pyRserve Documentation

Release 1.0.3

Ralph Heinkel

Jul 27, 2023

Contents

pyRserve is a library for connecting Python to an R process running under Rserve. Through such a connection variables can be get and set in R from Python, and also R-functions can be called remotely.

This documentation applies to pyRserve release V 1.0.3

Contents:

CHAPTER 1

Overview

1.1 What pyRserve does

pyRserve is a library for connecting Python to R (an excellent statistic package). Running Rserve in R attaches the R-interpreter to a network socket, waiting for pyRserve to connect to it. Through such a connection, variables can be get and set in R from Python, and also R-functions can be called remotely.

In contrast to rpy or rpy2 the R process does not have to run on the same machine, it can run on a remote machine and all variable access and function calls will be delegated there through the network.

Furthermore - and this makes everything feel very pythonic - all data structures will automatically be converted from native R to native Python and numpy types and back.

1.2 Supported platforms

This package has been mainly developed under Linux, and hence should run on all standard unix platforms, as well as on MacOS. pyRserve has also been successfully used on Windows machines. Unittests have been used on the Linux and MacOS side, however they might just work fine for Windows.

It has been tested to work with Python 2.7.x, 3.6 to 3.9.

The latest development has been tested with some previous and current versions of R and Rserve.

1.3 License

pyRserve has been written by Ralph Heinkel (ralph-heinkel.com) and is released under MIT license.

1.4 Quick installation

From your unix/macOS,windows command line run:

```
pip install pyRserve
```

For a fully functional setup also R and Rserve have to be installed. See section installation in the pyRserve documentation for instructions.

1.5 Quick usage

Open a **first shell** and start up the R server, by calling the module *Rserve* that provides the actual network connectivity for R:

```
$ R CMD Rserve
```

R (Rserve) will now listen on port 6311 (on localhost). Of course Rserve can be configured to listen on an exposed port and hence will be accessible from remote hosts as well.

Open a **second shell**, start Python, import pyRserve, and initialize the connection to Rserve:

```
$ python
>>> import pyRserve
>>> conn = pyRserve.connect()
```

The default connection will be done on `localhost:6311`. Other hosts can be reached by calling `pyRserve.connect(host=..., port=...)` as well.

The `conn` object provides a namespace called `conn.r` that directly maps all variables and other global symbols (like functions etc) and hence makes them accessible from Python.

Now create a vector in R, access the vector from Python (will be converted into a numpy array), and call the `sum()`-function in R:

```
>>> conn.r("vec <- c(1, 2, 4)")
>>> conn.r.vec                  # access vector 'vec' as an attribute of 'conn.r'
array([1., 2., 4.])
>>> conn.r.sum(conn.r.vec)      # 'sum' in running in the R-interpreter, returning the␣
↪result to Python
7.0
```

The other way around also works:

```
>>> conn.r.somenumber = 444         # set a variable called 'somenumber' in the R␣
↪interpreter...
>>> conn.r("somenumber * 2")        # ... and double the number
888.0
```

1.6 Source code repository

pyRserve is now hosted on GitHub at https://github.com/ralhei/pyRserve.

1.7 Documentation

Documentation can be found at https://pyrserve.readthedocs.io.

1.8 Support

For discussion of pyRserve and getting help please use the Google newsgroup available at http://groups.google.com/group/pyrserve.

Issues with the code (like bugs, etc.) should be reported on GitHub at https://github.com/ralhei/pyRserve/issues.

1.9 Missing features

- Authentication is implemented in Rserve but not yet in pyRserve
- TLS encryption is not implemented yet in pyRserve. However using ssh tunnels can solve security issues in the meantime (see documentation).

CHAPTER 2

Installation

Before pyRserve can be used, R and Rserv have to be installed properly. Installation instructions for both packages are available on their corresponding websites at http://www.r-project.org/ and http://www.rforge.net/Rserve/

2.1 Installing R from sources

For R being able to run Rserve properly it has to be installed with the `--enable-R-shlib` option.

The following command show how to do this for the sources. Make sure you have a fortran compiler installed, otherwise installation will not be possible.

Note: You need a couple of LINUX packages and libraries to be installed, like a fortran compile and readline/bzip2/... development libraries. On OpenSuse these can be installed with `zypper install -y gcc-fortran readline-devel libbz2-devel xz-devel pcre2-devel libcurl-devel` Other Linux distributions provide packages with similar names.

On installing R then looks like:

```
R_VER=4.3.1   # possibly find the latest version, or use the version you require
curl -LO https://cran.r-project.org/src/base/R-4/R-${R_VER}.tar.gz
tar -xzf R-${R_VER}.tar.gz
cd R-${R_VER}
./configure --enable-R-shlib -with-x=no
make
make install
```

For Windows it might be just enough to install a prebuilt R package. The same might be true for some Linux distributions, just make sure to install a version which also contains all headers necessary for compiling Rserve in the next step.

