Regular expressions and reshaping using data tables and the nc package

by Toby Dylan Hocking

Abstract Regular expressions are powerful tools for extracting tables from non-tabular text data. Capturing regular expressions that describe information to extract from column names can be especially useful when reshaping a data table from wide (one row with many columns) to tall (one column with many rows). We present the R package **nc**, which provides functions for data reshaping, regular expressions, and a uniform interface to three C libraries (PCRE, RE2, ICU). We describe the main features of **nc**, then provide detailed comparisons with related R packages (**stats**, **utils**, **data.table**, **tidyr**, **reshape2**, **cdata**).

Introduction

Regular expressions are powerful tools for text processing that are available in many programming languages, including R. A regular expression *pattern* defines a set of *matches* in a *subject* string. For example, the pattern `.*[.].*` matches zero or more non-newline characters, followed by a period, followed by zero or more non-newline characters. It would match the subjects `Sepal.Length` and `Petal.Width`, but it would not match in the subject `Species`.

The focus of this article is patterns with capture groups, which are typically defined using parentheses. For example, the pattern `(.*)[.](.*)` results in the same matches as the pattern in the previous paragraph, and it additionally allows the user to capture and extract the substrings by group index (e.g. group 1 matches `Sepal`, group 2 matches `Length`).

Named capture groups allow extracting the a substring by name rather than by index. Using names rather than indices is useful in order to create more readable regular expressions (names document the purpose of each sub-pattern), and to create more readable R code (it is easier to understand the intent of named references than numbered references). For example, the pattern `(?<part>.*)[.](?<dimension>.*)` documents that the flower part appears before the measurement dimension; the `part` group matches `Sepal` and the `dimension` group matches `Length`.

Recently, Hocking (2019a) proposed a new syntax for defining named capture groups in R code. Using this new syntax, named capture groups are specified using named arguments in R, which results in code that is easier to read and modify than capture groups defined in string literals. For example, the pattern in the previous paragraph can be written as `part=".*", "[.]", dimension=".*"`. Sub-patterns can be grouped for clarity and/or re-used using lists, and numeric data may be extracted by specifying group-specific type conversion functions.

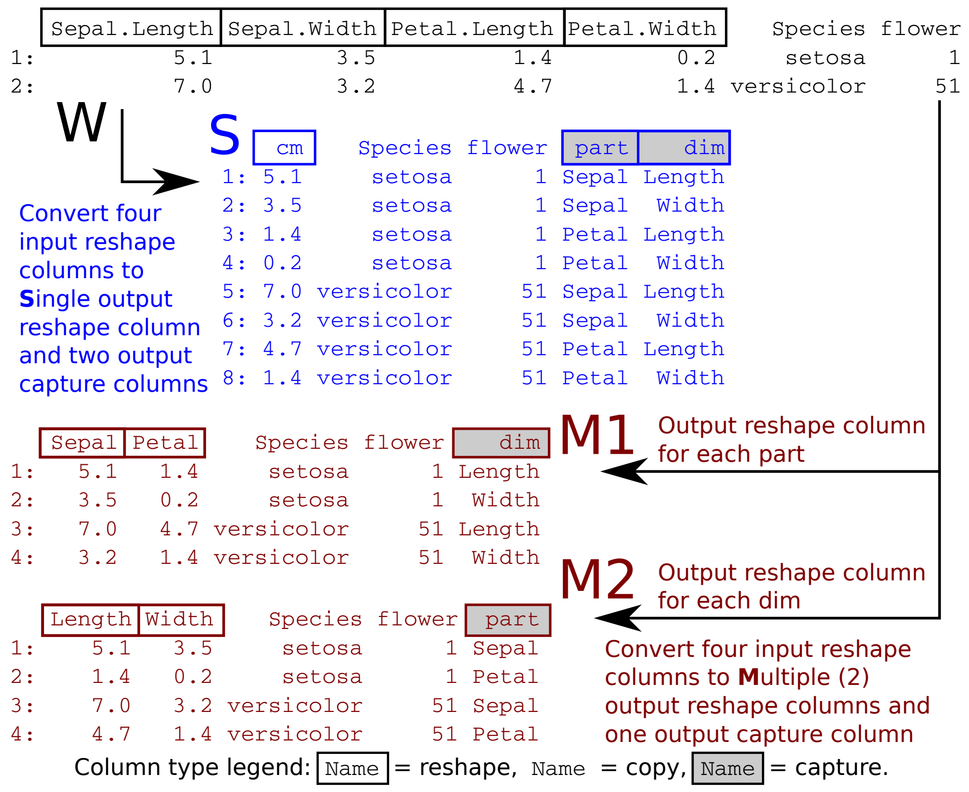
A main thesis of this article is that regular expressions can greatly simplify the code required to specify wide-to-tall data reshaping operations. For one such operation the input is a "wide" table with many columns, and the desired output is a "tall" table with more rows, and some of the input columns converted into a smaller number of output columns (Figure 1). To clarify the discussion we first define three terms that we will use to refer to the different types of columns involved in this conversion:

Reshape columns contain the data which is present in the same amount but in different shapes in the input and output. There are equivalent terms used in different R packages: `varying` in `utils::reshape`, `measure.vars` in `melt` (**data.table**, **reshape2**), etc.

