Developing R Packages

Simon Andrews
simon.andrews@babraham.ac.uk
@simon_andrews

v2021-01-13
Writing R Functions

```r
bmi <- function(weight, height) {
  height/100 -> height
  height^2 -> height
  return(weight/height)
}

> bmi(90,175)
[1] 29.38776

> bmi(c(90,102), c(175,183))
[1] 29.38776 30.45776
```
Making functions robust

• More important when your function will be run by people other than you!

• Two main changes:
  1. Make sure your function isn’t affected by the environment in the script in which it’s run
  2. Validate the input data and provide helpful error messages if anything is wrong
Encapsulation

• A function should only refer to data passed in as an argument.

• It shouldn’t read or modify data in the users environment

• It should only send data back via the return statement and let the user decide how to use it
The wrong way to do it...

```r
read_csv("Child_Variants.csv") -> variants

how_many_genes <- function() {
  variants %>%
    distinct(GENE) %>%
    nrow() -> gene_count

  print(paste0("There are ",gene_count," genes"))
}

how_many_genes()

[1] "There are 10321 genes"
```
The right way to do it...

read_csv("Child_Variants.csv") -> variants

how_many_genes <- function(variant_data) {
  variant_data %>%
    distinct(GENE) %>%
    nrow() -> gene_count

  return(gene_count)
}

print(paste0("There are ", how_many_genes(variants)," genes"))

[1] "There are 10321 genes"
Data Guarantees

```r
how_many_genes <- function(variant_data, chr=NULL) {

  if (! is.null(chr)) {
    variant_data %>%
      filter(CHR==chr) -> variant_data
  }

  variant_data %>%
    distinct(GENE) %>%
    nrow() -> gene_count

  return(gene_count)
}

print(paste0("There are ", how_many_genes(variants)," genes"))
[1] "There are 10321 genes"

print(paste0("There are ", how_many_genes(variants,2)," genes on chr 2"))
[1] "There are 647 genes on chr 2"
```
function(variant_data, chr=NULL)

• **variant** should be a tibble (or data frame)
• **chr** should be either text or integer
  
  – Could get the wrong data type
  – Could get multiple values when only one is required
  – Could have an empty dataset
  – Could get an invalid value
    • Non-existent chromosome
    • NA, negative or infinite value

• R provides no guarantees – it’s all up to the function author
Actions upon problems

• `message()`: print a message to inform the user about something
• `warning()`: print a warning message but keep going
• `stop()`: print an error message and stop the script execution

```plaintext
> message("This is a message")
This is a message

> warning("This is a warning")
Warning message:
This is a warning

> stop("I will go no further")
Error: I will go no further
```
value <- -10

if (value < 0) {
  warning(paste("Values should be positive, yours was", value))
} else {
  message(paste("The value", value, "was fine"))
}

Warning message:
Values should be positive, yours was -10

value <- c(1, 2, -10)
Warning message:
In if (value < 0) {
  the condition has length > 1 and only the first element will be used
Collapsing logical vectors

c(1,5,-10) > 0
[1] TRUE TRUE FALSE

any(c(1,5,-10) > 0)
[1] TRUE

all(c(1,5,-10) > 0)
[1] FALSE

value <- c(1,2,-10)

if (any(value < 0)) {
  warning("Values should be positive, one of yours wasn't")
}
Argument checking

```r
how_many_genes <- function(variant_data, chr=NULL) {
  if (length(chr) != 1) {
    stop("You can only analyse one chromosome")
  }
  if (!is.null(chr) & !chr %in% variant_data$CHR) {
    warning(paste("No data for chr",chr))
  }
}

how_many_genes(variants,"Z")
Warning message:
In how_many_genes(variants, "Z") : No data for chr Z

how_many_genes(variants,c(1,2,"X"))
Error in how_many_genes(variants, c(1, 2, "X")) :
  You can only analyse one chromosome
```
# Argument Checking
(with `assertthat`)

