




rdata: A Python library for R datasets

Carlos Ramos-Carreño ¹ and Tuomas Rossi ²

1 Universidad Autónoma de Madrid, Spain 2 CSC – IT Center for Science Ltd., Finland

DOI: [10.21105/joss.07540](https://doi.org/10.21105/joss.07540)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Arfon Smith](#) 

Reviewers:

- [@has2k1](#)
- [@rich-iannone](#)

Submitted: 27 October 2024

Published: 01 December 2024

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](#)).

Summary

Research work usually requires the analysis and processing of data from different sources. Traditionally in statistical computing, the R language has been widely used for this task, and a huge amount of datasets have been compiled in the Rda and Rds formats, native to this programming language. As these formats contain internally the representation of R objects, they cannot be directly used from Python, another widely used language for data analysis and processing. The library `rdata` allows to load and convert these datasets to Python objects, without the need of exporting them to other intermediate formats which may not keep all the original information. This library has minimal dependencies, ensuring that it can be used in contexts where an R installation is not available. The capability to write data in Rda and Rds formats is also under development. Thus, the library `rdata` facilitates data interchange, enabling the usage of the same datasets in both languages (e.g. for reproducibility, comparisons of results against methods in both languages, or the creation of complex processing pipelines that involve steps in both R and Python).

Statement of need

The datasets of the R programming language, such as those from the CRAN repository, are often stored in the R specific formats Rda and Rds. In Python, there were a few packages that could parse these file formats, albeit all of them presented some limitations.

The package `rpy2` ([Gautier, 2024](#)) can be used to interact with R from Python. This includes the ability to load data in the Rda and Rds formats, and to convert these data to equivalent Python objects. Although this is arguably the best package to achieve interaction between both languages, it has many disadvantages if one wants to use it just to load R datasets. In the first place, the package requires an R installation, as it relies in launching an R interpreter and communicating with it. Secondly, launching R just to load data is inefficient, both in time and memory. Finally, this package inherits the GPL license from the R language, which is not compatible with most Python packages, typically released under more permissive licenses.

The package `pyreadr` ([Fajardo, 2024](#)) also provides functionality to read and write some R datasets. It relies in the C library `librdata` in order to perform the parsing of the R data files. This adds an additional dependency from C building tools, and requires that the package is compiled for all the desired operating systems. Moreover, this package is limited by the functionalities available in `librdata`, which at the moment of writing does not include the parsing of common objects such as R lists and S4 objects. The license can also be a problem, as it is part of the GPL family and does not allow commercial use.

As existing solutions were unsuitable for our needs, the package `rdata` was developed to parse data in the Rda and Rds formats. This is a small, extensible, efficient, and very complete implementation in pure Python of an R data parser, that is able to read and convert most datasets in the CRAN repository to equivalent Python objects, such as the built-in types of the Python standard library, NumPy arrays ([Harris et al., 2020](#)), or Pandas dataframes ([McKinney,](#)

2010; [The Pandas Development Team, 2024](#)). It has a permissive license and can be extended to support additional conversions from custom R classes.

The package `rdata` has been designed as a pure Python package with minimal dependencies, so that it can be easily integrated inside other libraries and applications. It currently powers the functionality offered in the `scikit-datasets` package ([Díaz-Vico & Ramos-Carreño, 2023](#)) for loading datasets from the CRAN repository of R packages. This functionality is used for fetching the functional datasets provided in the `scikit-fda` library ([Ramos-Carreño et al., 2024](#)), whose development was the main reason for the creation of the `rdata` package itself.

Features

The package `rdata` is intended to be both flexible and easy to use. In order to be flexible, the parsing of the R data file formats and the conversion of the parsed structures to appropriate Python objects have been splitted in two steps. This allows advanced users to perform custom conversions without losing information. Most users, however, will want to use the default conversion routine, which attempts to convert data to a standard Python representation which preserves most part of the information. Converting an Rda dataset to Python objects using the package `rdata` can be easily done as follows:

```
import rdata
```

```
converted = rdata.read_rda("dataset.rda")
```

This is equivalent to the following code, in which the two steps are performed separately:

```
import rdata
```

```
parsed = rdata.parser.parse_file("dataset.rda")
converted = rdata.conversion.convert(parsed)
```

The function `parse_file()` of the `parser` module is used to parse Rda and Rds files, returning a tree-like structure of Python objects that contains a representation of the basic R objects conforming the dataset. The function `convert()` of the `conversion` module transforms that representation to the final Python objects, such as lists, dictionaries or dataframes, that users can manipulate.

