Documentation and R Markdown

January 21, 2019

Practices for style & code documentation

In a previous section, we briefly discussed some conventions for variable naming. Other aspects of style are also worth a mention. Spacing and indentation is one area. In particular, blank lines should be chosen so that expressions are grouped into logical categories. Consistent indentation should be used as well. The contents of a function, conditional, or loop should be indented one level deeper than the enclosing block. There is much debate as to how far to indent and whether one should use spaces or tabs (I use 4 spaces myself, and I map the tab key in my editor to 4 spaces). The truth is it doesn't really matter all that much as long as you are consistent and follow any guidelines established by your team.

For code comments, the key is to not under-comment or over-comment. Firstly, make your code as self-documenting as possible. This is done by choosing sensible variable names, keeping code logically organized, and favoring simple, idiomatic code over fancy tricks or non-idiomatic styles. Self-documenting code isn't always possible. It is these cases where you should provide comments.

Writing idiomatic R code primarily involves writing functions. These can be put into packages, which have their own form of formal documentation. For non-packaged functions, it is a good idea to put some comments at the beginning of the function covering preconditions and postconditions. Preconditions specify the data types and other assumptions about the parameters of the functions, as well as any other expectations about the calling or external environment (e.g., the existence of a file). Postconditions specify what the function does, namely what its return value is and what its side effects, if any, are.

R Markdown

R Markdown is a framework for integrating R code, R output, and commentary. It comes from the idea of "literate programming" where we describe what a program does, inserting code snippets as needed, and the resulting document serves as the program's source code. R Markdown can be used for creating reports for third parties who are merely interested in results, or it can be used to better explain your program to other programmers (or your future self).

The tool chain for R Markdown allows for you to produce output files in a variety of formats including pdf, Word, and html.

Here is an example of a simple markdown file, which are usually stored in simple text files with the .Rmd extension:

````
---
title: "Example document"
date: 2019-01-22
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(warning = FALSE)
knitr::opts_chunk$set(message = FALSE)
knitr::opts_chunk$set(cache = TRUE)
```

# This is a header
````

````
This following is an R code chunk.

```{r imports}
library(magrittr)
```

## This is a smaller header

*I'm in italics!*

**I'm in bold!**

### Even smaller

Another R code chunk. This depends on the "imports" chunk.

```{r pipe_example, dependson = "imports", echo = FALSE}

1 %>% `+`(1) # A very strange way to calculate 1 + 1

```

Another R code chunk follows. This one is unnamed.

```{r eval = FALSE}
1 + 1 # A better way to calculate 1 + 1
```

  * Unordered list - item 1
    + sub-item
  * Item 2

And now an ordered list...

  1. Item 1
  2. Item 2

Here is inline R code: `x <- 5`. It won't run, but it will be displayed in a
fixed format font appropriate for code. [^1]

[^1]: Here is my footnote. It is a very nice footnote. Too bad no one reads footnotes.
````

There are three components here.

- An optional YAML header for specifying the default output formats, the name of the document, etc.
- **Chunks** of R code.
- Formatted text.

You can compile R Markdown using the `rmarkdown::render` function. However, an even easier way is to load the markdown source into RStudio and hit the knitr button.

R code chunks can be named or unnamed, and a variety of options can be provided. Some import ones include `echo` which determines if the code appears in the output document, `eval` which determines if the code is run, and `warning` which determines if warnings are suppressed in the output. These options can be set individually for each chunk, but you can also set global defaults via `knitr::opts_chunk$set`.

Caching

The programs embedded in an R Markdown document may take a long time to run. Because of this, you don't want to rerun everything every time a small change is made to your Markdown document. In this case, you would want to cache R code chunks.

You can use `knitr::opts_chunk$set` to turn on caching as seen in our example. Caching can be turned off in individual R code blocks (`cache = FALSE`). If caching is used, R chunks will not run when the R Markdown file is recompiled unless a change is made to the chunk or any of its dependencies. Dependencies must be manually specified using the `dependson` option.

The R Markdown tool chain does not check to see if changes are substantive or not, so adding a single blank line is enough to force R Markdown to ignore the cache and rerun the block. This is a useful trick for when external dependencies not tracked by R Markdown caching have changed.