2.2 Installing Rserve

If you have already downloaded the tar file then from your command line run:

```
curl -LO http://www.rforge.net/Rserve/snapshot/Rserve_1.8-12.tar.gz
R CMD INSTALL Rserve_1.8-12.tar.gz
```

Older versions of Rserve might also work, the earliest function version however seems to be 0.6.6.

Note: Rserve usually daemonizes itself after starting from the command line. If you want to prevent this from happening (e.g. because you would like to control Rserve by a process management tool like `supervisord` or want to control Rserve running the unittests with `pytest --run-rserve`) then Rserve has to be install with the special `-DNODAEMON` compiler flag:

```
PKG_CPPFLAGS=-DNODAEMON  R CMD INSTALL Rserve_1.8-12.tar.gz
```

2.3 Installing pyRserve

From your unix/windows command line run:

```
pip install pyRserve
```

If you want to develop or test locally, then also install extra packages for testing:

```
pip install pyRserve[testing]
```

Currently supported Python versions are 3.6 to 3.11. It might still run on Python 2.7 but this is not supported anymore and will be deprecated in future versions.

In the next section you'll find instructions how to use everything together.

2.4 Running unittests

After installation is completed - and for those who want to contribute to pyRserve's developement - unittests can be run straight from the command line. Remember to have pyRserve installed with the testing dependencies, as described in the previous section.

In the current setup pytest is able to automatically fire up an Rserve-process which needs to be available for the unittests to run against. This is achieved by calling:

```
$ pytest testing --run-rserve
============================= test session starts ==============================
platform linux -- Python 3.11.3, pytest-7.4.0, pluggy-1.2.0
rootdir: /home/user/pyRserve
collected 50 items

testing/test_rparser.py ........................................                 ⌊
 ↪                              [ 84%]
testing/test_taggedContainers.py ........                                       ⌊
 ↪                              [100%]
============================== 50 passed in 4.19s ==============================
```

In case you have Rserve already running on localhost, it is sufficient to call `pytest testing`.

CHAPTER 3

pyRserve manual

This manual is written in sort of a *walk-through*-style. All examples can be tried out on the Python command line as you read through it.

3.1 Setting up a connection to Rserve

3.1.1 Running both Rserve and pyRserve locally on one host

This is the most simple solution, and we will begin with it before explaining remote connections.

First of all startup Rserve if it is not yet running:

```
$ R CMD Rserve
```

By default Rserve is listening on port port 6311 (its default) on localhost (or 127.0.0.1) only, for security reasons. This means that no connection from any other machine is possible to it. For now, and for simplicity, we stick with running everything (Rserve and pyRserve) on the same host.

R puts itself into daemon mode, meaning that your shell comes back, and you have no way to shutdown R via `ctrl-C` (you need to call `kill` with it's process id). However `Rserve` can be started in debug mode during development. In this mode it'll print messages to stdout helping you to see whether your connection works etc. To do so *Rserve* needs to be started like:

```
$ R CMD Rserve.dbg
```

Now we can try to connect to it. From the python interpreter import the pyRserve package and by omitting any arguments to the `connect()` function setup the connection to your locally running `Rserve`:

```
$ python
>>> import pyRserve
>>> conn = pyRserve.connect()
```

The resulting connection handle can tell you where it is connected to:

```
>>> conn
<Handle to Rserve on localhost:6311>
```

The connection will be closed automatically when conn is deleted, or by explicitly calling the `close()`-method:

```
>>> conn.close()
>>> conn
<Closed handle to Rserve on localhost:6311>
```

Running operations on a closed pyRserve connector results in an exception. However a connection can be reopened by calling the `connect()` method. It reuses the previously given values (or defaults) for `host` and `port`:

```
>>> conn.connect()
<Handle to Rserve on localhost:6311>
```

To check the status of the connection use:

```
>>> conn.isClosed
False
```

3.1.2 Setting up a remote connection to Rserve

Variant 1: Make Rserve listen to a public port

To allow Rserve accept connections from remote hosts on a public port a special flag needs to be set in its configuration file (which might be missing initially). Once the `remote enable`-flag is set there, Rserve needs to be restarted in order to honor it.

Warning: Opening Rserve on a port which is publically (or maybe within an organization like a company) accessible allows anyone who has access to this machine to connect to the Rserve server process.

Warning: Traffic between Rserve and pyRserve is not encrypted - so anyone with access to the network would in principle be able to sniff your communication, or even manipulate it.

By default Rserve tries to load the configuration file from `/etc/Rserv.conf`. So if you have root privileges on your host you can enable remote connections with the following command:

```
$ sudo echo "remote enable" > /etc/Rserv.conf
```

Then restart Rserve.

In case you don't have sudo privileges the config file can be created anywhere else, e.g.:

```
$ echo "remote enable" > ~/.config/Rserv.conf
```

Then restart Rserve like `$ R CMD Rserve --RS-conf ~/.config/Rserv.conf`.

