Combining R and Python for Scientific Computing

Thomas Zwagerman

2024-07-11

Table of contents

Preface **4**

1 Introduction **5**
- 1.1 Language Agnostic Data Science . . . 5
- 1.2 Python for the R user . . . 5
- 1.3 Using R and Python together . . . 6

I Python for the R User **7**

2 Differences between R and Python **8**

II Using R and Python Together **9**

3 Language bridges: reticulate / rpy2 **10**
- 3.1 Using R in Python: rpy2 . . . 10
- 3.2 Using Python in R: reticulate . . . 11

4 Quarto: language agnostic publishing **12**

5 Web Applications: Shiny for R / Shiny for Python **13**

6 Using R and Python in the shell **14**
- 6.1 R . . . 14
  - 6.1.1 Executing scripts . . . 14
  - 6.1.2 Rscript vs R -f . . . 14
  - 6.1.3 Logging methods . . . 14
  - 6.1.4 A note on developing R packages . . . 14

7 Managing Environments **15**
- 7.1 Context . . . 15
  - 7.1.1 How R package installation works . . . 15
  - 7.1.2 How Python package installation works . . . 15
- 7.2 With conda . . . 15
  - 7.2.1 Setting up an environment from scratch . . . 15

7.2.2 Exporting conda environment: environment.yml 17
7.2.3 Installing from an environment.yml 17
7.3 Without conda 17
7.3.1 Python without conda 17
7.3.2 R without conda 19

Preface

Hello!

1 Introduction

1.1 Language Agnostic Data Science

The trend of the past few years is that Data Science is becoming more language-agnostic.

Tools such as Quarto, Apache Arrow… enable working across languages with less friction.

RStudio (the company) renaming itself to "Posit" was also a clear statement of intent. Tools like Shiny and Pins are being ported to Python, and the latest exciting installment is Posit's new language agnostic data science IDE, Positron.

We often receive questions from R users on whether we can deliver an "Introduction to Python" course, or something similar. While yes, we can certainly do that, the question really is… *why?*

Another aspect to this we have noticed, is that a lot of "Introduction to Python" courses are almost always aimed at complete beginners of programming *full stop*. Most, if not all R users know full-well what a variable or a dataframe is and how to `print("Hello World!")`. These courses are not pitched at the right level.

Returning to the question of why R users might want to learn Python, the answer is that there could be a plethora of reasons. But sometimes, simply being able to use R in combination with Python could be enough to achieve your aims.

In other scenarios, you **will** need to learn some Python, but a quick crash course on the differences between R and Python and the particular topic you're interested in could be enough.

This book aims to provide some guidance on where R and Python are different, but also show the ever-increasing range of methods to combine the two languages together. This book is split in two major sections, **Python for the R user** and **Using R and Python together**.

1.2 Python for the R user

This chapter attempts to minimally break down the differences between R and Python, written for the R user.

Topics:

- matplotlib, instead of ggplot2
- numpy, instead of dplyr

1.3 Using R and Python together

This chapter outlines the range of tools available to implement R and Python together, and allow you to collaborate with Python users, without actually learning much Python.

Topics:

- Language bridges: reticulate / rpy2
- Quarto: language agnostic publishing
- Web Applications: Shiny for R / Shiny for Python
- In Production: Shell execution
- In Production: Managing environments

Part I

Python for the R User

2 Differences between R and Python

Differences

Part II

Using R and Python Together

3 Language bridges: reticulate / rpy2

Below are some examples of how to incorporate R code in Python, and Python in R.

3.1 Using R in Python: rpy2

In a Python, you can incorporate R scripts using the rpy2 package.

Below shows an example where we have some R scripts that contains functions we want to use in Python:

```
from rpy2 import robjects
from rpy2.robjects import pandas2ri

# Load R functions
# Read R scripts from files
def load_r_script(filename):
    with open(filename, "r") as file:
        return file.read()

script_location = "src/R/"

script_filenames = [
    "read_in_data.R",
    "do_something.R"
]

scripts = [script_location + script for script in script_filenames]

for script in scripts:
    robjects.r(load_r_script(script))

# Assign the functions by loading them with robjects
read_in_data_r = robjects.r["read_in_data"]
do_something_r = robjects.r["do_something"]
```

We can now use the functions `read_in_data_r` and `do_something_r` in our script.

For example, reading in some csv data, and doing something to it:

```
r_df = read_in_data_r("/data/my_file.csv")
r_df = do_something_r(r_df)
```

We could keep using R throughout, but the `r_df` is an R dataframe object.

If we wanted to continue our analysis, but using `pandas`, we will have to convert our R dataframe, to a pandas dataframe:

```
with (robjects.default_converter + pandas2ri.converter).context():
    df = robjects.conversion.get_conversion().rpy2py(r_df)
```

Here, `df` is now a pandas dataframe that can be used natively in Python.