Advanced users will probably require loading datasets which contain non standard S3 or S4 classes, translating each of them to a custom Python class. This can be achieved using `rdata` by creating a constructor function that receives the converted object representation and its attributes, and returns a Python object of the desired type. As an example, consider the following short code that constructs a Pandas ([The Pandas Development Team, 2024](#)) `Categorical` object from the internal representation of an R factor.

```
import pandas
```

```
def factor_constructor(obj, attrs):
    values = [attrs['levels'][i - 1] if i >= 0 else None for i in obj]
    return pandas.Categorical(values, attrs['levels'], ordered=False)
```

Then, a dictionary containing as keys the original class names to convert and as values the constructor functions can be passed as the `constructor_dict` parameter of the `read_rda()` (or `convert()` if we do it in two steps) function. In the previous example, this could be done using the following code:

```
converted = rdata.read_rda(
    "dataset.rda",
    constructor_dict={"factor": factor_constructor},
)
```

When the default conversion routine is being executed, if an object belonging to an S3 or S4 class is found, the appropriate constructor will be called passing to it the partially constructed object. If no constructor is available for that class, a warning will be emitted and the constructor of the most immediate parent class available will be called. If there are no constructors for any of the parent classes, the basic underlying Python object will be left without transformation.

By default, a dictionary named `DEFAULT_CLASS_MAP` is passed to `convert()` including constructors for commonly used classes, such as `data.frame`, `ordered` or the aforementioned `factor`. In case the user desires different conversions for basic R objects, it would be enough to create a subclass of the `Converter` class. Several utility functions, such as the routines `convert_char()` and `convert_list()`, are exposed by the conversion module in order for users to be able to reuse them for that purpose.

Ongoing work

To broaden the utility of the `rdata` library for data processing pipelines with steps in both R and Python, we are currently extending the library with the capability to write compatible Python objects to Rda and Rds files. As an example, such a pipeline is present in the Hmsc-HPC code (Rahman et al., 2024). The continuous development of this code has also been driving the ongoing work on the writing functionality of the `rdata` library. The writing of Rda and Rds files is implemented as a two-step process similar to reading: first, the Python object is converted to the tree-like intermediate representation used in parsing, and then this intermediate representation is written to a file of the chosen format. Currently, the writing functionality supporting common types is available in the development branch of the `rdata` library.

Acknowledgements

This work has received funding from the Spanish Ministry of Education and Innovation, projects PID2019-106827GB-I00 / AEI / 10.13039/501100011033 and PID2019-109387GB-I00, from an FPU grant (Formación de Profesorado Universitario) from the Spanish Ministry of Science, Innovation and Universities (MICIU) with reference FPU18/00047, and from the European Union's Horizon Europe research and innovation programme under grant agreement No 101057437 (BioDT project, <https://doi.org/10.3030/101057437>). Views and opinions expressed are those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the European Commission can be held responsible for them. The authors gratefully acknowledge the use of the computational facilities provided by Centro de Computación Científica (CCC) at Universidad Autónoma de Madrid and by CSC – IT Center for Science, Finland.

References

- Díaz-Vico, D., & Ramos-Carreño, C. (2023). *scikit-datasets: Scikit-learn-compatible datasets*. <https://doi.org/10.5281/zenodo.6383047>
- Fajardo, O. (2024). *Pyreadr*. Zenodo. <https://doi.org/10.5281/zenodo.7110169>
- Gautier, L. (2024). *Rpy2: R in Python*. GitHub. <https://github.com/rpy2/rpy2>
- Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Río, J. F. del, Wiebe, M., Peterson, P., ... Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585(7825), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>

- McKinney, Wes. (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt & Jarrod Millman (Eds.), *Proceedings of the 9th Python in Science Conference* (pp. 56–61). <https://doi.org/10.25080/Majora-92bf1922-00a>
- Rahman, A. U., Tikhonov, G., Oksanen, J., Rossi, T., & Ovaskainen, O. (2024). Accelerating joint species distribution modelling with Hmsc-HPC by GPU porting. *PLOS Computational Biology*, 20(9), e1011914. <https://doi.org/10.1371/journal.pcbi.1011914>
- Ramos-Carreño, C., Torrecilla, J. L., Carbajo-Berrocal, M., Marcos, P., & Suárez, A. (2024). Scikit-fda: A Python Package for Functional Data Analysis. *Journal of Statistical Software*, 109, 1–37. <https://doi.org/10.18637/jss.v109.i02>
- The Pandas Development Team. (2024). *pandas-dev/pandas: pandas* (latest). Zenodo. <https://doi.org/10.5281/zenodo.3509134>