Input and Output

Ulrich Bergmann

6 February 2020

So far we have only used datasets from `R` packages or created toy tibbles. In this section, we will learn how to read in data from a variety of sources.

1 Reading `csv` files with `readr`

We will start with the `readr` packages which is useful to a) read in `csv` files and to b) correctly parsing data columns. Before we start, let's load some packages.

```
library(readr)
library(tibble)
library(dplyr)
```

1.1 Working directory

First steps first: To read in data from a specific location on our hard drive we should specify a working directory. You can get the current working directory with `getwd()` and set a new one with `setwd()` (not recommended anymore, now that we have projects but still nice to know). In this case, I set the working directory to the `r_public` folder that I created before.

```
cat("Current working directory:")
```

```
## Current working directory:
```

```
getwd()
```

```
## [1] "/home/bergmul/documents/teaching/pp4rs/2020/dev/R/io"
```

1.2 Reading in *-delimited data

The `readr` package provides several functions to read in delimited data:

- `read_csv()`: comma delimited
- `read_csv2()`: semicolon delimited
- `read_tsv()`: tab delimited
- `read_delim()`: any delimiter

To see how they work, let's create some data and read them in. (I know, I know still no real data. Be patient!)

```
my_csv = "a, b, c, d
          1, 2, 3, 4
          5, 6, 7, 8"

read_csv(my_csv)
```

```
## # A tibble: 2 x 4
##       a     b     c     d
##   <dbl> <dbl> <dbl> <dbl>
## 1     1     2     3     4
## 2     5     6     7     8
```

As you can see, the function correctly interpreted the first line of our string as variable names and the remaining elements as comma-separated integers. The other functions work in a similar way. There is one exception though: `read_delim`. It allows for more general specifications.

```
my_csv = "a_ b_ c_ d
          1_ 2_ Maria_ female
          5_ 6_ Teresa_ female"

read_delim(my_csv, delim = "_", trim_ws = TRUE)
```

```
## # A tibble: 2 x 4
##       a     b c      d
##   <dbl> <dbl> <chr>  <chr>
## 1     1     2 Maria  female
## 2     5     6 Teresa female
```

In this case, we specified an underscore as the delimiter and told the function to trim leading and trailing whitespace. Sometimes the `csv` file includes lines which we want `readr` to ignore. To do this we use the `skip` argument.

```
my_csv = "Sometimes you can read some rubbish here
          We don't want to import this
          name, age
          Julian, 29"
read_csv(my_csv, skip = 2)
```

```
## # A tibble: 1 x 2
##   name     age
##   <chr>  <dbl>
## 1 Julian    29
```

At other times, the `csv` file does not provide variable names, we can provide them by passing a character vector as the `col_names` argument.

```
my_csv = "Julian, 29\nTeresa, 25"

read_csv(my_csv, col_names = c("Name", "Age"))
```

```
## # A tibble: 2 x 2
##   Name     Age
##   <chr>  <dbl>
## 1 Julian    29
## 2 Teresa    25
```

There are more options and you can explore them by looking them up in the help file. For now, I will only show you one more useful option: You can use the `na` argument to specify the characters used in the `csv` file to indicate missing values.

```
my_csv = "Julian, 29
          Teresa, 25
          Jonas, .
          ., 64"
read_csv(my_csv, col_names = c("Name", "Age"), na = ".")
```

```
## # A tibble: 4 x 2
##   Name     Age
##   <chr>  <dbl>
## 1 Julian    29
## 2 Teresa    25
## 3 Jonas     NA
## 4 <NA>      64
```

2 Parsing data correctly with the parse functions

Sometimes columns are not correctly interpreted when they are read in. For these vectors, we can parse them differently using the `parse` functions. Each of these functions takes a vector and returns a vector. For example, assume in the following that I want to parse the `age` column not as an integer vector but as a character vector. I can use the `parse_character` function for this.

```
my_csv = "Julian, 29
          Teresa, 25
          Jonas, .
          ., 64"
my_tibble = read_csv(my_csv, col_names = c("name", "age"), na = ".")
head(my_tibble)
str(parse_character(my_tibble$age))
```

There are a bunch of functions, each for a different kind of data type:

- `parse_logical`
- `parse_number` (`parse_double`, `parse_integer`)
- `parse_character`
- `parse_datetime` (`parse_date`, `parse_datetime`)

Let's look at some of them in the following.