Variant 2: Connect to Rserve through an SSH tunnel

This option is definitely more secure than variant 1. First of all communication is encrypted. Secondly you can easily control who is allowed to access Rserve from outside the host Rserve is running on.

The approach could be:

1. Create a generic account on the host running Rserve, called e.g. `rserveuser`. For this example let the host be called `rservehost`.
2. In the `rserveuser`'s home directory, inside the `~/.ssh` directory, add the public ssh key of allowed users to the `~/.ssh/authorized_keys` file. This can be done in a very special ways that only enables access to Rserve, without any other privilege like opening a remote shell etc.

   To achive this, a line in the `~/.ssh/authorized_keys` must look like:

   ```
   command="echo 'Rserve only account.'",restrict,port-forwarding,permitopen=
   ↪"localhost:6311" ssh-ed25519 AAAAC3..pxfm user1@someuserhost
   ```

3. Start rserve in normal mode, without the `remote enable` flag, so it only listens on localhost.
4. `user1` (owning the public ssh key added in 2.) then opens an SSH tunnel to `rservehost`:

   ```
   $ ssh -N -L 6311:localhost:6311 rservehost
   ```

   This command forwards traffic from port 6311 on `user`'s client machine to `localhost:6311` on `rservehost`.
5. `user1` on his/her client machine opens Python and establishes an Rserve connection with:

   ```
   >>> import pyRserve
   >>> conn = pyRserve.connect()
   ```

   The connection to `localhost:6311` on the client machine will be forwarded to Rserve listening on `localhost:6311` on `rservehost`.

Variant 3: Connect to Rserve through a Unix socket

This option might be more flexible for concurrent/dynamic connections than variants 1 and 2 and is slightly more secure than variant 1 (and less than variant 2), as the Unix socket can only be accessed from within the server.

To enable Unix sockets in Rserve a flag needs to be enabled:

```
R CMD Rserve --RS-socket /tmp/rserve.sock
```

That socket can now be used from pyRserve:

```
>>> import pyRserve
>>> conn = pyRserve.connect(unix_socket='/tmp/rserve.sock')
```

Warning: Just as in Variant 1, communication between pyRserve and Rserve is not encrypted. The only additional security is that this socket cannot be accessed from the network, but a user with access to the system can still sniff/manipulate your connection.

3.1.3 Shutting down Rserve remotely

If you need to shutdown Rserve from your client connection the following command can be called:

```
>>> conn.shutdown()
```

3.2 String evaluation in R

Having established a connection to Rserve you can run the first commands on it. A valid R command can be executed by making a call to the R name space via the connection's *eval()* method, providing a string as argument which contains valid R syntax:

```
>>> conn.eval('3 + 5')
8.0
```

In this example the string `"3 + 5"` will be sent to the remote side and evaluated by the R interpreter. The result is then delivered back into a native Python object, a floating point number in this case. As an R expert you are probably aware of the fact that R uses vectors for all numbers internally by default. But why did we receive a single floating point number? The reason is that pyRserve looks at arrays coming from Rserve and converts arrays with only one single item into an atomic value. This behaviour is for convenience reasons only.

There are two ways to override this behaviour so that the result is a real (numpy) array:

- Apply *atomicArray=True* to the *eval()*-method:

  ```
  >>> conn.eval('3 + 5', atomicArray=True)
  array([ 8.])
  ```

  This behaviour is then valid for one single call.

- Apply *atomicArray=True* to the *connect()*-function to make it the default for all calls to *eval()*:

  ```
  conn = pyRserve.connect(atomicArray=True)
  ```

  Then calling *eval()* would return a *numpy* array in every case:

  ```
  >>> conn.eval('3 + 5')
  array([ 8.])
  ```

`conn.atomicArray` will tell you how the connection handles results. This attribute contains the value of the `atomicArray` kw-argument given to connect. It can also be changed directly for a running connection.

```
>>> conn.atomicArray
True
>>> conn.atomicArray = False  # change value
```

3.3 More expression evaluation

Of course also more complex data types can be sent from R to Python, e.g. lists or real arrays. Here are some examples:

```
>>> conn.eval("list(1, 'otto')")
[1, 'otto']
>>> conn.eval('c(1, 5, 7)')
array([ 1., 5., 7.])
```

As demonstrated here R-lists are converted into plain Python lists whereas R-vectors are converted into numpy arrays on the Python side.