Copy columns contain data in the input which are each copied to multiple rows in the output (`id.vars` in `melt`).

Capture columns are only present in the output, and contain data which come from matching a capturing regex pattern to the input reshape column names.

For example the wide iris data (W in Figure 1) have four numeric columns to reshape: `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`. For some purposes (e.g. displaying a histogram of each reshape input column using facets in **ggplot2**) the desired reshaping operation results in a table with a single reshape output column (S in Figure 1), two copied columns, and two columns captured from the names of the reshaped input columns. For other purposes (e.g. scatterplot to compare Petal and Sepal sizes) the desired reshaping operation results in a table with multiple reshape output columns (M1 with `Sepal` and `Petal` columns in Figure 1), two copied columns, and one column captured from the names of the reshaped input columns. We propose to use the new regular expression syntax of Hocking (2019a), e.g. `part=".*", "[.]", dimension=".*"`, to define both types of wide-to-tall data reshaping operations. In particular, we propose using a single capturing regular expression for



```
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species flower
1:          5.1         3.5          1.4         0.2     setosa      1
2:          7.0         3.2          4.7         1.4 versicolor     51
```

```
    cm    Species flower  part    dim
1: 5.1     setosa      1 Sepal Length
2: 3.5     setosa      1 Sepal  Width
3: 1.4     setosa      1 Petal Length
4: 0.2     setosa      1 Petal  Width
5: 7.0 versicolor     51 Sepal Length
6: 3.2 versicolor     51 Sepal  Width
7: 4.7 versicolor     51 Petal Length
8: 1.4 versicolor     51 Petal  Width
```

```
   Sepal Petal    Species flower    dim
1:   5.1   1.4     setosa      1 Length
2:   3.5   0.2     setosa      1  Width
3:   7.0   4.7 versicolor     51 Length
4:   3.2   1.4 versicolor     51  Width
```

```
   Length Width    Species flower  part
1:    5.1   3.5     setosa      1 Sepal
2:    1.4   0.2     setosa      1 Petal
3:    7.0   3.2 versicolor     51 Sepal
4:    4.7   1.4 versicolor     51 Petal
```

W
S
Convert four input reshape columns to **S**ingle output reshape column and two output capture columns
M1
Output reshape column for each part
M2
Output reshape column for each dim
Convert four input reshape columns to **M**ultiple (2) output reshape columns and one output capture column
Column type legend: Name = reshape, Name = copy, Name = capture.

Figure 1: Two rows of the iris data set (W, black) are considered as the input to a wide-to-tall reshape operation. Four input reshape columns are converted to either a single output reshape column (S, blue) or multiple (2) output reshape columns (M1, M2, red). Other output columns are either copied from the non-reshaped input data, or captured from the names of the reshaped input columns.

defining both (1) the subset of reshape input columns to convert, and (2) the additional capture output columns. We will show that this results in a simple, powerful, non-repetitive syntax for wide-to-tall data reshaping.

In this article our original contribution is the R package **nc** which provides a new implementation of the previously proposed named capture regex syntax of Hocking (2019a), in addition to several new functions that perform wide-to-tall data reshaping using regular expressions. The main new ideas are (1) using un-named capture groups in the regex string literal to provide a uniform interface to three regex C libraries, (2) integration of capture groups and **data.table** functionality (Dowle and Srinivasan, 2019), and (3) specifying wide-to-tall reshape operations with a concise syntax which results in less repetitive user code than other packages. A secondary contribution of this article is a detailed comparison of current R functions for reshaping data with regular expressions.

The organization of this article is as follows. The rest of this introduction provides an overview of current R packages for regular expressions and data reshaping. The second section describes the proposed functions of the **nc** package. The third section provides detailed comparisons with other R packages, in terms of syntax and computation times. The article concludes with a summary and discussion.

Related work

There are many R functions which can extract tables from non-tabular text using regular expressions. Recommended R package functions include `base::regexpr` and `base::gregexpr` as well as `utils::strcapture`. CRAN packages include **namedCapture** (Hocking, 2019b), **rematch2** (Csárdi, 2017), **rex** (Ushey et al., 2017), **stringr** (Wickham, 2018), **stringi** (Gagolewski, 2018), **tidyr** (Wickham and Henry, 2018), and **re2r** (Wenfeng, 2017). Hocking (2019a) provides a detailed comparison of these packages in terms of features, syntax, and computation time.