2.1 Parsing numbers

The `parse_number` function is pretty amazing. It can recover number vectors from almost anything! Just look at the following example.

```
my_column = c("100$", "20%", "Something with a 100")
parse_number(my_column)
```

```
## [1] 100  20 100
```

You use the locale function to control for country-specific delimiters for the decimal point and grouping.

```
# comma instead of decimal point
my_column = c("1,23", "1,23", "1,245")
parse_number(my_column, locale = locale(decimal_mark = ","))
```

```
## [1] 1.230 1.230 1.245
```

```
# ' to group numbers
my_column = c("123'456'789")
parse_number(my_column, locale = locale(grouping_mark = "'"))
```

```
## [1] 123456789
```

2.2 Parsing characters

You would think parsing characters is pretty easy. There can be difficulties though because not everybody uses the same character encoding. To learn more about this topic, take a look at this website: http://www.w3.org/International/articles/definitions-characters/. We only need to know that different encodings exist and they can lead to problems. R usually assumes to 'UTF-8' encoding (and you should use it too!). See what happens if we read in characters with `Latin-1` encoding:

```
x1 = "El Ni\xf1o was particularly bad this year"
parse_character(x1)
```

```
## [1] "El Ni\xf1o was particularly bad this year"
```

Well, that does not look nice. But luckily enough, we can use the `locale` function to tell `readr` that the string is encoded with `Latin-1`.

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

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

Now, this time the parsing is correct! You can also use `readr` to try to guess the encoding with the `guess_encoding` function. Look up its help file if you want to know more.

2.3 Parsing dates

```
# specify time zone maybe
parse_datetime("2016-09-08T0708")
```

```
## [1] "2016-09-08 07:08:00 UTC"
```

```
parse_datetime("20160908T0708")
```

```
## [1] "2016-09-08 07:08:00 UTC"
```

```
parse_date("2016-09-08")
```

```
## [1] "2016-09-08"
```

```
parse_date("2016/09/08")
```

```
## [1] "2016-09-08"
```

```
library(hms)
parse_time("01:10 am")
```

```
## 01:10:00
```

```
parse_date("27/05/1987", "%d/%m/%Y")
```

```
## [1] "1987-05-27"
```

3 Parsing and reading at the same time

Each `parse` function has a corresponding `col` function. This allows you to use the `parse` function to find out how to correctly parse a column and then specify the correct parsing right at the beginning of the data processing using the corresponding `col` function. I usually read in data in three steps.

1. First, I read in all columns as character vectors. This allows me to browse the data and determine the correct parsing. To read in every column as a character vector, you can use the `.default` argument in the `col_types` function.

```
challenge1 = read_csv(readr_example("challenge.csv"),
                     col_types = cols(
                       .default = col_character()
                     ))
```

