readr: Summary

Abhinav Mishra, Kristin Köhler, Sanket Gosavi, Utkarsha Kandale

2022-05-11

readr is a part of the core tidyverse package.

It is used to *import* ('read') and *write* files. *readr*'s functions are used for turning the most common flat files into data frames:

- *read_csv()* reads comma delimited files, *read_csv2()* reads semicolon separated files (common in countries where , is used as the decimal place), *read_tsv()* reads tab delimited files, and *read_delim()* reads in files with any delimiter.
- *read_fwf()* reads fixed width files. You can specify fields either by their widths with *fwf_widths()* or their position with *fwf_positions()*. *read_table()* reads a common variation of fixed width files where columns are separated by white space.

They have similar syntax. In this tutorial, we will demonstrate how to use the *read_tsv* and *read_csv2* function with an example.

```
library(tidyverse)
```

We can use the function *read_tsv()* to read the tab-delimited text file *peptides.txt* into a tibble.

```
peptides <- read_tsv("peptides.txt")
peptides
```

```
## # A tibble: 73,903 x 69
##    Sequence  `N-term cleava~` `C-term cleava~` `Amino acid be~` `First amino a~`
##    <chr>     <chr>            <chr>            <chr>            <chr>
##  1 AAAAAAAA~ QSRFQVDLVSENAGR~ AAAAAAAAGAGAGAK~ R                A
##  2 AAAAAAAA~ _____________~ LSGPAEVGPGAVGER~ K                A
##  3 AAAAAAAA~ _____________~ AAAAAAAAAGAAGGR~ M                A
##  4 AAAAAAAA~ LEYYENARKFRHSVR~ AAASGAAIPPLIPPR~ R                A
##  5 AAAAAAAA~ LEYYENARKFRHSVR~ AASGAAIPPLIPPRR~ R                A
##  6 AAAAAAAA~ TTSSRVLRGGRDRGR~ RGRAAAAAAAAAVSR~ R                A
##  7 AAAAAAAG~ _____________~ SWDADAFSVEDPVRK~ M                A
##  8 AAAAAAAL~ _____________~ AAAAALESWQAAAPR~ M                A
##  9 AAAAAAAL~ TILRQARNHKLRVDK~ RVDKAAAAAAALQAK~ K                A
## 10 AAAAAAAP~ _____________~ SGGGGGGEEERLEEK~ M                A
## # ... with 73,893 more rows, and 64 more variables: `Second amino acid` <chr>,
## #   `Second last amino acid` <chr>, `Last amino acid` <chr>,
## #   `Amino acid after` <chr>, `A Count` <dbl>, `R Count` <dbl>,
## #   `N Count` <dbl>, `D Count` <dbl>, `C Count` <dbl>, `Q Count` <dbl>,
## #   `E Count` <dbl>, `G Count` <dbl>, `H Count` <dbl>, `I Count` <dbl>,
## #   `L Count` <dbl>, `K Count` <dbl>, `M Count` <dbl>, `F Count` <dbl>,
## #   `P Count` <dbl>, `S Count` <dbl>, `T Count` <dbl>, `W Count` <dbl>, ...
```

Since this dataset is huge, we will shift our focus to a smaller, customized dataset in the following examples that we will generate with *read_csv2*. All its functionalities are also applicable on other readr functions.

Compared to base *R*, *read.tsv*, etc. functions, the *readr*'s *read_tsv*, etc. functions are much **faster** (approx. 10x). Furthermore, they are more user friendly (they don't convert character vectors to factors, use row names, or merge the column names) and more **reproducible**.

There are a lot of helpful, customizable parameters usable in all of *readr()* functions. Some of them are demonstrated, above.

