PypeR, A Python Package for Using R in Python

Xiao-Qin Xia
Vaccine Research Institute of San Diego

Michael McClelland
Vaccine Research Institute of San Diego

Yipeng Wang
Vaccine Research Institute of San Diego

Abstract

This article describes PypeR, a Python package which allows the R language to be called in Python using the pipe communication method. By running R through pipe, the Python program gains flexibility in sub-process controls, memory control, and portability across popular operating system platforms, including Windows, GNU Linux and Mac OS X. PypeR can be downloaded at http://rinpy.sourceforge.net/.

Keywords: PypeR, Python, R language.

1. Introduction

A rapidly growing open source programming language, Python has a clean object-oriented design and extensive support libraries, which significantly increase programmer productivity. Python is widely used in many kinds of software development, including computation-intensive scientific programming and web applications. There are also some projects for scientific computing in Python. NumPy (Oliphant 2006) provides fundamental mathematical functions, including convenient N-dimensional array manipulation; ScientificPython (Hinsen 2007) integrates an extensive collection of modules, such as 3D visualization, parallel programming, and so on. Python is very attractive to bioinformaticians as a programming language; however, one barrier to the use of Python in biological applications is the lack of biology-related packages.

R (R Development Core Team 2010) is a language and environment for statistical computing and graphics. It is very popular and powerful in scientific computation, especially in bioinformatics. Bioconductor (Gentleman et al. 2004), an R-based open source software project that aims to provide innovative and reliable tools for computational biology and bioinformatics, has rapidly growing user communities and developer communities with advances in high throughput genome analysis technology, such as DNA microarrays and high-throughput
sequencing. Most Bioconductor tools are presented as R packages. Therefore, R can be a good complement to Python in bioinformatics computation.

There are two software projects currently facilitating the integration of Python and R; RSPython (Temple Lang 2005), a Python interface to the R system for statistical computing, and RPy (Cock 2005), a project inspired by the former. Both packages are based on the application programming interfaces (APIs) of Python and R.

RSPython allows users to call functions in R from Python and vice versa. However the last update was in 2005 and it seems to no longer be in development. The dependence on obsolete libraries makes it difficult to use on modern computer frameworks.

RPy presents a simple and efficient way of accessing R from Python. It is robust and very convenient for frequent interaction operations between Python and R. This package allows Python programs to pass Python objects of basic data types to R functions and return the results in Python objects. Such features make it an attractive solution for the cases in which Python and R interact frequently. However, there are still limitations of this package as listed below.

- **Performance:**

  RPy may not behave very well for large-size data sets or for computation-intensive duties. A lot of time and memory are inevitably consumed in producing the Python copy of the R data because in every round of a conversation RPy converts the returned value of an R expression into a Python object of basic types or NumPy array. RPy2, a recently developed branch of RPy, uses Python objects to refer to R objects instead of copying them back into Python objects. This strategy avoids frequent data conversions and improves speed. However, memory consumption remains a problem. This can be illustrated by the memory and CPU consumption of a simple Python function (see Table 1):

  ```python
  def foo(r):
      r("a <- NULL")
      for i in range(20):
          r("a <- rbind(a, seq(1000000) * 1.0 * %d)" % i)
      print r("sum(a)")
  ```

Table 1 lists system loads of this function running on a Dell Precision WorkStation T7400 with Intel Xeon Quad Core 2.33Ghz and 4GB memory. Although RPy2 improved speed compared to RPy, it used even more memory than RPy.
When we were implementing WebArray (Xia et al. 2005), an online platform for microarray data analysis, a job consumed roughly one quarter more computational time if running R through RPy instead of through R's command-line user interface. Therefore, we decided to run R in Python through pipes in subsequent developments, e.g., WebArrayDB (Xia et al. 2009), which retained the same performance as achieved when running R independently. We do not know the exact reason for such a difference in performance, but we noticed that RPy directly uses the shared library of R to run R scripts. In contrast, running R through pipes means running the R interpreter directly.

- **Memory:**

R has been denounced for its uneconomical use of memory. The memory used by large-size R objects is rarely released after these objects are deleted. Sometimes the only way to release memory from R is to quit R. RPy module wraps R in a Python object. However, the R library will stay in memory even if the Python object is deleted. In other words, memory used by R cannot be released until the host Python script is terminated.