## Raw Data Types and Values
- **is.scalar**: a single value
- **is.number**: a single number
- **is.numeric**: a numeric vector
- **is.count**: a positive integer
- **is.string**: a single text value
- **is.character**: a text vector
- **is.flag**: a single logical
- **is.logical**: a logical vector
- **is.date**: a date
- **noNA**: no NA values
- **not_empty**: some data in it

## Data structure checks
- **is.tibble**: a tibble
- **is.data.frame**: a data frame
- **is.list**: a list

## File operations
- **is.dir**: a directory
- **is.readable**: a readable file
- **is.writeable**: a writeable file
- **has.extension**: correct extension
Simpler checking with `assert_that`

- `assert_that(x)` is equivalent to `if(x){stop()}
- Constructs nice messages (or make your own)

```r
> assert_that(is.number("X"))
Error: "X" is not a number (a length one numeric vector).

> assert_that(is.readable("missing.txt"))
Error: Path 'missing.txt' does not exist

> assert_that(is_tibble(4))
Error: is_tibble(x = 4) is not TRUE

> assert_that(is_tibble(4), msg = "This was not a tibble")
Error: This was not a tibble
```
Easier Checking with `assertthat`

```r
library(assertthat)

how_many_genes <- function(variant_data, chr=NULL) {
  assert_that(is.data.frame(variant_data), msg="Not a data frame")

  assert_that("GENE" %in% colnames(variant_data), msg="No GENE column")

  if(!is.null(chr)) {
    assert_that(is.scalar(chr))
    assert_that(is.count(chr)|is.string(chr), msg="chr must be text or int")
  }
}

how_many_genes(variants)
how_many_genes("variants")
how_many_genes(variants,c(1,2,3))
how_many_genes(variants,TRUE)
```
Keeping going when testing

• Any error statement will cause execution to stop
• You can wrap any code you expect to fail in `try()` so this doesn’t happen
• Especially useful in notebooks (also `error=TRUE` in the block)

```r
try(how_many_genes("variants"))
Error: variant_data is not a data frame

try(how_many_genes(variants,c(1,2,3)))
Error: is.scalar(chr) | is.null(chr) is not TRUE

try(how_many_genes(variants,TRUE))
Error: is.null(chr) | is.count(chr) | is.string(chr) is not TRUE
```
Writing and debugging functions

```r
mean_centre_distance <- function(xvalues, yvalues) {
  mean(xvalues) -> centrex
  mean(yvalues) -> centrey

  xvalues-centrex ** 2 -> xdiff2
  yvalues-centrey ** 2 -> ydiff2

  sqrt(xdiff2+ydiff2) -> distances

  return(mean(distances))
}

mean_centre_distance(positions$x, positions$y)

[1] NaN
Warning message:
In sqrt(xdiff2 + ydiff2) : NaNs produced
```
Writing Functions

xvalues <- positions$x
yvalues <- positions$y

mean(xvalues) -> centrex
mean(yvalues) -> centrey

xvalues-centrex ** 2 -> xdiff2
yvalues-centrey ** 2 -> ydiff2

sqrt(xdiff2+ydiff2) -> distance

> head(xdiff2)
Writing Functions

```r
xvalues <- positions$x
yvalues <- positions$y

mean_center_distance <- function(xvalues, yvalues) {
  mean(xvalues) -> centresx
  mean(yvalues) -> centresy

  (xvalues-centresx) ** 2 -> xdiff2
  (yvalues-centresy) ** 2 -> ydiff2

  sqrt(xdiff2+ydiff2) -> distance

  return(distance)
}
```
Exercise 1 – Writing Robust Functions
Putting functions into packages
Toolchain

• Operating system packages
  – Rtools
    • Compiler
    • Make system

• R helper packages
  – devtools (many convenience methods)
  – roxygen2 (documentation)
  – testthat (test suite)
  – knitr (vignettes)
Rtools