1. *skip* allows one to skip certain amount of lines from the top. This is very handy since in many files the first few lines are comments which explain the file/data.
2. *comment* does the same, however, here you have to specify which character represents a comment. Every line which starts with that particular character will be skipped.
3. *na* allows one to choose how the NAs are read. They can just be read as NAs or a specific value like 0.
4. *col_names* allows one to choose whether to import column names from the original file or to choose your own column names like we have done above.

Example 1

```
columns <- c("Name","Apples","Date","Time","Cost","Satisfied")

example <- read_csv2("This data is about breakfast routines of people
                     %It contains breakfast related info on four students
                     Abhinav;2;4 May 22;8AM;1;y
                     Kristin;4;4 May 22;7:45AM;2;y
                     Pushpa;3;4 May 22;6:30AM;1,50;y
                     Sanket;0;4 May 22;na;0;n",
                     skip = 1, comment = "%", na = "na",
                     col_names = columns)
example
```

```
## # A tibble: 4 x 6
##   Name    Apples Date     Time    Cost Satisfied
##   <chr>    <dbl> <chr>    <chr>  <dbl> <chr>
## 1 Abhinav      2 4 May 22 8AM      1   y
## 2 Kristin      4 4 May 22 7:45AM   2   y
## 3 Pushpa       3 4 May 22 6:30AM   1.5 y
## 4 Sanket       0 4 May 22 <NA>     0   n
```

Parsing: An important aspect of *readr* package. It handles everything related to the class of data/observation, e.g. it allows one to decide whether observations are numeric, double, logical, factors, etc.

Example 2

```
parse_number("It costs 123.456.789,50 Euros",
             locale = locale(grouping_mark = ".",
                             decimal_mark = ","))
```

```
## [1] 123456790
```

```
parse_double("123.456.789,50",
             locale = locale(decimal_mark = ","))
```

```
## [1] NA
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected               actual
##   <int> <int> <chr>                  <chr>
## 1     1    NA no trailing characters 123.456.789,50
```

```
parse_double("It costs 123.456.789,50 Euros",
             locale = locale(decimal_mark = ","))
```

```
## [1] NA
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected actual
##   <int> <int> <chr>    <chr>
## 1     1    NA a double It costs 123.456.789,50 Euros
```

Usually, numerical data will have certain additional characters such as the currency symbol ($, etc), grouping marks to make it easier to read, decimal marks, etc. Different regions have different ways of symbolizing these marks. For example, Germany uses '.' as grouping mark and ',' as a decimal mark, but the opposite is true for USA. To deal with these issues and make it harder to work with your data *tidyr* contains parse_number and parse_double functions.

parse_numbers converts a variable to numeric. This function also drops any strings associated with the observation and just keeps the number. However, care must be taken that it is not always good at reading decimal marks. This can be solved by using parse_double function instead which correctly recognizes decimal marks and grouping marks. However, parse_double cannot separate strings from the numeric (Example 2).

In case of observation like "It costs 123.456.789,50 Euros", it would be better to separate the observation into multiple columns (one containing the numeric and the others containing strings) and then using parse_double.

Parsing strings: It is generally not needed nowadays. However, it can be useful if someone encounters data which is not encoded in *UTF-8* (this would be the case in older data). Then one can use *parse_character* function to select the appropriate encoding for the data. Furthermore, if one is unsure about the type of encoding one can also use the *guess_encoding* function on an observation from the dataset and then determine which encoding to use.

```
x1 <- "El Ni\xf1o was particularly bad this year"

parse_character(x1, locale =
                  locale(encoding = "Latin1"))
```

```
## [1] "El Niño was particularly bad this year"
```

```
guess_encoding(charToRaw(x1))
```

```
## # A tibble: 2 x 2
##   encoding   confidence
##   <chr>           <dbl>
## 1 ISO-8859-1       0.46
## 2 ISO-8859-9       0.23
```

Parsing dates: It converts the observations containing dates into a specialized vector. This makes it much easier to work with data with dates (for example, adding or subtracting days, months, years, etc.). The code below shows how one can parse dates. The type of values are represented by %.