For reshaping data from wide (one row with many columns) to tall (one column with many rows), there are several different R functions that provide similar functionality. Each function supports a different set of features (Table 1); each feature/column is explained in detail below:

pkg::function	single	multiple	regex	na.rm	types	list
nc::capture_melt_multiple	no	unsorted	capture	yes	any	yes
nc::capture_melt_single	yes	no	capture	yes	any	yes
tidyr::pivot_longer	yes	unsorted	capture	yes	some	yes
stats::reshape	yes	sorted	capture	no	some	no
data.table::melt,patterns	yes	sorted	match	yes	no	yes
tidyr::gather	yes	no	no	yes	some	yes
reshape2::melt	yes	no	no	yes	no	no
cdata::rowrecs_to_blocks	yes	unsorted	no	no	no	yes
cdata::unpivot_to_blocks	yes	no	no	no	no	yes
utils::stack	yes	no	no	no	no	no

Table 1: Reshaping functions in R support various features: "single" for converting input columns into a single output column; "multiple" for converting input columns (either "sorted" in a regular order, or "unsorted" for any order) into multiple output columns of different types; "regex" for regular expressions to "match" input column names or to "capture" and create new output column names; "na.rm" for removal of missing values; "types" for converting input column names to non-character output columns; "list" for output of list columns.

single refers to support for converting input reshape columns of the same type to a single reshape output column.

multiple refers to support for converting input reshape columns of possibly different types to multiple output reshape columns; "sorted" means that conversion works correctly only if the input reshape columns are sorted in a regular order, e.g. `Sepal.Length`, `Sepal.Width`, `Petal.Length`, `Petal.Width`; "unsorted" means that conversion works correctly even if the they are not sorted, e.g. `Sepal.Length`, `Sepal.Width`, `Petal.Width`, `Petal.Length`.

regex refers to support for regular expressions; "match" means a pattern is used to match the input column names; "capture" means that the specified pattern is used to create new output capture columns — this is especially useful when the names consist of several distinct pieces of information, e.g. `Sepal.Length`; "no" means that regular expressions are not directly supported (although `base::grep` can always be used).

na.rm refers to support for removing missing values.

types refers to support for converting captured text to numeric output columns.

list refers to support for output of list columns.

Recommended R package functions include `stats::reshape` and `utils::stack` for reshaping data from wide to tall. Of the features listed in Table 1, `utils::stack` only supports output with a single reshape column, whereas `stats::reshape` supports the following features. For data with regular input column names (output column, separator, time value), regular expressions can be used to specify the separator (e.g. in `Sepal.Length`, `Sepal` is output column, dot is separator, `Length` is time value). Multiple output columns are supported, but incorrect output may be computed if input columns are not sorted in a regular order. The time value is output to a capture column named `time` by default. Automatic type conversion is performed on time values when possible, but custom type conversion functions are not supported. There is neither support for missing value removal nor list column output.

The **tidyr** package provides two functions for reshaping data from wide to tall format: `gather` and `pivot_longer`. The older `gather` function only supports converting input reshape columns to a single output reshape column (not multiple). The input reshape columns to convert may not be directly specified using regular expressions; instead R expressions such as `x:y` can be used to indicate all columns starting from `x` and ending with `y`. It does support limited type conversion; if the `convert=TRUE` argument is specified, the `utils::type.convert` function is used to convert the input column names to numeric, integer, or logical. In contrast the newer `pivot_longer` also supports multiple output reshape columns (even if input reshape columns are unsorted), and regular expressions for specifying output capture columns (but to specify input reshape columns with a regex, `grep` must be used). Limited type conversion is also supported in `pivot_longer`, via the `names_ptypes` argument, which should be a list with names corresponding to output columns and values corresponding to prototypes (zero-length atomic vectors, e.g. `numeric()`). Both functions support list columns and removing missing values, although different arguments are used (`na.rm` for `gather`, `values_drop_na` for `pivot_longer`).

The **reshape2** and **data.table** packages each provide a melt function for converting data from wide to tall (Wickham, 2007; Dowle and Srinivasan, 2019). The older **reshape2** version only supports converting input reshape columns to a single output reshape column, whereas the newer **data.table** version also supports multiple output reshape columns. Regular expressions are not supported in **reshape2**, but can be used with data.table::patterns to match input column names to convert (although the output can be incorrect if columns are not sorted in a regular order). Neither function supports type conversion, and both functions support removing missing values from the output using the na.rm argument. List column output is supported in **data.table** but not **reshape2**.

The **cdata** package provides several functions for data reshaping, including rowrecs_to_blocks and unpivot_to_blocks which can convert data from wide to tall (Mount and Zumel, 2019). The simpler of the two functions is unpivot_to_blocks, which supports a single output reshape column (interface similar to reshape2::melt/tidyr::gather). The user of rowrecs_to_blocks must provide a control table that describes how the input should be reshaped into the output. It therefore supports multiple output reshape columns, for possibly unsorted input columns. Both functions support list column output, but other features from Table 1 are not supported (regular expressions, missing value removal, type conversion).

