Using BloomR as a Report Generating Tool

What is all this stuff about?

You will be able to generate reports from BloomR in PDF or HTML.
For doing so you will write a file in special format in which:

- The narrative will be formatted with a very very simple markup language, named *Markdown* format (basically the one used when you write a Wikipedia article).
- In the middle of the narrative you can insert special code blocks, denoted as *code chunks*.

When you convert this file to PDF or HTML format the narrative will be pretty formatted, with bold, italic and all the bells and whistles; in place of the code chunks you will find their outputs, that is the tables, the plots or whatever the code generates.

If you want, you might *echo* the code, that is you can let display the R code too in the PDF/HTML report, so your reader will learn about the calculation you implemented to produce the report output.

Since these files include R code chunks inside a markdown narrative, they are named *Rmd* files (short for R-markdown).

In the next sections you will learn how easy is the markdown syntax and the commands needed to convert Rmd files to HTML or PDF files.

Setup BloomR Core or BloomR Lab for creating HTML reports

The best, if you want to make reports, is to use BloomR Lab or BloomR Studio, because you can leverage an Rmd report editor, plus in BloomR Studio you can create PDF reports (and there is zero setup to do). In *BloomR Core or Lab you can only create HTML reports.*
To try this feature in BloomR Core or Lab, start by loading `knitr` and `markdown` libraries, with the following commands:

```
library(knitr)
library(markdown)
```

As for every `library` command, they are per session. They stay loaded in memory until BloomR session is running. If you close and reopen BloomR, to use them you will need to load them again.

Now you are ready to create ...

Your First R Markdown File

We start by creating an *Rmd* file containing only the narrative and no R code.

With the usual File menu create a new file. For example, create a new file named `my-first-report.Rmd` in the directory `mybloomr`. Remember to add the `Rmd` extension when you name the file.

In BloomR Core you can do this with R commands too, like follows.

Change to `mybloomr` directory identified with the tilde:

```
setwd('~')
```

Create the empty file named `my-first-report.Rmd`. Note the `Rmd` extension.

```
file.create('my-first-report.Rmd')
dir() # Check the file was actually created!
```

To start editing the file. Use:

```
file.edit('my-first-report.Rmd')
```

The editor window pops up: copy and paste in it the following material:

Warning: For your convenience, inside `mybloomr/examples`, there is already a sample `my-first-report.Rmd`, pay attention not to overwrite it!

Type the following Wikipedia-style content (markdown) in `my-first-report.Rmd`:

```
This is a header for the section
==============================

**Bold**, *Italic*, _Italic again_

These lines will be
printed as a single line.

To insert a line break leave two spaces after me.
Now I will be printed on a new line!

The following is the typewriter style. Use it if you want to comment/show code templates.

    # This is the main project function
    myfunction(arg1, arg2, arg3)

Leave four spaces before lines and (at least) one blank line above and below the block.
To use the typewriter style *inline*, enclose the text inside backticks,
therefore this `text` will be printed in typewriter style.
If you use dashes, `-`, instead of equals, `=`, you will get a subsection header.

Using Itemisations and Enumerations
---------------------------------

*  Item 1
*  Item 2
*  ...

1.  Item 1
2.  Item 2
3.  ...
```

After the conversion (that we are going to show ahead), your text will be formatted like this:

This is a header for the section

Bold, *Italic*, *Italic again*

These lines will be printed as a single line.

To insert a line break leave two spaces after me.
Now I will be printed on a new line!

The following is the typewriter style. Use it if you want to comment/show code templates.

```
# This is the main project function
myfunction(arg1, arg2, arg3)
```

Leave four spaces before lines and (at least) one blank line above and below the block. To use the typewriter style *inline*, enclose the text inside backticks, therefore this `text` will be printed in typewriter style.
If you use dashes, `-`, instead of equals, `=`, you will get a subsection header.

Using Itemisations and Enumerations

- Item 1

- Item 2
- ...

1. Item 1
2. Item 2
3. ...

Carefully compare the initial markdown input with the formatted output and understand how it works and why.