To set a variable inside the R namespace do:

```
>>> conn.eval('aVar <- "abc"')
'abc'
```

and to request its value just do:

```
>>> conn.eval('aVar')
'abc'
```

3.4 Expression evaluation without expecting a result

In the example above setting a variable in R did not only set the variable but also returned it back to Python:

```
>>> conn.eval('aVar <- "abc"')
'abc'
```

This is usually not something one would expect or need, and especially in the case of very large data this can cause unnecessary network traffic. The solution to this is to either call *eval()* with another option *void=True*, or to use *conn.voidEval()* directly. The following two calls are identical and do not return the string *'abc'*:

```
>>> conn.eval('aVar <- "abc"', void=True)
>>> conn.voidEval('aVar <- "abc"')
```

3.5 Defining functions and calling them through expression evaluation

It is also possible to create functions inside the R interpreter through the connector's namespace, or even to execute entire scripts. Basically you can do everything which is possible inside a normal R console:

```
# create a function and execute it:
>>> conn.voidEval('doubleit <- function(x) { x*2 }')
>>> conn.eval('doubleit(2)')
4.0

# store a mini script definition in a Python string ...
>>> my_r_script = '''
squareit <- function(x)
  { x**2 }
squareit(4)
'''
# .... and execute it in R:
>>> conn.eval(my_r_script)
16.0
```

3.6 The R namespace - setting and accessing variables in a more Pythonic way

Previous sections explained how to set a variable inside R by evaluation a statement in string format:

```
>>> conn.voidEval('aVar <- "abc"')
```

This is not very elegant and has limited ways to provide values already stored in Python variables. A much nicer way to do this is by setting the variable name in R as an attribute to a special variable *conn.r* which points to the namespace in R directly. The following statement does the same thing as the one above, just "more Pythonic":

```
>>> conn.r.aVar = "abc"
```

So of course it is then possible to compute values or copy them from Python variables into R:

```
>>> conn.r.aVar = some_python_number * 1000.505
```

To retrieve a variable from R just use it as expected:

```
>>> print('A value from R:', conn.r.aVar)
```

In its current implementation pyRserve allows to set and access the following base types:

- None (NULL)
- boolean
- integers (32-bit only)
- floating point numbers (64 bit only), i.e. doubles
- complex numbers
- strings

Furthermore the following containers are supported:

- lists
- numpy arrays
- TaggedList
- AttrArray
- TaggedArray

Lists can be nested arbitrarily, containing other lists, numbers, or arrays. `TaggedList`, `AttrArray`, and `TaggedArray` are special containers to handle very R-specific result types. They will be explained further down in the manual.

The following example shows how to assign a python list with mixed data types to an R variable called `aList`, and then to retrieve it again:

```
>>> conn.r.aList = [1, 'abcde', numpy.array([1, 2, 3], dtype=int)]
>>> conn.r.aList
[1, 'abcde', array([1, 2, 3])]
```

Numpy arrays can also contain dimension information which are translated into R matrices when assigned to the R namespace:

```
>>> arr = numpy.array(range(12))
>>> arr.shape = (3, 4)
>>> conn.r.aMatrix = arr
>>> conn.r('dim(aMatrix)')  # give me the dimension of aMatrix on the R-side
array([3, 4])
```

The result of the shape information is - in contrast to what one gets from numpy arrays - an array itself. There is nothing special about this, this is just the way R internally deals with that information.

3.7 Expression evaluation through the R namespace

Instead of using *conn.eval('1+1')* expressions can also be evaluate by making a function call on the R namespace directly. The following calls are producing the same result:

```
>>> conn.r('1+1')
>>> conn.eval('1+1')
```

conn.r('...') also accepts the 'void-option in case you want to suppress that a result is returned. Again the following three calls are producing the same result:

```
>>> conn.r('1+1', void=True)
>>> conn.eval('1+1', void=True)
>>> conn.voidEval('1+1')
```

3.8 Calling functions in R

Functions defined in R can be called as if they were a Python methods, declared in the namespace of R.

Before the examples below are usable we need to define a couple of very simple functions within the R namespace: `func0()` accepts no parameters and returns a fixed string, `func1()` takes exactly one parameter and `funcKKW()` takes keyword arguments with default values:

```
conn.voidEval('func0 <- function() { "hello world" }')
conn.voidEval('func1 <- function(v) { v*2 }')
conn.voidEval('funcKW <- function(a1=1.0, a2=4.0) { list(a1, a2) }')
```

Now calling R functions is as trivial as calling plain Python functions:

```
>>> conn.r.func0()
"hello world"
>>> conn.r.func1(5)
10
>>> conn.r.funcKW(a2=6.0)
[1.0, 6.0]
```

Of course you can also call functions built-in to R:

```
>>> conn.r.length([1,2,3])
3
```

3.9 Getting help with functions

If R is properly installed including its help messages those can be retrieved directly. Also here no surprise - just do it the Python way through the `__doc__` attribute:

```
>>> print(conn.r.sapply.__doc__)
lapply                  package:base                  R Documentation

Apply a Function over a List or Vector

Description:
```

(continues on next page)

(continued from previous page)

```
'lapply' returns a list of the same length as 'X', each element of
which is the result of applying 'FUN' to the corresponding element
of 'X'.
[...]
```

Of course this only works for functions which provide documentation. For all others `__doc__` just returns `None`.

