JSON

Jim Harner

1/6/2020

The `jsonlite` package is part of the `tidyverse`. It is a powerful parser for statistical data. `jsonlite` makes it easy to build data pipelines and to use the web APIs.

```
library(jsonlite)
```

The material for the section is extracted from: the jsonlite package vignette, by Jeroen Ooms.

2.2 JSON

JavaScript Object Notation (JSON) is a text format for the serialization of structured data. The design of JSON is a simple and concise text-based format, particularly when compared to XML. The syntax is:

- easy for humans to read and write,
- easy for machines to parse and generate, and
- completely described in a single page here.

The character encoding of JSON text is Unicode, using UTF-8 by default, making it naturally compatible with non-latin alphabets. Over the past years, JSON has become hugely popular on the internet as a general purpose data interchange format. High quality parsing libraries are available for almost any programming language. Several R packages are available to assist the user in generating, parsing and validating JSON through CRAN, including `rjson`, `RJSONIO`, and `jsonlite`.

The emphasis of this section is to show how R data structures are most naturally represented in JSON. This is not a trivial problem, particularly for complex or relational data as they frequently appear in statistical applications. Several R packages implement `toJSON` and `fromJSON` functions which directly convert R objects into JSON and vice versa. However, the exact mapping between the various R data classes and JSON structures is not self evident. The most basic data structures in R do not perfectly map to their JSON counterparts, and leave some ambiguity for edge cases.

The JSON Specification

The JSON format specifies 4 primitive types (`string`, `number`, `boolean`, `null`) and two universal structures:

- A JSON object: an unordered collection of zero or more name-value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.
- A JSON array: an ordered sequence of zero or more values.

These structures are heterogeneous; i.e. they are allowed to contain elements of different types. Therefore, the native R realization of these structures is a **named** list for JSON objects, and **unnamed** list for JSON arrays.

However, a list is an awkward and inefficient type for storing and manipulating data in R. Most statistical applications work with (homogeneous) vectors and matrices or with data frames. In order to give these data structures a JSON representation, we can define certain special cases of JSON structures which get parsed into specific R types. For example, one convention which all current implementations have in common is that a homogeneous array of primitives gets parsed into an atomic vector instead of a list.

```
txt <- '[12, 3, 7]'
x <- fromJSON(txt)
is(x)
```

```
## [1] "integer"              "double"               "numeric"
## [4] "vector"               "data.frameRowLabels"
```

```
x
```

```
## [1] 12  3  7
```

This is reasonable, but this simplification can compromise type-safety for dynamic data.

For example, suppose an R package uses `fromJSON` to pull data from a JSON API on the web and that for some particular combination of parameters the result includes a null value, e.g., `[12, null, 7]`.

```
txt <- '[12, null, 7]'
x <- fromJSON(txt)
x
```

```
## [1] 12 NA  7
```

When relying on JSON as a data interchange format, the behavior of the parser must be consistent and unambiguous. Clients relying on JSON to get data in and out of R must know exactly what to expect in order to facilitate reliable communication, even if the content of the data is dynamic.

Similarly, R code using dynamic JSON data from an external source is only reliable when the conversion from JSON to R is consistent. In the example above, we could argue that instead of falling back on a list, the array is more naturally interpreted as a numeric vector where the `null` becomes a missing value (`NA`).

We now look at examples of how the common R classes are represented in JSON. The `toJSON` function relies on method dispatch, which means that objects get encoded according to their class attribute. If an object has multiple class values, R uses the first occurring class which has a `toJSON` method. If none of the classes of an object has a `toJSON` method, an error is raised. In some cases we reverse the operation using `fromJSON`.

2.2.1 Atomic Vectors

The most basic data type in R is the atomic vector. Atomic vectors hold an ordered, homogeneous set of values of type `logical` (booleans), `character` (strings), `raw` (bytes), `numeric` (doubles), `complex` (complex numbers with a real and imaginary part), or `integer`.

Because R is fully vectorized, there is no user level notion of a primitive: a scalar value is considered a vector of length 1. Atomic vectors map to JSON arrays:

```
x <- c(1, 2, pi)
toJSON(x)
```

```
## [1,2,3.1416]
```

The JSON array is the only appropriate structure to encode a vector, even though vectors in R are homogeneous, whereas the JSON array is actually heterogeneous, but JSON does not make this distinction.

