How to build a package for Bioconductor

You Zhou

January 6, 2021

Contents

Introduction

This R markdown file includes the notes and steps for building a package in *Bioconductor* standard. *Bioconductor* provides tools for the analysis and comprehension of high-throughput genomic data. *Bioconductor* uses the R statistical programming language, and is open source and open development. It has two releases each year, and an active user community.

Some other useful resources:
Bioconductor guild
How to make a package 1
How to make a package 2
A good coding example
youtube video for how to build a packge

The workflows

1 Idea

For building a package the first and most important thing is have a good idea, what do you want to achieve with your package? Which function you would like to include in this package.

2 Name of the package

Then we need to define a name for the package. The library availble can be used to check the availability of the name and also give a suggest for the name.

```
library(available) # Check if the Title of a Package is Available,
# Appropriate and Interesting
# Check for potential names
available::suggest("Easily extract information about your sample")
available::suggest("Plot iCLIP data")
# Check whether it's available (you can also name it just as you want)
available::available("whyitworks", browse = FALSE)
```

2.1 Version Format

All Bioconductor packages should have a version number in `x.y.z` format. Examples of good version numbers:

```
1.2.3
0.99.5
2.3.0
3.12.44
```

New packages submitted to Bioconductor should set Version: `0.99.0` in the DESCRIPTION file. Specifying y=99 triggers a bump in x at the next release which in this case results in version 1.0.0.

3 Using the 'Devel' Version of Bioconductor

Package authors should develop against the *version of R* that will be available to users when the Bioconductor devel branch becomes the Bioconductor release branch.

R has a '.y' release in x.y.z every year (typically mid-April), but Bioconductor has a .y release (where current devel becomes release) every 6 months (mid-April and mid-October).

This means that, from mid-October through mid-April, Bioconductor developers should be developing against R-devel. From mid-April to mid-October, developers should use R-release (actually, the R snapshot from the R-x-y-branch) for Bioconductor development.

Sometimes we may need to upgrade your *R version* to match the the *Devel*. By default Bioconductor requires the *newest version* of R for the package development.

```
if (!requireNamespace("BiocManager", quietly=TRUE))
    install.packages("BiocManager")
BiocManager::install(version = "3.12") # BiocManager::install(version = "devel")
BiocManager::valid()                   # Checks for out of date packages
```

4 Open a R package project and link it to the GitHub

After install the correct version of *devel*, we should open a new R package project in our RStudio. This new project show link to the GitHbub.
Version control is particularly important for software development. This is because we will want to keep track of every change, so in case we accidentally break something we can go back in time and fix our errors.

5 Some useful packages

The following packages can help for making a Bioconductor package:

```
library(usethis)       # For the package set up
library(roxygen2)      # Organizing the files in the package, e.g. DESCRIPTION
library(devtools)      # The tools for developing package
library(goodpractice) # To correct your coding style, and get the advices
library(BiocCheck)     # To do the last step-checking
```

6 Making the functions

After all these preparation steps, now we can code our functions. The common requirement for the coding of Bioconductor is showed down below:

- ***Use 4 spaces for indenting. No tabs***
- ***No lines longer than 80 characters*** For these two requirements, we can use the `Global Option` in the RStudio, to help.

