# Testing R Code by Richie Cotton



## Today's itinerary

1. Introduction
    - What we'll do today
    - Icebreaker
2. Run-time testing with `assertive`
    - *assert* functions
    - *is* and *has* functions
    - Case study: `geomean`
    - Exercises
3. Development-time testing with `testthat`
    - Creating unit tests
    - Integrating tests into packages
    - Exercises
4. Writing maintainable and testable code
    - Don't Repeat Yourself
    - Keep It Simple, Stupid
    - Fail Early, Fail Often
    - Exercises

## R Packages that you need today

Today, we'll be using the following packages. You may wish to install them now, if you haven't get them already.

```
assertive, testthat, ggplot2, sig, Hmisc, data.table, plyr, dplyr, mice
```

## The very short version

- Run-time testing is for checking that people using your code aren't doing stupid things.

- Development-time testing is for checking that you didn't do stupid things when writing your code.
- If you break your code down into small pieces, it will usually be easier to maintain and easier to test.

# Run-time testing of code with *assertive*

## *assert* functions

There are times when it is a good idea to check the state of your variables, to ensure that they have the properties that you think they have. For example, if you have a count variable, you might want to check that it is numeric, that all the values are non-negative, and that all the values are whole numbers.

Base-R has a function called `stopifnot` that lets you perform such checks.

```
counts <- c(1, 2, 3, 4.5)
stopifnot(
  is.numeric(counts),
  all(counts >= 0),
  isTRUE(all.equal(counts, round(counts)))
)
```

```
## Error: isTRUE(all.equal(counts, round(counts))) is not TRUE
```

This is OK, but the code isn't that easy to read. Worse, the error messages that it produces in the event of failure aren't very user-friendly.

`assertive` provides lots of *assert* functions that provide checks for specific conditions. (An *assertion* is software development jargon for a check.) They are designed to make your code easier to read, and to return helpful error messages to users in the event of a check failing.

Here's the same example again, written in an `assertive` style.

```
library(assertive)
counts <- c(1, 2, 3, 4.5)
assert_is_numeric(counts)
assert_all_are_non_negative(counts)
assert_all_are_whole_numbers(counts)
```

```
## Error in eval(expr, envir, enclos): counts are not all whole numbers.
## There was 1 failure:
##   Position Value      Cause
## 1        4   4.5 fractional
```

Here you see that the error message contains a human readable sentence, followed by information on the values that caused problems, along with their positions and reasons for failure.

## *is* and *has* functions

Each of the *assert* functions has an underlying *is* or *has* function. For example, `assert_is_numeric` calls `is_numeric`, `assert_all_are_non_negative` calls `is_non_negative`, and so on.

Some *is* and *has* functions, such as `is_numeric`, return a single logical value.

```
is_numeric(1:6)
```

```
## [1] TRUE
```

```
is_numeric(letters)
```

```
## [1] FALSE
## attr(,"cause")
## [1] letters is not of type 'numeric'; it has class 'character'.
```

When the check passed, `is_numeric` returned `TRUE`, and when it failed, `is_numeric` returned `FALSE` with a `cause` attribute explaining the problem.

Where *is* functions return a single value, they have a single corresponding *assert* function prefixed by `assert_`. For example, `is_numeric` is paired with `assert_is_numeric`.

Some *is* and *has* functions, such as `is_non_negative`, return a logical vector.

```
is_non_negative(rnorm(6))
```

```
## -0.687605702476876 -0.669164001253048   1.99914264432859
##              FALSE              FALSE              TRUE
##  0.293517442709438 -0.417878025432502 -0.328072635233795
##               TRUE              FALSE             FALSE
## attr(,"cause")
## [1] too low too low                 too low too low
```

`is_non_negative` returned a logical vector which was `TRUE` where the check passed, and `FALSE` where the check failed. This time the `cause` attribute was also vectorised, returning an empty string for the passes and a brief explanation of the problem for the failures.

Where *is* functions return a vector, there are two corresponding *assert* functions, prefixed `assert_all_are_`, and `assert_any_are_`. For example, `is_non_negative` is paired with `assert_all_are_non_negative` and `assert_any_are_non_negative`.

## Why do we use *assertive*?

The main alternative is *assertthat*. This is a much lighter weight package that provides only the basics, like `is.string`, and `is.dir`.

## What can assertive do?

The types of checks available in the assertive package can be grouped into a few categories.

*Testing types*

You can test for a particular type of object using `is_numeric`, `is_character`, `is_matrix`, `is_data.frame`, through to more obscure types like `is_qr`, `is_name`, and `is_relistable`.

`is_s4`, `is_atomic` and `is_recursive` test properties of variables.

`is_a_number`, `is_a_string`, `is_a_bool`, etc. combine tests for types with `is_scalar` (see below) to check for a single numeric/character/logical value respectively.

*Testing sizes*

`is_scalar` tests for objects of length one, or with one element (this can be different for lists; you choose the metric).

Similarly `is_empty` and `is_non_empty` test for objects of zero length/containing zero elements.

More generally `is_of_length` and `has_elements` test for a particular length/number of elements.

*Testing missing values*

You can test for NAs, NaNs, and NULLs using `is_na`, `is_nan` and `is_null`, or their opposites `is_not_na`, `is_not_nan` and `is_not_null`.

*Testing numbers*

`is_in_range` tests if a number is in a numeric range, along with the more specialised wrappers: `is_in_open_range`, `is_in_closed_range`, `is_in_left_open_range`, `is_in_right_open_range`, `is_positive`, is_negative,is_non_positive,is_non_neg

Finiteness can be tested with `is_finite`, `is_infinite`, `is_positive_infinity` and `is_negative_infinity`.

`is_odd` and `is_even` test for those qualities, and are generalized by `is_divisible_by`.

`is_whole_number` tests whether a number is an integer, give or take some tolerance.

`is_real` and `is_imaginary` test for real/imaginary numbers.

*Testing attributes*

Rows, columns and dimensions can be tested for using, `has_rows`, `has_cols` and `has_dims`.

Similarly their names can be tested for using `has_rownames`, `has_colnames`, `has_dimnames` and `has_names`.

Duplicates can be tested for using `has_duplicates` or its opposite has_no_duplicates‘.

Attributes can be tested for using `has_attributes` and `has_any_attributes`.

Functions arguments can be tested using `has_arg`.

Model terms can be tested using `has_terms`.

*Testing files and connections*

`is_dir` tests if a path refers to an existing directory.