- **OSX**
  - Command line developer tools
    xcode-select --install

- **Linux**
  - R development package
    sudo apt -y install r-base-dev

- **Windows**
  - Rtools40
  - Install from [https://cran.r-project.org/bin/windows/Rtools/](https://cran.r-project.org/bin/windows/Rtools/)
R helper packages

• `install.packages(c(
   "devtools", # Development helpers
   "roxygen2", # Documentation
   "testthat", # Testing
   "knitr")) # Writing vignettes`
Starting a package - naming

• Rules!
  – Only letters, numbers and dots (not hyphens or underscores)
  – Can’t start with a number
  – Can’t end with a dot

• Guidelines
  – Don’t use dots
  – All lowercase is best
  – Don’t use a name already in CRAN or Bioconductor
Starting a package – create git repository

• It’s easiest to create a new git repository first rather than creating a package and adding the repository later.
Starting a package – `create_package()`

- Check out your git repository
- Load devtools
- Use `create_package()` to make the basic file structure
- Open the newly created project

```
git clone https://github.com/s-andrews/sangr.git

library(devtools)
create_package("C:/Users/andrewss/git/sangr/")

File > Open Project [C:/Users/andrewss/git/sangr/sangr.Rproj]

Commit and Push to github (R folder won’t add until something is in it)
```
> create_package("C:/Users/andrewss/git/sangr/")

√ Setting active project to 'C:/Users/andrewss/git/sangr'
√ Creating 'R/'
√ Writing 'DESCRIPTION'
Package: sangr
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
    * First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
√ Writing 'NAMESPACE'
√ Writing 'sangr.Rproj'
√ Adding '.Rproj.user' to '.gitignore'
√ Adding '^sangr\.Rproj$', '^\\.Rproj\\.user$' to '.Rbuildignore'
√ Opening 'C:/Users/andrewss/git/sangr/' in new RStudio session
√ Setting active project to '<no active project>'
Configuring your package

• Package metadata is in the DESCRIPTION file

• Edit this file to specify suitable values for the metadata

• Just do the Title, Authors and Description to start with

Package: sangr
Title: Simulate Sanger Sequencing Chromatograms
Version: 0.0.0.9000
Authors@R:
  person(given = "Simon", family = "Andrews", role = c("aut", "cre"), email = "simon.andrews@babraham.ac.uk", comment = c(ORCID = "0000-0002-5006-3507"))
Description: Simulate Sanger Sequencing Chromatograms
License: `use_mit_license()` or `use_gpl3_license()` or friends to pick a license
Encoding: UTF-8
LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
Package development workflow

• Write R code
  – Editing .R files under the R folder in the package

• Reload the package to test the new code
  – Call `load_all()` to bring the new code into your environment

• Update Tests
  – ....
Writing R Code

• Create a new .R file under the R directory
  – Best to use the `use_r()` function to create the file

• Write one or more functions within the newly created file

```r
> use_r("simulate_sanger_data")
✓ Setting active project to 'C:/Users/andrewss/git/sangr'
* Modify 'R/simulate_sanger_data.R'
* Call `use_test()` to create a matching test file
```
Functions and Files

- How many R files do you make?
- How do you divide your functions between them?

- Put each main function in its own file
- Name the file after the function (foo() goes in foo.R)

- Helper functions can either go alongside the main ones or be in a separate file of their own (eg utils.R)
Functions and Files

simulate_sanger_data.R
- `simulate_sanger_data`
- `add_base`
- `add_noise`
- `degrade_signal`

draw_chromatogram.R
- `draw_chromatogram`

merge_sanger_data.R
- `merge_sanger_data`
library(tidyverse)
theme_set(theme_classic(base_size = 16))

simulate_sanger_data <- function(sequence, sd=5, noise=0.2, degrade=0.8) {
    trace_length <- 20*(nchar(sequence)+2)
    start_signal <- rep(0,trace_length)

    tibble(
        POS=1:trace_length,
        G=start_signal,A=start_signal,T=start_signal,C=start_signal
    ) -> base_data

    for (i in 1:nchar(sequence)) {
        base <- substr(sequence,i,i)
        position <- 20 * (i+1)
        add_base(base_data,base,position,sd) -> base_data
    }

    add_noise(base_data,noise) -> base_data
    degrade_signal(base_data, degrade) -> base_data

    return(base_data)
}
Making functions robust

• More important when your function will be run by people other than you!

