Comparing Non-Identical Objects

Introducing the 'compare' package

by Paul Murrell

The **compare** package provides functions for comparing two R objects for equality, while allowing for a range of "minor" differences. Objects may be reordered, rounded, or resized, they may have names or attributes removed, or they may even be coerced to a new class if necessary in order to achieve equality.

The results of comparisons report not just whether the objects are the same, but also include a record of any modifications that were performed.

This package was developed for the purpose of partially automating the marking of coursework involving R code submissions, so functions are also provided to convert the results of comparisons into numeric grades and to provide feedback for students.

Motivation

STATS 220 is a second year university course run by the Department of Statistics at the University of Auckland.[1] The course covers a range of "Data Technologies", including HTML, XML, databases, SQL, and, as a general purpose data processing tool, R.

In addition to larger assignments, students in the course must complete short exercises in weekly computer labs.

For the R section of the course, students must write short pieces of R code to produce specific R objects. Figure 1 shows two examples of basic, introductory exercises.

The students submit their answers to the exercises as a file containing R code, which means that it is possible to recreate their answers by calling `source()` on the submitted files.

At this point, the R objects generated by the students' code can be compared with a set of model R objects in order to establish whether the students' answers are correct.

How this comparison occurs is the focus of this article.

Black and white comparisons

The simplest and most strict test for equality between two objects in the base R system (R Development Core Team, 2008) is provided by the function `identical()`. This returns `TRUE` if the two objects are *exactly* the same, otherwise it returns `FALSE`.

The problem with this function is that it is very strict indeed and will fail for objects that are, for all practical purposes, the same. The classic example is the comparison of two real (floating-point) values, as demonstrated in the following code, where differences can arise simply due to the limitations of how numbers are represented in computer memory (see R FAQ 7.31, Hornik, 2008).

```
> identical(0.3 - 0.2, 0.1)

[1] FALSE
```

Using the function to test for equality would clearly be unreasonably harsh when marking any student answer that involves calculating a numeric result.

The `identical()` function, by itself, is not sufficient for comparing student answers with model answers.

Shades of grey

The recommended solution to the problem mentioned above of comparing two floating-point values is to use the `all.equal()` function. This function allows for "insignificant" differences between numeric values, as shown below.

```
> all.equal(0.3 - 0.2, 0.1)

[1] TRUE
```

This makes `all.equal()` a much more appropriate function for comparing student answers with model answers.

What is less well-known about the `all.equal()` function is that it also works for comparing other sorts of R objects, besides numeric vectors, *and* that it does more than just report equality between two objects.

If the objects being compared have differences, then `all.equal()` does not simply return `FALSE`. Instead, it returns a character vector containing messages that describe the differences between the objects. The following code gives a simple example, where `all.equal()` reports that the two character vectors have different lengths, and that, of the two pairs of strings that can be compared, one pair of strings does not match.

```
> all.equal(c("a", "b", "c"), c("a", "B"))

[1] "Lengths (3, 2) differ (string compare on first 2)"
[2] "1 string mismatch"
```

This feature is actually very useful for marking student work. Information about whether a student's answer is correct is useful for determining a raw mark, but it is also useful to have information about what the student did wrong. This information can be used as the basis for assigning partial marks

[1] `http://www.stat.auckland.ac.nz/courses/stage2/#STATS220`

1. Write R code to create the three **vectors** and the **factor** shown below, with names `id`, `age`, `edu`, and `class`.

   You should end up with objects that look like this:

```
> id
[1] 1 2 3 4 5 6

> age
[1] 30 32 28 39 20 25

> edu
[1] 0 0 0 0 0 0

> class
[1] poor   poor   poor   middle
[5] middle middle
Levels: middle poor
```

2. Combine the objects from Question 1 together to make a **data frame** called `IndianMothers`.

   You should end up with an object that looks like this:

```
> IndianMothers
  id age edu  class
1  1  30   0   poor
2  2  32   0   poor
3  3  28   0   poor
4  4  39   0 middle
5  5  20   0 middle
6  6  25   0 middle
```

Figure 1: Two simple examples of the exercises that STATS 220 students are asked to perform.

for an answer that is close to the correct answer, and for providing feedback to the student about where marks were lost.