`is_existing_file` tests whether a file exists.

`is_executable_file`, `is_readable_file`, and `is_writable_file` test your permissions to access a file (though they are based on `file.access`, which is slightly unreliable on Windows).

`is_connection` tests whether an object is a connection, and there are many specialized types including `is_file_connection`, `is_fifo_connection`, `is_pipe_connection`, `is_readable_connection`, `is_open_connection`, `is_writable_connection`, `is_stdin`, `is_stdout` and `is_stderr`.

*Testing time*

`is_in_future` and `is_in_past` test when a date-time object is.

*Testing sets*

`is_set_equal` tests if two vectors contain the same elements, regardless of order.

`is_subset` and `is_superset` test if one vector contains another.

*Testing complex data types*

`is_email_address`, `is_credit_card_number`, `is_date_string`, `is_honorific`, `is_ip_address`, `is_hex_color`, `is_cas_number` and `is_isbn_code` check for

`is_uk_car_licence`, `is_uk_national_insurance_number`, `is_uk_postcode` and `is_uk_telephone_number` test the United Kingdom-specific data types.

`is_us_telephone_number`, `is_us_zip_code` test the United States-specific data types.

## Case study: calculating geometric means

By far the most common use of assertions is for checking input to functions. Consider this function for calculating the geometric mean:

```
geomean <- function(x, na.rm = FALSE)
{
  exp(mean(log(x), na.rm = na.rm))
}
```

In a statically-typed language, we could enforce `x` being a numeric vector. R's dynamic typing (while mostly helping us be more productive) gives us some rope to hang ourselves with: `x` and `na.rm` can be absolutely anything. We need to handle the cases when `x` is not numeric, or when x contains negative values, or when `na.rm` is not a single logical value.

The built-in functions `exp`, `mean` and `log` have some of their own logic for handling bad inputs, and it is possible to simply rely on that logic rather than writing your own. For demonstration purposes, let's see how they behave, and then see if we can improve upon it.

If we pass a non-numeric `x`, we see:

```
geomean("a")
```

```
## Error in log(x): non-numeric argument to mathematical function
```

The error message is OK, but it since it is appearing to come from `log(x)`, it isn't totally clear to the user where the problem originates. The *assertive* fix is to include `assert_is_numeric(x)` in the function.

Where should this line go? In accordance with the first clause of the programming principle "fail early, fail often", the assertion belongs at the start of the function.

```
geomean2 <- function(x, na.rm = FALSE)
{
  assert_is_numeric(x)
  exp(mean(log(x), na.rm = na.rm))
}
```

```
geomean2("a")
```

```
## Error in geomean2("a"): x is not of type 'numeric'; it has class 'character'.
```

The geometric mean doesn't make any mathematical sense for (real) negative numbers, and will return `NaN` if the input contains any.

```
geomean2(rnorm(20))
```

```
## Warning in log(x): NaNs produced
```

```
## [1] NaN
```

The warning here is not so informative (why were the NaNs produced?), and appears to come from `log(x)`. We could be strict and throw an error if there are negative values by adding a call to `assert_all_are_non_negative`. To replicate the base-R behaviour we can define custom actions based upon the result of `is_non_negative`:

```
geomean3 <- function(x, na.rm = FALSE)
{
  assert_is_numeric(x)
  if(any(is_negative(x), na.rm = TRUE)) # Don't worry about NAs here
  {
    warning("x contains negative values, so the geometric mean makes no sense.")
    return(NaN)
  }
  exp(mean(log(x), na.rm = na.rm))
}
```

```
geomean3(rnorm(20))
```

```
## Warning in geomean3(rnorm(20)): x contains negative values, so the
## geometric mean makes no sense.
```

```
## [1] NaN
```

For `na.rm`, the `mean` function coerces its input to be a logical value, warning if the value's length is more than one.

```
x <- rlnorm(20)
x[sample(20, 5)] <- NA
geomean(x, c(1.5, 0))
```

```
## Warning in if (na.rm) x <- x[!is.na(x)]: the condition has length > 1 and
## only the first element will be used
```

```
## [1] 0.9569963
```

The warning about the length is OK, but again its source is not totally clear for users. The coercion to logical happens silently, which isn't ideal.

Again, we could be strict and throw an error when `na.rm` isn't a scalar logical value using `assert_is_a_bool`. (This is a compound assertion checking both the type and the length of the object). In this case, to replicate the base-R behaviour, we will use some utility functions provided by *assertive*.

```
geomean4 <- function(x, na.rm = FALSE)
{
  assert_is_numeric(x)
  if(!all(is_non_negative(x), na.rm = TRUE)) # Don't worry about NAs here
  {
    warning("x contains negative values, so the geometric mean makes no sense.")
    return(NaN)
  }
  na.rm <- coerce_to(use_first(na.rm), "logical")
  exp(mean(log(x), na.rm = na.rm))
}
```

`use_first` takes the first element of an object, warning if it has length more than one. `coerce_to` checks and object's `class`, then converts it to the requested type, with a warning, using an appropriate `as.*` function if it exists, or `as` if it doesn't.

```
geomean4(x, c(1.5, 0))
```

```
## Warning: Only the first value of 'na.rm' will be used.
```

```
## Warning: Coercing use_first(na.rm) to class 'logical'.
```

```
## [1] 0.9569963
```

Whether to throw an error on bad input or fix it is personal preference and depends upon context. For end-user functions should should usually try to be flexible and fix things unless they've done something very silly. For lower-level functions, you can often afford to be stricter.

## Development time testing with the *testthat* package.

The assertions that we used in the run-time testing section are mainly used for checking that your user's haven't broken your code. Development-time testing is

more about checking that your code is gives the right answer in the first place. In R, these checks usually happen at the level of a function. That is, you have a test that calls a function, and checks whether the return value is the same as what you expected. The computer science jargon name for this sort of test is a *unit test*.

## Why do we use *testthat*?

The main alternative is *RUnit*. This follows the syntax of *xUnit* more closely, so if you've done any unit testing in other programming languages, then it's a bit easier to learn. However, *RUnit* doesn't have test caching, so it's much slower, especially for large projects, and it is missing some key features like checking warnings. You can automatically convert *RUnit* tests to *testthat* tests using the *runittotestthat* package.