New features in nc

The **nc** package provides new regular expression functionality based on the syntax recently proposed by Hocking (2019a). In this section we first discuss how the **nc** package implements this syntax as a front-end to three regex engines, and we then discuss the new features for data reshaping using regular expressions.

Uniform interface to three regex engines

Several C libraries providing regular expression engines are available in R. The standard R distribution has included and the Perl-Compatible Regular Expressions (PCRE) C library since 2002 (R Core Team, 2002). CRAN package **re2r** provides the RE2 library, and **stringi** provides the ICU library. Each of these regex engines has a unique feature set, and may be preferred for different applications. For example, PCRE is installed by default, RE2 guarantees matching in polynomial time, and ICU provides strong unicode support. For a more detailed comparison of the relative strengths of each regex library, we refer the reader to the recent paper of (Hocking, 2019a).

Each regex engine has also a different R interface, so switching from one engine to another may require non-trivial modifications of user code. In order to make switching between engines easier, Hocking (2019a) introduced the **namedCapture** package, which provides a uniform interface for capturing text using PCRE and RE2. The user may specify the desired engine via e.g. options(namedCapture.engine="PCRE"); the **namedCapture** package provides the output in a uniform format. However **namedCapture** requires the engine to support specifying capture group names in regex pattern strings, and to support output of the group names to R (which ICU does not support).

Our proposed **nc** package provides support for the ICU engine in addition to PCRE and RE2. The **nc** package implements this functionality using un-named capture groups, which are supported in all three regex engines. In particular, a regular expression is constructed in R code that uses named arguments to indicate captures, which are translated to un-named groups when passed to the regex engine. For example, consider a user who wants to capture the two pieces of the column names of the iris data, e.g. Sepal.Length. The user would typically specify the capturing regular expression as a string literal, e.g. "(.*)[.](.*)". Using **nc** the same pattern can be applied to the iris data column names via

```
> nc::capture_first_vec(
+   names(iris), part=".*", "[.]", dim=".*", engine="ICU", nomatch.error=FALSE)

    part    dim
1: Sepal Length
2: Sepal  Width
3: Petal Length
4: Petal  Width
5:  <NA>   <NA>
```

Above we see an example usage of nc:capture_first_vec, which is for capturing the first match of a regex from a character vector subject (the first argument). There are a variable number of other arguments (...) which are used to define the regex pattern. In this case there are three pattern arguments: `part=".*","[.]",dim=".*"`. Each named R argument in the pattern generates an un-named capture group by enclosing the specified value in parentheses, e.g. `(.*)`. All of the sub-patterns are pasted together in the sequence they appear in order to create the final pattern that is used with the specified regex engine. The `nomatch.error=FALSE` argument is given because the default is to stop with an error if any subjects do not match the specified pattern (the fifth subject `Species` does not match). Under the hood, the following function is called to parse the pattern arguments:

```
> str(compiled <- nc::var_args_list(part=".*", "[.]", dim=".*"))

List of 2
 $ fun.list:List of 2
  ..$ part:function (x)
  ..$ dim :function (x)
 $ pattern : chr "(.*)[.](.*)"
```

This function is intended mostly for internal use, but can be useful for viewing the generated regex pattern (or using it as input to another regex function). The return value is a named list of two elements: `pattern` is the capturing regular expression which is generated based on the input arguments, and `fun.list` is a named list of type conversion functions. Group-specific type conversion functions are useful for converting captured text into numeric output columns Hocking (2019a). Note that the order of elements in `fun.list` corresponds to the order of capture groups in the pattern (e.g. first capture group named `part`, second `dim`). These data can be used with any regex engine that supports un-named capture groups (including ICU) in order to get a capture matrix with column names, e.g.

```
> m <- stringi::stri_match_first_regex(names(iris), compiled$pattern)
> colnames(m) <- c("match", names(compiled$fun.list))
> m

     match          part    dim
[1,] "Sepal.Length" "Sepal" "Length"
[2,] "Sepal.Width"  "Sepal" "Width"
[3,] "Petal.Length" "Petal" "Length"
[4,] "Petal.Width"  "Petal" "Width"
[5,] NA             NA      NA
```

Again, this is not the recommended usage of **nc**, but here we give these details in order to explain how it works. Note that the result from **stringi** is a character matrix with three columns: first for the entire match, and another column for each capture group. Using the same pattern with `base::regexpr` (PCRE engine) or `re2r::re2_match` (RE2 engine) yields output in varying formats. The **nc** package takes care of converting these different results into a standard format which makes it easy to switch regex engines (by changing the value of the `engine` argument).

Note that the standard format used by **nc**, as shown above with `nc::capture_first_vec`, is a data table. The main reason that data tables are always output by **nc** matching functions is in order to support numeric output columns, when type conversion functions are specified. A secondary reason is for the convenient syntax, as we will show in the next section.

