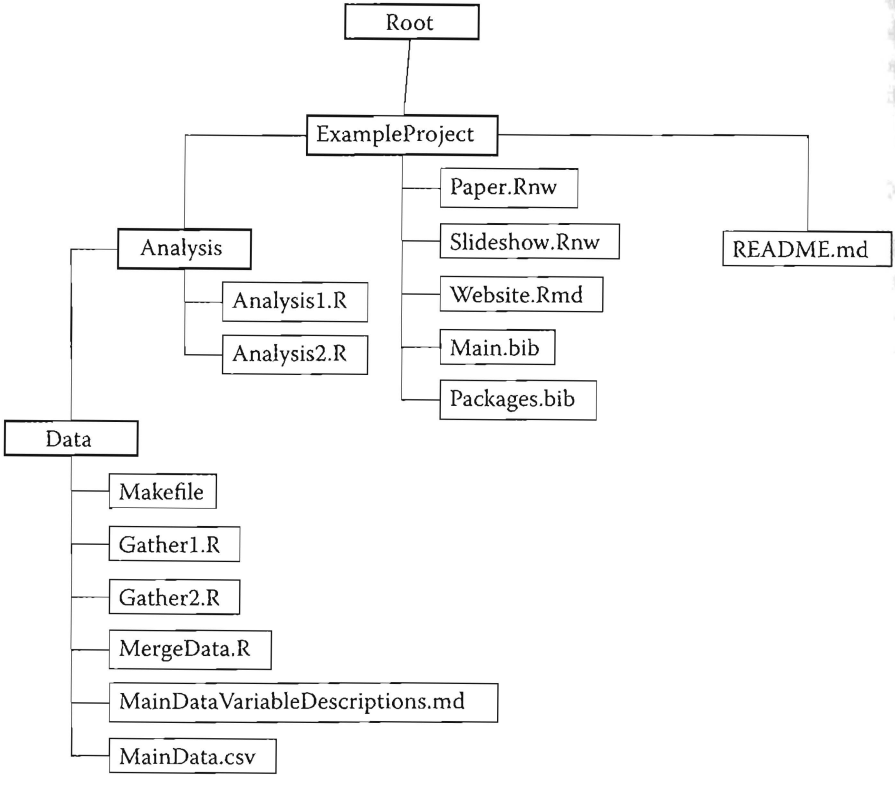
For reproducible research, collaborative research, and even if you ever change the computer you work on, it is a good idea to use relative rather than absolute file paths. Absolute file paths give the entire path of a given file or directory on a specific system. For example, `/ExampleProject/Data` is an absolute path as it specifies the path of the *Data* child directory all the way back to the root directory. However, if our current working directory is *ExampleProject* and we want to link to the *Data* child directory or a file in it, we don't need the absolute path. We could simply use `Data/`, i.e. the path relative to the working directory.

It is good practice to use relative paths when possible and organize your files such that using relative paths is easy. This makes your code less dependent on the particular file structure of a particular computer. For example, imagine you use `C:\\ExampleProject\\Data` in your source code to link to the *Data* directory. Someone–a collaborator, a researcher reproducing your work, or even you–then tries to run the code on a different computer. The code will break if they are, for instance, using a Unix-like system or have placed *ExampleProject* in a different partition of their hard drive. This can be fixed relatively straightforwardly by changing the file path in the source. However, this is tedious (often not well documented) and unnecessary if you use relative file paths.

4.1.5 Spaces in directory & file names

It is generally good practice to avoid putting spaces in your file and directory names. For example, I called the example project parent directory "ExampleProject" rather than "Example Project". Spaces in file and directory names can sometimes create problems for computer programs trying to read the file path. The program may believe that the space indicates that the path name has ended. To make multi-word names easily readable without using spaces, adopt a convention such as CamelBack. In CamelBack new words are indicated with capital letters, while all other letters are lower case. For example, "ExampleProject".

FIGURE 4.1
Example Research Project File Tree



4.2 Organizing Your Research Project

Figure 4.1 gives an example of how the files in a simple reproducible research project could be organized. The project's parent directory is called *ExampleProject*. Inside this directory are the primary knittable documents (*Paper.Rnw Slideshow.Rnw*, and *Website.Rmd*). In addition there is an *Analysis* sub-directory with the R files to run the statistical analyses followed by a further *Data* child directory.

The nested file structure allows you to use relative file paths. The knittable documents can call *Analysis1.R* with the relative path *Analysis/Analysis1.R*, which in turn could call a file in the *Data/* subdirectory. If all of the directories were at the same level of the file tree then you would need to use absolute file paths.