3.10 Applying an R function as argument to another function

A typical application in R is to apply a vector to a function, especially via `sapply` and its brothers (or sisters, depending how how one sees them).

Fortunately this is as easy as you would expect:

```
>>> conn.voidEval('double <- function(x) { x*2 }')
>>> conn.r.sapply(numpy.array([1, 2, 3]), conn.r.double)
array([ 2.,  4.,  6.])
```

Here a Python array and a function defined in R are provided as arguments to the R function `sapply`.

Of course the following attempt to provide a Python function as an argument into R makes no sense:

```
>>> def double(v): return v*2
...
>>> conn.r.sapply(array([1, 2, 3]), double)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'double' is not defined
```

This will result in a NameError error because the connector tries to reference the function 'double' inside the R namespace. It should be obvious that it is not possible to transfer function implementations from Python to R.

3.11 Applying a variable already defined in R to a function

To understand why this is an interesting feature one has to understand how Python and pyRserve works. The following code is pretty inefficient:

```
>>> conn.r.arr = numpy.array([1, 2, 3])
>>> conn.r.sapply(conn.r.arr, conn.r.double)
```

To see why it is inefficient it is reproduced here more explicitly, but doing exactly the same thing:

```
>>> conn.r.arr = numpy.array([1, 2, 3])
>>> arr = conn.r.arr
>>> conn.r.sapply(arr, conn.r.double)
```

Now it is clear that the value of `conn.r.arr` is first set inside R, then retrieved back to Python (in the second line) and then again sent back to the `sapply` function. This is pretty inefficient, it would be much better just to set the array in R and then to refer to `conn.r.arr` instead of sending it back and forth. Here the "reference" namespace called `ref` comes into play:

```
>>> conn.ref.arr
<RVarProxy to variable "arr">
```

Through `conn.ref` it is possible to only reference a variable (or a function) in the R namespace without actually bringing it over to Python. Such a reference can then be passed as an argument to every function called from `conn.r`. So the proper way to make the call above is:

```
>>> conn.r.arr = numpy.array([1, 2, 3])
>>> conn.r.sapply(conn.ref.arr, conn.r.double)
```

However it is still possible to retrieve the actual content of a variable proxy through its `value()` method:

```
>>> conn.ref.arr.value()
array([1., 2., 3.])
```

So using `conn.ref` instead of `conn.r` primarily returns a reference to the remote variable in the R namespace, instead of its value. Actually we have done that before with the function `conn.r.double`. This doesn't return the R function to Python - something which would be pretty useless. Instead only a proxy to the R function is returned:

```
>>> conn.r.double
<RFuncProxy to function "double">
```

Actually functions are always returned as proxy references, both in the `conn.r` and the `conn.ref` namespace, so `conn.r.<function>` is the same as `conn.ref.<function>`.

Using reference to R variables is indeed absolutely necessary for variable content which is not transferable into Python, like special types of R classes, complex data frames etc.

3.12 Handling complex result objects from R functions

Some functions in R (especially those doing statistical calculations) return quite complex result objects.

The T-test is such an example. In the R shell you would see something like this (please ignore the silly values applied to the t test):

```
> t.test(c(1,2,3,1),c(1,6,7,8))

     Welch Two Sample t-test

data:  c(1, 2, 3, 1) and c(1, 6, 7, 8)
t = -2.3054, df = 3.564, p-value = 0.09053
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -8.4926941  0.9926941
sample estimates:
mean of x mean of y
     1.75      5.50
```

This is what you would get to see directly in your R shell.

Now, how would this convoluted result be transferred into Python objects? For this to be possible pyRserve has defined three special classes that allow for a mapping from R to Python objects. These classes are explained the the following sections. Afterwards - with that knowledge - we have a final look at the result of the t-test again.

3.12.1 TaggedLists

The first special type of container is called "TaggedList". It reflects a list-type object in R where items can be accessed in two ways as shown here (this is now pure R code):

```
> t <- list(husband="otto", wife="erna", "5th avenue")
> t[1]
$husband
[1] "otto"

> t['husband']
$husband
[1] "otto"
```

So items in the list can be either accessed via their index position, or through their "tag". Please note that the third list item ("5th avenue") is not tagged, so it can only be accessed via its index number, i.e. `t[3]` (indexing in R starts at 1 and not at zero as in Python!).

There is no direct match to any standard Python construct for a `TaggedList`. Python dictionaries do not preserve their elements' order and also don't allow for missing keys (which is why an OrderDict also doesn't help). NamedTuples on the other side would do the job but don't allow items to be appended or deleted since they are immutable.