• Two main changes:
  1. Make sure your function isn’t affected by the environment in the script in which it’s run
  2. Validate the input data and provide helpful error messages if anything is wrong
Making functions robust

• More important when your function will be run by people other than you!

• Two main changes:

  1. Make sure your function isn’t affected by the environment in the script in which it’s run

  2. Validate the input data and provide helpful error messages if anything is wrong
Being a good citizen

• We should ensure our function can’t be broken by the environment in which it’s run (ie what the user has already done in their R session)

• We should ensure that nothing our function does could break other code in the users R session
How R finds functions

tibble(x=1:10,y=21:30) -> data

how_many_rows <- function(x) {
  return(nrow(x))
}

> library(MASS)

Attaching package: ‘MASS’

The following object is masked from ‘package:dplyr’:

select

> search()

[1] ".GlobalEnv"       "package:MASS"
[3] "package:dplyr"    "tools:rstudio"
[9] "package:datasets" "package:methods"

how_many_rows(data)
[1] 5
Stop Relying on `.GlobalEnv`

tibble(x=1:10, y=21:30) -> data

how_many_rows <- function(x) {
  return(base::nrow(x))
}

nrow <- function(x) {
  return(5)
}

how_many_rows(data)
[1] 50
Don’t modify `.GlobalEnv`

**User’s Code**

```r
library(MASS)
library(your_package)
select(some_data)
```

**Your Package’s Code**

```r
library(dplyr)
myfunc <- function(x){
  select(x,1:5)
}
```

You just broke their script!
Things your package code shouldn’t do!

- **Use** `library` **to load other packages**
- **Rely on** `.GlobalEnv` **to find functions that it uses**
- Change or rely on the working directory
- Create or rely on any global variables

- Change or rely on any global options
  - Graphics parameters (e.g., ggplot theme or `par`)
  - System options (e.g., locale)
  - Random number generator seed (restore it if used)
Using other packages in your R code

- Add the package as a dependency in your metadata
- Call functions with explicit package names

```r
tibble::tibble(
    POS=1:trace_length,
    G=start_signal,A=start_signal,T=start_signal,C=start_signal
)
```

- Core functions mostly come from the `base::` package, but there are other default packages (eg `stats::`)
Adding Dependencies

• You need to tell R if your code relies on functions from other packages. Dependencies are recorded in the DESCRIPTION file.

• Two levels
  – A package is used and your package won’t work if it’s not there
    • Add an ‘Imports’ dependency
  – A package can be used but your package will still work without it
    • Add a ‘Suggests’ dependency

  – By default Imports are automatically installed, but Suggests aren’t
Adding Dependencies

• The easiest way to add dependencies to your project is with 
  use_package()

use_package("tibble")
use_package("tibble","Suggests")
use_package("tibble","Suggests",min_version="3.0.0")

use_package("tibble")
✓ Adding 'tibble' to Imports field in DESCRIPTION
* Refer to functions with `tibble::fun()`

LazyData: true
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.1.1
Imports:
tibble
Unusual dependencies

• If you use the %>% pipe then you need to add that as a dependency. It’s not a function so you do need to import it into the environment with either

  `usethis::use_package("magrittr","Depends")`
  `usethis::use_pipe()`
Trying out new code

- You can simulate updating and loading a modified package using the `load_all()` function

- Functions from the package will be imported into your environment similar to if you’d used `library()`

```r
> load_all()
Loading sangr
> packageVersion("sangr")
[1] ‘0.0.0.9000’
```
More thorough testing

• To do a more complete check of the structure, code and metadata in your package you can do a full build using the `check()` function.