This should be enough to create pretty documents. Later on you may want to learn more about formatting the narrative here: daringfireball.net.

NB: It is possible (and very common) to use an alternative syntax for section headers, that is:

```
# Section Header
## Subsection Header
```

Anyway I am not emphasizing it, since it can be confused with R syntax for comments.

When you are down with your editing, **save** the file. How?
Well, you might hit `Ctrl-S` on your keyboard, use the save button on BloomR toolbar, or use the menu `File->Save`. Anyway make sure that the editor is selected and not the console, otherwise you will save the command history!

Convert Rmd files to HTML files

Before adding *code chunks* to the narrative, let us see how to operate the conversion for the end-user document (perhaps your employer or your customer).

First of all some before-flight-checks: have you loaded the `knitr` and `markdown` libs? This is easily forgotten, therefore causing errors.
Also, can you see your Rmd-file in the current directory? If so, you avoid to provide the full path to it. To check this type in the console:

```
dir()
```

Now it is time to create your first HTML report. First, when you are finished with editing, remember to *save the Rmd* file.
In BloomR Core and BloomR Lab, use the following two commands to generate the report:

```
library(knitr)
library(markdown)
knit("my-first-report.Rmd") # ->will give: my-first-report.md
markdownToHTML("my-first-report.md", "my-first-report.html")
```

For your convenience, I have retyped the `library` commands, but you need to type them only once and they libraries will stay in memory until you close BloomR.

The first function *knits* the Rmd file, that is transforms the R markdown in a standard markdown (like the one used for Wikipedia articles). To be honest, for such a simple file it is pretty useless. The second function is self-explanatory: open with your browser `my-first-report.html` and see the report.

Creating reports in BloomR Studio edition

While you could use the commands above, there is a better and suggested way in BloomR Studio edition, just type:

```
br.rmd2html("my-first-report.Rmd")
```

There is nothing to setup.

R Code Chunks

We can now add R code chunks to the narrative. Code chunks have this general template:

````
```{{{R}} someLabel, option=value}
Put R code lines here
```
````

Note the three backticks and don't leave any space before them!
`someLabel` can be whatever, but it should be unique (you cannot reuse it in other chunks). Depending on what you want to obtain, there can be many options for `option = value` or even none.

Normally there is a first chunk setting global options for all document chunks, similar to the following:

````
```{{{R}} setup}
# set global chunk options: images will be 7x5 inches
opts_chunk$set(fig.width=7, fig.height=5)
```
````

These options will apply to all chunks unless explicitly overridden.

Your first code chunk can be as follows:

````
```{{{R}} myFirstChunk}
x = 1+1 # a simple calculator
set.seed(123)
rnorm(5)  # boring random numbers
```
````

When you convert your Rmd file to PDF/HTML, you will see, near the location of the chunk, *both* the code (pretty formatted) and the resulting output, i.e. five normal random numbers. As noted, you can change the label "myfirstchunk" to whatever you want, but the name must be unique.

What if you want only the results of the code (without the code itself)?
You should disable the *echo*.

````
```{{{R}}  myChunkNoecho, echo=FALSE}
set.seed(123)
rnorm(6)  # boring random numbers
```
````

Now, given the option `echo=FALSE`, after the conversion you will see only the random numbers, so without the code producing them.

Is everything clear? If so, you might rewrite `my-first-report.Rmd` with some R code.
In your editor replace the text with the following new one (as usual copy & paste to avoid typo):

```
My First Report
===============

Good morning, first of all I will set the default figure dimensions:

```{{{R}} setup}
# set global chunk options: images will be 7x5 inches
opts_chunk$set(fig.width=7, fig.height=5)
```

I will implement some calculations. See the code and results below:

```{{{R}} myFirstChunk}
x <- 1+1 # a simple calculator
set.seed(123)
rnorm(5)  # boring random numbers
```

I will now generate six random numbers.
Well, you already know the code behind, so I don't show it:

```{{{R}} myChunkNoecho, echo=FALSE}
set.seed(321)
rnorm(6)  # boring random numbers
```

And now let me plot my ideas:

```{{{R}} myNicePlot, fig.cap="We compare Miles per Gallon with HP."}
par(mar = c(4, 4, 2, .1))
with(mtcars, {
  plot(mpg~hp, pch=20, col='darkgray', main="My First Plot")
  lines(lowess(hp, mpg))
})
```
```

The conversion can be done with the same commands shown above and will produce the following formatted output:

My First Report

Good morning, first of all I will set the default figure dimensions:

```
# set global chunk options: images will be 7x5 inches
opts_chunk$set(fig.width=7, fig.height=5)
```

I will implement some calculations. See the code and results below:

```
x <- 1+1 # a simple calculator
set.seed(123)
rnorm(5)  # boring random numbers
```

```
## [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
```

I will now generate six random numbers. Well, you already know the code behind, so I don't show it:

```
## [1]  1.7049032 -0.7120386 -0.2779849 -0.1196490 -0.1239606  0.2681838
```

And now let me plot my ideas:

```
par(mar = c(4, 4, 2, .1))
with(mtcars, {
  plot(mpg~hp, pch=20, col='darkgray', main="My First Plot")
  lines(lowess(hp, mpg))
})
```