The `all.equal()` function has some useful features that make it a helpful tool for comparing student answers with model answers. However, there is an approach that can perform better than this.

The `all.equal()` function looks for equality between two objects and, if that fails, provides information about the sort of differences that exist. An alternative approach, when two objects are not equal, is to try to *transform* the objects to make them equal, and report on which transformations were necessary in order to achieve equality.

As an example of the difference between these approaches, consider the two objects below: a character vector and a factor.

```
> obj1 <- c("a", "a", "b", "c")
> obj1

[1] "a" "a" "b" "c"

> obj2 <- factor(obj1)
> obj2

[1] a a b c
Levels: a b c
```

The `all.equal()` function reports that these objects are different because they differ in terms of their fundamental mode—one has attributes and the other does not—and because each object is of a different class.

```
> all.equal(obj1, obj2)
[1] "Modes: character, numeric"
[2] "Attributes: < target is NULL, current is list >"
[3] "target is character, current is factor"
```

The alternative approach would be to allow various transformations of the objects to see if they can be transformed to be the same. The following code shows this approach, which reports that the objects are equal, if the second one is coerced from a factor to a character vector. This is more information than was provided by `all.equal()` and, in the particular case of comparing student answers to model answers, it tells us a lot about how close the student got to the right answer.

```
> library(compare)
> compare(obj1, obj2, allowAll=TRUE)

TRUE
  coerced from <factor> to <character>
```

Another limitation of `all.equal()` is that it does not report on some other possible differences between objects. For example, it is possible for a student to have the correct values for an R object, but have the values in the wrong order. Another common mistake is to get the case wrong in a set of string values (e.g., in a character vector or in the `names` attribute of an object).

In summary, while `all.equal()` provides some desirable features for comparing student answers to model answers, we can do better by allowing for a wider range of differences between objects and by taking a different approach that attempts to transform the student answer to be the same as the model answer, if at all possible, while reporting which transformations were necessary.

The remainder of this article describes the **compare** package, which provides functions for producing these sorts of comparisons.

The `compare()` function

The main function in the **compare** package is the `compare()` function. This function checks whether two objects are the same and, if they are not, carries out various transformations on the objects and checks them again to see if they are the same after they have been transformed.

By default, `compare()` only succeeds if the two objects are identical (using the `identical()` function) *or* the two objects are numeric and they are equal (according to `all.equal()`). If the objects are not the same, no transformations of the objects are considered. In other words, by default, `compare()` is simply a convenience wrapper for `identical()` and `all.equal()`. As a simple example, the following comparison takes account of the fact that the values being compared are numeric and uses `all.equal()` rather than `identical()`.

```
> compare(0.3 - 0.2, 0.1)

TRUE
```

Transformations

The more interesting uses of `compare()` involve specifying one or more of the arguments that allow transformations of the objects that are being compared. For example, the `coerce` argument specifies that the second argument may be coerced to the class of the first argument. This allows for more flexible comparisons such as between a factor and a character vector.

```
> compare(obj1, obj2, coerce=TRUE)

TRUE
  coerced from <factor> to <character>
```

It is important to note that there is a definite order to the objects; the *model* object is given first and the *comparison* object is given second. Transformations attempt to make the comparison object like the model object, though in a number of cases (e.g., when ignoring the case of strings) the model object may also be transformed. In the example above, the comparison object has been coerced to be the same class as the model object. The following code demonstrates the effect of reversing the order of the objects in the comparison. Now the character vector is being coerced to a factor.

```
> compare(obj2, obj1, coerce=TRUE)

TRUE
  coerced from <character> to <factor>
```

Of course, transforming an object is not guaranteed to produce identical objects if the original objects are genuinely different.

```
> compare(obj1, obj2[1:3], coerce=TRUE)

FALSE
  coerced from <factor> to <character>
```

Notice, however, that even though the comparison failed, the result still reports the transformation that was attempted. This result indicates that the comparison object was converted from a factor (to a character vector), but it *still* did not end up being the same as the model object.