## The basic unit test structure

Consider a simple function for calculating hypotenuses of right angled triangles (on a flat surface).

```
hypotenuse <- function(x, y)
{
  sqrt(x ^ 2 + y ^ 2)
}
```

To test this, we can use some triangles where we know the answer.

```
library(testthat)
test_that(
  "hypotenuse, with inputs x = 3 and y = 4, returns 5",
  {
    expected <- 5
    actual <- hypotenuse(3, 4)
    expect_equal(actual, expected)
  }
)
```

The basic test structure often looks very similar to this. The first argument is a description of the test, to remind what the point of it is when you come back to it a few months later, and the test code usually contains those three lines: declare what you think the answer should be, calculate it, and then check that the two numbers are equal.

As well as testing that for correct answers, we can test that the behaviour is as expected when bad inputs are passed. This is even easier than our first test.

```
test_that(
  "hypotenuse, with no inputs, throws an error",
  {
    expect_error(
      hypotenuse(),
      'argument "x" is missing, with no default'
    )
  }
)
```

Other common variants of expectations are:

- `expect_true`, `expect_false` and `expect_null`, which are shortcuts for checking those common return types.
- `expect_warning`, `expect_message` and `expect_output`, for testing feedback, which work like `expect_error`.

You may also occasionally come across these rare expectations:

- `expect_less_than` and `expect_greater_than`, for numeric inequalities.
- `expect_identical`, a stricter check than `expect_equal`.
- `match`, for matching strings using regular expressions.
- `is`, for checking the class of variables.

#### Tests with multiple expectations

You can combine multiple expectations into a single test. This is a common thing to do when checking warnings.

```
test_that(
  "min, with a zero-length input, returns infinity with a warning",
  {
    expected <- Inf
    expect_warning(
      actual <- min(numeric()),
      "no non-missing arguments to min; returning Inf"
    )
    expect_equal(actual, expected)
  }
)
```

### Labelling test error messages

In the event that a test fails, *testhat* does reasonably well at explaining what the problem is. Sometimes however, a little more information can be informative to discover what went wrong, and *testthat* allows you to provide additional information in the failure message.

Consider this test that doesn't work.

```
test_that(
  "hypotenuse, with a NULL input, returns NULL",
  {
    expect_null(hypotenuse(3, NULL))
  }
)
```

```
## Error: Test failed: 'hypotenuse, with a NULL input, returns NULL'
## Not expected: hypotenuse(3, NULL) isn't null.
```

On the face of it, the test seemed reasonable: put `NULL` into the function and get `NULL` out again. The test failure message correctly tells us that `hypotenuse(3, NULL)` isn't null, but it leaves us hanging with the question: what did the function return? By customising the `label` argument, we can find out straight away. In the next chunk of code, `deparse` turns a value into the code that would generate that value.

```
test_that(
  "hypotenuse, with a NULL input, returns NULL",
  {
    actual <- hypotenuse(3, NULL)
    label <- paste("hypotenuse(3, NULL) =", deparse(actual))
    expect_null(actual, label = label)
  }
)
```

```
## Error: Test failed: 'hypotenuse, with a NULL input, returns NULL'
## Not expected: hypotenuse(3, NULL) = numeric(0) isn't null.
```

### Integrating tests into packages

The file structure of R packages is quite strict. In the simplest case, at the top level, you have files named `DESCRIPTION` (containing the name, version, and authors of the package, amongst other things), `NAMESPACE` (containing the names of the functions to export to users), and directories `R` (containing your R code) and `man` (containing help files for your functions and datasets.

`testthat` tests go in a directory `tests/testthat`. You also need a file named `tests/testthat.R`, which contains the following code:

```
library(testthat)
library(yourpackage)
test_package("yourpackage")
```

Then whenever you check your package (using `R CMD check` or `devtools::check`), R will automatically run all the tests.

### Testing complex objects

Where `expected` is a complex object, say an S3/S4/Reference/R6 class or an environment, it is very easy to end up with very complex tests containing many expectations.

For example, if you are testing a linear modelling function, the resulting model has a lot of components.

```
model <- lm(dist ~ speed, cars)
str(model)
```

```
## List of 12
##  $ coefficients : Named num [1:2] -17.58 3.93
##   ..- attr(*, "names")= chr [1:2] "(Intercept)" "speed"
##  $ residuals    : Named num [1:50] 3.85 11.85 -5.95 12.05 2.12 ...
##   ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:50] -303.914 145.552 -8.115 9.885 0.194 ...
##   ..- attr(*, "names")= chr [1:50] "(Intercept)" "speed" "" "" ...
##  $ rank         : int 2
##  $ fitted.values: Named num [1:50] -1.85 -1.85 9.95 9.95 13.88 ...
##   ..- attr(*, "names")= chr [1:50] "1" "2" "3" "4" ...
##  $ assign       : int [1:2] 0 1
##  $ qr           :List of 5
##   ..$ qr   : num [1:50, 1:2] -7.071 0.141 0.141 0.141 0.141 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:50] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:2] "(Intercept)" "speed"
##   .. ..- attr(*, "assign")= int [1:2] 0 1
##   ..$ qraux: num [1:2] 1.14 1.27
##   ..$ pivot: int [1:2] 1 2
##   ..$ tol  : num 1e-07
##   ..$ rank : int 2
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 48
```

```
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = dist ~ speed, data = cars)
##  $ terms        :Classes 'terms', 'formula' length 3 dist ~ speed
##   .. ..- attr(*, "variables")= language list(dist, speed)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "dist" "speed"
##   .. .. .. ..$ : chr "speed"
##   .. ..- attr(*, "term.labels")= chr "speed"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(dist, speed)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. ..- attr(*, "names")= chr [1:2] "dist" "speed"
##  $ model        :'data.frame':   50 obs. of  2 variables:
##   ..$ dist : num [1:50] 2 10 4 22 16 10 18 26 34 17 ...
##   ..$ speed: num [1:50] 4 4 7 7 8 9 10 10 10 11 ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula' length 3 dist ~ speed
##   .. .. ..- attr(*, "variables")= language list(dist, speed)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "dist" "speed"
##   .. .. .. .. ..$ : chr "speed"
##   .. .. ..- attr(*, "term.labels")= chr "speed"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(dist, speed)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "dist" "speed"
##  - attr(*, "class")= chr "lm"
```

To test everything: the coefficients, the fitted values, the residuals, the call, and all the obscure little parameters contained in this object, takes a lot of time and effort, and will result in difficult to maintain testing code. So how much do you need to test in a case like this?