2. I can try out different parsers using the `parse` functions. (Note that you can use the `parse` functions from the `readr` package together with other packages such as `readxl`). In this case, browsing and parsing will lead you to conclude that the correct parsers are `parse_double` and `parse_data`, respectively.
3. Finally, we specify the correct parsers directly at the beginning of the data processing stage using the `col` functions that correspond to the `parse` functions.

```
challenge2 = read_csv(readr_example("challenge.csv"),
                      col_types = cols(
                        country = col_factor(),
                        continent = col_factor(),
                        year = col_date(),
                        lifeExp = col_double(),
                        pop = col_double(),
                        gdpPercap = col_double()
                      ))
```

```
## Warning: The following named parsers don't match the column names: country,
## continent, year, lifeExp, pop, gdpPercap
```

```
## Warning: 1000 parsing failures.
##  row col           expected     actual
## 1001   y 1/0/T/F/TRUE/FALSE 2015-01-16 '/home/bergmul/R/x86_64-pc-linux-gnu-library/3.6/readr/extdat
## 1002   y 1/0/T/F/TRUE/FALSE 2018-05-18 '/home/bergmul/R/x86_64-pc-linux-gnu-library/3.6/readr/extdat
## 1003   y 1/0/T/F/TRUE/FALSE 2015-09-05 '/home/bergmul/R/x86_64-pc-linux-gnu-library/3.6/readr/extdat
## 1004   y 1/0/T/F/TRUE/FALSE 2012-11-28 '/home/bergmul/R/x86_64-pc-linux-gnu-library/3.6/readr/extdat
## 1005   y 1/0/T/F/TRUE/FALSE 2020-01-13 '/home/bergmul/R/x86_64-pc-linux-gnu-library/3.6/readr/extdat
## .... ... .................. .......... ..........................................................
## See problems(...) for more details.
```

```
head(challenge2)
```

```
## # A tibble: 6 x 2
##       x y
##   <dbl> <lgl>
## 1   404 NA
## 2  4172 NA
## 3  3004 NA
## 4   787 NA
## 5    37 NA
## 6  2332 NA
```

4 Writing csv files

You can use the `readr` package to write `csv` files as well. In this case, we want to save our cleaned up dataframe as a `csv` file in a `dataframes` sub-folder. For this, we first check whether the sub-folder already exists. If it does not, we create it.

```r
if (!file.exists("dataframes")) {
  dir.create("dataframes")
}
```

Then, we write the `csv` file using the `write_csv` function.

```r
write_csv(challenge2, "./dataframes/challenge2.csv")
```

You can check now in your working directory whether this worked. Note that a `csv` file does not store the information about the correct parsing of the data columns.

```r
read_csv("./dataframes/challenge2.csv")
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_logical()
## )
```

```
## # A tibble: 2,000 x 2
##        x y
##    <dbl> <lgl>
##  1   404 NA
##  2  4172 NA
##  3  3004 NA
##  4   787 NA
##  5    37 NA
##  6  2332 NA
##  7  2489 NA
##  8  1449 NA
##  9  3665 NA
## 10  3863 NA
## # ... with 1,990 more rows
```

We have to specify the correct parsing again! If you only work with `R` and the dataframe is not too big, you can store the dataframe as an `RDS` file instead.

5 Write and read `RDS` files with `readr`

There is not much to say here apart from the fact that the `RDS` file 'remembers' the correct parsing.

```r
write_rds(challenge2, "./dataframes/challenge2.rds")
challenge_rds = read_rds("./dataframes/challenge2.rds")
head(challenge_rds)
```

```
## # A tibble: 6 x 2
##       x y
##   <dbl> <lgl>
## 1   404 NA
## 2  4172 NA
## 3  3004 NA
## 4   787 NA
## 5    37 NA
## 6  2332 NA
```

6 Reading in Excel sheets and Stata data with `readxl` and `haven`

Many datasets are stored in Excel sheets. You can read them in using the `readxl` package. A similar package, `haven`, exists for Stata files. I will not use them here, but can look at the helpfiles for the commands `read_excel` and `read_dta`.

```r
library(readxl)
help(read_excel)
```

```r
library(haven)
help(read_dta)
```

If you want to try them out, you can use the data files from the 'Baby-Wooldridge' here: http://www.cengage.com/aise/economics/wooldridge_3e_datasets/.

Sources

These notes were created by Julian Langer for the 2019 edition of PP4RS (https://pp4rs.github.io/2019-foundations-uzh/) and have only been minorly updated. His exposition is heavily inspired by the notes for a new book on R data science by Garrett Grolemund and Hadley Wickham. You can find detailed outlines here: http://r4ds.had.co.nz.