Paper CT01

Decoding in R

Himani Narang, Novartis Healthcare Pvt. Ltd., Hyderabad, India

ABSTRACT

Creation of ADaM datasets often requires generating variables by decoding values from codelists metadata as per CDISC submissions values. The newly created variables can be either in numeric or character formats, and creating them individually by merging with codelists dataset can be a cumbersome task. Many programming languages contain data structures that allow storing key-value pairs that can make this task easier. On the other hand, R does not have a direct built-in data structure for storing key-value pairs. However, other approaches can be used to mimic the functionality, such as named vectors and named lists. We have created a function in R named decode() which creates named vectors from codelists dataset. By simply accessing the vectors in a repetitive manner using mapply() function, the input values, or the keys, are decoded to the desired values and multiple variables in different formats can be generated at the same time.

INTRODUCTION

Many programming languages contain data structures that allow storing key-value pairs. These data structures are known by different names in different languages, such as associative arrays, maps, symbol tables and dictionaries, but have a common functionality. They allow easy access of values with the help of unique keys. Unlike many other languages, R does not have a direct built-in data structure for storing key-value pairs.

Data structures containing key-value pairs have a wide range of applications in various domains. One such application in the clinical domain is decoding of raw data to controlled terminology as per CDISC standards during the creation of ADaM datasets. Codelists metadata can be used for this purpose, which would contain all possible raw values as well as their corresponding decoded values in different formats, as shown in Figure 1. The different kinds of decoded values can be extended terms, numeric codes or abbreviated values, depending on the user's requirements. Storing the information as key-value pairs can help the user for easy conversion of the keys to any desired format.

(a)

CODELIST Codelist name	KEY SDTM	VALE Extended	VALN Numeric	VALA Abbreviated
COUNTRY	ABW	Aruba	1	ABW
COUNTRY	AFG	Afghanistan	2	AFG
COUNTRY	AGO	Angola	3	AGO
COUNTRY	AIA	Anguilla	4	AIA
COUNTRY	ALA	Aland Islands	5	ALA
COUNTRY	ALB	Albania	6	ALB

(b)

CODELIST Codelist name	KEY SDTM	VALE Extended	VALN Numeric	VALA Abbreviated
RACE	CAUCASIAN	Caucasian	10	CA
RACE	BLACK	Black	20	BL
RACE	ASIAN	Asian	30	AS
RACE	NATIVE AMERICAN	Native American	40	NA
RACE	PACIFIC ISLANDER	Pacific Islander	50	PI
RACE	AMERICAN INDIAN OR ALASKA NATIVE	American Indian or Alaska Native	60	AIAN
RACE	BLACK OR AFRICAN AMERICAN	Black or African American	70	BLAA

Figure 1. Snapshots from codelists metadata containing codelist name, raw values from SDTM to be used as keys, and controlled term values in different formats (Extended, Numeric and Abbreviated)

IMPLEMENTING KEY-VALUE PAIRS IN R

Even though there isn't a direct data structure in R for storing key-value pairs, there are a number of other approaches that can be used for the same:

1. **R Hashed Environment:** An environment object generated from the env() function has a hash table structure that allows storage and retrieval of key-value pairs. Even though this approach possesses the efficiency of a hash table, the implementation of an environment is slightly more complex as compared to many other data structures in R. Moreover, the syntax requires each key-value pair to be inserted individually, therefore, it is not as straightforward to create a large number of key-value pairs derived from a dataset.
2. **Named Lists and Vectors:** The primary difference between a named list/vector and a regular list/vector in R is that a named list/vector allows accessing values using names, as well as indices, as opposed to just indices in a regular list/vector. This makes them useful in generating key-value pairs with easy retrieval using the names of individual elements.
3. **R Packages – dict, hash, ht:** Since R is an open-source software, users have developed various packages that allow the storage of key-value pairs in the form of hash tables. Some of these packages are dict, hash and ht packages, each of them possessing their own unique strengths.

This is not a comparative paper and will not discuss the individual strengths and weaknesses of the mentioned approaches. While some of the newer packages may offer better efficiency and advanced solutions, this paper focuses on the flexibility and elegance of using a simple vector for achieving complex tasks. The aim of this paper is to highlight the use of named vectors for easy storage and retrieval of key-value pairs, with the help of a user-defined function named decode().