?parse_date can be used to check the appropriate '%' time to use for your data

```
parse_date("22/05/04", "%y/%m/%d")
```

```
## [1] "2022-05-04"
```

```
parse_date("22-05-04", "%y-%m-%d")
```

```
## [1] "2022-05-04"
```

```
parse_date("4 May 2022", "%d %B %Y")
```

```
## [1] "2022-05-04"
```

Moreover, you can also parse time as factors (for categorical data), and logical (TRUE/FALSE).

Important: functions starting with *parse_** do not work while you are reading a *data.frame*.

When *readr* reads a file, it goes through the first 1000 observations and guesses the type of data for each column based on these 1000 rows (*guess_inf*). This can be demonstrated by using the code below:

```
guess_parser("2010-10-01")
```

```
## [1] "date"
```

```
guess_parser(c("TRUE", "FALSE"))
```

```
## [1] "logical"
```

```
guess_parser(c("12,352,561"))
```

```
## [1] "number"
```

readr package uses a function like this to guess the class of variables in a column, and automatically assign it when it reads the file. However, *readr* is prone to error and in some rare cases, it may not be able to assign the correct class of a column. For example, if a certain column in a dataset only has NA's as the first 1000 observations, readr would probably assign the logical class to this column which may or may not be correct. To fix this issue one can use the *col_types* function to manually assign the class for each column.

Take a look at the code below:

```
example
```

```
## # A tibble: 4 x 6
##   Name    Apples Date     Time    Cost Satisfied
##   <chr>    <dbl> <chr>    <chr>  <dbl> <chr>
## 1 Abhinav      2 4 May 22 8AM      1   y
## 2 Kristin      4 4 May 22 7:45AM   2   y
## 3 Pushpa       3 4 May 22 6:30AM   1.5 y
## 4 Sanket       0 4 May 22 <NA>     0   n
```

```
example_parsed <- read_csv2("This data is about breakfast routines
                    of people%It contains breakfast related info on
                    four students
                    Abhinav;2;4 May 22;8:00 AM;1;y
                    Kristin;4;4 May 22;7:45 AM;2;y
                    Pushpa;3;4 May 22;6:30 AM;1,50;y
                    Sanket;0;4 May 22;na;0;n",
                     skip = 1, comment = "%", na = "na",
                    col_names = columns,
                     col_types = cols(
                       Name = col_character(),
                       Apples = col_double(),
                       Date = col_date(format = "%d %B %y"),
                       Time = col_time(format = "%I:%M %p"),
                       Cost = col_number(),
                       Satisfied = col_factor()))

example_parsed
```

```
## # A tibble: 6 x 1
##   Name
##   <chr>
## 1 of people
## 2 four students
## 3 Abhinav;2;4 May 22;8:00 AM;1;y
## 4 Kristin;4;4 May 22;7:45 AM;2;y
## 5 Pushpa;3;4 May 22;6:30 AM;1,50;y
## 6 Sanket;0;4 May 22;na;0;n
```

If we revisit the previous dataset, we notice that the **readr** package has made some mistakes with parsing the columns (i.e. it has parsed the columns *Date*, *Time*, and *Satisfied* as characters rather than dates, time, and category, respectively). This issue can be solved by manually parsing the data using the col_types function as demonstrated in the example_parsed code block.

readr also has functions to write your data into a file and save it on your disc. This can be done using functions like *write_csv* or *write_tsv*. The code below saves the example_parsed tibble as a *.csv* file in the working directory.

```
write_csv(example_parsed, "example_parsed.csv")
```

Further information: packages for importing types of data

1. *haven* reads SPSS, Stata, and SAS files.
2. *readxl* reads excel files (both .xls and .xlsx).
3. *DBI,* along with a database specific backend (e.g. RMySQL, RSQLite, RPostgreSQL, etc..) allows you to run SQL queries against a database and return a data frame.