- **Portability:**

As a module with extensions written in C, the RPy source package has to be compiled with a specific R version on POSIX (Portable Operating System Interface for Unix) systems, and the R must be compiled with the shared library enabled. Also, the binary distributions for Windows are bound to specific combinations of different versions of Python/R, so it is quite frequent that a user has difficulty in finding a distribution that fits the user's software environment.

The above concerns and problems can be addressed by running a standalone R interpreter through pipes. A pipe provides a one-way communication stream between running processes. Full-duplex communication can be achieved by using two pipes. By using pipes a Python program can pass R scripts to a standalone R interpreter and run them at full-speed. The Python program can also recycle memory by terminating the R interpreter at the other side of a pipe. Since an R session is launched by a shell command and the Python program is not associated with a specific R library, any R installed on the system can be used by the Python program. On POSIX systems, it is even possible to R on remote computer.

Although pipes are not fast at passing data between processes, it is useful to connect Python and R when the performance of R scripts and portability are important concerns. Here we present a module - PypeR, which allows Python scripts to call R by using pipes. PypeR is especially useful when there is no need for frequent interactive data transfers between Python and R, as demonstrated by the comparison in Table 1.

2. Description

There are two main tasks achieved by PypeR in order to use R from Python through pipes: (1) conversion of data between Python data types and R data types; (2) communication between Python and R.

A function Str4R has been implemented in PypeR for conversions of Python objects of basic data types or NumPy arrays to R data objects in the form of a string. The conventions are listed in Table 2. All Python objects listed in Table 2 and their subclasses can be converted by
Objects in Python | Objects in R
--- | ---
None | NULL
Bool | Logical
Integer | Integer
Long | Integer
Float | Real
Complex | Complex
String | String
Tuple/List/Set/FrozenSet/Iterators | Vector/List
Dictionary | Named list
1-dimension NumPy array | Vector
1-dimension NumPy record array | Data frame
2-dimension NumPy array | Matrix
Other NumPy array | Array

Table 2: Conventions for conversion of Python objects to R objects. If all elements in a Tuple/List/Set/FrozenSet/iterators are of the same basic type (string, bool, integer, long integer, float or complex), the sequence will be converted to an R vector, otherwise an R list. An exception comes from a practical consideration: If a sequence consists of combinations of objects of integer, long integer, and float, it will be converted to a numeric vector.

RPy converts an R named list to a Python dictionary. However, a Python dictionary does not keep the original order of data in an R list. Also a dictionary will keep only the last of any elements that share the same name. PypeR solves this problem by using a Python list of two-element tuples, in which the first element is the name and the second is the value. If there is a need, the user can simply apply the Python built-in function dict to convert such a list to a Python dictionary. An R data frame is a table-like data structure, in which rows represent records and columns represent elements in each record. Different columns can be of different data types. RPy can only convert data frames into Python lists with each column in the data frame as an element. However, PypeR can convert an R data frames to a 1-dimension NumPy record array, a data structure similar to a data frame in R.

There are additional differences between PypeR and RPy in the conversions of NumPy arrays. While RPy uses the R array as the only destination data type for NumPy arrays, PypeR converts NumPy arrays to R vectors, matrices, arrays, or data frames, depending on the data.
type and shape of the source object (see Table 2). It is important to distinguish different data types in R since some NumPy arrays cannot be converted to R arrays correctly. For example, in RPy a NumPy 1-dimension record array, the analogue of an R data frame, will be converted to a one-column R array and it will not be treated as a table any more.

More importantly, R and NumPy “understand” and treat “dimension” in different ways. A multiple-dimension array can be viewed as a number of nesting arrays. Given a list of integers representing the dimension of an array, NumPy uses the values from left to right to represent dimension size from outermost to innermost in the nesting structure. Unfortunately, R does this in the opposite direction to NumPy. For example, the shape of a N-dimension array in NumPy is \((d_1, d_2, \cdots, d_{n-2}, d_{n-1}, d_n)\), where \(d_1\) is the size of the outermost dimension, \(d_{n-1}\) and \(d_n\) are the row number and column number of the innermost sub-arrays. To keep the same nesting structure, the correct dimension vector in R should be \((d_{n-1}, d_n, d_{n-2}, \cdots, d_2, d_1)\). PypeR implements such a conversion while RPy does not.