A number of other transformations are available in addition to coercion. For example, differences in length, like in the last case, can also be ignored.

```
> compare(obj1, obj2[1:3],
+         shorten=TRUE, coerce=TRUE)

TRUE
  coerced from <factor> to <character>
  shortened model
```

It is also possible to allow values to be sorted, or rounded, or to convert all character values to upper case (i.e., ignore the case of strings).

Table 1 provides a complete list of the transformations that are currently allowed (in version 0.2 of **compare**) and the arguments that are used to enable them.

A further argument to the `compare()` function, `allowAll`, controls the default setting for most of these transformations, so specifying `allowAll=TRUE` is a quick way of enabling all possible transformations. Specific transformations can still be *excluded* by explicitly setting the appropriate argument to `FALSE`.

The `equal` argument is a bit of a special case because it is `TRUE` by default, whereas almost all others are `FALSE`. The `equal` argument is also especially influential because objects are compared after every transformation and this argument controls what sort of comparison takes place. Objects are always compared using `identical()` first, which will only succeed if the objects have exactly the same representation in memory. If the test using `identical()` fails and `equal=TRUE`, then a more lenient comparison is also performed. By default, this just means that numeric values are compared using `all.equal()`, but various other arguments can extend this to allow things like differences in case for character values (see the asterisked arguments in Table 1).

The `round` argument is also special because it always defaults to `FALSE`, even if `allowAll=TRUE`. This means that the `round` argument must be specified explicitly in order to enable rounding. The default is set up this way because the value of the `round` argument is either `FALSE` or an integer value specifying the number of decimal places to round to. For this

Table 1: Arguments to the `compare()` function that control which transformations are attempted when comparing a model object to a comparison object.

Argument	Meaning
`equal`	Compare objects for "equality" as well as "identity" (e.g., use `all.equal()` if model object is numeric).
`coerce`	Allow coercion of comparison object to class of model object.
`shorten`	Allow either the model or the comparison to be shrunk so that the objects have the same "size".
`ignoreOrder`	Ignore the original order of the comparison and model objects; allow both comparison object and model object to be sorted.
`ignoreNameCase`	Ignore the case of the `names` attribute for both comparison and model objects; the `name` attributes for both objects are converted to upper case.
`ignoreNames`	Ignore any differences in the `names` attributes of the comparison and model objects; any `names` attributes are dropped.
`ignoreAttrs`	Ignore all attributes of both the comparison and model objects; all attributes are dropped.
`round`*	Allow numeric values to be rounded; either `FALSE` (the default), or an integer value giving the number of decimal places for rounding, or a function of one argument, e.g., `floor`.
`ignoreCase`*	Ignore the case of character vectors; both comparison and model are converted to upper case.
`trim`*	Ignore leading and trailing spaces in character vectors; leading and trailing spaces are trimmed from both comparison and model.
`ignoreLevelOrder`*	Ignore original order of levels of factor objects; the levels of the comparison object are sorted to the order of the levels of the model object.
`dropLevels`*	Ignore any unused levels in factors; unused levels are dropped from both comparison and model objects.
`ignoreDimOrder`	Ignore the order of dimensions in array, matrix, or table objects; the dimensions are reordered by name.
`ignoreColOrder`	Ignore the order of columns in data frame objects; the columns in the comparison object are reordered to match the model object.
`ignoreComponentOrder`	Ignore the order of components in a list object; the components are reordered by name.