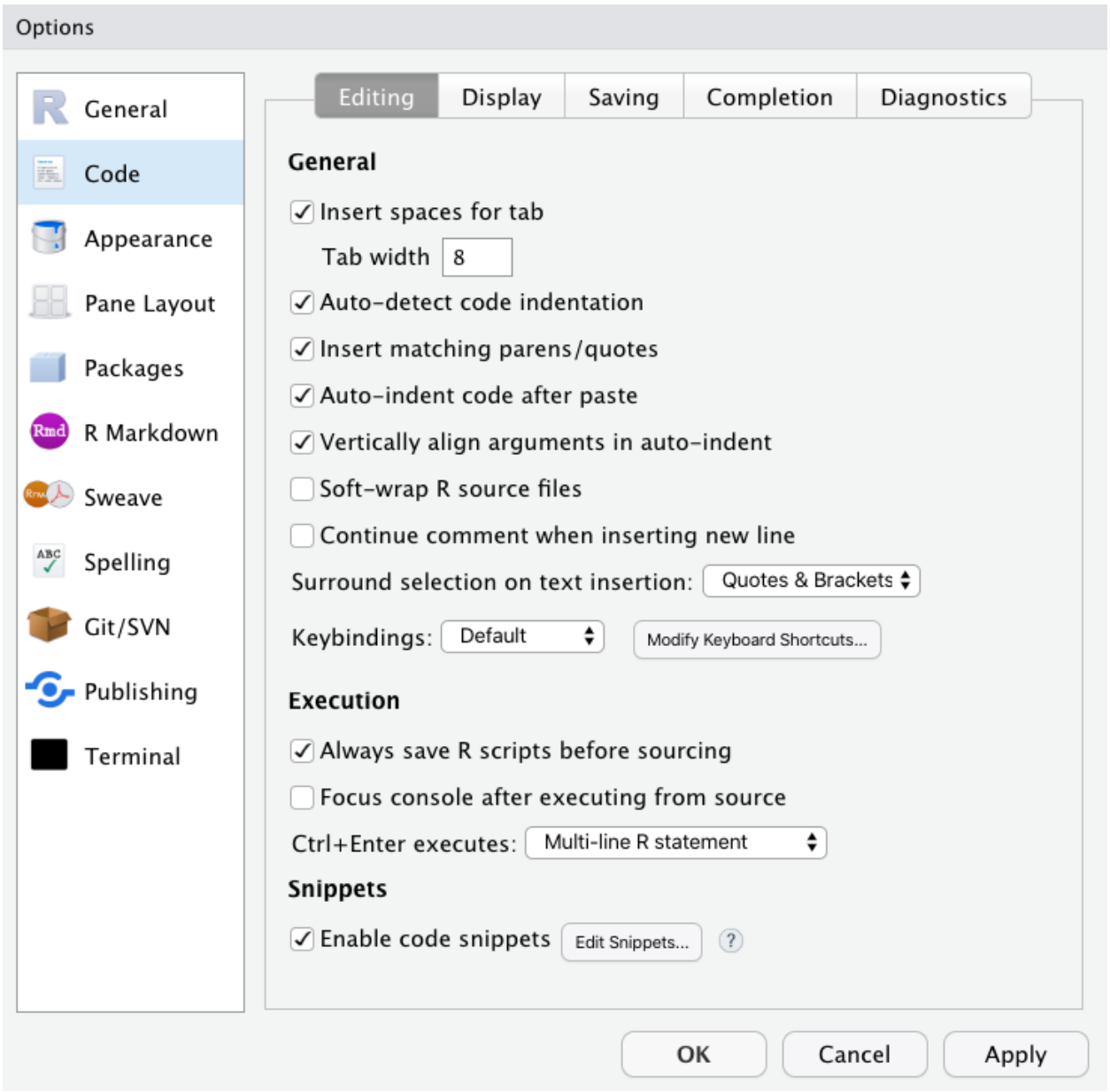

Figure 1: **Setting 1**

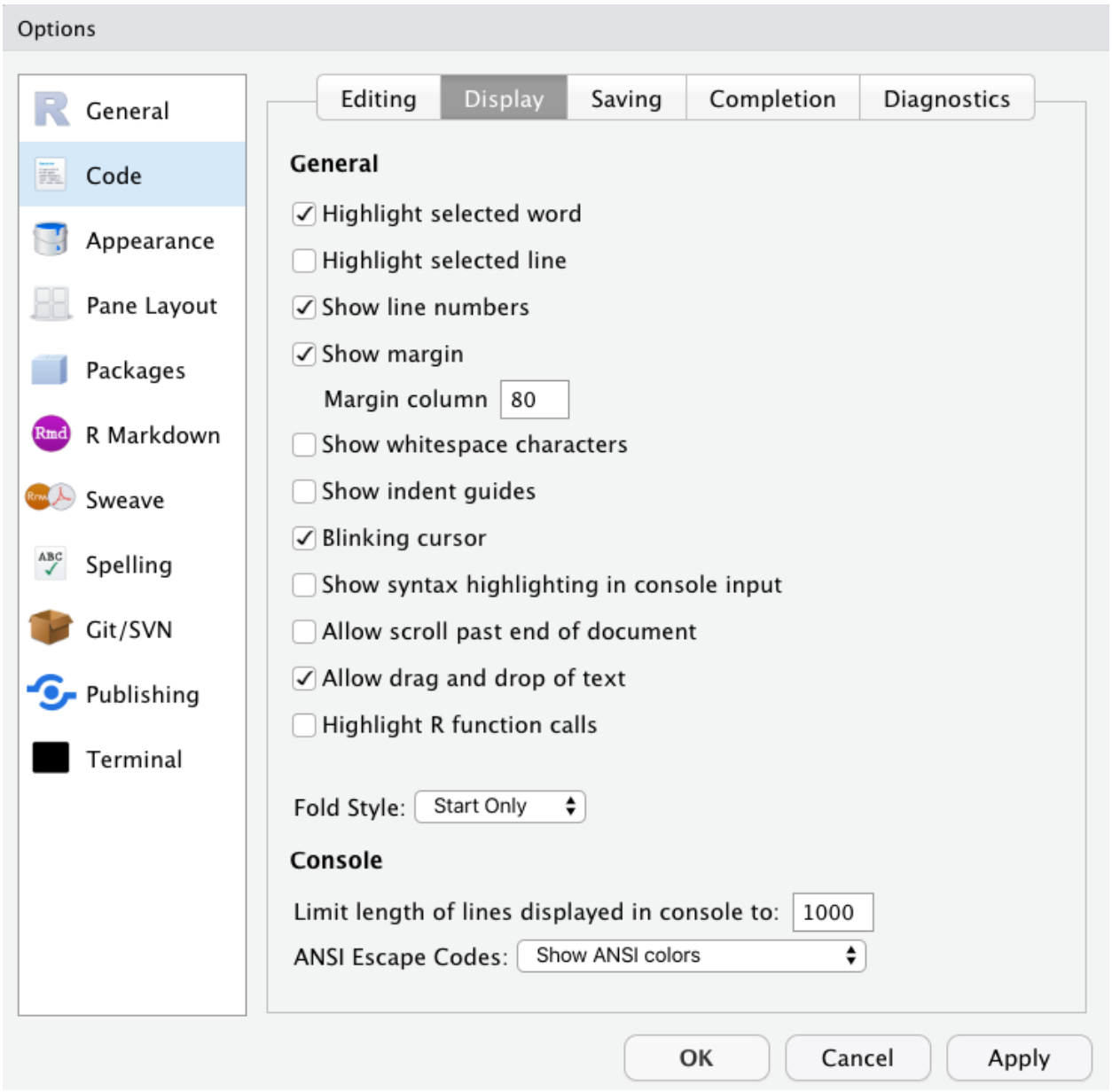

Figure 2: **Setting 2**

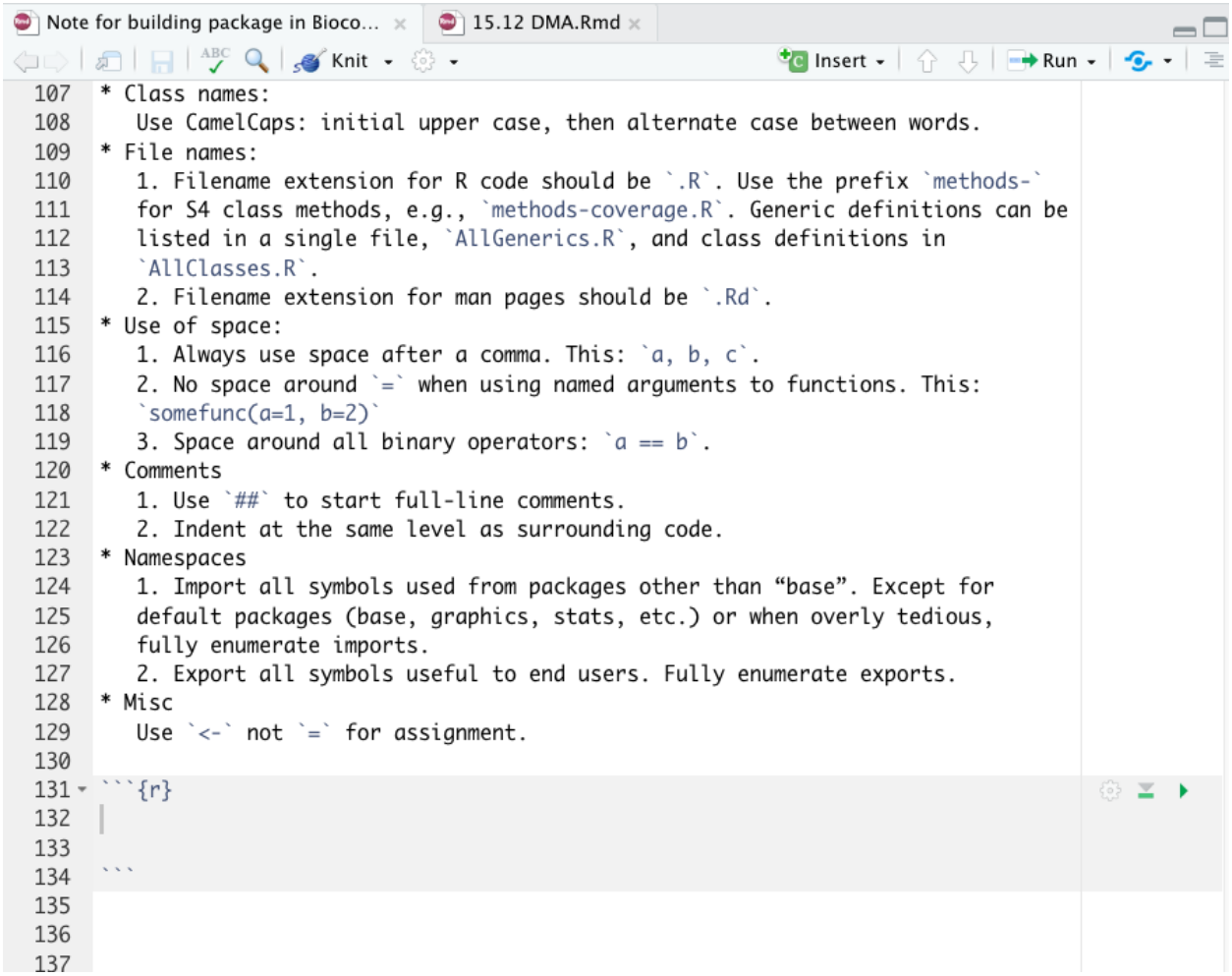

Figure 3: **Then we can see the line which indicates the 80 characters in the RStudio**

- ***Variable names***:
  Use camelCaps: initial lowercase, then alternate case between words.
- ***Function names***:
  1. Use camelCaps: initial lower case, then alternate case between words.
  2. Do not use `.` (in the S3 class system, some(x) where x is class A will dispatch to some.A).
  3. Prefix *non-exported functions* with a `..`
- ***Class names***:
  Use CamelCaps: initial upper case, then alternate case between words.
- ***File names***:
  1. Filename extension for R code should be `.R`. Use the prefix `methods-` for S4 class methods, e.g., `methods-coverage.R`. Generic definitions can be listed in a single file, `AllGenerics.R`, and class definitions in `AllClasses.R`.
  2. Filename extension for man pages should be `.Rd`.
- ***Use of space***:
  1. Always use space after a comma. This: `a, b, c`.
  2. No space around `=` when using named arguments to functions. This: `somefunc(a=1, b=2)`
  3. Space around all binary operators: `a == b`.
- ***Comments***
  1. Use `##` to start full-line comments.
  2. Indent at the same level as surrounding code.
- ***Namespaces***
  1. Import all symbols used from packages other than "base". Except for default packages (base, graphics, stats, etc.) or when overly tedious, fully enumerate imports.
  2. Export all symbols useful to end users. Fully enumerate exports.
- ***Misc*** Use `<-` not `=` for assignment.