In addition to the main files and subdirectories in *ExampleProject* you will probably notice a file called *README.md*. The *README.md* file gives an overview of all the files in the project. It should briefly describe the project including things like its title, author(s), topic, any copyright information, and so on. It should also indicate how the folders in the project are organized and give instructions for how to reproduce the project. The README file should be in the main project folder–in our example this is called *ExampleProject*–so that it is easy to find. If you are storing your project as a GitHub repository (see Chapter 5) and the file is called *README*, its contents will automatically be displayed on the repository's main page. If the *README* file is written using Markdown (e.g. *README.md*), it will also be properly formatted. Figure 5.2 shows an example of this.

FIGURE 4.2
An Example RStudio Project Menu

It is good practice to dynamically include the system information for the R session you used to create the project. To do this you can write your README file with R Markdown. Simply include the `sessionInfo()` command in a *knitr* code chunk in the R Markdown document. If you knit this file immediately after knitting your presentation document, it will record the information for that session.

You can also dynamically include session info in a LaTeX document. To do this, use the `toLatex` command in a code chunk. The code chunk should have the option `results='asis'`. The code is:

```
toLatex(sessionInfo())
```

4.3 Setting Directories as RStudio Projects

If you are using RStudio, you may want to organize your files as Projects. You can turn a normal directory into an RStudio Project by clicking on `File` in the RStudio menu bar and selecting `New Project...`. A new window will pop-up. Select the option `Existing Directory`. Find the directory you want to turn into an RStudio Project by clicking on the `Browse` button. Finally, select `Create Project`. You will also notice in the Create Project pop-up window that you can build new project directories and create a project from a directory already under version control (we'll do this at the end of Chapter 5). When you create a new project you will see that RStudio has put a file with the extension `.Rproj` into the directory.

Making your research project directories RStudio Projects is useful for a number of reasons:

- The project is listed in RStudio's Project menu where it can be opened easily (see Figure 4.2).
- When you open the project in RStudio it automatically sets the working directory to the project's directory and loads the workspace, history, and source code files you were last working on.
- You can set project specific options like whether PDF presentation documents should be compiled with *Sweave* or *knitr*.
- When you close the project your R workspace and history are saved in the project directory if you want.
- It helps you version control your files.
- You can build your Project–run the files in a specific way–with makefiles.
- Gives you an easy-to-use interface for managing the R packages that your project depends on.

We will look at many of these points in more detail in the next few chapters.

4.4 R File Manipulation Commands

R has a range of commands for handling and navigating through files. Including these commands in your source code files allows you to more easily replicate your actions.

`getwd`

To find your current working directory use the `getwd` command:

```
getwd()
```

```
## [1] "/git_repositories/Rep-Res-Book/Source/Children/Chapter4"
```

The example here shows you the current working directory that was used while knitting this chapter.

`list.files`

Use the `list.files` command to see all of the files and subdirectories in the current working directory. You can list the files in other directories too by adding the directory path as an argument to the command.

```
list.files()
```

```
## [1] "chapter4.Rnw" "images4"
```

You can see that the *Chapter4* folder has the file *chapter4.Rnw* (the markup file used to create this chapter) and a child directory called *images4* where I stored the original versions of the figures included in this chapter.

`setwd`

The `setwd` command sets the current working directory. For example, if we are on a Mac or other Unix-like computer we can set the working directory to the *GatherSource* directory in our Example Project (see Figure 4.1) like this:

```
setwd("/ExampleProject/Data/GatherSource")
```

Now R will automatically look in the *GatherSource* folder for files and will save new files into this folder, unless we explicitly tell it to do otherwise.

When working with a knittable document, setting the working directory once in a code chunk changes the working directory for all subsequent code chunks.

`root.dir`

By default the root (or working) directory for all of the code chunks in a knittable document is the directory where this document is located. You can reset the directory by feeding a new file path to the `root.dir` option. We can set this globally[4] for all of the chunks in the document by including the following code in the document's first chunk.

```
opts_knit$set(root.dir = '/ExampleProject/Analysis')
```

Here we set the */ExampleProject/Analysis* sub-directory as the root directory for all of the chunks in our presentation document.

Note: In general it is preferable to use a nested file structure, as we saw before, rather than specify `root.dir`. A nested file structure creates one less step for those trying to reproduce your work on a different computer. They do not need to change the `root.dir` file path.