Missing values

Statistical data often have missing values: a concept foreign to many other languages. Besides regular values, each vector type in R except for `raw` can hold `NA` as a value. Vectors of type `double` and `complex` define three additional types of non finite values: `NaN`, `Inf` and `-Inf`. The JSON format does not natively support any of these types; therefore, such values values need to be encoded in some other way. There are two obvious approaches:

- use the JSON null type;
- encode missing values as strings by wrapping them in double quotes.

Both methods result in valid JSON, but both have a limitation:

- the problem with the null type is that it is impossible to distinguish between different types of missing data (e.g., `Inf`, `NA`), which could be a problem for numeric vectors.
- The problem with encoding missing values as strings is that this method can not be used for character vectors, because the user won't be able to distinguish the actual string "NA" and the missing value `NA`.

`jsonlite` uses the following defaults:

- Missing values in non-numeric vectors (`logical`, `character`) are encoded as `null`.
- Missing values in numeric vectors (`double`, `integer`, `complex`) are encoded as strings.

```
toJSON(c(TRUE, NA, NA, FALSE))
```

```
## [true,null,null,false]
```

```
toJSON(c("FOO", "BAR", NA, "NA"))
```

```
## ["FOO","BAR",null,"NA"]
```

```
toJSON(c(3.14, NA, NaN, 21, Inf, -Inf))
```

```
## [3.14,"NA","NaN",21,"Inf","-Inf"]
```

Note that we can specify a `na` argument to override the defaults.

```
toJSON(c(3.14, NA, NaN, 21, Inf, -Inf), na = "null")
```

```
## [3.14,null,null,21,null,null]
```

Special vector types: dates, times, and factors

Besides missing values, JSON also lacks native support for some of the basic vector types in R. These include vectors of class `Date`, `POSIXt` (timestamps), and `factor`. By default, the `jsonlite` package coerces these types to strings (using `as.character`).

2.2.2 Matrices

R has the ability to interface with basic linear algebra subprograms such as `LAPACK`. These libraries provide well tuned, high performance implementations of important linear algebra operations for calculating inner products, eigenvalues, singular value decompositions, etc. These are building blocks of statistical methods such as linear regression and principal component analysis.

Linear algebra methods operate on matrices, which are 2-dimensional structures of homogeneous values as discussed previously.

```
x <- matrix(1:12, nrow=3, ncol=4)
print(x)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

A matrix is stored in memory as a single atomic vector with an attribute called `dim` defining the dimensions of the matrix. The product of the dimensions is equal to the length of the vector.

Even though the matrix is stored as a single vector, the way it is printed and indexed makes it conceptually a 2 dimensional structure. In `jsonlite` a matrix maps to an array of equal-length subarrays:

```
x <- matrix(1:12, nrow=3, ncol=4)
(x.json <- toJSON(x))
```

```
## [[1,4,7,10],[2,5,8,11],[3,6,9,12]]
```

We can reconstruct the matrix in R from JSON.

```
fromJSON(x.json)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Because the matrix is implemented in R as an atomic vector, it automatically inherits the conventions mentioned earlier with respect to edge cases and missing values.

Matrix row and column names

Besides the `dim` attribute, the `matrix` class has an additional, optional attribute: `dimnames`. This attribute holds names for the rows and columns in the matrix. However, this information is not in the default JSON mapping for matrices for several reasons.

1. This attribute is optional so either row or column names or both could be `NULL`.
2. The names in matrices are mostly there for annotation only; they are not actually used in calculations.

When row or column names of a matrix contain vital information, we might want to transform the data into a more appropriate structure. Wickham calls this "tidying" the data.

In the following example, the predictor variable (treatment) is stored in the column headers rather than the actual data. As a result, these values do not get included in the JSON output:

```
x <- matrix(c(NA, 1, 2, 5, NA, 3), nrow=3)
row.names(x) <- c("Joe", "Jane", "Mary");
colnames(x) <- c("Treatment A", "Treatment B")
print(x)
```

```
##      Treatment A Treatment B
## Joe           NA           5
## Jane           1          NA
## Mary           2           3
```

```
toJSON(x)
```

```
## [["NA",5],[1,"NA"],[2,3]]
```