Data table integration and `nc::field` to avoid repetition

In the previous section we mentioned that every **nc** matching function outputs a data table, which can be queried/summarized/joined/etc using the convenient syntax of the **data.table** package. In this section we show an example of how this syntax makes is easy to parse non-tabular text output files from a bioinformatics program. We also show an example of how the new `nc::field` function can be used to avoid repetition when defining a pattern to match fields of the form `variable=value`.

We consider a bioinformatics program called SweeD which outputs two data files that are linked by an integer alignment ID number (Pavlidis et al., 2013). The “Report” file is similar to tab-separated values (TSV), except that there are some blank lines, followed by two forward slashes, followed by the integer alignment number. For example, the first few lines of the first two alignments look like:

```
> report.txt.gz <- system.file("extdata", "SweeD_Report.txt.gz", package="nc")
> report.vec <- readLines(report.txt.gz)
> cat(report.vec[1:5], sep="\n")

//1
Position        Likelihood        Alpha
```

```
700.0000        4.637328e-03        2.763840e+02
130585.6172        3.781283e-01        8.490200e-04

> cat(report.vec[1003:1008], sep="\n")

129756434.0000        3.850623e-01        4.812648e+01

//2
Position        Likelihood        Alpha
135.0000        7.282316e-01        3.163686e+01
111533.0625        2.548831e+00        4.932014e-04
```

The file above can be easily parsed by `nc::capture_all_str`, which is for finding all matches of a regular expression in a multi-line text file:

```
> report.alignments <- nc::capture_all_str(
+   report.vec,
+   "//",
+   Alignment="[0-9]+",
+   TSV="[^/]+")
> class(report.alignments)

[1] "data.table" "data.frame"

> dim(report.alignments)

[1] 10  2

> nchar(report.alignments$TSV)

 [1] 40170 40028 39996 39946 39924 39914 39907 39906 39894 39892

> substr(as.matrix(report.alignments[1:2]), 1, 50)

     Alignment TSV
[1,] "1"       "\nPosition\tLikelihood\tAlpha\n700.0000\t4.637328e-03\t2"
[2,] "2"       "\nPosition\tLikelihood\tAlpha\n135.0000\t7.282316e-01\t3"
```

The code above computes a data table with ten rows, one for each match in the text file. There are two columns: `Alignment` is a unique identifier, and `TSV` contains the tab-separated values in each block. Each `TSV` entry is a long text string (about 40,000 characters); we display the first 50 characters of the first two blocks above. Because the result is a data table, we can parse the `TSV` using a call to `fread` inside of a by block:

```
> (report.positions <- report.alignments[, data.table::fread(text=TSV), by=Alignment])

       Alignment   Position   Likelihood        Alpha
    1:         1      700.0 4.637328e-03 2.763840e+02
    2:         1   130585.6 3.781283e-01 8.490200e-04
    3:         1   260471.2 3.602315e-02 4.691340e-03
    4:         1   390356.9 7.618749e-01 5.377668e-04
    5:         1   520242.5 2.979971e-08 1.411765e-01
   ---
 9996:        10 82991564.8 8.051006e-03 1.357819e-03
 9997:        10 83074967.8 7.048433e-03 1.825764e-03
 9998:        10 83158370.8 1.012360e-07 7.999999e-03
 9999:        10 83241773.8 3.977189e-08 9.999997e-01
10000:        10 83325174.0 3.980538e-08 1.200000e+03
```

A bioinformatics analyst may desire to plot the `Likelihood` or `Alpha` values in a genome browser, as a function of `Position` along the chromosome. Each alignment has a chromosome name which therefore needs to be added/joined to this table; the second "Info" file has the chromosome name, as well as some other summary statistics for each alignment:

```
> info.txt.gz <- system.file("extdata", "SweeD_Info.txt.gz", package="nc")
> info.vec <- readLines(info.txt.gz)
> info.vec[24:40]
```

```
 [1] " Alignment 1"                      ""
 [3] "\t\tChromosome:\t\tscaffold_0"     "\t\tSequences:\t\t14"
 [5] "\t\tSites:\t\t\t1670366"           "\t\tDiscarded sites:\t1264068"
 [7] ""                                  "\t\tProcessing:\t\t155.53 seconds"
 [9] ""                                  "\t\tPosition:\t\t8.936200e+07"
[11] "\t\tLikelihood:\t\t4.105582e+02"   "\t\tAlpha:\t\t\t6.616326e-06"
[13] ""                                  ""
[15] " Alignment 2"                      ""
[17] "\t\tChromosome:\t\tscaffold_1"
```

Again, the data above can be parsed to a data table via a regular expression:

```
> nc::capture_all_str(
+   info.vec,
+   " ",
+   "Alignment", " ", Alignment="[0-9]+",
+   "\n\n\t\t",
+   "Chromosome", ":\t\t", Chromosome=".*")

    Alignment Chromosome
 1:         1 scaffold_0
 2:         2 scaffold_1
 3:         3 scaffold_2
 4:         4 scaffold_3
 5:         5 scaffold_4
 6:         6 scaffold_5
 7:         7 scaffold_6
 8:         8 scaffold_7
 9:         9 scaffold_8
10:        10 scaffold_9
```