`dir.create`

Sometimes you may want to create a new directory. You can use the `dir.create` command to do this.[5] For example, to create a *ExampleProject* file in the root *C* directory on a Windows computer type:

```
dir.create("C:\\ExampleProject")
```

`file.create`

Similarly, you can create a new blank file with the `file.create` command. To add a blank R source code file called *SourceCode.R* to the *ExampleProject* directory on the *C* drive use:

```
file.create("C:\\ExampleProject\\SourceCode.R")
```

[4] See the discussion of global chunk options in Chapter 3, page 53.

[5] Note: you will need the correct system permissions to be able to do this.

`cat`

If you want to create a new file and put text into it use the `cat` (concatenate and print) command. For example, to create a new file in the current working directory called *ExampleEcho.md* that includes the text "Reproducible Research with R and RStudio" type:

```
cat("Reproducible Research with R and RStudio",
    file = "ExampleCat.md")
```

In this example we created a Markdown formatted file by using the `.md` file extension. We could Of course, change the file extension to `.R` to set it as an R source code file, `.Rnw` to create a *knitr* LaTeX file, and so on.

You can use `cat` to print the contents of one or more objects to a file. **Warning:** The `cat` command will overwrite existing files with the new contents. To add the text to existing files use the `append = TRUE` argument.

```
cat("More Text", file = "ExampleCat.md",
    append = TRUE)
```

`unlink`

You can use the `unlink` command to delete files and directories.

```
unlink("C:\\ExampleProject\\SourceCode.R")
```

Warning: the `unlink` command permanently deletes files, so be very careful using this command.

`file.rename`

You can use the `file.rename` to, obviously, rename a file. It can also be used to move a file from one directory to another. For example, imagine that we want to move the *ExampleCat.md* file from the directory *ExampleProject* to one called *MarkdownFiles* that we already created.[6]

[6]The `file.rename` command won't create new directories. To move a file to a new directory you will need to create the directory first with `dir.create`.

```
file.rename(from = "C:\\ExampleProject\\ExampleCat.md",
            to = "C:\\MarkdownFiles\\ExampleCat.md")
```

`file.copy`

The `file.rename` fully moves a file from one directory to another. To copy the file to another directory use the `file.copy` command. It has the same syntax as `file.rename`:

```
file.copy(from = "C:\\ExampleProject\\ExampleCat.md",
            to = "C:\\MarkdownFiles\\ExampleCat.md")
```

4.5 Unix-like Shell Commands for File Management

Though this book is mostly focused on using R for reproducible research it can be useful to use a Unix-like shell program to manipulate files in large projects. Unix-like shell programs including Bash on Mac and Linux and Windows PowerShell allow you to type commands to interact with your computer's operating system.[7] We will especially return to shell commands in the next chapter when we discuss Git version control and makefiles for collecting data in Chapter 6, as well as the command-line program[8] Pandoc (Chapter 12 and 13). We don't have enough space to fully introduce shell programs or even all of the commands for manipulating files. We are just going to cover some of the basic and most useful commands for file management. For good introductions for Unix and Mac OS 10 computers see William E. Shotts Jr.'s (2012) book on the Linux command-line. For Windows users, Microsoft maintains a tutorial on Windows PowerShell at `http://technet.microsoft.com/en-us/library/hh848793`. The commands discussed in this chapter should work in both Unix-like shells and Windows PowerShell.

It's important at this point to highlight a key difference between R and Unix-like shell syntax. In shell commands you don't need to put parentheses

[7] You can access Bash via the Terminal program on Mac OS 10 and Linux computers. It is the default shell on Mac and Linux, so it loads automatically when you open the Terminal. Windows PowerShell comes installed with Windows.

[8] A command-line program is just a program you run from a shell.

around your arguments. For example, if I want to change my working directory to my Mac Desktop in a shell using the `cd` command I simply type:[9]

```
cd /Users/Me/Desktop
```

In this example `Me` is my user name.

`cd`

As we just saw, to change the working directory in the shell just use the `cd` (change directory) command. Here is an example of changing the directory in Windows PowerShell:

```
cd C:/Users/Me/Desktop
```

If you are in a child directory and want to change the working directory to the previous working directory you were in, simply type:

```
cd -
```

If, for example, our current working directory is */User/Me/Desktop* and we typed `cd` followed by a minus sign (`cd -`) then the working directory would change to */User/Me*. Note this will not work in PowerShell.

`pwd`

To find your current working directory, use the `pwd` command (present working directory). This is essentially the same as R's `getwd` command.

```
pwd

## /Users/Me/Desktop
```

[9]Many shell code examples in other sources include the shell prompt, like the `$` in Bash or `>` in PowerShell. These are like R's `>` prompt. I don't include the prompt in code examples in this book because you don't type them.