• This is a more complete (and slower) option than `load_all()` it does much more than just re-import your functions.
Using Tidyverse Column Names

• The `check` function produces a spurious warning if you use a tibble column name. Define a variable with that name, set to NULL to work round this.

```r
# Example code snippet

# This avoids the spurious warnings from check()

# Define a variable
POS <- NULL

# Use the variable in the code

data %>%
  tidyr::pivot_longer(
    cols=-POS,
    names_to="base",
    values_to="density"
  )
```

> checking R code for possible problems ... NOTE
add_noise: no visible binding for global variable 'POS'

```r
> POS <-

# Use the variable in the code

data %>%
  tidyr::pivot_longer(
    cols=-POS,
    names_to="base",
    values_to="density"
  )
```
-- R CMD check results ---------------------------------- sangr 0.0.0.9000 ----
Duration: 23s

> checking DESCRIPTION meta-information ... WARNING
 Non-standard license specification:
   `use_mit_license()``, `use_gpl3_license()` or friends to pick a license
 Standardizable: FALSE

> checking top-level files ... NOTE
 File LICENSE is not mentioned in the DESCRIPTION file.

> checking R code for possible problems ... NOTE
 add_base: no visible global function definition for ':='
 add_base: no visible global function definition for 'sym'
 add_base: no visible global function definition for 'dnorm'
 Undefined global functions or variables:
   := dnorm sym
 Consider adding
   importFrom("stats", "dnorm")
to yourNAMESPACE file.

0 errors √ | 1 warning x | 2 notes x
Exercise 2: Creating your package
Metadata and Documentation
Package Versions

• Package versions are generally split into numeric sections, separated by dots eg (2.10.5)

  – Major version (2)
  – Minor version (10)
  – Patch version (5)

• Sometimes the version will be followed by a dev version – always above 9000 for development (never releases) eg 2.10.5.9000
Package versions

• Major
  – Starts at 0 and moves to 1 when all basic functionality is complete.
  – Increments when adding significant new functionality, or breaking backwards compatibility

• Minor
  – Starts at 0 and increases within the same major version any time any functionality is added or changed

• Patch
  – Starts at 0 and increases every time a bug is fixed within the same minor version
Package Version Examples

• 1.0.0 to 1.0.1 – Bug fix in a function or documentation update

• 1.0.0 to 1.1.0 – Added a new function or improved the functionality of an existing function

• 1.0.0 to 2.0.0 – Made a major change in the way that something works – possibly breaking existing code
Choosing a License

• The `DESCRIPTION` file specifies the license under which your code is to be made available.

• You need to use a standard license if you want to submit to CRAN.

• Private code stored on github can use any license (including none).
Standard CRAN licenses (for code)

- **MIT** (*use_mit_license()*):
  - Very simple and permissive
  - Anyone can use the code in any way they like with no conditions
  - Standard disclaimer to protect you from litigation

- **Apache2** (*use_apl2_license()*):
  - Very similar to MIT, very permissive
  - Provides some protection against the use of software patents

- **GNU Public License** (*use_gpl3_license()*):
  - Permissive in that people can *use* the code without restriction
  - Restrictive in that if others modify and distribute the code they must share their changes under the same license
Applying a license

> `use_gpl3_license(name="Simon Andrews")`
✓ Setting License field in DESCRIPTION to 'GPL-3'
✓ Writing 'LICENSE.md'
✓ Adding 'LICENSE\.md$' to '.Rbuildignore'

License: GPL-3 + file LICENSE

- You need to set the name of the code’s copyright holder (who wrote it)
- Multiple people can be added by separating names with semi-colons
- Manually add the LICENSE file to the License file in DESCRIPTION
Adding Function Documentation

- Use the roxygen2 package to add and edit documentation
- Adds both text documentation and additional metadata

