Polyglot programming for single-cell analysis

Louise Deconinck Benjamin Rombaut Robrecht Cannoodt

2024-09-12

Introduction

1. How do you interact with a package in another language?
2. How do you make you package useable for developers in other languages?

We will be focusing on R & Python

Summary

Interoperability between languages allows analysts to take advantage of the strengths of different ecosystems

On-disk interoperability uses standard file formats to transfer data and is typically more reliable

In-memory interoperability transfers data directly between parallel sessions and is convenient for interactive analysis

While interoperability is currently possible developers continue to improve the experience

Single-cell best practices: Interoperability

How do you interact with a package in another language?

1. In-memory interoperability
2. Disk-based interoperability

How do you make your package useable for developers in other languages?

1. Package-based interoperability
2. Best practices

Package-based interoperability

or: the question of reimplementation.

- Consider the pros:

    1. Discoverability
    2. Can your package be useful in other domains?
    3. Very user friendly

- Consider the cons:

    1. Think twice: is it worth it?
    2. **It's a lot of work**
    3. How will you keep it up to date?
    4. How will you ensure parity?

Package-based interoperability

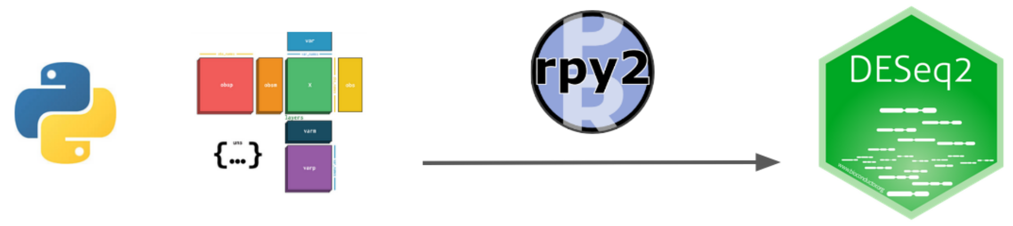
Please learn both R & Python

Best practices

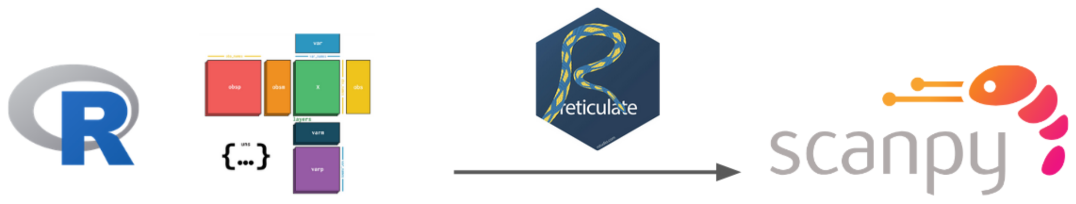
1. Work with the standards
2. Work with matrices, arrays and dataframes
3. Provide vignettes on interoperability

In-memory interoperability

A Python user with an anndata object can use rpy2 to run the DESeq2 method in R



An R user with an anndata object can use reticulate to run scanpy functions in Python



Overview

1. Advantages & disadvantages
2. Pitfalls when using Python & R
3. Rpy2
4. Reticulate

in-memory interoperability advantages

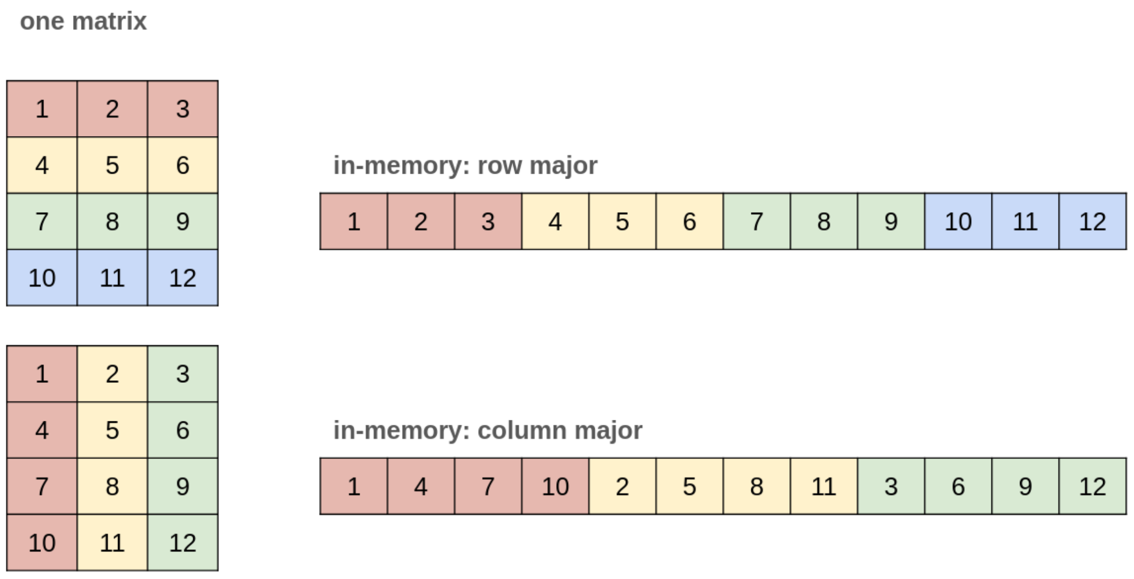
- no need to write & read results
- useful when you need a limited amount of functions in another language