The answer depends upon what you are trying to test. If you are the author of the `lm` (or other modelling) function, then you need to make sure that it returns a particular structure, and some in depth testing is necessary. However, if you are focussed on analysing some data, and you have a function that wraps `lm`, then you probably only need to test that the coefficients are what you expected.

That is, the "keep it simple, stupid" principle applies here: don't write overly

complicated tests if you don't have to.

# Writing maintainable and testable code

## Don't Repeat Yourself

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

c2.com

Or equivalently,

> Duplicated code is bad code: anything that appears in two or more places in a program will eventually be wrong in at least one.

softwarecarpentry.org

### An example

You've created some plots to gain insight into the diamond market, but when you show them to your boss, she complains that the tasteful royal blue color scheme that you chose doesn't follow corporate marketing procedures, and that you should have used lime green instead.

```
library(ggplot2)

(scatter_price_vs_carat <- ggplot(diamonds, aes(carat, price)) +
  geom_point(color = "royalblue")
)


(density_cut_vs_price <- ggplot(diamonds, aes(price, color = cut)) +
  geom_density(color = "royalblue")
)


(bar_clarity_by_cut <- ggplot(diamonds, aes(clarity)) +
  geom_bar(fill = "royalblue") +
  facet_wrap(~ cut)
)
```