Wickham recommends that the data be melted into its tidy form. Once the data is tidy, the JSON encoding will naturally contain the treatment values:

```
library(reshape2)
y.df <- melt(x, varnames=c("Subject", "Treatment"))
y.df
```

```
##   Subject   Treatment value
## 1     Joe Treatment A    NA
```

```
## 2    Jane Treatment A     1
## 3    Mary Treatment A     2
## 4     Joe Treatment B     5
## 5    Jane Treatment B    NA
## 6    Mary Treatment B     3
```

```
(y.json <- toJSON(y.df, pretty=TRUE))
```

```
## [
##   {
##     "Subject": "Joe",
##     "Treatment": "Treatment A"
##   },
##   {
##     "Subject": "Jane",
##     "Treatment": "Treatment A",
##     "value": 1
##   },
##   {
##     "Subject": "Mary",
##     "Treatment": "Treatment A",
##     "value": 2
##   },
##   {
##     "Subject": "Joe",
##     "Treatment": "Treatment B",
##     "value": 5
##   },
##   {
##     "Subject": "Jane",
##     "Treatment": "Treatment B"
##   },
##   {
##     "Subject": "Mary",
##     "Treatment": "Treatment B",
##     "value": 3
##   }
## ]
```

We can reconstruct the data frame in R from JSON.

```
fromJSON(y.json)
```

```
##   Subject   Treatment value
## 1     Joe Treatment A    NA
## 2    Jane Treatment A     1
## 3    Mary Treatment A     2
## 4     Joe Treatment B     5
## 5    Jane Treatment B    NA
## 6    Mary Treatment B     3
```

The `melt` function will be discussed in Module 3.

2.2.3 Lists

The `list` is the most general purpose data structure in R. It holds an ordered set of elements, including other lists, each of arbitrary type and size. Two types of lists are distinguished: *named* lists and *unnamed* lists. A list is considered named if it has an attribute called `names`. In practice, a named list is any list for which we can access an element by its name, whereas elements of an unnamed list can only be accessed using their index number:

```
mylist1 <- list("foo" = 123, "bar"= 456)
mylist1$bar
```

```
## [1] 456
```

```
mylist2 <- list(123, 456)
mylist2[[2]]
```

```
## [1] 456
```

Unnamed lists

Just like vectors, an unnamed list maps to a JSON array:

```
toJSON(list(c(1,2), "test", TRUE, list(c(1,2))))
```

```
## [[1,2],["test"],[true],[[1,2]]]
```

Note that even though both vectors and lists are encoded using JSON arrays, they can be distinguished from their contents:

- an R vector results in a JSON array containing only primitives;
- a list results in a JSON array containing only objects and arrays.

This allows the JSON parser to reconstruct the original type from encoded vectors and arrays:

```
x <- list(c(1,2,NA), "test", FALSE, list(foo="bar"))
x
```

```
## [[1]]
## [1]  1  2 NA
##
## [[2]]
## [1] "test"
##
## [[3]]
## [1] FALSE
##
## [[4]]
## [[4]]$foo
## [1] "bar"
```

```
(y <- toJSON(x))
```

```
## [[1,2,"NA"],["test"],[false],{"foo":["bar"]}]
```

```
fromJSON(y)
```

```
## [[1]]
## [1]  1  2 NA
##
```

```
## [[2]]
## [1] "test"
##
## [[3]]
## [1] FALSE
##
## [[4]]
## [[4]]$foo
## [1] "bar"
```

The only exception is the empty list and empty vector, which are both encoded as `[ ]` and therefore indistinguishable.

Named lists

A named list in R maps to a JSON object:

```
toJSON(list(foo=c(1,2), bar="test"))
```

```
## {"foo":[1,2],"bar":["test"]}
```

Because a list can contain other lists, this works recursively:

```
toJSON(list(foo=list(bar=list(baz=pi))))
```

```
## {"foo":{"bar":{"baz":[3.1416]}}}
```

Named lists map almost perfectly to JSON objects with one exception: list elements can have empty names:

```
x <- list(foo=123, "test", TRUE)
attr(x, "names")
```

```
## [1] "foo" ""    ""
```

```
x$foo
```

```
## [1] 123
```

```
x[[2]]
```

```
## [1] "test"
```

In a JSON object, each element in an object must have a valid name. To ensure this property, `jsonlite` uses the same solution as the `print` method, which is to fall back on indices for elements that do not have a proper name:

```
x <- list(foo=123, "test", TRUE)
x
```

```
## $foo
## [1] 123
##
## [[2]]
## [1] "test"
##
## [[3]]
## [1] TRUE
```

```
toJSON(x)
```

```
## {"foo":[123],"2":["test"],"3":[true]}
```

This behavior ensures that all generated JSON is valid. However, named lists with empty names should be avoided.

2.2.4 Data Frames

The data frame is the most used data structure in R. This class holds tabular data in which each column is named and (usually) homogeneous. Conceptually it is very similar to a table in relational data bases such as PostgreSQL, where fields are referred to as column names, and records are called rows.