in-memory interoperability drawbacks

- not always access to all classes
- data duplication
- you need to manage the environments

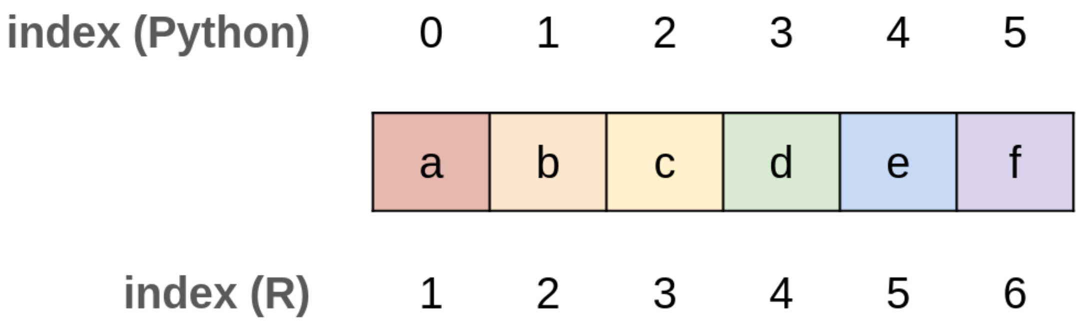
Pitfalls when using Python and R

Column major vs row major matrices In R: every dense matrix is stored as column major



Pitfalls when using Python and R

Indexing



Pitfalls when using Python and R

dots and underscores

- mapping in rpy2

```
from rpy2.robjects.packages import importr

d = {'package.dependencies': 'package_dot_dependencies',
     'package_dependencies': 'package_uscore_dependencies'}
tools = importr('tools', robject_translations = d)
```

Pitfalls when using Python and R

Integers

```
library(reticulate)
bi <- reticulate::import_builtins()

bi$list(bi$range(0, 5))
# TypeError: 'float' object cannot be interpreted as an integer
```

```
library(reticulate)
bi <- reticulate::import_builtins()

bi$list(bi$range(0L, 5L))
# [1] 0 1 2 3 4
```

Rpy2: basics

- Accessing R from Python
  - `rpy2.rinterface`, the low-level interface
  - `rpy2.robjects`, the high-level interface

```
import rpy2
import rpy2.robjects as robjects

vector = robjects.IntVector([1,2,3])
rsum = robjects.r['sum']

rsum(vector)
```

–
6
–

Rpy2: basics

```
str_vector = robjects.StrVector(['abc', 'def', 'ghi'])
flt_vector = robjects.FloatVector([0.3, 0.8, 0.7])
int_vector = robjects.IntVector([1, 2, 3])
mtx = robjects.r.matrix(robjects.IntVector(range(10)), nrow=5)
print(mtx)
```

```
     [,1] [,2]
[1,]    0    5
[2,]    1    6
[3,]    2    7
[4,]    3    8
[5,]    4    9
```

Rpy2: numpy

```
import numpy as np

from rpy2.robjects import numpy2ri
from rpy2.robjects import default_converter

rd_m = np.random.random((5, 4))

with (default_converter + numpy2ri.converter).context():
    mtx = robjects.r.matrix(rd_m, nrow = 5)
    print(mtx)
```

```
[[0.49591958 0.3843478  0.77713133 0.1873487 ]
 [0.72041163 0.16727186 0.98250235 0.81832831]
 [0.9284306  0.63418864 0.17022845 0.83715282]
 [0.8960735  0.15571451 0.70520788 0.45977798]
 [0.67306021 0.03015352 0.56079185 0.54337257]]
```