- Documentation is added to the source files as comments above each function

- Rstudio can generate a template for each function
#' Title
#' 
#' @param sequence
#' @param sd
#' @param noise
#' @param degrade
#' 
#' @return
#' @export
#' 
#' @examples
simulate_sanger_data <- function(sequence, sd=5, noise=0.2, degrade=0.8) {

#' Simulate sanger chromatogram density data
#' @param sequence A string of nucleotides to use (GATC)
#' @param sd The standard deviation of the peak width
#' @param noise What proportion of the signal to make from random noise
#' @param degrade What proportion of the signal should be lost by the end
#' @return A tibble with positions (20 per base) and G/A/T/C signal
#' @export
#' @examples
#' simulate_sanger_data("GAATTC")
simulate_sanger_data <- function(sequence, sd=5, noise=0.2, degrade=0.8) {

Using data in examples

• You can add data files to your package
  – Data designed to be accessed directly
  – Example data to show how to parse for example

• R data should be put into a folder called ‘data’ in Rda format

• Example data should go in a folder called ‘inst’
  – Can access using `system.file("good.fq", package = "fastqR")`
Checking Function Documentation

• Update documentation using the `document()` function

• Read the compiled version using `?function_name`

Simulate sanger chromatogram density data

Description
Simulate sanger chromatogram density data

Usage
`simulate_sanger_data(sequence, sd = 5, noise = 0.2, degrade = 0.8)`

Arguments

- `sequence`: A string of nucleotides to use (GATC)
- `sd`: The standard deviation of the peak width
- `noise`: What proportion of the signal to make from random noise
- `degrade`: What proportion of the signal should be lost by the end

Value
A tibble with positions (20 per base) and G/A/T/C signal

Examples
`simulate_sanger_data("GAAATC")`
Adding imports to documentation

• You sometimes need to add `@importFrom` statements to your roxygen documentation

• Normally to import additional operators (eg `%>%` or `:=`)

• Roxygen will add these to your `NAMESPACE` file automatically
Adding imports to documentation

#' @return A tibble with positions (20 per base)
#' @export
#' @importFrom magrittr %>%
#' @examples
#' simulate_sanger_data("GAATTC")
Documenting the package as a whole

• It can be useful to add package level documentation rather than just documenting functions

• Still use roxygen2 to do this, but you have to document NULL which then flags this as package level documentation

• Can read with `package?sangr` (doesn’t work from devtools though)

• Probably best to put this into a fresh R file
Documenting the package as a whole

#' sangr: A package for simulating sanger sequencing chromatograms
#' 
#' The sangr package provides three main functions which you might want to use.
#' 
#' @section sangr functions:
#' simulate_sanger_data
#' 
#' draw_chromatogram
#' 
#' merge_sanger_data
#' 
#' @docType package
#' @name sangr
NULL
Writing Vignettes

• Longer form documentation – not just function documents, but a description of how to use the package
• Created similarly to R notebooks using Markdown.

```r
> usethis::use_vignette("sangr_usage")
√ Adding 'knitr' to Suggests field in DESCRIPTION
√ Setting VignetteBuilder field in DESCRIPTION to 'knitr'
√ Adding 'inst/doc' to '.gitignore'
√ Creating 'vignettes/
√ Adding '*.html', '*.R' to 'vignettes/.gitignore'
√ Adding 'rmarkdown' to Suggests field in DESCRIPTION
√ Writing 'vignettes/sangr_usage.Rmd'
* Modify 'vignettes/sangr_usage.Rmd'
```
Writing Vignettes

```{r, include = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>
)
```

```{r setup}
library(sangr)
```
Exercise 3: Licensing and Documentation
Testing and Installation
Writing a Test Suite

• R packages can integrate with the testthat package to create a test suite for your code.

• The test suite is automatically run whenever someone builds your package to check that the code is working on their machine.