The result above is a data table with ten rows, one for each alignment. There are two columns: the first is the `Alignment` identifier, and the second is the `Chromosome` name. Note the repetition in the code above: `Alignment`/`Chromosome` appear as capture group names as well as in the pattern itself. Such repetition can be avoided by using the `nc::field` helper function, which uses its first argument for the capture group name as well as a pattern to match:

```
> (info.alignments <- nc::capture_all_str(
+   info.vec,
+   " ",
+   nc::field("Alignment", " ", "[0-9]+"),
+   "\n\n\t\t",
+   nc::field("Chromosome", ":\t\t", ".*")))

    Alignment Chromosome
 1:         1 scaffold_0
 2:         2 scaffold_1
 3:         3 scaffold_2
 4:         4 scaffold_3
 5:         5 scaffold_4
 6:         6 scaffold_5
 7:         7 scaffold_6
 8:         8 scaffold_7
 9:         9 scaffold_8
10:        10 scaffold_9
```

Note that the repetition in the code has been removed, and the result is the same (because the generated regex pattern is the same). Also note that the order of the regex string literals is unchanged, so the pattern is as easy to read/understand as the original repetitive version. Internally, `nc::field` generates a pattern that consists of pasting all of its arguments together; the first argument is also used as the capture group name for the third (and subsequent) arguments. In general, `nc::field` is useful whenever there are several fields to parse, each of the form `variable=value`.

In order to create the desired output table with `Chromosome` and `Position` columns, we join the two data tables:

```
> info.alignments[report.positions, on="Alignment"]
```

```
       Alignment Chromosome   Position   Likelihood        Alpha
    1:         1 scaffold_0      700.0 4.637328e-03 2.763840e+02
    2:         1 scaffold_0   130585.6 3.781283e-01 8.490200e-04
    3:         1 scaffold_0   260471.2 3.602315e-02 4.691340e-03
    4:         1 scaffold_0   390356.9 7.618749e-01 5.377668e-04
    5:         1 scaffold_0   520242.5 2.979971e-08 1.411765e-01
   ---
 9996:        10 scaffold_9 82991564.8 8.051006e-03 1.357819e-03
 9997:        10 scaffold_9 83074967.8 7.048433e-03 1.825764e-03
 9998:        10 scaffold_9 83158370.8 1.012360e-07 7.999999e-03
 9999:        10 scaffold_9 83241773.8 3.977189e-08 9.999997e-01
10000:        10 scaffold_9 83325174.0 3.980538e-08 1.200000e+03
```

To conclude this section, we have shown that `nc::capture_all_str` inputs a multi-line text file subject, then outputs a data table with one row per match and one column per capture group. We have also shown how these data tables may be further processed (e.g. `fread`, `by=Alignment`) and joined using the convenient data table syntax. Finally, we showed two examples of using `nc::field` to define a pattern that matches/captures fields of the form `variable=value`.

Wide-to-tall data reshaping

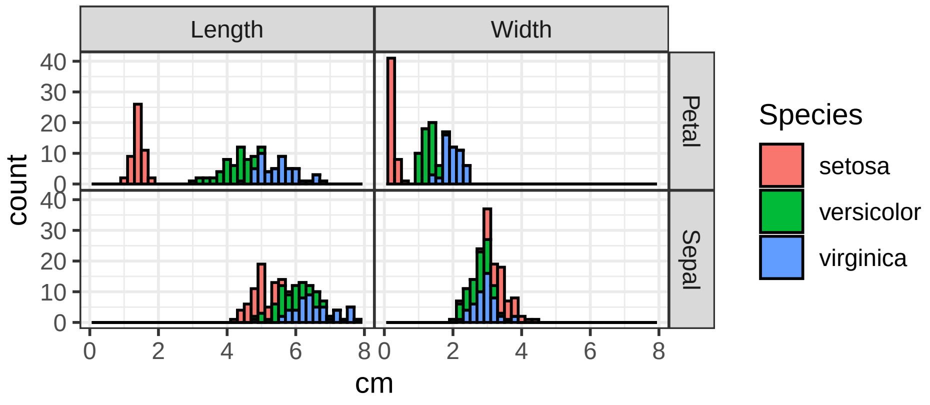
In this section we show how new **nc** functions can be used to reshape wide data (with many columns) to tall data (with fewer columns, and more rows). We begin by considering the two data visualization problems mentioned in the introduction, involving the familiar iris data set. First, suppose we would like to examine the largest/smallest values or ranges. One way would be to use a histogram of each numeric variable, with row facets for flower part and column facets for measurement dimension. Our desired output therefore needs a single column with all of the reshaped numeric data to plot (Figure 1, W→S). We can perform this operation using `nc::capture_melt_single`, which inputs a data frame and a pattern describing the columns to melt:

```
> (iris.tall.single <- nc::capture_melt_single(
+   iris, part=".*", "[.]", dim=".*", value.name="cm"))

       Species  part    dim  cm
  1:    setosa Sepal Length 5.1
  2:    setosa Sepal Length 4.9
  3:    setosa Sepal Length 4.7
  4:    setosa Sepal Length 4.6
  5:    setosa Sepal Length 5.0
 ---
596: virginica Petal  Width 2.3
597: virginica Petal  Width 1.9
598: virginica Petal  Width 2.0
599: virginica Petal  Width 2.3
600: virginica Petal  Width 1.8
```

