Plain Text

Jim Harner

6/7/2019

The `readr` package is part of the `tidyverse`. It is used for importing rectangular text files into R as tibbles.

```
library(readr)
```

Material in this section is based partially on material in Paul Murrell's Introduction to Data Technologies.

2.1 Plain Text Files

Plain text files are the simplest way to store data. Generally, the data is stored in rows representing observations (or records) with columns representing variables (or fields). The beginning of the file may contain **metadata**, i.e., information about the data. It is sometimes called the **header**, which may represent the variable names.

The **data model** is the representation of the data structure. Plain text files often lack metadata, e.g., the delimiters. However, it is common for the first row to represent the variable names, particularly in cases where fields are separated by commas.

For example, consider the Pacific Pole of Inaccessibility data from `pointnemotemp.txt`.

```
             VARIABLE : Mean TS from clear sky composite (kelvin)
             FILENAME : ISCCPMonthly_avg.nc
             FILEPATH : /usr/local/fer_dsets/data/
             SUBSET   : 93 points (TIME)
             LONGITUDE: 123.8W(-123.8)
             LATITUDE : 48.8S
                        123.8W
                         23
 16-JAN-1994 00 /  1:  278.9
 16-FEB-1994 00 /  2:  280.0
 16-MAR-1994 00 /  3:  278.9
 16-APR-1994 00 /  4:  278.9
 16-MAY-1994 00 /  5:  277.8
 16-JUN-1994 00 /  6:  276.1
```

The first eight rows are metadata. The data of interest is in the first column (date) and the last column (temperature) for the space-delimited file.

The following are two subtypes of plain text files depending on how values are separated:

- Delimited formats: values in a row are separated by a special character (one or more spaces, tabs, commas, etc.);
- Fixed-width formats: each value in a row is allocated a fixed number of characters.

In both cases, all information is stored as text.

2.1.1 Advantages and Disadvantages of Plain Text Files

The plain text format is simple and portable. However, the disadvantages for using rows and columns are that:

- representing complex data structures is difficult;
- storing everything as characters is inefficient in terms of memory.

Hierarchical or **multi-level** or **stratified** data are complex and cannot easily be stored as plain text. For example, consider family data which has two types of objects: parents and children. Data models that do not satisfy the **relational model** (see chapter 2, section 4) are often inefficient due to repetition of values, i.e., the data representation violates the DRY (Don't Repeat Yourself) principle. Thus, using a plain text format should only be used for relatively small flat files.

The `read.table` base R function can be used to read the `pointnemotemp.txt` text file. For data science applications, it is better to use the `read_table` function (or the `read_fwf` function if the columns are in fixed positions) in the `readr` package (part of Hadley Wickham's **Tidyverse**) to read the text file. This is a very good choice if the file is large since the result is a tibble—not a `data.frame`. Although a tibble inherits from a `data.frame`, it both enhances and restricts the behavior of a `data.frame`. Only the first 10 rows are printed and the type of each column is given.

```
nemotemp_tbl <- read_table("pointnemotemp.txt", skip=8,
                           col_types = c("c---d"),
                           col_names = c("date", "temp"))
class(nemotemp_tbl)
```

```
## [1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

```
nemotemp_tbl
```

```
## # A tibble: 93 x 2
##    date         temp
##    <chr>       <dbl>
##  1 16-JAN-1994  279.
##  2 16-FEB-1994  280
##  3 16-MAR-1994  279.
##  4 16-APR-1994  279.
##  5 16-MAY-1994  278.
##  6 16-JUN-1994  276.
##  7 16-JUL-1994  276.
##  8 16-AUG-1994  276.
##  9 16-SEP-1994  276.
## 10 16-OCT-1994  277.
## # ... with 83 more rows
```

The `col_types` are given by `c` = character, `-` = skip, and `d` = double. To be useful we need to convert `date` to an R Date format since the `D` column type does not work here. The `lubridate` R package provides an easy way to convert strings to R dates (or times or date-times). For now we will use the `parse_date` function in the `readr` package.

```
nemotemp_tbl$date <- parse_date(nemotemp_tbl$date, format = "%d-%b-%Y")
nemotemp_tbl
```

```
## # A tibble: 93 x 2
##    date        temp
##    <date>     <dbl>
##  1 1994-01-16  279.
##  2 1994-02-16  280
##  3 1994-03-16  279.
##  4 1994-04-16  279.
##  5 1994-05-16  278.
##  6 1994-06-16  276.
```

```
##  7 1994-07-16  276.
##  8 1994-08-16  276.
##  9 1994-09-16  276.
## 10 1994-10-16  277.
## # ... with 83 more rows
```

The result is a `POSIXct` vector with a time zone attribute. In the `format` argument, `%d` is a 2-digit specification for day, `%b` is the abbreviated name for month, and `%Y%` is the 4-digit representation for year. At this point we can do arithmetical operations on `date`. For a complete list of formatting options for dates (and times) see the documentation for `parse-datetime`.

A regular data frame can be converted to a tibble by the `as_tibble` function in the `tibble` package, which is part of the `tidyverse`. The packages in the `tidyverse` form a coherent approach to data manipulation and analyses using R, particularly for big data.

Using one or more blanks to separate fields is dangerous if there is missing data (not the case here). We can write comma-separated data to a file using `write_csv`.

2.1.2 CSV Files

The CSV (Comma-Separated Value) format is a special case of the plain text format. It is very reliable and common and solves the problem of where the fields are in a row.

Rules for CSV files:

- Comma delimited: each field is separated by a comma;
- Double-quotes are special: fields containing commas are enclosed by double quotes;
- Double-quote escape sequence: fields containing double-quotes must be surrounded by double-quotes and each embedded double-quote must be represented using two double-quotes;
- Header information: an optional single header line can contain the names of the fields.

CSV files are often used for transferring data from spreadsheets. The `write_csv` function in the `readr` package is used for writing a data frame or tibble to the local file system.