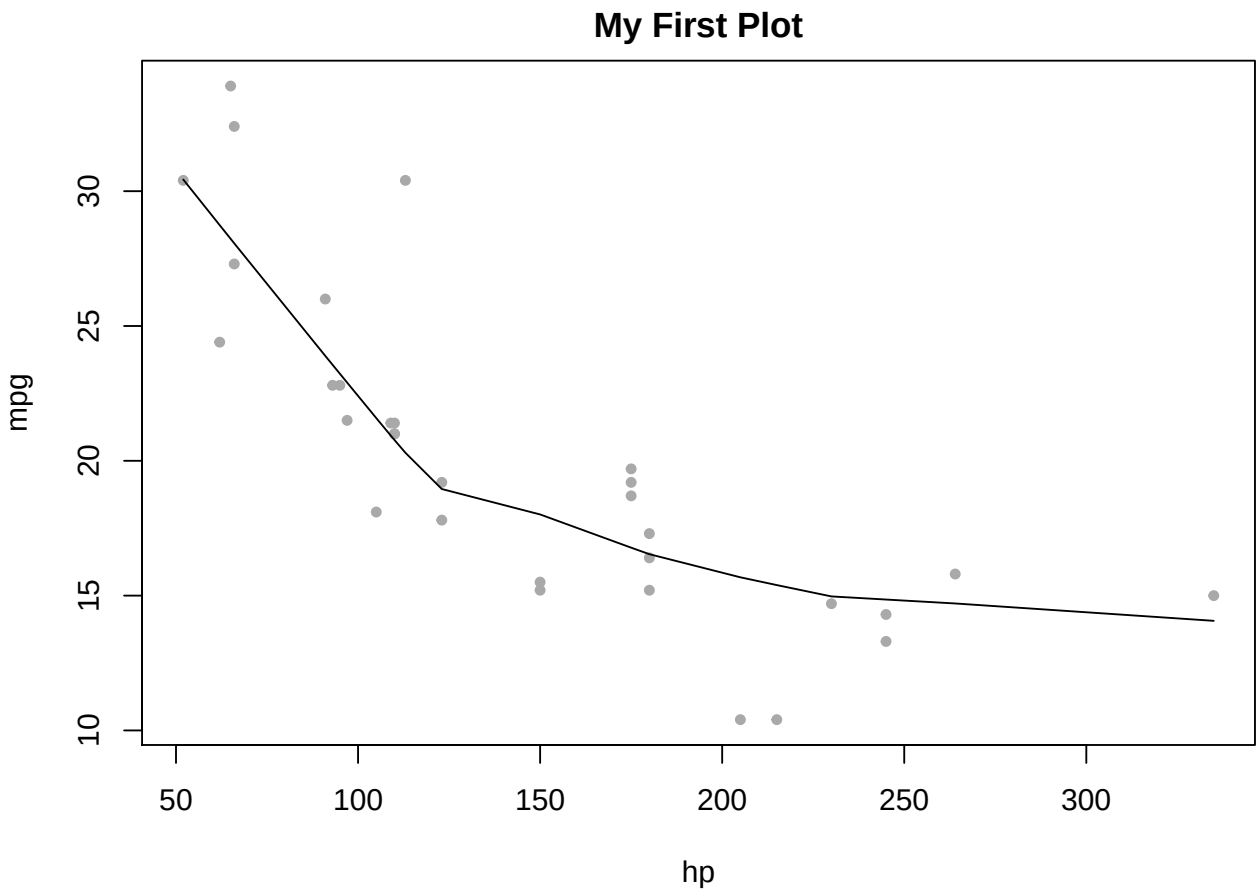

Figure 1: In this plot we compare Miles per Gallon with HP.

As you can see code chunks are now followed by their output.

When a code chunk contains plotting commands, here the command `plot(mpg~hp, pch=20, col='darkgray')`, then the report will also show the generated figure. There are some differences concerning figure positioning.
In HTML-format output the figure will be placed *immediately after* the code chunk generating it (so here soon after `myNicePlot` chunk).
That would be different for PDF output. Because PDF's want to resemble the physical paper, figures are automatically placed in the document in such a way to optimise the text flow. In fact, sometimes there is not enough space on the current page to insert the picture, which is therefore placed on another page (perhaps the next one). So, if in this very moment you are reading me from a PDF output, don't be surprised to find the plot placed in what is only apparently a weird place.

A final notice about figure generating chunks. The code for `myNicePlot` uses the not so mysterious `fig.cap` option. Yes, you guessed it: it is the text for the plot caption. It is normally not used in HTML format, so I will talk about this later.

Maths

You can use LaTeX style maths in your report.

The following markdown:

```
$$f(\alpha, \beta) \propto x^{\alpha-1}(1-x)^{\beta-1}$$
```

will be rendered as:

$$f(\alpha, \beta) \propto x^{\alpha-1}(1-x)^{\beta-1}$$

Note that `_` and `^` set indices and exponents. LaTeX commands/symbols are preceeded by a backslash, e.g.: `\frac`, `\sum`, and arguments (if any) are encolesed in curly braces. To use braces literally escape them with : `{` and `\}`. A single index or exponent does not require enclosing braces.

Now consider the following more involved equation:

$$f(\alpha) = \frac{1}{c+1}\sum_{k=0}^{\infty} x_k^{\alpha-1}, \mathrm{ where } c \in \mathbf{n} = \{1 \ldots \bar{n}\}$$

.

You can generate it with commands:

```
$$f(\alpha) = \frac{1}{c+1}\sum_{k=0}^\infty  x^{\alpha-1}_k,
              \mathrm{ where } c \in \mathbf{n} = \{1 \ldots \bar{n} \}$$
```

For inline expressions use a single dollar. So `$x_k^j$` gives: x_k^j.

Try to exercise on some online engine like codecogs.com.

Convert the rich Rmd document to an HTML report

There are no special commands here. Just the same used for the simple Rmd without code. However, in BloomR Core and BloomR Lab, when you check for the file produced:

```
dir() # a strange "figure" directory will appear
```

together with `my-first-report.html`, you see also a `figure` directory listed. Here you find the report's figures, in case you need to use them separately. You can delete this directory.