The solution was to provide a special class in Python which is called `TaggedList`. When accessing the list `t` from the example above you'll obtain an instance of a TaggedList in Python:

```
>>> t = conn.eval('list(husband="otto", wife="erna", "5th avenue")')
>>> t
TaggedList(husband='otto', wife='erna', '5th avenue')
```

This `TaggedList` instance can be accessed in the same way as its R pendant, except for the fact the indexing is starting at zero in the usual Pythonic way:

```
>>> t[0]
'otto'
>>> t['husband']
'otto'
>>> t[2]
'5th avenue'
```

To retrieve its data suitable for instantiating another `TaggedList` on the Python side get its data as a list of tuples. This also demonstrates how a `TaggedList` can be created directly in Python:

```
>>> from pyRserve import TaggedList
>>> t.astuples
[('husband', 'otto'), ('wife', 'erna'), (None, '5th avenue')]
>>> new_tagged_list = TaggedList(t.astuples)
```

Note: `TaggedList` does not provide the full list API that one would expect, some methods are just to entirely implemented yet. However it is useful enough to retrieve all information obtained out of a R result object.

3.12.2 AttrArrays

An `AttrArray` is simply an normal numpy array, with an additional dictionary attribute called `attr`. This dicionary is used to store meta data associated to an array retrieved from R.

Let's create such an `AttrArray` in R, and transfer it into to the Python side:

```
>>> conn.voidEval("t <- c(-8.49, 0.99)")
>>> conn.voidEval("attributes(t) <- list(conf.level=0.95)")
>>> conn.r.t
AttrArray([-8.49, 0.99], attr={'conf.level': array([ 0.95])})
```

To create such an array from Python in R is also possible via:

```
>>> from pyRserve import AttrArray
>>> conn.r.t = AttrArray.new([-8.49, 0.99], {'conf.level': numpy.array([ 0.95])})
```

Instead of a list argument the `new` function also accepts a numpy array as well:

```
>>> conn.r.t = AttrArray.new(numpy.array([-8.49, 0.99]), {'conf.level': numpy.array([␣
↪0.95])})
```

3.12.3 TaggedArrays

The third special data type provided by pyRserve is the so called `TaggedArray`. It provides basically the same features as `TaggedList` above, however the underlying data type is a numpy-Array instead of a Python list. In fact, a TaggedArray is a direct subclass of `numpy.ndarray`, enhanced with some new features like accessing array cells by name as in `TaggedList`.

For the moment `TaggedArrays` only make real sense if they are 1-dimensional, so please do not change its shape. The results would not really be predictable.

To create a `TaggedArray` on the R side and transfer it to Python type:

```
>>> res = conn.eval('c(a=1.,b=2.,3.)')
>>> res
TaggedArray([ 1.,  2.,  3.], key=['a', 'b', ''])
>>> res[1]
2.0
>>> res['b']
2.0
```

The third element in the array did not obtain a name on the R side, so it is represented by an empty string in the `TaggedArray` object.

Although `TaggedArray``s are normal numpy arrays they loose their tags when further processed in Python, but still present themselves (via ``__repr__)` as `TaggedArray`. This is a current flaw in their implementation.

To create a `TaggedArray` directly in Python there is a constructor function `new()` which takes a normal 1-d numpy array as the first argument and a list of tags as the second. Both arguments must match in their size:

```
>>> from pyRserve import TaggedArray
>>> arr = TaggedArray.new(numpy.array([1, 2, 3]), ['a', 'b', ''])
>>> arr
TaggedArray([1, 2, 3], key=['a', 'b', ''])
```

3.13 Back to the t-test example

After `TaggedList` and `TaggedArray` have been introduced we can now go back to the t-test mentioned before. Let's make the same call to the test function, this time just from the Python side, and then look at the result. Again there are two ways to call it, one via string evaluation by the R interpreter, one by directly providing native Python parameters. So:

```
>>> res = conn.eval('t.test(c(1,2,3,1),c(1,6,7,8))')
```

and:

```
>>> res = conn.r.test(numpy.array([1,2,3,1]), numpy.array([1,6,7,8]))
```

does actually the same thing.

Looking at the result we get::

```
>>> res
<TaggedList(statistic=TaggedArray([-2.30541984]),
 parameter=TaggedArray([ 3.56389482], tags=['df']),
 p.value=0.090532640733331213,
 conf.int=TaggedArray([-8.49269413,  0.99269413], attr={'conf.level': array([ 0.
↪95])}),
 estimate=TaggedArray([ 1.75,  5.5 ], tags=['mean of x', 'mean of y']),
 null.value=TaggedArray([ 0.], tags=['difference in means']),
 alternative='two.sided',
 method='Welch Two Sample t-test',
 data.name='c(1, 2, 3, 1) and c(1, 6, 7, 8)')>
```

The result is an instance of a `TaggedList`, containing different types of list items.