*These transformations only occur if `equal=TRUE`

argument, the value TRUE corresponds to rounding to zero decimal places.

Finally, there is an additional argument colsOnly for comparing data frames. This argument controls whether transformations are only applied to columns (and not to rows). For example, by default, a data frame will only allow columns to be dropped, but not rows, if shorten=TRUE. Note, however, that ignoreOrder means ignore the order of *rows* for data frames and ignoreColOrder must be used to ignore the order of columns in comparisons involving data frames.

The compareName() function

The compareName() function offers a slight variation on the compare() function.

For this function, only the *name* of the comparison object is specified, rather than an explicit object. The advantage of this is that it allows for variations in case in the names of objects. For example, a student might create a variable called indianMothers rather than the desired IndianMothers. This case-insensitivity is enabled via the ignore.case argument.

Another advantage of this function is that it is possible to specify, via the compEnv argument, a particular environment to search within for the comparison object (rather than just the current workspace). This becomes useful when checking the answers from several students because each student's answers may be generated within a separate environment in order to avoid any interactions between code from different students.

The following code shows a simple demonstration of this function, where a comparison object is created within a temporary environment and the name of the comparison object is upper case when it should be lowercase.

```
> tempEnv <- new.env()
> with(tempEnv, X <- 1:10)
> compareName(1:10, "x", compEnv=tempEnv)
```

```
TRUE
  renamed object
```

Notice that, as with the transformations in compare(), the compareName() function records whether it needed to ignore the case of the name of the comparison object.

A pathological example

This section shows a manufactured example that demonstrates some of the flexibility of the compare() function.

We will compare two data frames that have a number of simple differences. The model object is a data frame with three columns: a numeric vector, a character vector, and a factor.

```
> model <-
+     data.frame(x=1:26,
+                y=letters,
+                z=factor(letters),
+                row.names=letters,
+                stringsAsFactors=FALSE)
```

The comparison object contains essentially the same information, except that there is an extra column, the column names are uppercase rather than lowercase, the columns are in a different order, the y variable is a factor rather than a character vector, and the z variable is a character variable rather than a factor. The y variable and the row names are also uppercase rather than lowercase.

```
> comparison <-
+     data.frame(W=26:1,
+                Z=letters,
+                Y=factor(LETTERS),
+                X=1:26,
+                row.names=LETTERS,
+                stringsAsFactors=FALSE)
```

The compare() function can detect that these two objects are essentially the same as long as we reorder the columns (ignoring the case of the column names), coerce the y and z variables, drop the extra variable, ignore the case of the y variable, and ignore the case of the row names.

```
> compare(model, comparison, allowAll=TRUE)
```

```
TRUE
  renamed
  reordered columns
  [Y] coerced from <factor> to <character>
  [Z] coerced from <character> to <factor>
  shortened comparison
  [Y] ignored case
  renamed rows
```

Notice that we have used allowAll=TRUE to allow compare() to attempt all possible transformations at its disposal.

Comparing files of R code

Returning now to the original motivation for the **compare** package, the compare() function provides an excellent basis for determining not only whether a student's answers are correct, but also how much incorrect answers differ from the model answer.

As described earlier, submissions by students in the STATS 220 course consist of files of R code. Marking these submissions consists of using source() to run the code, then comparing the resulting objects with model answer objects. With approximately 100 students in the STATS 220 course, with weekly labs, and with multiple questions per lab, each of which

may contain more than one R object, there is a reasonable marking burden. Consequently, there is a strong incentive to automate as much of the marking process as possible.

The `compareFile()` function

The `compareFile()` function can be used to run R code from a specific file and compare the results with a set of model answers. This function requires three pieces of information: the name of a file containing the "comparison code", which is run within a local environment, using `source()`, to generate the comparison values; a vector of "model names", which are the names of the objects that will be looked for in the local environment after the comparison code has been run; and the model answers, either as the name of a binary file to `load()`, or as the name of a file of R code to `source()`, or as a list object containing the ready-made model answer objects.