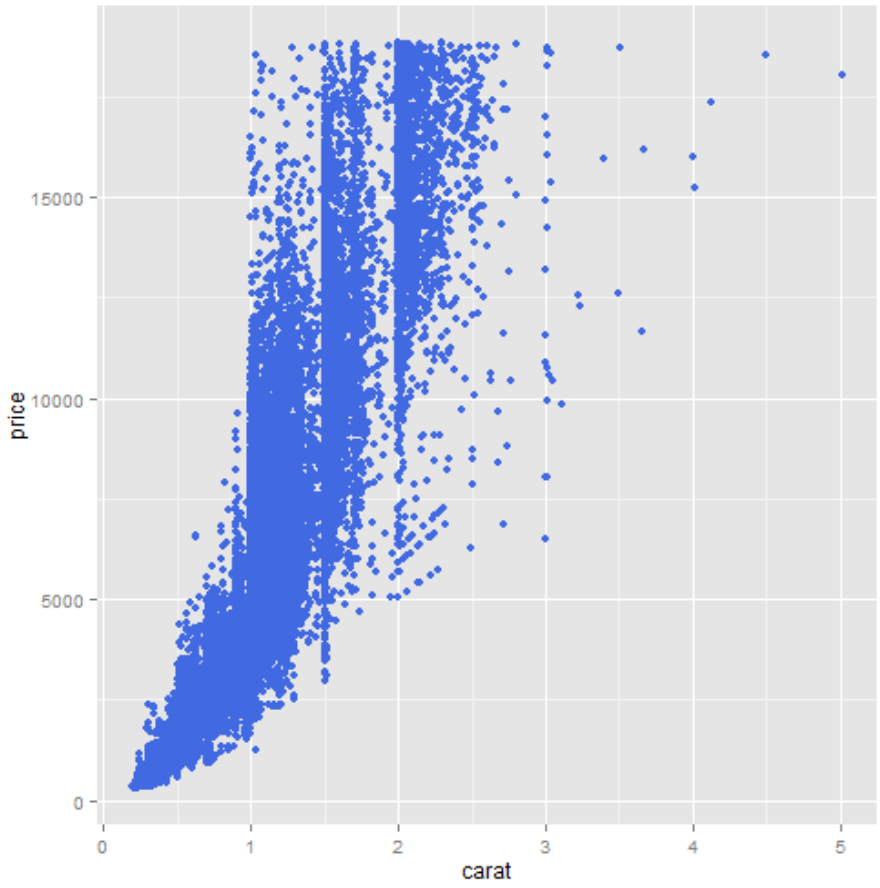


Figure 1: plot of chunk ggplot2_dupes

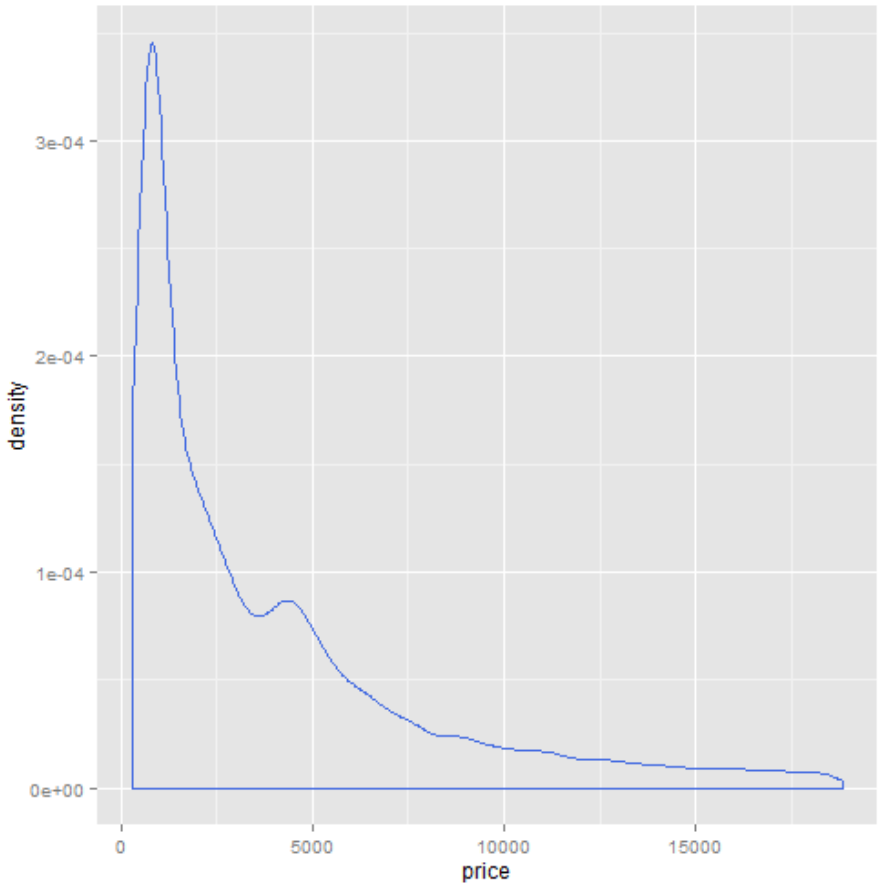


Figure 2: plot of chunk ggplot2_dupes

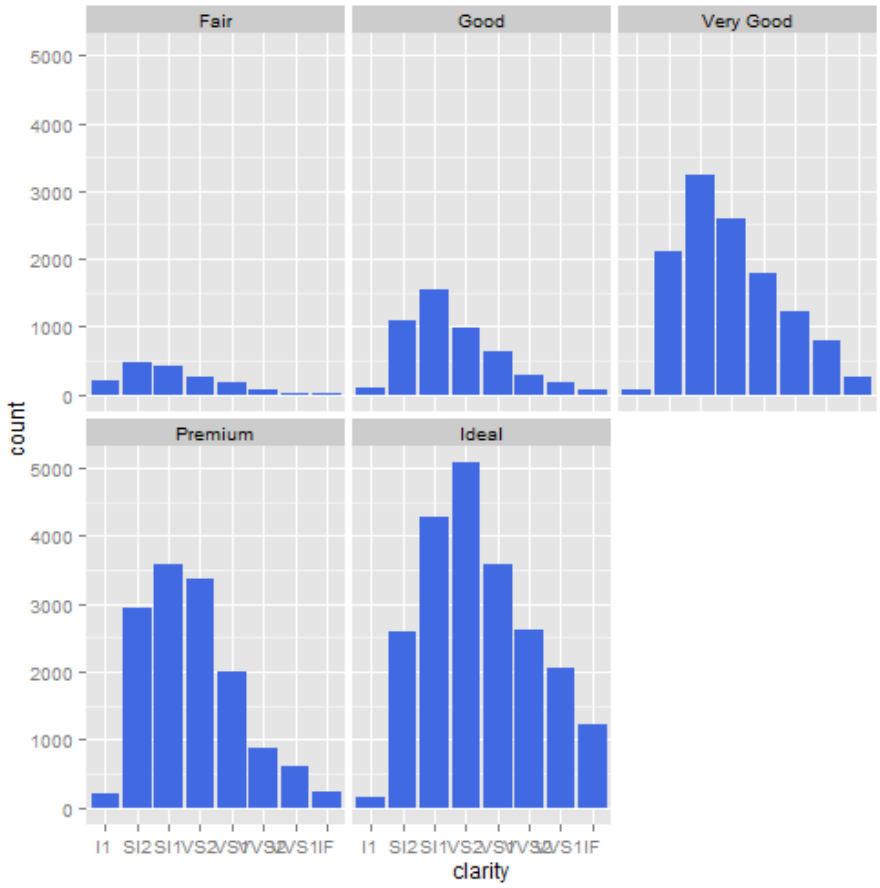


Figure 3: plot of chunk ggplot2_dupes

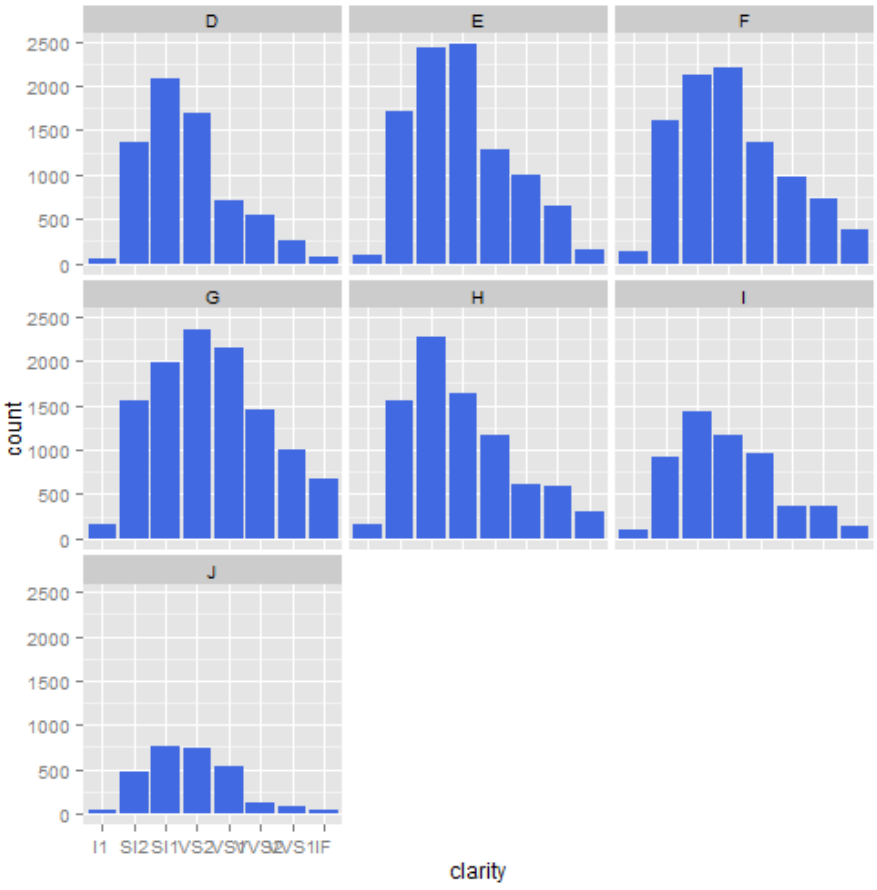


Figure 4: plot of chunk ggplot2_dupes

```
(bar_clarity_by_color <- ggplot(diamonds, aes(clarity)) +
  geom_bar(fill = "royalblue") +
  facet_wrap(~ color)
)
```

As the code is currently written, to change the color, we have to change code in every single plot. What will inevitably happen is that after changing the color in two plots, we'll be phoned by Irene from Accounts who wants something urgently and then we'll forget to change the other plots and things will be inconsistent. Our goal is to have ways of changing things in a single place.