```
dir <- getwd()
write_csv(nemotemp_tbl, file.path(dir, "nemotemp.csv"))
```

We can then read it back in using `read_csv`, which uses `,` as a separator by default. The `read_csv` function in the `readr` package results in a tibble and is preferred over base R's `read.csv` for big data.

```
nemotemp_csv <- read_csv("nemotemp.csv")
```

```
## Parsed with column specification:
## cols(
##   date = col_date(format = ""),
##   temp = col_double()
## )
```

```
nemotemp_csv
```

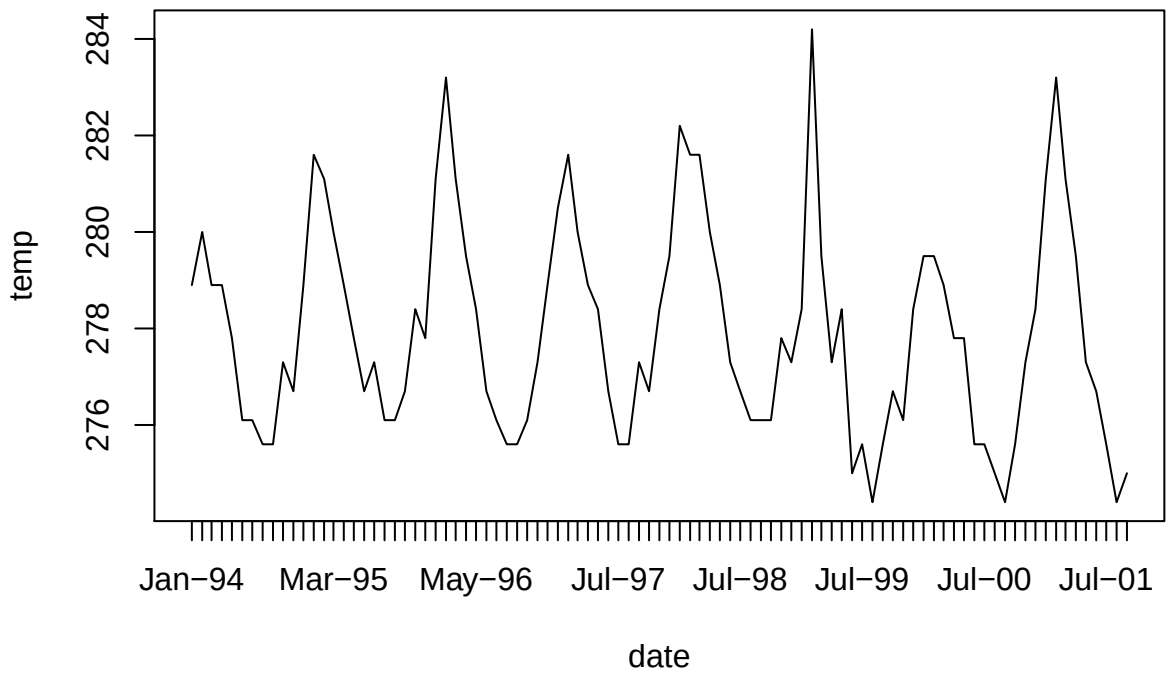
```
## # A tibble: 93 x 2
##    date        temp
##    <date>     <dbl>
##  1 1994-01-16  279.
##  2 1994-02-16  280
##  3 1994-03-16  279.
##  4 1994-04-16  279.
```

```
##  5 1994-05-16  278.
##  6 1994-06-16  276.
##  7 1994-07-16  276.
##  8 1994-08-16  276.
##  9 1994-09-16  276.
## 10 1994-10-16  277.
## # ... with 83 more rows
```

Here the column types were determined by default. Note that the write-read sequence preserved the `date` format.

Once `date` has been converted to an R date format, the time series for temperature is easy to plot.

```
attach(nemotemp_csv)
plot(date, temp, xaxt = "n", type = "l")
axis(1, at = date, labels = format(date, "%b-%y"), cex.axis = 1)
```



This was done with R base graphics. In a later chapter, time series plots will be done with ggplot2.

Underlying the printed representation of a date is a numeric value equal to the number of days since Jan. 1, 1970 (a UNIX convention).

```
head(data.frame(date, as.numeric(date)))
```

```
##         date as.numeric.date.
## 1 1994-01-16             8781
## 2 1994-02-16             8812
## 3 1994-03-16             8840
## 4 1994-04-16             8871
## 5 1994-05-16             8901
```

```
## 6 1994-06-16             8932
```

2.1.3 Text Encodings

How do lines end? UNIX-based systems differ from Windows.

- macOS and Linux: `\n`
- Windows: `\r\n`

Many programs convert automatically, but this is why Mac and Linux users sometimes see emailed text as double spaced.

How a single character is stored in memory is called **character encoding**. Originally, data was stored using the ASCII (American Standard Code for Information Interchange) encoding. In this encoding data is stored as characters and each character is stored in memory as a byte (8 bits). Only $2^8 = 256$ characters can be represented.

Other languages have special characters, e.g., Latin1 and Latin2 are not compatible. Asian and other characters use **multi-byte** encoding.

UNICODE lets computers work with all characters in all languages. Every character has its own number called a **code point** often coded as `U+xxxxxx`, where `x` is a hexadecimal digit.

There are two encodings to store a UNICODE code point in memory:

- UTF-16: two bytes per character;
- UTF-8: one or more bytes per character depending on the character.

For ASCII, UTF-8 uses one byte per character. UTF-8 is the most common encoding.