Sync Rmd directory in BloomR Lab or Studio and reduce potential errors

As noted, before generating the report you need to be sure that the current working directory is set to the one containing the Rmd file `my-first-report.Rmd`. To do this use `setwd` command, as noted above, and to be sure check the content of the current working directory with `dir()`.
There is an easier way in BloomR Lab and BloomR Studio. Here you will notice a button like an "R" with an enter symbol, the **Set directory to this file**. Move your cursor on a place in the Rmd file where there is R code (not the narrative), when you do so the button toolbar modifies and, when you hover the appropriate button, you will read from the mouse pointer tooltip *Set directory to this file*. If you click this button. You will see a message on the R console informing about the changing of current directory.
One way to avoid setting the working directory is specifying the full path of the file, such as `D:\\some-dir\\my-first-report.Rmd`. Note that you need to use `\\` (double backslashes) in the paths.

Let us step ahead to for a real Rmd file with code chunks.

Generate a PDF report in BloomR Studio

To tranform our Rmd file into a PDF, type in the console:

```
br.rmd2pdf("my-first-report.Rmd")
```

As usual remember to sync R working directory before running the function.
Note that you will get an error if you are not using the Studio edition.

As we have observed above, in LaTeX (PDF) the plot is automatically placed in such a way to optimise the text flow. Figures and tables are therefore named by those who speak properly *floats*.

One thing to note is that, *the order by which floats are displayed in the report reflects that of the code chunks generating them*. Besides, below them, there is an automatic generated sequential identifier, e.g.: "Figure 1". You will use this number to address the floats inside the report body text.

As hinted above, you can do even more. Set the `fig.cap` property in your plot generating chunks as follows:

````
```{{{R}} myNicePlot, fig.cap="In this plot we compare Miles per Gallon with HP."}
    ...
```
````

Then in the PDF output (but not in HTML output), after the plot number, you get also the given caption, which adds more insights and makes the report more eye catching.

if you want both a PDF and HTML report, instead of running two command you can use:

```
br.rmd2both("my-first-report.Rmd")
```

Presenting matrix-like objects as tables

To print matrix-like objects as tables, you use the command `kable` and the chunk option `results="asis"`.

First of all, normally you don't get table captions using Pandoc. But there is a trick to get them: add a chunk on top of your `my-first-report.Rmd` with the usual *chunk-syntax* containing the following:

```
opts_knit$set(rmarkdown.pandoc.to = "latex")
```

Add a chunk for creating a generic matrix to print:

```
i=4; j=6
(M=matrix(rnorm(i*j), i, dimnames= list(letters[1:i], LETTERS[1:j])))
```

```
##            A         B          C          D           E          F
## a  0.7268415 0.3477014 -1.1534395  0.5775845  0.98833540 -2.3309312
## b  0.2331354 1.4845918 -0.8046717  0.4463561 -1.07223880  0.4175160
## c  0.3391139 0.1883255  0.4560691  0.9172555 -0.75801528 -1.1203274
## d -0.5519147 2.4432598  0.4203326 -0.1070615  0.09500072 -0.4746847
```

To print `M` with four decimals, we use this chunk:

```
kable(M, digits=4, caption="This is a test matrix")
```

Table 1: This is a test matrix

	A	B	C	D	E	F
a	0.7268	0.3477	-1.1534	0.5776	0.9883	-2.3309
b	0.2331	1.4846	-0.8047	0.4464	-1.0722	0.4175
c	0.3391	0.1883	0.4561	0.9173	-0.7580	-1.1203

	A	B	C	D	E	F
d	-0.5519	2.4433	0.4203	-0.1071	0.0950	-0.4747

One can show a tabular regression output like follows:

```
x=rnorm(100)
y=x^2
kable(summary(lm(y ~ x))$coefficients)
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	0.8405290	0.1127115	7.457347	0.0000000
x	-0.1923836	0.1228085	-1.566534	0.1204472

I am not a literate girl

If one emphasises the code side of this document vs. the English description, she speaks of *literate programming*; but there are situations in which you want to *purl* the document, that is stripping the narrative and leave only the code, in other words we want to convert the Rmd file containing words+code in a standard R script where only the code survives. That's as simple as that:

```
purl("my-first-report.Rmd")
```

Conclusion

That's it. Now you can fully appreciate the difference between a spreadsheet, which is undisclosed and unrepeatable, and *reproducible research*.
For more info go to knitr homepage.