Like a matrix, a data frame can be subsetted with two indices, to extract certain rows and columns of the data:

```
is(iris)
```

```
## [1] "data.frame" "list"       "oldClass"   "vector"
```

```
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
# select columns by indices
iris[1:3, c(1,5)]
```

```
##   Sepal.Length Species
## 1          5.1  setosa
## 2          4.9  setosa
## 3          4.7  setosa
```

The behavior of `jsonlite` is designed for compatibility with conventional ways of encoding table-like structures outside the R community. The implementation is more involved, but results in a powerful and more natural way of representing data frames in JSON.

Column based versus row based tables

Generally speaking, tabular data structures can be implemented in two different ways:

- A column based way. A column based structure is efficient for inserting or extracting certain columns of the data, but it is inefficient for manipulating individual rows.
- A row based way. For row-based implementations we can easily manipulate a particular record, but to insert/extract a whole column we would need to iterate over all records in the table and read/modify the appropriate field in each of them.

The data frame class in R is implemented in a column based fashion: it consists of a named list of equal-length vectors. Thereby the columns in the data frame naturally inherit the properties from atomic vectors discussed before, such as homogeneity, missing values, etc.

Another argument for column-based implementation is that statistical methods generally operate on columns. For example, the `lm` function fits a linear regression by extracting the columns from a data frame as specified

by the `formula` argument. R simply binds the specified columns together into a matrix `X` and calls out to a highly optimized FORTRAN subroutine to calculate the OLS estimates.

Unfortunately R is an exception in its preference for column-based storage: most languages, systems, databases, API's, etc, are optimized for record based operations. For this reason, the conventional way to store and communicate tabular data in JSON seems to almost exclusively be row based. This discrepancy presents various complications when converting between data frames and JSON.

Row based data frame encoding

The encoding of data frames is one of the major differences between `jsonlite` and implementations from other currently available packages. Instead of using the column-based encoding also used for lists, `jsonlite` maps data frames by default to an array of records:

```
(x <- toJSON(iris[1:2,], pretty=TRUE))
```

```
## [
##   {
##     "Sepal.Length": 5.1,
##     "Sepal.Width": 3.5,
##     "Petal.Length": 1.4,
##     "Petal.Width": 0.2,
##     "Species": "setosa"
##   },
##   {
##     "Sepal.Length": 4.9,
##     "Sepal.Width": 3,
##     "Petal.Length": 1.4,
##     "Petal.Width": 0.2,
##     "Species": "setosa"
##   }
## ]
```

```
fromJSON(x)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
```

This output looks a bit like a list of named lists. However, there is one major difference: the individual records contain JSON primitives, whereas lists always contain JSON objects or arrays:

```
toJSON(list(list(Species="Foo", Width=21)), pretty=TRUE)
```

```
## [
##   {
##     "Species": ["Foo"],
##     "Width": [21]
##   }
## ]
```

This leads to the following convention when encoding R objects:

- JSON primitives only appear in vectors and data-frame rows;
- primitives within a JSON array indicate a vector;

- primitives inside a JSON object indicate a data-frame row;
- a JSON encoded list, (named or unnamed) will never contain JSON primitives.

Missing values in data frames

The section on atomic vectors discussed two methods of encoding missing data appearing in a vector. Use:

- strings, or
- the JSON null type.

When a missing value appears in a data frame, there is a third option:

- do not include this field in the JSON record.

```
x <- data.frame(foo=c(FALSE, TRUE, NA, NA),
                bar=c("Aladdin", NA, NA, "Mario"))
x
```

```
##     foo     bar
## 1 FALSE Aladdin
## 2  TRUE    <NA>
## 3    NA    <NA>
## 4    NA   Mario
```

```
(y <- toJSON(x, pretty=TRUE))
```

```
## [
##   {
##     "foo": false,
##     "bar": "Aladdin"
##   },
##   {
##     "foo": true
##   },
##   {},
##   {
##     "bar": "Mario"
##   }
## ]
```

```
fromJSON(y)
```

```
##     foo     bar
## 1 FALSE Aladdin
## 2  TRUE    <NA>
## 3    NA    <NA>
## 4    NA   Mario
```

The default behavior of `jsonlite` is to omit missing data from records in a data frame. This seems to be the most conventional method used on the web, and we expect this encoding will most likely lead to the correct interpretation of missingness, even in languages without an explicit notion of `NA`.