Some considerations:

1. This is great method to quickly incorporate some R code into a Python script.
2. You will need to manage dependencies across both R and Python, see Chapter 7.
3. Performance - the conversion step can be computationally expensive in some cases. This is especially important to consider when using this method in a Shiny for Python application. If your app relies on snappy performance, use this method as a prototype, and subsequently consider whether the effort of porting your R scripts to a Python-based method is worth the performance improvement. If sticking with this method, try to minimise the number of conversions back and forth.

3.2 Using Python in R: reticulate

4 Quarto: language agnostic publishing

5 Web Applications: Shiny for R / Shiny for Python

6 Using R and Python in the shell

6.1 R

Running R in the shell bring slightly different concerns from running R interactively - which is likely what most R users are used to.

There are a few extra quirks and considerations to running R in the shell, some of which are listed below.

6.1.1 Executing scripts

If you are intending to run an R script in the shell, the following line of text should be at the top of your script:

```
#!/usr/bin/env Rscript
```

6.1.2 Rscript vs R -f

What is the difference?

6.1.3 Logging methods

Base R's `sink()` function

The `logging` and `logger` packages.

6.1.4 A note on developing R packages

When writing functions as part of an R package, it could be useful to keep in mind that some users might not be using your package interactively, but through the shell.

There is no difference in functionality here, but feedback your package provides, say through `print()` (discouraged), `message()/warning()/stop()`, or `cli::cat_*()` functions, behave slightly differently.

7 Managing Environments

7.1 Context

This is a big one for the R user. When you are unfamiliar with how Python virtual environments work, it is probably one of the most frustrating concepts to pick up from scratch.

But fear not - to fully get to grips with how Python virtual environments work and package installation works, it is good to remind ourselves how this works in R.

7.1.1 How R package installation works

Well, *remind ourselves…*. Python installation in R is likely something you have not really thought about, beyond writing `install.packages()`.

This is one of the huge benefits that makes R easy to pick up for "non-programmers", a lot of this stuff is handled for you.

7.1.1.1 Building from binaries vs. building from source

7.1.2 How Python package installation works

pip, conda, miniconda, mambo PyPi… where do I start?

7.2 With conda

7.2.1 Setting up an environment from scratch

conda is …

To create a project environment which manages both R (r-base) and Python (python):

```
conda create -n env_name r-base python=3.12
```

This sets up a new environment called `env_name` with both R and Python install.

Notice you can specify which version of R and/or Python to install.

7.2.1.1 Python packages

You can then install Packages packages with:

```
conda install numpy
```

7.2.1.2 Python packages on Github

For Python packages on Github:

```
conda install git pip

# Now use pip install with git+<repo-url>
pip install git+https://github.com/davidwilby/amundsen-sea-low-index

# If you need a specific optional dependency:
pip install asli[s3]@git+https://github.com/davidwilby/amundsen-sea-low-index
```

7.2.1.3 R packages

R packages are installed with the same `conda install` command. Note that in this case, package names are pre-pended with `r-`:

```
conda install r-dplyr
```

7.2.1.4 R packages on Github, or not on CRAN

For R packages on Github:

Taken from this issue, but does not work for me

```
conda install conda build

conda skeleton cran <github-url>

# Replace with your version of R
conda build --R=4.2.2 r-<lowercase-packagename>
```

This creates a "conda skeleton" of the R package hosted on Github.

note, explain what a conda skeleton is?

Alternative approach, specify R version in conda but use `renv::restore()` for non-CRAN packages

All packages installed and used should now be recorded in the environment, with their appropriate version number.

7.2.2 Exporting conda environment: environment.yml

To access and share your environment, run:

```
conda env export > environment.yml
```

This will create an environment.yml file in your project directory. This will allow for this environment to be set up on other systems.

7.2.3 Installing from an environment.yml

For others (or for yourself on a different system) to set up an identical environment, run:

```
conda -f create env_name environment.yml
```

If you can't (or don't want to) use conda, it is possible to manage the Python and R environments in the same project, but treat their management as seperate processes.

7.3 Without conda

7.3.1 Python without conda

In Python, this is fairly straightforward, and you can use this method regardless of the level of permissions you have on your machine. Here, you would use `venv`, and `requirements.txt`.

7.3.1.1 Setting up a venv and requirements.txt

```
# To create your venv
python -m venv env_name
```

This creates you virtual environment. To use it, and install your package you need to "activate" it.

```
# To activate your venv
source env_name/bin/activate
```

Now you have activated your environment, you can install a packages using `pip`.

```
pip install pathlib
```

To keep track of the packages your project depends on, record them in a file called `requirements.txt`. It is a very simple file that looks like this:

```
configparser==7.1.0
pathlib==1.0.1
xarray==2024.6.0
```

All it records is `packagename==1.0.0`, very simple!

If you are not sure what packages you have imported.

```
pip freeze
```

This should print them in the terminal.

7.3.1.2 Installation in a new venv from a requirements.txt

To set up a new environment and install the required packages from a specified requirements.txt:

```
# To create your venv
python -m venv env_name
source env_name/bin/activate

pip install -r requirements.txt
```

7.3.2 R without conda

To manage R dependencies, you have a couple of options, depending on the level of permissions you have on the system you are working on.

