8. Using Textual and Binary Formats for Storing Data

Watch a video of this chapter[1]

There are a variety of ways that data can be stored, including structured text files like CSV or tab-delimited, or more complex binary formats. However, there is an intermediate format that is textual, but not as simple as something like CSV. The format is native to R and is somewhat readable because of its textual nature.

One can create a more descriptive representation of an R object by using the `dput()` or `dump()` functions. The `dump()` and `dput()` functions are useful because the resulting textual format is editable, and in the case of corruption, potentially recoverable. Unlike writing out a table or CSV file, `dump()` and `dput()` preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again. For example, we can preserve the class of each column of a table or the levels of a factor variable.

Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files. In addition, textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem because one can just open the file in an editor and look at it (although this would probably only be done in a worst case scenario!). Finally, textual formats adhere to the Unix philosophy[2], if that means anything to you.

There are a few downsides to using these intermediate textual formats. The format is not very space-efficient, because all of the metadata is specified. Also, it is really only partially readable. In some instances it might be preferable to have data stored in a CSV file and then have a separate code file that specifies the metadata.

8.1 Using `dput()` and `dump()`

One way to pass data around is by deparsing the R object with `dput()` and reading it back in (parsing it) using `dget()`.

[1] https://youtu.be/5mIPigbNDfk
[2] http://www.catb.org/esr/writings/taoup/

```
> ## Create a data frame
> y <- data.frame(a = 1, b = "a")
> ## Print 'dput' output to console
> dput(y)
structure(list(a = 1, b = structure(1L, .Label = "a", class = "factor")), .Names = c("a",
"b"), row.names = c(NA, -1L), class = "data.frame")
```

Notice that the `dput()` output is in the form of R code and that it preserves metadata like the class of the object, the row names, and the column names.

The output of `dput()` can also be saved directly to a file.

```
> ## Send 'dput' output to a file
> dput(y, file = "y.R")
> ## Read in 'dput' output from a file
> new.y <- dget("y.R")
> new.y
  a b
1 1 a
```

Multiple objects can be deparsed at once using the dump function and read back in using `source`.

```
> x <- "foo"
> y <- data.frame(a = 1L, b = "a")
```

We can `dump()` R objects to a file by passing a character vector of their names.

```
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
```

The inverse of `dump()` is `source()`.

```
> source("data.R")
> str(y)
'data.frame':        1 obs. of  2 variables:
 $ a: int 1
 $ b: Factor w/ 1 level "a": 1
> x
[1] "foo"
```

8.2 Binary Formats

The complement to the textual format is the binary format, which is sometimes necessary to use for efficiency purposes, or because there's just no useful way to represent data in a textual manner. Also, with numeric data, one can often lose precision when converting to and from a textual format, so it's better to stick with a binary format.

The key functions for converting R objects into a binary format are `save()`, `save.image()`, and `serialize()`. Individual R objects can be saved to a file using the `save()` function.

```
> a <- data.frame(x = rnorm(100), y = runif(100))
> b <- c(3, 4.4, 1 / 3)
>
> ## Save 'a' and 'b' to a file
> save(a, b, file = "mydata.rda")
>
> ## Load 'a' and 'b' into your workspace
> load("mydata.rda")
```

If you have a lot of objects that you want to save to a file, you can save all objects in your workspace using the `save.image()` function.

```
> ## Save everything to a file
> save.image(file = "mydata.RData")
>
> ## load all objects in this file
> load("mydata.RData")
```

Notice that I've used the `.rda` extension when using `save()` and the `.RData` extension when using `save.image()`. This is just my personal preference; you can use whatever file extension you want. The `save()` and `save.image()` functions do not care. However, `.rda` and `.RData` are fairly common extensions and you may want to use them because they are recognized by other software.

The `serialize()` function is used to convert individual R objects into a binary format that can be communicated across an arbitrary connection. This may get sent to a file, but it could get sent over a network or other connection.

When you call `serialize()` on an R object, the output will be a raw vector coded in hexadecimal format.

```
> x <- list(1, 2, 3)
> serialize(x, NULL)
 [1] 58 0a 00 00 00 02 00 03 03 02 00 02 03 00 00 00 00 13 00 00 00 03 00
[24] 00 00 0e 00 00 00 01 3f f0 00 00 00 00 00 00 00 00 00 0e 00 00 00 01
[47] 40 00 00 00 00 00 00 00 00 00 00 0e 00 00 00 01 40 08 00 00 00 00 00
[70] 00
```

If you want, this can be sent to a file, but in that case you are better off using something like `save()`.

The benefit of the `serialize()` function is that it is the only way to perfectly represent an R object in an exportable format, without losing precision or any metadata. If that is what you need, then `serialize()` is the function for you.