**Idea 1: Use variables rather than hard-coded values**

If we change the code to define the color in a variable, then we only need to change the value in that assignment line.

```
library(ggplot2)

col <- "royalblue"  # <- change this to "limegreen"

(scatter_price_vs_carat <- ggplot(diamonds, aes(carat, price)) +
  geom_point(color = col)
)

(density_cut_vs_price <- ggplot(diamonds, aes(price, color = cut)) +
  geom_density(color = col)
)

(bar_clarity_by_cut <- ggplot(diamonds, aes(clarity)) +
  geom_bar(fill = col) +
  facet_wrap(~ cut)
)

(bar_clarity_by_color <- ggplot(diamonds, aes(clarity)) +
  geom_bar(fill = col) +
  facet_wrap(~ color)
)
```

**Idea 2: For values that you want to change everywhere, update global settings.**

If we have to use a corporate color scheme in every plot, it makes sense to set the default colors to this value. For just four plots, it isn't worth bothering with, but if you have hundreds of plots, then this will save you some time.

```
library(ggplot2)
```

```r
col <- "royalblue"
# Currently need to use Brit spelling for aesthetic names
update_geom_defaults("point", list(colour = col))
update_geom_defaults("density", list(colour = col))
update_geom_defaults("bar", list(fill = col))

# Now we can drop the code specifying the colors
(scatter_price_vs_carat <- ggplot(diamonds, aes(carat, price)) +
  geom_point()
)

(density_cut_vs_price <- ggplot(diamonds, aes(price, color = cut)) +
  geom_density()
)

(bar_clarity_by_cut <- ggplot(diamonds, aes(clarity)) +
  geom_bar() +
  facet_wrap(~ cut)
)

(bar_clarity_by_color <- ggplot(diamonds, aes(clarity)) +
  geom_bar() +
  facet_wrap(~ color)
)
```

**Idea 3: For bigger repetitions, it's better to wrap the contents into a function**

The third and fourth plots share a lot of code, so we can avoid repetition by wrapping it into a function. The code for the last two plots can be simplified to:

```r
barplot_diamond_clarity <- function(facet_var)
{
  facet_formula <- as.formula(paste("~", facet_var))
  ggplot(diamonds, aes(clarity)) +
    geom_bar() +
    facet_wrap(facet_formula)
}

bar_clarity_by_cut <- barplot_diamond_clarity("cut")
bar_clarity_by_color <- barplot_diamond_clarity("color")
```

**Idea 4: For code to be reused across projects, include in a package.**

This is beyond the scope of this example, but if you want to re-use code across projects, then put it in a package.

## Keep It Simple, Stupid

> On a good day, you can keep 7 (give or take 2) things in your working memory

Millers law, paraphrased

Miller's law is a result from cognitive psychology that dates back to the 1950s. Since R is a high-level programming language where one line of code *very* roughly translates into one useful thought, I present the Cotton corollary:

> Most of your R functions should be seven lines or less

To hammer a point home, it's perfectly acceptable for a function to be one line long. These are usually the easiest kind of function to debug.

### Simplifying function interfaces

As well as keeping functions short, the other side of keeping things simple is to make it to call your function. Making functions do more things makes their interface harder to use, and means that you need more tests for them.

The `sig` package has a function to find all the functions in an environment (or package) that are very long or have too many inputs. The `Hmisc` package is a particularly bad offender.

```
library(sig)
sig_report(pkg2env(Hmisc))
```

```
## The environment contains 539 variables of which 536 are functions.
## Distribution of the number of input arguments to the functions:
##   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
##   2  70 117  91  42  46  26  19  16  18  18  10   8   4   7   2   2   5
##  18  19  20  21  22  24  25  26  27  29  30  32  33  35  52  66
##   4   4   3   6   3   2   1   1   2   1   1   1   1   1   1   1
## These functions have more than 10 input args:
##  [1] areg.boot                       aregImpute
##  [3] bpplotM                         bpplt
##  [5] ciapower                        cleanup.import
##  [7] confbar                         cpower
##  [9] csv.get                         curveRep
## [11] datadensity.data.frame          dotchart2
## [13] dotchart3                       Dotplot
## [15] Ecdf.default                    Ecdf.formula
## [17] errbar                          event.chart
```

```
## [19] event.history                   fit.mult.impute
## [21] format.df                       formatCats
## [23] formatCons                      ggplot.summaryP
## [25] histSpike                       histSpikeg
## [27] knitrSet                        labcurve
## [29] latex.default                   latex.describe
## [31] latex.describe.single           latex.responseSummary
## [33] latex.summary.formula.response latex.summary.formula.reverse
## [35] latex.summaryM                  latexDotchart
## [37] panel.bpplot                    panel.Ecdf
## [39] panel.plsmo                     panel.xYplot
## [41] plot.curveRep                   plot.rm.boot
## [43] plot.summary.formula.response  plot.summary.formula.reverse
## [45] plot.summaryM                   plot.summaryP
## [47] plot.summaryS                   plotMultSim
## [49] plsmo                           print.char.list
## [51] print.char.matrix               print.summary.formula.reverse
## [53] print.summaryM                  putKey
## [55] putKeyEmpty                     rcspline.plot
## [57] rcspline.restate                redun
## [59] responseSummary                 rlegend
## [61] rm.boot                         sas.get
## [63] scat1d                          stata.get
## [65] summary.formula                 summaryD
## [67] summaryM                        summaryRc
## [69] transcan                        upData
## [71] xYplot
## Distribution of the number of lines of the functions:
##          1          2      [3,4]      [5,8]     [9,16]     [17,32]
##          0         52         13         61         98         112
##    [33,64]   [65,128]  [129,256]  [257,512] [513,1024]
##         94         64         31         10          1
## These functions have more than 50 lines:
##   [1] areg                           areg.boot
##   [3] aregImpute                     aregTran
##   [5] binconf                        bpplotM
##   [7] bpplt                          bystats
##   [9] bystats2                       ciapower
##  [11] cleanup.import                 cnvrt.coords
##  [13] confbar                        contents.data.frame
##  [15] cpower                         curveRep
##  [17] curveSmooth                    cut2
##  [19] datadensity.data.frame         dataframeReduce
##  [21] dataRep                        describe.vector
##  [23] dotchart2                      dotchart3
##  [25] drawPlot                       Ecdf.data.frame
```

```
##  [27] Ecdf.default                    errbar
##  [29] event.chart                     event.history
##  [31] fit.mult.impute                 format.df
##  [33] formatCats                      formatCons
##  [35] formatTestStats                 Function.areg.boot
##  [37] getRs                           ggplot.summaryP
##  [39] hdquantile                      hist.data.frame
##  [41] histbackback                    histSpike
##  [43] histSpikeg                      hoeffd
##  [45] html.contents.data.frame        improveProb
##  [47] impute.transcan                 inverseFunction
##  [49] invertTabulated                 knitrSet
##  [51] labcurve                        largest.empty
##  [53] latex.default                   latex.describe
##  [55] latex.describe.single           latex.summary.formula.cross
##  [57] latex.summary.formula.response latex.summary.formula.reverse
##  [59] latex.summaryM                  latex.summaryP
##  [61] latexDotchart                   list.tree
##  [63] logrank                         matchCases
##  [65] matxv                           medvPanel
##  [67] na.detail.response              nobsY
##  [69] ordGridFun                      panel.bpplot
##  [71] panel.Dotplot                   panel.Ecdf
##  [73] panel.plsmo                     panel.xYplot
##  [75] plot.areg.boot                  plot.curveRep
##  [77] plot.drawPlot                   plot.rm.boot
##  [79] plot.summary.formula.response  plot.summary.formula.reverse
##  [81] plot.summaryM                   plot.summaryP
##  [83] plot.summaryS                   plotMultSim
##  [85] plsmo                           predict.transcan
##  [87] print.char.list                 print.char.matrix
##  [89] print.contents.data.frame       print.describe.single
##  [91] print.redun                     print.summary.formula.cross
##  [93] print.summary.formula.response print.summary.formula.reverse
##  [95] print.summaryM                  putKey
##  [97] Quantile2                       rcspline.eval
##  [99] rcspline.plot                   rcspline.restate
## [101] redun                           replace.substring.wild
## [103] reShape                         responseSummary
## [105] rlegend                         rm.boot
## [107] sas.get                         sasxport.get
## [109] scat1d                          spearman2.default
## [111] spss.get                        stata.get
## [113] stratify                        strgraphwrap
## [115] summarize                       summary.areg.boot
## [117] summary.formula                 summaryD
```

```
## [119] summaryM                    summaryP
## [121] summaryRc                   summaryS
## [123] symbol.freq                 t.test.cluster
## [125] transcan                    upData
## [127] varclus                     wtd.table
```