Rpy2: pandas

```
import pandas as pd

from rpy2.robjects import pandas2ri

pd_df = pd.DataFrame({'int_values': [1,2,3],
                      'str_values': ['abc', 'def', 'ghi']})

with (default_converter + pandas2ri.converter).context():
    pd_df_r = robjects.DataFrame(pd_df)
    print(pd_df_r)
```

```
  int_values str_values
0          1        abc
1          2        def
2          3        ghi
```

Rpy2: sparse matrices

```
import scipy as sp

from anndata2ri import scipy2ri

sparse_matrix = sp.sparse.csc_matrix(rd_m)

with (default_converter + scipy2ri.converter).context():
    sp_r = scipy2ri.py2rpy(sparse_matrix)
    print(sp_r)
```

```
5 x 4 sparse Matrix of class "dgCMatrix"

[1,] 0.4959196 0.38434780 0.7771313 0.1873487
[2,] 0.7204116 0.16727186 0.9825023 0.8183283
[3,] 0.9284306 0.63418864 0.1702285 0.8371528
[4,] 0.8960735 0.15571451 0.7052079 0.4597780
[5,] 0.6730602 0.03015352 0.5607919 0.5433726
```

Rpy2: anndata

```python
import anndata as ad
import scanpy.datasets as scd

import anndata2ri

adata_paul = scd.paul15()

with anndata2ri.converter.context():
    sce = anndata2ri.py2rpy(adata_paul)
    ad2 = anndata2ri.rpy2py(sce)
```

Rpy2: interactivity

```
%load_ext rpy2.ipython  # line magic that loads the rpy2 ipython extension.
                        # this extension allows the use of the following cell magic

%%R -i input -o output  # this line allows to specify inputs
                        # (which will be converted to R objects) and outputs
                        # (which will be converted back to Python objects)
                        # this line is put at the start of a cell
                        # the rest of the cell will be run as R code
```

Reticulate

R	Python	Examples
Single-element vector	Scalar	`1`, `1L`, `TRUE`, `"foo"`
Multi-element vector	List	`c(1.0, 2.0, 3.0)`, `c(1L, 2L, 3L)`
List of multiple types	Tuple	`list(1L, TRUE, "foo")`
Named list	Dict	`list(a = 1L, b = 2.0)`, `dict(x = x_data)`
Matrix/Array	NumPy ndarray	`matrix(c(1,2,3,4), nrow = 2, ncol = 2)`
Data Frame	Pandas DataFrame	`data.frame(x = c(1,2,3), y = c("a", "b", "c"))`
Function	Python function	`function(x) x + 1`
Raw	Python bytearray	`as.raw(c(1:10))`
NULL, TRUE, FALSE	None, True, False	`NULL`, `TRUE`, `FALSE`

Reticulate

```r
library(reticulate)

bi <- reticulate::import_builtins()
rd <- reticulate::import("random")

example <- c(1,2,3)
bi$max(example)
# [1] 3
rd$choice(example)
# [1] 2
cat(bi$list(bi$reversed(example)))
# [1] 3 2 1
```

Reticulate numpy

```
np <- reticulate::import("numpy")

a <- np$asarray(tuple(list(1,2), list(3, 4)))
b <- np$asarray(list(5,6))
b <- np$reshape(b, newshape = tuple(1L,2L))

np$concatenate(tuple(a, b), axis=0L)
#      [,1] [,2]
# [1,]    1    2
# [2,]    3    4
# [3,]    5    6
```

Reticulate conversion

```
np <- reticulate::import("numpy", convert = FALSE)

a <- np$asarray(tuple(list(1,2), list(3, 4)))
b <- np$asarray(list(5,6))
b <- np$reshape(b, newshape = tuple(1L,2L))

np$concatenate(tuple(a, b), axis=0L)
# array([[1., 2.],
#        [3., 4.],
#        [5., 6.]])
```

You can explicitly convert data types:

```
result <- np$concatenate(tuple(a, b), axis=0L)

py_to_r(result)
#      [,1] [,2]
# [1,]    1    2
# [2,]    3    4
# [3,]    5    6

result_r <- py_to_r(result)
```

```r
r_to_py(result_r)
# array([[1., 2.],
#        [3., 4.],
#        [5., 6.]])
```