So to access e.g. the confidence interval one would type in Python:

```
>>> res['conf.int']
AttrArray([-8.49269413,  0.99269413], attr={'conf.level': array([ 0.95])})
```

This returns an AttrArray where the confidence level is stored in an attribute called `conf.level` in the `attr`-dictionary:

```
>>> res['conf.int'].attr['conf.level']
array([ 0.95])
```

In the `res`-result data structure above there are also objects of a container called TaggedArray:

```
>>> res['estimate']
TaggedArray([ 1.75,  5.5 ], tags=['mean of x', 'mean of y'])
>>> res['estimate'][1]
5.5
>>> res['estimate']['mean of y']
5.5
```

3.14 Out Of Bounds messages (OOB)

Starting with version 1.7, Rserve allows OOB messages to be sent from R to Rserve clients, i.e. it allows for nested communication during an `eval` call.

This capability requires to start Rserve with a configuration enabling it, and loading Rserve itself as a library into the server. Both is easily accomplished in a config file (e.g. `oob.config`) like this:

```
oob enable
eval library(Rserve)
```

Then start Rserve using this config file:

```
R CMD Rserve --RS-conf oob.conf
```

OOB messaging works by calling `self.oobSend` or `self.oobMessage` in R, e.g.:

```
>>> conn.eval('self.oobSend(1)')
True
```

This does nothing but to indicate that it works. For real usefulness, one needs to register a callback that gets called with the sent data and user code as parameters:

```
>>> def printoobmsg(data, code): print(data, code)
...
>>> conn.oobCallback = printoobmsg
>>> conn.eval('self.oobSend("foo")')  # user code is 0 per default
<<< foo 0
True
```

The other function, `self.oobMessage` executes the callback and gives its return value to R:

```
>>> conn.oobCallback = lambda data, code: data**code
>>> conn.voidEval('dc <- self.oobMessage(2, 3)')
>>> conn.r.dc
8
```

The user code might be useful to create a callback convention used for switching callbacks based on agreed-upon codes:

```
>>> C_PRINT = conn.r.C_PRINT = 0
>>> C_ECHO  = conn.r.C_ECHO  = 1
>>> C_STORE = conn.r.C_STORE = 2
>>> store = []
>>> functions = {
...     C_PRINT: lambda data: print('<<<', data),
...     C_ECHO:  lambda data: data,
...     C_STORE: store.append,
... }
>>> def dispatch(data, code):
...     return functions[code](data)
>>> conn.oobCallback = dispatch
>>>
>>> conn.eval('self.oobMessage("foo", C_PRINT)')
<<< foo
>>> conn.eval('self.oobMessage("foo", C_ECHO)')
'foo'
>>> conn.eval('self.oobMessage("foo", C_STORE)')
>>> store
['foo']
>>> conn.eval('self.oobMessage('foo', 3)')
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
  File "<stdin>", line 1, in <module>
KeyError: 3
```

3.14.1 An example showing how nesting of OOB messages works

The previous examples were showing the bare application of OOB messages, but the real power of it comes when one understands how messages are getting nested within a `eval` call.

For that first create an R function which returns progress information during a "complicated" calculation:

```
>>> r_func = """
... big_job <- function(x)
... {
...     a <- x*2
...     self.oobSend('25% done')
...     b <- a * a
...     self.oobSend('50% done')
...     c <- a + b
...     self.oobSend('75% done')
...     d <- c**2
...     self.oobSend('100% done')
...     -1 * d
... }"""
>>> conn.eval(r_func)
```

Then create a progress report function, register it as a callback and then call the actual R function:

```
>>> def progress(msg, code): print(msg)
...
>>> conn.oobCallback = progress
>>> res = conn.r.big_job(5)
25% done
50% done
75% done
100% done
>>> res
-12100.0
```

CHAPTER 4

Changelog

- **V.1.0.1 (2023-01-10)**
  - Replace deprecated numpy.bool8 with **numpy.bool_**
  - Upgraded installation instructions in INSTALL file (more up-to-date R and Rserve)
  - Added Dockerfile for installing R and Rserve into container (used for github actions)
  - Enabled github actions for testing
- **V 1.0.0 (2022-10-13)**
  - Added docu for secure connection to Rserve via SSH tunnel
  - Updated meta data for pyRserve package
  - Added deprecation warning for Python 2
  - Corrected links to documentation
- **V 1.0.0b3 (2021-06-25)** Brought usage of pytest into the year 2021.
  - use fixtures for setting up an Rserve connection
  - put fixtures into conftest.py
  - added command line option for controlling rserve startup
  - properly named rserve-test.conf file.
- **V 1.0.0b2 (2021-06-22)**
  - Added missing version.txt file to wheel
- **V 1.0.0b1 (2021-06-22)**
  - Updated and cleanup documentation
  - Updated for more recent versions of R and Rserve
  - Added pre-commit hooks
  - Separated packages for dev/testing from production ones