Once you know which functions that you need to simplify, how do you go about making them simpler?

There's a rather obvious piece of advice ("You Ain't Gonna Need It") that states that you shouldn't write code for features that you don't need.

> Always implement things when you *actually* need them, never when you just *foresee* that you need them.
>
> c2.com

Assuming that you've only implemented what you need to implement, and it's still a bit tricky, how can you make it easier to use? Let's take a look at a few complicated functions, and see how they manage complexity.

**Idea 1: Pass arguments for advanced functionality to another function**

One technique used reduce the complexity of function signatures is to force advanced arguments to be passed to another function that checks those arguments and returns them as a list. The `grid` package makes heavy use of this technique. We'll use the `sig` package to look at the function signature.

```
library(grid)
sig(pointsGrob)
```

```
## pointsGrob <- function(x = stats::runif(10), y = stats::runif(10), pch =
##           1, size = unit(1, "char"), default.units = "native", name =
##           NULL, gp = gpar(), vp = NULL)
```

In the function signature for many of the functions in `grid`, you see `gp = gpar()`. `gpar` just checks that its inputs are sensible, then returns them as a list:

```
gpar(col = "red", cex = 3)
```

```
## $col
## [1] "red"
##
## $cex
## [1] 3
```

You can use them as follows:

```
pointsGrob(1:10, runif(10), gp = gpar(col = "red", cex = 3))
```

```
## points[GRID.points.896]
```

This allows all the argument checking to be written once (in `gpar`) rather than appearing in every `grob` function.

**Idea 2: Having wrapper functions for specific use cases**

The `read.table` function in the `utils` package is pretty complicated. It needs to cope with a lot of variation in file formats, so it necessarily has a lot of arguments.

```
sig(read.table)
```

```
## read.table <- function(file, header = FALSE, sep = "", quote = ""'", dec =
##           ".", numerals = c("allow.loss", "warn.loss", "no.loss"),
##           row.names, col.names, as.is = !stringsAsFactors, na.strings =
##           "NA", colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
##           fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip
##           = TRUE, comment.char = "#", allowEscapes = FALSE, flush = FALSE,
##           stringsAsFactors = default.stringsAsFactors(), fileEncoding =
##           "", encoding = "unknown", text, skipNul = FALSE)
```

The `utils` package also provides four wrapper functions for the most common cases: `read.csv` and `read.delim` for reading comma and tab delimited files respectively, and their variants `read.csv2` and `read.delim2` which use European-style commas for decimal places.

Each of the wrapper functions provides default arguments suitable for their specific use-case, meaning that the user doesn't have to bother setting them.

```
sig(read.csv)
```

```
## read.csv <- function(file, header = TRUE, sep = ",", quote = """, dec =
##         ".", fill = TRUE, comment.char = "", ...)
```

```
sig(read.delim)
```

```
## read.delim <- function(file, header = TRUE, sep = " ", quote = """, dec =
##           ".", fill = TRUE, comment.char = "", ...)
```

```
sig(read.csv2)
```

```
## read.csv2 <- function(file, header = TRUE, sep = ";", quote = """, dec =
##          ",", fill = TRUE, comment.char = "", ...)
```

```
sig(read.delim2)
```

```
## read.delim2 <- function(file, header = TRUE, sep = " ", quote = """, dec =
##            ",", fill = TRUE, comment.char = "", ...)
```

**Idea 3: Auto-guessing defaults**

`fread` in the `data.table` package is an attempt at improving on 'read.table. While the focus is mainly on the speed of reading files, it is interesting to see how the authors have tried to improve on speed of usage. The function signature isn't much simpler:

```
library(data.table)
sig(fread)
```

```
## fread <- function(input = "", sep = "auto", sep2 = "auto", nrows = -1,
##      header = "auto", na.strings = "NA", stringsAsFactors = FALSE, verbose
##      = getOption("datatable.verbose"), autostart = 30, skip = -1, select =
##      NULL, drop = NULL, colClasses = NULL, integer64 =
##      getOption("datatable.integer64"), showProgress =
##      getOption("datatable.showProgress"), data.table =
##      getOption("datatable.fread.datatable"))
```

`fread` does however make its usage easier (and hence faster) than `read.table`. It's trick is to do a lot of automated guesswork as to what arguments to use. For example, rather than having to look in your file and see what the separator is, it reads a few lines and chooses a value that seems sensible. So you almost never need to use the `sep` argument. Similarly, there is smarter guessing of column class types and the number of rows.