Reticulate scanpy

```r
library(anndata)
library(reticulate)
sc <- import("scanpy")

adata_path <- "../usecase/data/sc_counts_subset.h5ad"
adata <- anndata::read_h5ad(adata_path)
```

We can preprocess & analyse the data:

```r
sc$pp$filter_cells(adata, min_genes = 200)
sc$pp$filter_genes(adata, min_cells = 3)
sc$pp$pca(adata)
sc$pp$neighbors(adata)
sc$tl$umap(adata)

adata
# AnnData object with n_obs × n_vars = 32727 × 20542
#     obs: 'dose_uM', 'timepoint_hr', 'well', 'row', 'col', 'plate_name', 'cell_id', 'cell_ty
#     var: 'highly_variable', 'means', 'dispersions', 'dispersions_norm', 'n_cells'
#     uns: 'cell_type_colors', 'celltypist_celltype_colors', 'donor_id_colors', 'hvg', 'leide
#     obsm: 'HTO_clr', 'X_pca', 'X_umap', 'protein_counts'
#     varm: 'PCs'
#     obsp: 'connectivities', 'distances'
```

Disk-based interoperability

Disk-based interoperability is a strategy for achieving interoperability between tools written in different programming languages by **storing intermediate results in standardized, language-agnostic file formats**.

- Upside:

    - Simple, just add reading and witing lines
    - Modular scripts

- Downside:
    - increased disk usage
    - less direct interaction, debugging…

Important features of interoperable file formats

- Compression
- Sparse matrix support
- Large images
- Lazy chunk loading
- Remote storage

General single cell file formats of interest for Python and R

File Format	Python	R	Sparse matrix	Large images	Lazy chunk loading	Remote storage
RDS						
Pickle						
CSV						
JSON						
TIFF						
.npy						
Parquet						
Feather						
Lance						
HDF5						
Zarr						
TileDB						

Specialized single cell file formats of interest for Python and R

File Format	Python	R	Sparse matrix	Large images	Lazy chunk loading	Remote storage
Seurat RDS						
Indexed OME-TIFF						
h5Seurat						
Loom						
HDF5						
AnnData h5ad						
AnnData Zarr						
TileDB-SOMA						
TileDB-BioImaging						
SpatialData Zarr						

Disk-based pipelines

Script pipeline:

```bash
#!/bin/bash

bash scripts/1_load_data.sh
python scripts/2_compute_pseudobulk.py
Rscript scripts/3_analysis_de.R
```

Notebook pipeline:

```bash
# Every step can be a new notebook execution with inspectable output
jupyter nbconvert --to notebook --execute my_notebook.ipynb --allow-errors --output-dir outpu
```

Just stay in your language and call scripts

```python
import subprocess

subprocess.run("bash scripts/1_load_data.sh", shell=True)
# Alternatively you can run Python code here instead of calling a Python script
subprocess.run("python scripts/2_compute_pseudobulk.py", shell=True)
subprocess.run("Rscript scripts/3_analysis_de.R", shell=True)
```

Pipelines with different environments

1. interleave with environment (de)activation functions
2. use rvenv
3. use Pixi

Pixi to manage different environments

```bash
pixi run -e bash scripts/1_load_data.sh
pixi run -e scverse scripts/2_compute_pseudobulk.py
pixi run -e rverse scripts/3_analysis_de.R
```

Define tasks in Pixi

```toml
...
[feature.bash.tasks]
load_data = "bash book/disk_based/scripts/1_load_data.sh"
...
[feature.scverse.tasks]
compute_pseudobulk = "python book/disk_based/scripts/2_compute_pseudobulk.py"
...
[feature.rverse.tasks]
analysis_de = "Rscript --no-init-file book/disk_based/scripts/3_analysis_de.R"
...
[tasks]
pipeline = { depends-on = ["load_data", "compute_pseudobulk", "analysis_de"] }
```

```
pixi run pipeline
```

Also possible to use containers

```
docker pull berombau/polygloty-docker:latest
docker run -it -v $(pwd)/usecase:/app/usecase -v $(pwd)/book:/app/book berombau/polygloty-do
```

Another approach is to use multi-package containers to create custom combinations of packages. - Multi-Package BioContainers - Seqera Containers

Workflows

You can go a long way with a folder of notebooks or scripts and the right tools. But as your project grows more bespoke, it can be worth the effort to use a **workflow framework** like Viash, Nextflow or Snakemake to manage the pipeline for you.

See https://saeyslab.github.io/polygloty/book/workflow_frameworks/

Takeaways