7.3.2.1 With sudo permissions

If you have sudo permissions, we can use the `pak` package. The great things about `pak` is that, unlike `install.packages()` and `renv::install()` (more on renv later), it will automatically fetch the pre-built binaries for your operating system, distribution and version. More detail on pak, and how it operates in [R in Production](https://r-in-production.org).

It will also automatically install any system-level dependencies that your R package may require. This is especially useful on linux systems, where system dependency installation can vastly differ between distributions.

If we have sudo permissions, and are using `pak`, your approach could be as simple as providing an `install.R` file which installs the right packages for you.

First install `pak`:

```
Rscript -e install.packages('pak')
```

Now it can be called in your set up script. You would normally only run this once on deployment.

```
#!/usr/bin/env Rscript
# Usage R -f install.R

# Selecting p3m.dev is an optional step for linux distros
# It will speed up installation and prevents the risk of installation
# failing on external C libraries

# This is because CRAN only provides source packages for linux
# and not binary
# see: https://r-in-production.org/packages.html#installing-a-package-on-linux

# For Ubuntu 24.04
options(repos = c(CRAN = "https://p3m.dev/cran/__linux__/noble/latest"))

# For Rocky 9
# options(repos = c(CRAN = "https://p3m.dev/cran/__linux__/rhel9/latest"))
```

```r
pak::pak("readr")
pak::pak("paws")
pak::pak("ini")
pak::pak("assertr")
pak::pak("dplyr")

# A package on Github:
pak::pak("thomaszwagerman/butterfly")
```

Call it with:

```
R -f install.R
```

7.3.2.2 Without sudo permission

Without sudo permission on your machine, you might have trouble running installation commands such as `install.packages()` or `pak::pak()`, as R might be trying to install your packages into a shared library, where you do not have 'write' permission.

In this case, the path of least resistance would be to use `renv`. To manage your environment.

If you have not used `renv` before, it is highly recommended you read Getting Started with renv before reading further.

To start using `renv`:

```r
install.packages('renv')

renv::init()
```

This will install and set up `renv` for you. `renv::init()` generates a `renv.lock` file based on the packages you have installed and used.

An extract from a `renv.lock` is shown below. You will notice it specifies the version of R used, which repositories it has used for installation, as well as packages and their associated version and download source.

```json
{
  "R": {
    "Version": "4.4.1",
    "Repositories": [
      {
```

```
        "Name": "P3M",
        "URL": "https://packagemanager.posit.co/cran/__linux__/centos7/latest"
      }
    ]
  },
  "Packages": {
    "MASS": {
      "Package": "MASS",
      "Version": "7.3-59",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
        "R",
        "grDevices",
        "graphics",
        "methods",
        "stats",
        "utils"
      ],
      "Hash": "0cafd6f0500e5deba33be22c46bf6055"
    },
    "R6": {
      "Package": "R6",
      "Version": "2.5.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Requirements": [
        "R"
      ],
      "Hash": "470851b6d5d0ac559e9d01bb352b4021"
    }
  }
}
```

This destination is set to a specific operating system, **centos7**. This URL is obtained from the Posit Package Manager.

When using `pak`, this is automatically fetched for us. Unfortunately for us, `renv` does not use `pak`. To prevent having to manually change this URL for each deployment on a different system, we need to insert this URL depending on the operating system we are working on.

Shannon Pileggi

With the above in mind, the `install.R` script would look like this:

```
#!/usr/bin/env Rscript
# Usage R -f hpc_setup.R
#
# This will not work with opensuse and sle,
# naming inconsistencies across distros is hard

# This is not an R project, so need to manually "activate" renv
source("renv/activate.R")

install.packages("pkgcache")

# Moving on to installing r and system dependencies with renv.lock
# Have R obtain the current platform distro and release
os <- data.frame(
  distribution = pkgcache::current_r_platform_data()$distribution,
  release = pkgcache::current_r_platform_data()$release
)

os$release <- round(as.numeric((os$release)))

# Some wrangling to make matching more reliable across distros
ppm_platforms <- pkgcache::ppm_platforms()

# Take the word "linux" out of distribution names
ppm_platforms$distribution <- gsub("linux", "", ppm_platforms$distribution)
# Makes matching rocky distro possible
ppm_platforms$release <- round(as.numeric((ppm_platforms$release)))

# Match with pak's ppm_platforms
os_table <- merge(
  os,
  ppm_platforms
)

if (os_table$os == "linux") {
  p3m_url <- paste0(
    "https://p3m.dev/cran/__linux__/",
  os_table$binary_url,
"/latest"
)
} else {
  p3m_url <- "https://p3m.dev/cran/latest"
```

```r
}

renv::lockfile_modify(repos = c(
  P3M = p3m_url
)) |>
renv::lockfile_write()

renv::restore()
```

In summary, this script should:

1. Activate `renv` and install `pkgcache`.
2. Detect which **os**, **distribution** and **version** R is being run on.
3. Concatenate the **package manager URL** and the **os binary URL**.
4. Modify the `renv.lock` file to point the repos URL to the correct binary URL.