Here is an example to help to understand these requirements.

```
## Non-exported function
.calculation <- function(x){
    x <- x + 1
}

## Exported function
testFunction <- function(x, y){
    if(x >= 2) {
        x <- .calculation(x)
        if (y > 0) {
            y <- .calculation(y)
        }
        if (y <= 0) {
            y <- y
        }
    }
    else{
        x <- x
    }
    output <- x + y
    return(output)
}
```

6.1 Coding advices

Bioconductor prefer to have robust and efficient code. Here are some coding advices:

- ***Vectorize***: use `lapply()`, `applly` instead of any kind of loop.

```
## Do something like this
y <- sapply(x, sqrt)
## Don't do it like this way
for (i in seq_along(x)) y[i] <- sqrt(x[i])
```

- ***Avoid 1:n style iterations*** Write `seq_len(n)` or `seq_along(x)` rather than `1:n` or `1:length(x)`. This protects against the case when n or length(x) is 0 (which often occurs as an unexpected 'edge case' in real code) or negative.
- ***Re-use existing functionality and classes*** Bioconductor strongly recommend reusing existing methods for importing data, and reusing established classes for representing data. Here are some suggestions for importing different file types and commonly used Bioconductor classes.
  - **Importing**

    1. GTF, GFF, BED, BigWig, etc., – `rtracklayer::import()`
    2. VCF – `VariantAnnotation::readVcf()`
    3. SAM / BAM – `Rsamtools::scanBam()`, `GenomicAlignments::readGAlignment*()`
    4. FASTA – `Biostrings::readDNAStringSet()`
    5. FASTQ – `ShortRead::readFastq()`
    6. MS data (XML-based and mgf formats) –`MSnbase::readMSData()`, `MSnbase::readMgfData()`

  - **Common Classes**

    1. Rectangular feature x sample data – `SummarizedExperiment::SummarizedExperiment()` (RNAseq count matrix, microarray, ...)
    2. Genomic coordinates – `GenomicRanges::GRanges()` (1-based, closed interval)
    3. DNA / RNA / AA sequences – `Biostrings::StringSet()`
    4. Gene sets – `BiocSet::BiocSet()`, `GSEABase::GeneSet()`, `GSEABase::GeneSetCollection()`
    5. Multi-omics data – `MultiAssayExperiment::MultiAssayExperiment()`
    6. Single cell data – `SingleCellExperiment::SingleCellExperiment()`
    7. Mass spec data – `MSnbase::MSnExp()`
- ***Give the functiona an evocative name that makes code easier to understand***
- ***Give your arguments evocative names that make them easier to understand***
- ***Use consistent coding style***
- ***Break your functions into smaller functions***
- ***Comment your code***

6.2 An example for function

Here I write a small function according to all the advices up there.

```
## Break your function into smaller functions
badExample <- function(x,y){
    if (x >= 0) {
        out <- (min(x,y)+1)/2
    }
    if (x < 0) {
        x <- x+10
        out <- (min(x,y)+1)/2
    }
    for (i in 1:length(x)) { ## should use seq_len()
```

```
        out <- out + i
    }
    out
}

.hidencalculation <- function(x,y){
    output <- (min(x,y)+1)/2
    output
}
goodExample <- function(x,y){
    if (x >= 0) {
        out <- .hidencalculation(x,y)  ## Here is the calculation step
    }
    if (x < 0) {
        x <- x+10
        out <- .hidencalculation(x,y)
    }
    for (i in seq_len(x)) {
        out <- out + i
    }
    out
}
```

7 biocViews

After all these steps, we need to run and generate the bicViews file for the Bioconductor. xx

Packages added to the *Bioconductor Project* require a **biocViews**: field in their DESCRIPTION file. The field name "biocViews" is case-sensitive and must begin with a lower-case 'b'.

biocViews terms are *keywords* used to describe a given package. They are broadly divided into three categories, representing the type of packages present in the Bioconductor Project -

a) Software
b) Annotation Data
c) Experiment Data

biocViews are available for the release and devel branches of Bioconductor. The devel branch has a check box under the tree structure which, when checked, displays biocViews that are defined but not used by any package, in addition to biocViews that are in use.

8 Checking

The ***BiocCheck*** can help us to do the Bioconductor specific checking.