The output above consists of one copy column (`Species`), two capture columns (`part`, `dim`), and a single reshape column (`cm`). The `value.name` argument is not considered part of the pattern, and instead specifies the name of the output reshape column. These data can be used to create the desired histogram with **ggplot2** via:

```
> library(ggplot2)
> ggplot(iris.tall.single)+facet_grid(part~dim)+
+   theme_bw()+theme(panel.spacing=grid::unit(0, "lines"))+
+   geom_histogram(aes(cm, fill=Species), color="black", bins=40)
```



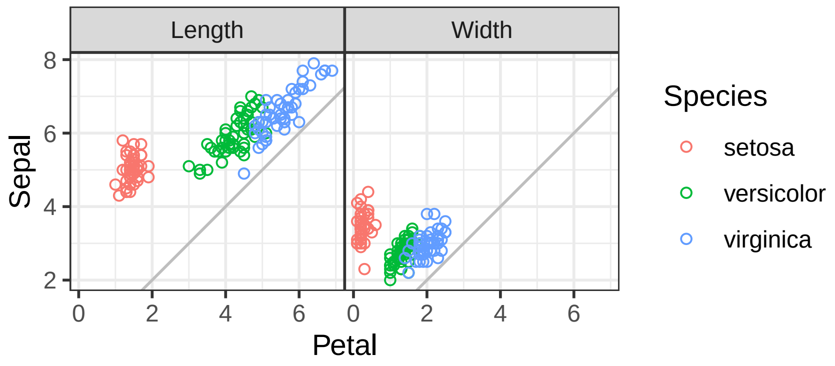
For the second data reshaping task, suppose we want to determine whether or not sepals are larger than petals, for each measurement dimension and species. We could use a scatterplot of sepal versus petal, with a facet for measurement dimension. We therefore need a data table with two reshape output columns: a `Sepal` column to plot against a `Petal` column (Figure 1, W→M1). We can perform this operation using `nc::capture_melt_multiple`, which inputs a data frame and a special pattern which must contain the `column` group and one other named group:

```
> (iris.parts <- nc::capture_melt_multiple(
+   iris, column=".*", "[.]", dim=".*"))

       Species    dim Petal Sepal
  1:    setosa Length   1.4   5.1
  2:    setosa Length   1.4   4.9
  3:    setosa Length   1.3   4.7
  4:    setosa Length   1.5   4.6
  5:    setosa Length   1.4   5.0
 ---
296: virginica  Width   2.3   3.0
297: virginica  Width   1.9   2.5
298: virginica  Width   2.0   3.0
299: virginica  Width   2.3   3.4
300: virginica  Width   1.8   3.0
```

The output above consists of one copy column (`Species`), one capture columns (`dim`), and two reshape columns (`Petal`, `Sepal`). These data can be used to create the scatterplot using **ggplot2** via:

```
> ggplot(iris.parts)+facet_grid(.~dim)+
+   theme_bw()+theme(panel.spacing=grid::unit(0, "lines"))+
+   coord_equal()+geom_abline(slope=1, intercept=0, color="grey")+
+   geom_point(aes(Petal, Sepal, color=Species), shape=1)
```



It is clear from the plot that sepals are larger than petals, for both dimensions, and for every measured flower.

As a final example we consider a data set from the World Health Organization:

```
> data(who, package="tidyr")
> set.seed(1);sample(names(who), 10)
```

```
 [1] "newrel_f3544" "year"         "new_ep_m65"   "country"      "new_ep_m1524"
 [6] "new_sn_m4554" "new_ep_f3544" "new_sp_f2534" "new_sp_f65"   "newrel_m4554"
```