`ls`

The `ls` (list) command works very similarly to R's `list.files` command. It shows you what is in the current working directory.

```
ls
```

```
## chapter4.Rnw images4
```

As we saw earlier, R also has an `ls` command. R's `ls` command lists items in the R workspace. The shell's `ls` command lists files and directories in the working directory.

`mkdir`

Use `mkdir` to create a new directory. For example, if I wanted to create a directory in my Linux root directory called *NewDirectory* I would type:

```
mkdir /NewDirectory
```

If running this code on Mac or Linux gives you an error message like this:

```
mkdir: /NewDirectory: Permission denied
```

you simply need to use the `sudo` command to run the command with higher privileges.

```
sudo mkdir /NewDirectory
```

Running this code will prompt you to enter your administrator password.

`echo`

There are a number of ways to create new files in Unix-like shells. One of the simplest ways is with the `echo` command. This command simply prints its arguments. For example:

```
echo Reproducible Research with R and RStudio
```

```
## Reproducible Research with R and RStudio
```

If you add the greater-than symbol (>) after the text you want to print and then a file name, `echo` will create the file (if it doesn't already exist) in the current working directory and then print the text into the file.

```
echo Reproducible Research with R and RStudio > ExampleEcho.md
```

Using only one greater-than sign will completely erase the *ExampleEcho.md* file's contents and replace them with `Reproducible Research with R and RStudio`. To add the text at the end of an existing file, use two greater-than signs (>>).

```
echo More text. >> ExampleEcho.md
```

There is also a `cat` shell command. It works slightly differently than the R version of the command and I don't cover it here.

`rm`

The `rm` command is similar to R's `unlink` command. It removes (deletes) files or directories. Again, be careful when using this command, because it permanently deletes the files or directories.

```
rm ExampleEcho.md
```

As we saw in Chapter 3, R also has an `rm` command. It is different because it removes objects from your R workspace rather than files from your working directory.

`mv`

To move a file from one directory to another with the shell, use the `mv` (move) command. For example, to move the file *ExampleEcho.md* from *ExamplePro-*

jects to *MarkdownFiles* use the following code and imagine both directories are in the root directory:[10]

```
mv /ExampleProject/ExampleEcho.md /MarkdownFiles
```

Note that the *MarkdownFiles* directory must already exist, otherwise it will simply rename the file. So this command is similar to the R command `file.rename`.

`cp`

The `mv` command completely moves a file from one directory to another. To copy a version of the file to a new directory use the `cp` command. The syntax is similar to `mv`:

```
cp /ExampleProject/ExampleEcho.md /MarkdownFiles
```

`system` *(R command)*

You can run shell commands from within R using R's `system` command. For example, to run the `echo` command from within R type:

```
system("echo Text to Add > ExampleEcho.md")
```

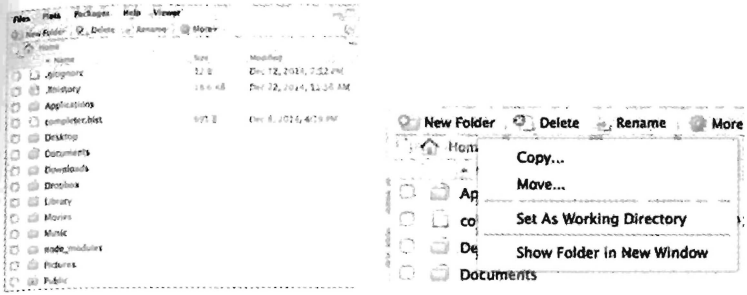
4.6 File Navigation in RStudio

The RStudio *Files* pane allows us to navigate our file tree and do some basic file manipulations. The left panel of Figure 4.3 shows us what this pane looks like. The pane allows us to navigate to specific files and folders and delete and rename files. To select a folder as the working directory tick the dialog box next to the file then click the `More` button and select `Set As Working Directory`. Under the `More` button () you will also find options to `Move` and `Copy` files (see the right pane of Figure 4.3).

The *Files* pane is a GUI, so our actions in the *Files* pane are not as easily reproducible as the commands we learned earlier in this chapter.

[10] If they were not in the root directory we would not place a forward slash at the beginning.

FIGURE 4.3
The RStudio Files Pane



Chapter summary

In this chapter we've learned how to organize our research files to enable dynamic replication. This included not only how they can be ordered in a computer's file system, but also the file path naming conventions the addresses– that computers use to locate files. Once we know how these addresses work we can use R and shell commands to refer to and manipulate our files. This skill is particularly useful because it allows us to place code in text-based files to manipulate our project files in highly reproducible ways. In the next few chapters we will begin to put these skills in practice when we learn how to store our files and create data files in reproducible ways.