Python version 2.4 introduced the package subprocess into the standard library to unify the interface for spawning new processes in Python. In brief, if a parent process starts a child process by pipes, the standard input (stdin) and standard output (stdout) of the child process will be redirected to two pipes, i.e., text that the parent process writes to one of the pipes will be used as stdin by the child process. Meanwhile anything that the child process sends to stdout can be read from the other pipe by the parent process. In order to get instant responses from R through a pipe, we adopt Josiah Carlson’s recipe (http://code.activestate.com/recipes/440554), i.e., the parent process (Python) keeps reading from the pipe in a non-blocking way within a short span of time. PypeR judges the end of execution of a series of R commands by recognizing a characteristic signal, which is produced by a specific standard command that is attached to the end of the R command sequences by PypeR.

Data flows in a pipe are byte streams, so all the data have to first be converted to strings. Str4R does this for Python objects. Commands are fed to the R process’s stdin by pipe after being converted to strings. Anything from the stdout of an R child process is collected by the Python script. R has limitations on the size of data read from stdin. So pipe is not suitable when huge data sets, such as matrices of thousands of elements, need to be passed on to R from Python. In such cases, PypeR will automatically create temporary files for communication with R.

A Python class named R is constructed to wrap R process. The class can create one or more callable instances (Python objects), each of which manages an R child process. These objects can even use different versions of R at the same time since R is launched by command line through pipe. Any R process can be terminated by deleting the Python object which is associated with it, and its memory will be released thereafter.

PypeR is written in pure Python (≥ 2.4), which allows it to be used on multiple operating system platforms. It has no predilection for versions of R or prerequisites on the compiling options for R, except that on Windows the pywin32 package is required. PypeR has been tested on Python 2.5.1/R 2.8.0 on CentOS 5.4 and Python 2.5.4/R 2.7.0 on Windows XP SP3.

### 3. Availability and Usage

PypeR is presented as a tarball for downloading at http://rinpy.sourceforge.net. It can be used directly if the package is decompressed in the Python searching path, or can be
installed in Python in the traditional way:

```python
# python setup.py install
```

**PypeR** is also included in Python’s Package Index ([http://pypi.python.org/pypi](http://pypi.python.org/pypi)), which provides a more convenient way for installation by using "easy_install":

```python
# easy_install pyper
```

To use the package, the first step is to import it into Python:

```python
>>> from pyper import *
```

For a single run of R, the function `runR` may be used:

```python
>>> outputs = runR("a <- 3; print(a + 5)")
```

or if there is an R script (e.g., “RScript.R”) to run:

```python
>>> runR("source('RScript.R')")
```

In cases where more interactive operations are involved, the better way is to create a Python object - an instance of the class `R`:

```python
>>> r = R(use_numpy=True)
```

The function `Str4R` can be applied to translate Python objects to R objects in the form of string. In the following examples, a Python list, an iterator, and a string are passed to the R child process as vectors, while a NumPy record array is converted as an R data frame:

```python
>>> r("a <- %s" % Str4R([0, 1, 2, 3, 4]))
```

```python
>>> r.assign("a", xrange(5))
```

```python
>>> r.An_IMPORTANT_Notice = """You CANNOT use dot (.) for a variable name in this format, and leading underscore (_) is INVALID in any R variable name."""
```

```python
>>> r["salary"] = numpy.array([(1, "Joe", 35820.0), (2, "Jane", 41235.0), (3, "Kate", 37932.0)], dtype=[("id", "<i4"), ("employee", "|S4"), ("salary", "<f4")])
```

```python
>>> print r["salary"]
```

```python
>>> del r["An_IMPORTANT_Notice"], r.salary
```

It is also possible to make plots, however the plotting is done in the background and the output are saved in a file:

```python
>>> r("png('test.png')")
```

```python
>>> r("plot(1:5)")
```

```python
>>> r("dev.off()")
```
Usage details and more examples can be found in the module documents and in the test script ("test.py") in the distribution package.

Acknowledgments

This work was supported in part by NIH grants R01AI075093, R21AI083964, U01CA0114810, U01AI52237, R01AI07397, R01AI077645, R01AI083646, BARD grant IS-4267-09, and DOD grant W81XWH-08-1-0720. We thank Fred Long for helpful comments.

References


Affiliation:

Xiao-Qin Xia
Vaccine Research Institute of San Diego
10835 Road to the Cure
San Diego, CA 92121, United States of America
E-mail: xqxia70@gmail.com