Each reshape column name starts with new and has three distinct pieces of information: diagnosis type (e.g. ep, rel), gender (m or f), and age range (e.g. 1524, 4554). As with the iris data, there are a number of interesting questions that can be answered by first reshaping the data (e.g. which age groups and diagnosis types are the most frequent).

```
> new.diag.gender <- list("new_?", diagnosis=".*", "_", gender=".")
> nc::capture_melt_single(who, new.diag.gender, ages=".*")

                         country iso2 iso3 year diagnosis gender ages value
    1:               Afghanistan   AF  AFG 1997        sp      m  014     0
    2:               Afghanistan   AF  AFG 1998        sp      m  014    30
    3:               Afghanistan   AF  AFG 1999        sp      m  014     8
    4:               Afghanistan   AF  AFG 2000        sp      m  014    52
    5:               Afghanistan   AF  AFG 2001        sp      m  014   129
   ---
76042:                  Viet Nam   VN  VNM 2013       rel      f   65  3110
76043: Wallis and Futuna Islands   WF  WLF 2013       rel      f   65     2
76044:                     Yemen   YE  YEM 2013       rel      f   65   360
76045:                    Zambia   ZM  ZMB 2013       rel      f   65   669
76046:                  Zimbabwe   ZW  ZWE 2013       rel      f   65   725
```

Above we define new.diag.list which we use as the first part of the pattern in the call to nc::capture_melt_single above (and we will use that sub-pattern again below). The result is a data table in which all three capture columns (diagnosis, gender, ages) are character vectors, since no type conversion functions were specified. If we want to extract two numeric columns from ages (e.g. to use as interval-censored outputs in a survival regression), we can provide a different pattern with type conversion functions:

```
> who.typed <- nc::capture_melt_single(
+   who, new.diag.gender, ages=list(
+     min.years="0|[0-9]{2}", as.numeric,
+     max.years="[0-9]{0,2}", function(x)ifelse(x=="", Inf, as.numeric(x))),
+   value.name="count")
> str(who.typed)

Classes ‘data.table’ and 'data.frame':	76046 obs. of  10 variables:
 $ country  : chr  "Afghanistan" "Afghanistan" "Afghanistan" "Afghanistan" ...
 $ iso2     : chr  "AF" "AF" "AF" "AF" ...
 $ iso3     : chr  "AFG" "AFG" "AFG" "AFG" ...
 $ year     : int  1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 ...
 $ diagnosis: chr  "sp" "sp" "sp" "sp" ...
 $ gender   : chr  "m" "m" "m" "m" ...
 $ ages     : chr  "014" "014" "014" "014" ...
 $ min.years: num  0 0 0 0 0 0 0 0 0 0 ...
 $ max.years: num  14 14 14 14 14 14 14 14 14 14 ...
 $ count    : int  0 30 8 52 129 90 127 139 151 193 ...
 - attr(*, ".internal.selfref")=<externalptr>
```

Comparison with other packages

TODO

Discussion and conclusions

TODO

Reproducible research statement. The source code for this article can be freely downloaded from https://github.com/tdhock/nc-article

Bibliography

G. Csárdi. *rematch2: Tidy Output from Regular Expression Matching*, 2017. URL https://CRAN.R-project.org/package=rematch2. R package version 2.0.1. [p2]

M. Dowle and A. Srinivasan. *data.table: Extension of 'data.frame'*, 2019. http://r-datatable.com. [p2, 4]

M. Gagolewski. *R package stringi: Character string processing facilities*, 2018. URL http://www.gagolewski.com/software/stringi/. [p2]

T. D. Hocking. Comparing namedcapture with other r packages for regular expressions. *R Journal*, 2019a. [p1, 2, 4, 5]

T. D. Hocking. *namedCapture: Named Capture Regular Expressions*, 2019b. R package version 2019.01.14. [p2]

J. Mount and N. Zumel. *cdata: Fluid Data Transformations*, 2019. URL https://CRAN.R-project.org/package=cdata. R package version 1.1.2. [p4]

P. Pavlidis, D. Živković, A. Stamatakis, and N. Alachiotis. SweeD: Likelihood-Based Detection of Selective Sweeps in Thousands of Genomes. *Molecular Biology and Evolution*, 30(9):2224–2234, 06 2013. ISSN 0737-4038. doi: 10.1093/molbev/mst112. [p5]

R Core Team. News for the 1.x series, 2002. URL https://github.com/tdhock/regex-tutorial/blob/master/R.NEWS.1.txt. [p4]

K. Ushey, J. Hester, and R. Krzyzanowski. *rex: Friendly Regular Expressions*, 2017. URL https://CRAN.R-project.org/package=rex. R package version 1.1.2. [p2]

Q. Wenfeng. *re2r: RE2 Regular Expression*, 2017. URL https://CRAN.R-project.org/package=re2r. R package version 0.2.0. [p2]

H. Wickham. Reshaping data with the reshape package. *Journal of Statistical Software*, 21(12):1–20, 2007. URL http://www.jstatsoft.org/v21/i12/. [p4]

H. Wickham. *stringr: Simple, Consistent Wrappers for Common String Operations*, 2018. URL https://CRAN.R-project.org/package=stringr. R package version 1.3.1. [p2]

H. Wickham and L. Henry. *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*, 2018. URL https://CRAN.R-project.org/package=tidyr. R package version 0.8.2. [p2]

Toby Dylan Hocking
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, Arizona
USA
toby.hocking@nau.edu