DECODE() FUNCTION

We have developed a function in R called decode() which involves creation of named vector from codelist metadata, and uses it to decode raw values to controlled terms. The function requires the data.table package to be loaded, and codelist_metadata dataset to be present in the global environment as a data.table object. This dataset must contain all possible raw values and their corresponding decoded values for each codelist, as shown in Figure 1.

ARGUMENTS

```
decode(codelist1, variable, intype, outtype)
```

- **codelist1:** String value containing the name of the codelist to be used. For example, 'COUNTRY', 'RACE' etc.
- **variable:** Name of variable containing the raw value. The user can also pass the raw value as a string to this argument. For example, 'USA'.
- **intype:** String value containing column name of codelists_metadata dataset that stores the raw values. For example, 'KEY'.
- **outtype:** String value containing column name of codelists_metadata dataset that stores the required decoded values. For example, 'VALE' for extended values, and 'VALA' for abbreviated values.

FUNCTIONALITY

The decode() function broadly consists of 3 main steps – Filter, Assignment and Retrieval.

```
decode <- function(codelist1, variable, intype, outtype){

    # Step 1 - Filter
    codelist_metadata2 = codelist_metadata[codelist == codelist1]

    # Step 2 - Assignment
    temp = setNames(codelist_metadata2[[outtype]], codelist_metadata2[[intype]])

    # Step 3 - Retrieval
    out = temp[variable]

    # If no match found, assign blank or NA
    if(is.null(out) == T && mode(codelist_metadata2[[outtype]]) == 'character'){ out
    = '' } else if
```

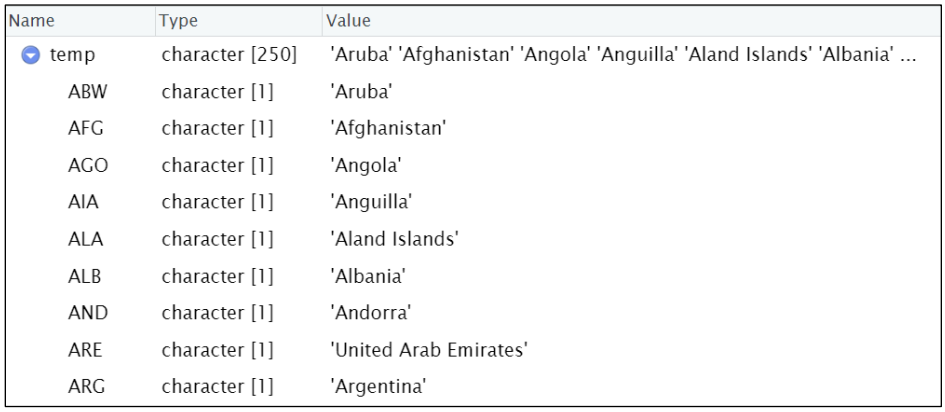
```
        (is.null(out) == T && mode(codelist_metadata2[[outtype]]) == 'numeric'){ out = NA
        }

        return(out)
}
```

Step 1 – Filter

The codelist_metadata dataset is filtered to retain only the observations for the particular codelist that the user passed to the function.

Step 2 – Assignment

The columns passed to the *outtype* and *intype* arguments are extracted from the codelist_metadata dataset, and the values in *intype* column are used to assign names to values in *outtype* using the setNames() function. This creates a named vector containing the desired key-value pairs, as shown in Figure 2.



Name	Type	Value
temp	character [250]	'Aruba' 'Afghanistan' 'Angola' 'Anguilla' 'Aland Islands' 'Albania' ...
ABW	character [1]	'Aruba'
AFG	character [1]	'Afghanistan'
AGO	character [1]	'Angola'
AIA	character [1]	'Anguilla'
ALA	character [1]	'Aland Islands'
ALB	character [1]	'Albania'
AND	character [1]	'Andorra'
ARE	character [1]	'United Arab Emirates'
ARG	character [1]	'Argentina'

Figure 2. Named vector created using setNames() function

Step 3 – Retrieval

In a named vector, the individual elements can be accessed by passing the names in square brackets as follows:

```
vector[name]
```

Using this approach, the raw value passed in the *variable* argument is used as a name to obtain its decoded value from the vector.

In case there are no matches found, the decode() function attempts to return NULL, which gives an error in certain scenarios. To avoid this, an additional set of conditions is added to the function, to return a blank string if outtype is of character type, and to return NA if outtype is of numeric type.