Any argument to `compare()` may also be included in the call.

Once the comparison code has been run, `compareName()` is called for each of the model names and the result is a list of `"comparison"` objects.

As a simple demonstration, consider the basic questions shown in Figure 1. The model names in this case are the following:

```
> modelNames <- c("id", "age",
+                 "edu", "class",
+                 "IndianMothers")
```

One student's submission for this exercise is in a file called `student1.R`, within a directory called `Examples`. The model answer is in a file called `model.R` in the same directory. We can evaluate this student's submission and compare it to the model answer with the following code:

```
> compareFile(file.path("Examples",
+                       "student1.R"),
+             modelNames,
+             file.path("Examples",
+                       "model.R"))

$id
TRUE

$age
TRUE

$edu
TRUE

$class
FALSE

$IndianMothers
FALSE
  object not found
```

This provides a strict check and shows that the student got the first three problems correct, but the last two wrong. In fact, the student's code completely failed to generate an object with the name `IndianMothers`.

We can provide extra arguments to allow transformations of the student's answers, as in the following code:

```
> compareFile(file.path("Examples",
+                       "student1.R"),
+             modelNames,
+             file.path("Examples",
+                       "model.R"),
+             allowAll=TRUE)

$id
TRUE

$age
TRUE

$edu
TRUE

$class
TRUE
  reordered levels

$IndianMothers
FALSE
  object not found
```

This shows that, although the student's answer for the `class` object was not perfect, it was pretty close; it just had the levels of the factor in the wrong order.

The `compareFiles()` function

The `compareFiles()` function builds on `compareFile()` by allowing a vector of comparison file names. This allows a whole set of student submissions to be tested at once. The result of this function is a list of lists of `"comparison"` objects and a special print method provides a simplified view of this result.

Continuing the example from above, the `Examples` directory contains submissions from a further four students. We can compare all of these submissions with the model answers and produce a summary of the results with a single call to `compareFiles()`. The appropriate code and output are shown in Figure 2.

The results show that most students got the first three problems correct. They had more trouble getting the fourth problem right, with one getting the factor levels in the wrong order and two others producing a character vector rather than a factor. Only one student, `student2`, got the final problem exactly right and only one other, `student4`, got essentially

```
> files <- list.files("Examples",
+                     pattern="^student[0-9]+[.]R$",
+                     full.names=TRUE)
> results <- compareFiles(files,
+                         modelNames,
+                         file.path("Examples", "model.R"),
+                         allowAll=TRUE,
+                         resultNames=gsub("Examples.|[.]R", "", files))
> results
```

```
         id    age   edu   class                                      IndianMothers
student1 TRUE  TRUE  TRUE  TRUE reordered levels                     FALSE object not found
student2 TRUE  TRUE  TRUE  TRUE                                       TRUE
student3 TRUE  TRUE  TRUE  TRUE coerced from <character> to <factor> FALSE object not found
student4 TRUE  TRUE  TRUE  TRUE coerced from <character> to <factor> TRUE renamed object
student5 TRUE  TRUE  TRUE  FALSE object not found                     FALSE object not found
```

Figure 2: Using the `compareFiles()` function to run R code from several files and compare the results to model objects. The result of this sort of comparison can easily get quite wide, so it is often useful to print the result with `options(width)` set to some large value and using a small font, as has been done here.

the right answer, though this student spelt the name of the object wrong.

Assigning marks and giving feedback

The result returned by `compareFiles()` is a list of lists of comparison results, where each result is itself a list of information including whether two objects are the same and a record of how the objects were transformed during the comparison. This represents a wealth of information with which to assess the performance of students on a set of R exercises, but it can be a little unwieldly to deal with.

The **compare** package provides further functions that make it easier to deal with this information for the purpose of determining a final mark and for the purpose of providing comments for each student submission.