**Idea 3: Split functionality into many functions**

Unix has a well-known philosophical standpoint that programs should

> Do One Thing and Do It Well

Doug McIlroy, 1978

In the `plyr` package, `ddply` is very powerful, but the interface takes a while to learn. This next example, gets the mean weight of chickens by feed group.

```
library(plyr)
ddply(chickwts, .(feed), summarize, MeanWeight = mean(weight))
```

```
##        feed MeanWeight
## 1    casein   323.5833
## 2 horsebean   160.2000
## 3   linseed   218.7500
## 4  meatmeal   276.9091
## 5   soybean   246.4286
## 6 sunflower   328.9167
```

This is already quite complex, but `ddply` also allows you to do other tasks like adding additional columns to the data, or calculating summary stats on multiple columns simultaneously. The `dplyr` approach, which (mostly) replaces `plyr`, is to split the functionality into several different functions. In this next example, `%>%` is the pipe operator, which passes the result of an expression into the first argument of the next function, allowing you to join multiple commands together.

```
library(dplyr)
chickwts %>%
  group_by(feed) %>%
  summarize(MeanWeight = mean(weight))
```

```
## Source: local data frame [6 x 2]
##
##        feed MeanWeight
## 1    casein   323.5833
## 2 horsebean   160.2000
## 3   linseed   218.7500
## 4  meatmeal   276.9091
## 5   soybean   246.4286
## 6 sunflower   328.9167
```

### Cyclomatic complexity

> The cyclomatic complexity number (CCN) is a measure of how many paths there are through a method. It serves as a rough measure of code complexity and as a count of the minimum number of test cases that are required to achieve full code-coverage of the method.

rkcole.com

Or,

> Cyclomatic complexity measures the number of linearly independent paths through the method, which is determined by the number and complexity of conditional branches. A low cyclomatic complexity generally indicates a method that is easy to understand, test, and maintain.

microsoft.com

As well as keeping functions short, another good idea is to reduce the number of possible paths through your code. Whenever you include `if` or `switch` statements, or loops, then you increase the number of possible ways that the function can run, and you make it harder to reason about what your code is doing.

The is a measure of the number of paths through your code called cyclomatic complexity, though it isn't well defined for R code. In the simplest case, we can write a function where there is only one path to take.

```
cyclo_single_path <- function()
{
  message("Hello World!")
}
```

Let's increase the complexity by including an `if` statement.

```
cyclo_if <- function(condition)
{
  if(condition)
  {
    message("Hello World!")
  }
}
```

In most programming languages, this has a cyclomatic complexity of two: you can pass either `TRUE` or `FALSE` to the condition, and it will change the behaviour. However in R, you can also pass `NA`, which throws an error, so the cyclomatic complexity is arguably three. (You can also pass numbers or strings or logical vectors with length greater than one, but these cases always resolve back to one of the three cases of `TRUE`/`FALSE`/`NA`.)

`switch` statements have a cyclomatic complexity of the number of choices, plus one for the default case (in this case when the data is `NA`), and arguably another one for the case of errors when you have bad inputs. The following example has a cyclomatic complexity of eight or nine.

```
cyclo_switch <- function(date)
{
  switch(
    weekdays(date),
    Monday    = message("The day is Monday!"),
    Tuesday   = message("The day is Tuesday!"),
    Wednesday = message("The day is Wednesday!"),
```

```
    Thursday  = message("The day is Thursday!"),
    Friday    = message("The day is Friday!"),
    Saturday  = message("The day is Saturday!"),
    Sunday    = message("The day is Sunday!"),
    message("The input date is missing")
  )
}
```

How to reduce cyclomatic complexity:

1. Due to R's troolean logic, dynamic typing, and lack of scalar types, there are a lot of "bad" cases, so it is good practise to check that you genuinely have a scalar logical value to pass to an `if` statement, as early as possible in your functions.
2. Nested `if` statements of loops quickly increase the cyclomatic complexity of the code. Try to avoid writing that sort of code.
3. Early returns for edge cases can be very useful; a typical code pattern looks like this:

   ```
   my_function <- function(x)
   {
     if(is.null(x))
     {
       warning("x is NULL, returning 0")
       return(0)
     }
     # Code for usual non-null case of x
   }
   ```

4. Refactoring the function into smaller functions reduces complexity until it doesn't.

## Fail Early, Fail Often

In order to make sure that your code gets the right answer, you need to have error-handling code that notifies you if something has gone wrong. You may wonder, where to put this error handling code. It turns out that there's a very well established answer:

> Make each module fail fast – either it does the right thing or it stops.

Jim Gray, 1985

As soon as you can detect that there is a problem, you should correct that problem or throw an error (fail). That's the "fail early" (or "fail fast") part.

The "fail often" part means that you need lots of checks to ensure the integrity of your code. This is especially true of R code: the flexibility of the language gives you a lot of scope for things going wrong.

### Writing good error messages

I'm sure you've had the problem of a trying to figure out why a really obscure error message is occurring. In R, most code involving non-standard evaluation is good for generating bad error messages:

```
library(ggplot2)
ggplot(cars, aes(spped, dist)) + geom_point()
```

```
## Error in eval(expr, envir, enclos): object 'spped' not found
```

This is hardly the worst offender, but it's still a long way from the explaining the real cause of the problem: "spped" is not one of the columns in the `cars` dataset; it should have read "speed".

If we apply our "fail early" principle, when is the earliest that we could handle this potential error?

We know which columns are in the dataset (and hence which columns can be used for aesthetics) when `ggplot` is called, so the check should come quite soon after that function is called.

What would this check look like? Well, we want to take each of the variables passed as a mapping (inside the `aes` function), convert them to strings, and check that they are all column names of the data frame. If that isn't true, we want to give an error message that explains the problem in a way that tells the person reading it exactly what the problem is using a real sentence.

```
check_aesthetics_are_in_data <- function(mapping, data)
{
  aesthetics <- sapply(mapping, deparse)
  are_the_aesthetics_present <- aesthetics %in% colnames(data)
  if(!all(are_the_aesthetics_present))
  {
    stop(
      "The aesthetics ",
      toString(sQuote(aesthetics[!are_the_aesthetics_present])),
      " are not columns in the dataset ",
      deparse(substitute(data)),
```

```
      "."
    )
  }
}
check_aesthetics_are_in_data(aes(spped, dist), cars)

## Error in check_aesthetics_are_in_data(aes(spped, dist), cars): The aesthetics 'spped' are
```

Of course, nobody has time to write all this sort of boilerplate code for every function, especially not enough times to fulfil the "fail often" part of the phrase.

That's why the `assertive` package exists: to provide pre-canned checks for the few hundred most common error types.