  - Enhanced handling of NA values (thanks to Max Taggart)
  - Fixed numpy deprecation warnings (thanks to chaddcw)
- **V 0.9.2 (2019-12-19)**
  - Replaced deprecated numpy.fromstring with numpy.frombuffer
  - Flake8/pep8 cleanup
  - Refactored exception hierarchy
- **V 0.9.1 (2017-05-19)**
  - Removed a bug on some Python3 versions
  - Added proper support for S4 objects (thanks to flying-sheep)
  - Added support for Python3 unitests on travis (thanks to flying-sheep)
- **V 0.9.0 (2016-04-11)**
  - Full support for data objects larger than 2**24 bytes
  - Maximum size of message sent to Rserv can now be 2**64 bytes
- **V 0.8.4 (2015-09-06)**
  - fixed missing requirements.txt in MANIFEST.in
  - fixed bug in installer (setup.py)
- **V 0.8.3 (2015-09-04)**
  - Fixed exception catching for Python 3.4 (thanks to eeue56)
  - Some pep8 cleanups
  - explicit initialization of a number of instance variables in some classes
  - cleanup of import statements in test modules
  - Allow for message sizes greater than 4GB coming from R server
- **V 0.8.2 (2015-07-11)**
  - Added support for S4 objects (generated when e.g. creating a db object in R)
- **V 0.8.1 (2014-07-17)**
  - Fixed errors in the documentation, updated outdated parts
  - For unittesting run Rserve on different port from the default 6311 to avoid clashes with regular Rserve running on the same server
  - Fixed but when passing a R-function as argument to a function call (e.g. to `sapply`), added unittest for this
- **V 0.8.0 (2014-06-26)**
  - Added support for remote shutdown of Rserve (thanks to Uwe Schmitt)
  - Added support for Out-Of-Bounds (OOB) messages (thanks to Philipp alias flying-sheep)
- **V 0.7.3 (2013-08-01)**
  - Added missing MANIFEST.in to produce a complete tgz package (now includes docs etc)
  - Fixed bug on x64 machines when handling integers larger than 2**31
- **V 0.7.2 (2013-07-19)**

  - Tested with Python 3.3.x, R 3.0.1 and Rserve 1.7.0
  - Updated documentation accordingly
  - Code cleanup for pep8 (mostly)
  - Marked code as production stable
- **V 0.7.1 (2013-06-23)**
  - Added link to new GitHub repository
  - fixed URL to documentation
- **V 0.7.0 (2013-02-25)**
  - Fixed problem when receiving very large result sets from R (added support for XT_LARGE header flag)
  - Correctly translate multi-dimensional R arrays into numpy arrays (preserve axes the right way) Removed 'arrayOrder' keyword argument as a consequence. THIS IS AN API CHANGE - PLEASE CHECK AND ADAPT YOUR CODE, ESPECIALLY IF YOU USE MULTI-DIM ARRAYS!!
  - Support for conn.voidEval and conn.eval and new 'defaultVoid'-kw argument in the connect() function
  - Fixed bug in receiving multi-dimensional boolean (logical) arrays from R
  - Added support for multi-dimensional string arrays
  - added support for XT_VECTOR_EXPR type generated e.g. via "expression()" in R (will return a list with the expression content as list content)
  - windows users can now connect to localhost by pyRserve.connect() (omitting 'localhost' parameter)
- **V 0.6.0 (2012-06-25)**
  - support for Python3.x
  - Python versions <= 2.5 no more supported (due to Py3 support)
  - support for unicode strings in Python 2.x
  - full support complex numbers, partial support for 64bit integers and arrays
  - suport for Fortran-style ordering of numpy arrays
  - elements of single-item arrays are now translated to native python data types
  - much improved documentation
  - better unit test coverage
  - usage of the deprecated conn(<eval-string>) is no more possible
  - pyRserve.rconnect() now also removed
- **V 0.5.2 (2011-12-02)**
  - Fixed problem with 32bit integers being mistakenly rendered into 64bit integers on 64bit machines
- **V 0.5.1 (2011-11-22)**
  - Fixed improper DeprecationWarning when evaluating R statements via conn.r(...)
- **V 0.5 (2011-10-03)**
  - Renamed pyRserve.rconnect() to pyRserve.connect(). The former still works but shows a DeprecationWarning

  - String evaluation should now only be executed on the namespace directly, not on the connection object anymore. The latter still works but shows a DeprecationWarning.
  - New kw argument *atomicArray=True* added to pyRserve.connect() for preventing single valued arrays from being converted into atomic python data types.

- **V 0.4 (2011-09-20)**
  - Added support for nested function calls. E.g. conn.r.t.test(....) now works.
  - Proper support for boolean variables and vectors
- **V 0.3 (2010-06-08)**
  - Added conversion of more complex R structures into Python
  - Updated documentation (installation, manual)
- V 0.2 (2010-03-19) Fixed rendering of TaggedArrays
- V 0.1 (2010-01-10) Initial version

CHAPTER 5

This is the MIT license

(see: http://www.opensource.org/licenses/mit-license.php)

Copyright (c) 2009, 2010, 2011 Ralph Heinkel (rh [at] ralph-heinkel.com)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search