In order to determine a final mark, we use the `questionMarks()` function to specify which object names are involved in a particular question, to provide a maximum mark for the question, and to specify a set of rules that determine how many marks should be deducted for various deviations from the correct answers.

The `rule()` function is used to define a marking rule. It takes an object name, a number of marks to deduct if the comparison for that object is `FALSE`, plus any number of transformation rules. The latter are generated using the `transformRule()` function, which associates a regular expression with a number of marks to deduct. If the regular expression is matched in the record of transformations for a comparison, then the appropriate number of marks are deducted.

A simple example, based on the second question in Figure 1, is shown below. This specifies that the question only involves an object named `IndianMothers`, that there is a maximum mark of 1 for this question, and that 1 mark is deducted if the comparison is `FALSE`.

```
> q2 <-
+     questionMarks("IndianMothers",
+                   maxMark=1,
+                   rule("IndianMothers", 1))
```

The first question from Figure 1 provides a more complex example. In this case, there are four different objects involved and the maximum mark is 2. The rules below specify that any `FALSE` comparison drops a mark *and* that, for the comparison involving the object named `"class"`, a mark should also be deducted if coercion was necessary to get a `TRUE` result.

```
> q1 <-
+     questionMarks(
+         c("id", "age", "edu", "class"),
+         maxMark=2,
+         rule("id", 1),
+         rule("age", 1),
+         rule("edu", 1),
+         rule("class", 1,
+              transformRule("coerced", 1)))
```

Having set up this marking scheme, marks are generated using the `markQuestions()` function, as shown by the following code.

```
> markQuestions(results, q1, q2)
```

```
         id-age-edu-class IndianMothers
student1                2             0
student2                2             1
```

```
student3                1             0
student4                1             1
student5                1             0
```

For the first question, the third and fourth students lose a mark because of the coercion, and the fifth student loses a mark because he has not generated the required object.

A similar suite of functions are provided to associate comments, rather than mark deductions, with particular transformations. The following code provides a simple demonstration.

```
> q1comments <-
+     questionComments(
+         c("id", "age", "edu", "class"),
+         comments(
+             "class",
+             transformComment(
+                 "coerced",
+                 "'class' is a factor!")))
> commentQuestions(results, q1comments)

         id-age-edu-class
student1 ""
student2 ""
student3 "'class' is a factor!"
student4 "'class' is a factor!"
student5 ""
```

In this case, we have just generated feedback for the students who generated a character vector instead of the desired factor in Question 1 of the exercise.

Summary, discussion, and future directions

The **compare** package is based around the `compare()` function, which compares two objects for equality and, if they are not equal, attempts to transform the objects to make them equal. It reports whether the comparison succeeded overall and provides a record of the transformations that were attempted during the comparison.

Further functions are provided on top of the `compare()` function to facilitate marking exercises where students in a class submit R code in a file to create a set of R objects.

This article has given some basic demonstrations of the use of the **compare()** package for comparing objects and marking student submissions. The package could also be useful for the students themselves, both to check whether they have the correct answer and to provide feedback about how their answer differs from the model answer. More generally, the `compare()` function may have application wherever the `identical()` and `all.equal()` functions are currently in use. For example, it may be useful when debugging code and for performing regression tests as part of a quality control process.

Obvious extensions of the **compare** package include adding new transformations and providing comparison methods for other classes of objects. More details about how the package works and how these extensions might be developed are discussed in the vignette, "Fundamentals of the Compare Package", which is installed as part of the **compare** package.

Acknowledgements

Many thanks to the editors and anonymous reviewers for their useful comments and suggestions, on both this article and the **compare** package itself.

Bibliography

K. Hornik. The R FAQ, 2008. URL `http://CRAN.R-project.org/doc/FAQ/R-FAQ.html`. ISBN 3-900051-08-9.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. URL `http://www.R-project.org`. ISBN 3-900051-07-0.