• You can also use the test suite to check that any changes you make don’t break the code.
Creating a Test Suite

• Run `testthat()` to create the basic structure

```r
> use_testthat()
√ Adding 'testthat' to Suggests field in DESCRIPTION
√ Creating 'tests/testthat/
√ Writing 'tests/testthat.R'
* Call `use_test()` to initialize a basic test file and open it for editing.
```
Adding tests

• Good idea to group tests by either function or type (parameters, errors etc)

• Run `use_test("simulate_sanger_data")` to create `test-simulate_sanger_data.R` in the `testthat` directory

• Write tests in the newly created file
library(sangr)

test_that("Simulation Parameters", {
  good_sequence <- "GATC"
  expect_equal(ncol(simulate_sanger_data(good_sequence)),5)
  expect_equal(nrow(simulate_sanger_data(good_sequence)),120)
  expect_equal(nrow(simulate_sanger_data(good_sequence,sd = 10)),120)
  expect_equal(nrow(simulate_sanger_data(good_sequence,degrade = 0.1)),120)
  expect_equal(nrow(simulate_sanger_data(good_sequence,noise = 0.5)),120)
})

test_that("Invalid Parameters", {
  expect_error(simulate_sanger_data("gatc"))
  expect_error(simulate_sanger_data("jeyc"))
  expect_error(simulate_sanger_data(1234))
  expect_error(simulate_sanger_data("GATC",sd=-1))
  expect_error(simulate_sanger_data("GATC",degrade=2))
  expect_error(simulate_sanger_data("GATC",noise=-1))
})
• `expect_equal(10, 10.0)`
• `expect_gt(20, 10)`
• `expect_lt(10, 20)`
• `expect_true()`
• `expect_false()`
• `expect_match("GGATCC", "GATC")`

• `expect_output(
  str(tibble()),
  "tibble")`

• `expect_is(1:10, "integer")`

• `expect_error()`
• `expect_warning()`
Running tests

• Explicitly with `test()`
• Automatically as part of `check()`

> `test()`
Loading sangr
Testing sangr
✓ | OK F W S | Context
✓ |   6      | `simulate_sanger_data` [0.3 s]

== Results =============================
Duration: 0.3 s

[ FAIL 0 | WARN 0 | SKIP 0 | PASS 6 ]
Fixing tests

```r
expect_error(simulate_sanger_data("GATC", sd=-1))
```

**Warning** (test-simulate_sanger_data.R:22:3): Invalid Parameters
NAs produced
Backtrace:
  1. testthat::expect_error(simulate_sanger_data("GATC", sd = -1)) test-simulate_sanger_data.R:22:2
  22. stats::runif(nrow(data), min = 0, max = biggest_signal * noise)
=====================================================================

== Results ==
Duration: 0.4 s

[ FAIL 0 | WARN 10 | SKIP 0 | PASS 10 ]
Fixing tests

```r
expect_error(simulate_sanger_data("GATC", sd=-1))
```

```r
simulate_sanger_data <- function(sequence, sd=5, noise=0.1, degrade=0.8) {
  if (sd<=0) {
    stop("SD must be more than zero")
  }
>
> test()
Loading sangr
Testing sangr
√ | OK F W S | Context
√ | 10       | simulate_sanger_data [0.3 s]

== Results =========================
Duration: 0.3 s

[ FAIL 0 | WARN 0 | SKIP 0 | PASS 10 ]
```
Installing the Package

• From github
  
  library(devtools)
  install_github("s-andrews/sangr", build_vignettes=TRUE)

• From source

  - R CMD build sangr
  - R CMD install sangr_0.1.tar.gz
Installing package into `/home/student/R/x86_64-pc-linux-gnu-library/4.0`
(as 'lib' is unspecified)
* installing *source* package 'sangr' ...
** using staged installation
** R
** inst
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
    converting help for package 'sangr'
        finding HTML links ... done
        draw_chromatogram.html
        merge_sanger_data.html
        sangr.html
        simulate_sanger_data.html
** building package indices
** installing vignettes
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (sangr)
Exercise 4: Testing and Installation