EXAMPLES

```
library(data.table)
library(haven)

codelist_metadata = data.table(read_sas('codelist_metadata.sas7bdat'))

# Original value
orig_val = 'IND'

# Original value decoded to extended values
decoded_val = decode(codelist1 = 'COUNTRY', variable = orig_val, intype = 'KEY',
                     outtype = 'VALE')
```

```
# Original value decoded to numeric values (Character to numeric decoding)
decoded_valn = decode(codelist1 = 'COUNTRY', variable = orig_val, intype = 'KEY',
                      outtype = 'VALN')

# Printing decoded values
decoded_val
[1] "India"

decoded_valn
[1] 104
```

GENERATING MULTIPLE DECODED COLUMNS AT ONCE

Creation of ADaM datasets can often require creation of multiple decoded columns from different variables at once using the codelist metadata. Creating each column one by one by calling the function can make the program lengthy and repetitive. The apply family of functions are useful in such scenarios to perform repetitive tasks on different chunks of data. There are a number of functions available under this family for the users to choose from based on their requirements. In this particular scenario, the mapply() function is useful, because it allows the user to pass different arguments for each column the function is applied to.

```
# Vector of new column names to be created from decode function
   cols_after_decode = c('COUNTRYL', 'COUNTRYN', 'COUNTRYS', 'ETHNICL', 'ETHNICN',
                         'ETHNICS', 'RACEL', 'RACEN', 'RACES', 'SEXL', 'SEXN', 'SEXS')

# Creating all decoded values from COUNTRY, ETHNIC, RACE and SEX columns in a single
step
   adsl = dm[, (cols_after_decode) :=
                    mapply(decode,
                           codelist1   =   c(rep('COUNTRY',   3),      rep('ETHNIC',   3),
                           rep('RACE', 3), rep('SEX', 3)),
                           variable = c(rep(COUNTRY, 3), rep(ETHNIC, 3), rep(RACE, 3),
                           rep(SEX, 3)),
                           intype = 'KEY',
                           outtype = c('VALE', 'VALN','VALA'),
                           SIMPLIFY = FALSE),
               by = row.names(dm)]
```

The function can be called inside a data table to create multiple new columns by passing the required arguments for each new column as vectors. The mapply() function returns a matrix by default, that cannot be inserted within a data table. By setting the *SIMPLIFY* argument to FALSE, the function can be modified to return a list instead, which can easily be inserted into a data table as new columns, as shown in Figure 3.

	COUNTRYL	COUNTRYN	COUNTRYS	ETHNICL	ETHNICN	ETHNICS	RACEL	RACEN	RACES	SEXL	SEXN	SEXS
1	United States	232	USA	Not Hispanic or Latino	281	NONHIS	White	90	WH	Male	1	M
2	United States	232	USA	Hispanic or Latino	100	HIS	White	90	WH	Female	2	F
3	United States	232	USA	Hispanic or Latino	100	HIS	White	90	WH	Female	2	F
4	United States	232	USA	Not Hispanic or Latino	281	NONHIS	White	90	WH	Male	1	M
5	United States	232	USA	Hispanic or Latino	100	HIS	White	90	WH	Female	2	F
6	United States	232	USA	Not Hispanic or Latino	281	NONHIS	Black or African American	70	BLAA	Female	2	F
7	United States	232	USA	Not Hispanic or Latino	281	NONHIS	Black or African American	70	BLAA	Female	2	F

Figure 3. Generating multiple columns at once using decode inside mapply()

CONCLUSION

Named vectors in R are useful to generate key-value pairs that can help the users to access values through their unique names. The decode() function uses this principle to decode values from codelist metadata during creation of ADaM datasets. While the function is designed only to generate controlled terminologies as per CDISC standards, the applications of this approach aren't just limited to this. Various implementations of key-value pairs have been shown to be useful in many domains. Even though R doesn't have an in-built data structure for storing key-value pairs, exploring different approaches for their implementation can open a number of possibilities for R users.

REFERENCES:

1. R Core Team (2020). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.
2. Matt Dowle and Arun Srinivasan (2021). data.table: Extension of `data.frame`. R package version 1.14.0. https://CRAN.R-project.org/package=data.table
3. Eduardo Arino de la Rubia (2016). Quick Benchmark of Hash Table Implementations in R. https://www.dominodatalab.com

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Himani Narang
Statistical Programmer
Novartis Healthcare Private Limited
Salarpuria-Sattva Knowledge City,
Raidurg, Rangareddy District,
Hyderabad - 500081
India
Email: himani.narang@novartis.com