The first section contains a quick introduction, as well as how to get started (requirements, install rpy2). This should be the natural place to start if you are new to R, or rpy2. Hints for porting existing code to a newer version of rpy2 are also given

Overview

Background

Python is a popular all-purpose scripting language, while R (an open source implementation of the S/Splus language) is a scripting language mostly popular for data analysis, statistics, and graphics. If you are reading this, there are good chances that you are at least familiar with one of both.

Having an interface between both languages to benefit from the libraries of one language while working in the other appeared desirable; an early option to achieve it was the RSPython project, itself part of the Omegahat project.

A bit later, the RPy project appeared and focused on providing simple and robust access to R from within Python, with the initial Unix-only releases quickly followed by Microsoft and MacOS compatible versions. This project is referred to as RPy-1.x in the rest of this document.

The present documentation describes RPy2, an evolution of RPy-1.x. Naturally RPy2 is inspired by RPy, but also by A. Belopolskys's contributions that were waiting to be included into RPy.

This effort can be seen as a redesign and rewrite of the RPy package, and this unfortunately means there is not enough left in common to ensure compatibility.

Installation

Upgrading from an older release of rpy2

In order to upgrade one will have to first remove older installed rpy2 packages then and only then install a version of rpy2.

To do so, or to check whether you have an earlier version of rpy2 installed, do the following in a Python console:

```
import rpy2
rpy2.__path__
```

An error during execution means that you do not have any older version of rpy2 installed and you should proceed to the next section.

If this returns a string containing a path, you should go to that path and remove all files and directories starting with *rpy2*. To make sure that the cleaning is complete, open a new Python session and check that the above code results in an error.

Requirements

Currently the development is done on UNIX-like operating systems with the following software versions. Those are the recommended versions to run rpy2 with.

Software	Versions
Python	2.7
R	2.15

Running Rpy2 will require compiled libraries for R, Python, and readline; building rpy2 will require the corresponding development headers (check the documentation for more information about builing rpy2).

At the time of writing, Python 2.6 has been reported to not work with rpy2 any longer, as over time an increasing number of features present in 2.7 (backported from the Python 3 series) have been used in rpy2.

Python 3.3, touted as “the Python 3.x that is production-ready”, is not yet the preferred and recommended Python version for rpy2 but it is expected to work just as well as Python 2.7. It is just a little less “battle tested”. From rpy2-2.4.x and onward, the main Python version will be the released 3.x at the time (3.3, or 3.4). Earlier version of Python 3 are not supported (they might work, they might not - you are on your own).

Older Python like 2.5 or even 2.4 might compile, but there is much less testing done with those platforms, if any, and likely limited hope for free support.

Rpy2 is not expected to work at all with an R version < 2.8. The use of the latest rpy2 with an R version older than the current release is not adviced (and mostly unsupported).

Alternative Python implementations

CPython is the target implementation, and because of presence of C code in rpy2 is it currently not possible to run the package on Jython. For that same reason, running it with Pypy is expected to require some effort.

Download

The following options are, or could be, available for download:

- Source packages. Released versions are available on Sourceforge as well as on Pypi. Snapshots of the development version can be downloaded from bitbucket

  Note

  The repository on bitbucket has several branches. Make sure to select the one you are interested in.

- Pre-compiled binary packages for
  - Microsoft's Windows - unofficial and unsupported binaries are provided by Christoph Gohlke (http://www.lfd.uci.edu/~gohlke/pythonlibs/); there is otherwise currently close to no support for this platform
  - Apple's MacOS X (although Fink and Macports are available, there does not seem to be binaries currently available)
  - Linux distributions

rpy2 has been reported compiling successfully on all 3 platforms, provided that development items such as Python headers and a C compiler are installed.

Note

Choose files from the *rpy2* package, not *rpy*.

Note

The *pip* or *easy_install* commands can be used, although they currently only provide installation from source (see *easy_install and pip*).

Linux precompiled binaries

Linux distribution have packaging systems, and rpy2 is present in a number of them, either as a pre-compiled package or a source package compiled on-the-fly.

Note

Those versions will often be older than the latest rpy2 release.

Known distributions are: Debian and related (such as Ubuntu - often the most recent thanks to Dirk Eddelbuettel), Suse, RedHat, Mandrake, Gentoo.

On, OS X rpy2 is in Macports and Fink.

Microsoft's Windows precompiled binaries

If available, the executable can be run; this will install the package in the default Python installation.

For few releases in the 2.0.x series, Microsoft Windows binaries were contributed by Laurent Oget from Predictix.

There is currently no binaries or support for Microsoft Windows (more for lack of ressources than anything else).

Install from source

easy_install and pip

The source package is on the PYthon Package Index (PYPI), and the *pip* or *easy_install* scripts can be used whenever available. The shell command will then just be:

```
# recommended:
pip install rpy2

# or
easy_install rpy2
```

Upgrading an existing installation is done with:

```
      # recommended:
pip install rpy2 --upgrade

# or
easy_install rpy2 --upgrade
```

Both utilities have a list of options and their respective documentation should be checked for details.

source archive

To install from a downloaded source archive *<rpy_package>*, do in a shell:

```
tar -xzf <rpy_package>.tar.gz
cd <rpy_package>
python setup.py build install
```

This will build the package, guessing the R HOME from the R executable found in the PATH.

Beside the regular options for `distutils`-way of building and installing Python packages, it is otherwise possible to give explicitly the location for the R HOME:

```
python setup.py build --r-home /opt/packages/R/lib install
```

Other options to build the package are:

```
--r-home-lib # for exotic location of the R shared libraries

--r-home-modules # for R shared modules
```

Compiling on Linux

Given that you have the libraries and development headers listed above, this should be butter smooth.

The most frequent errors seem to be because of missing headers.

Compiling on OS X

XCode tools will be required in order to compile rpy2. Please refer to the documentation on the Apple site for more details about what they are and how to install them.

On OS X “Snow Leopard” (10.6.8), it was reported that setting architecture flags was sometimes needed

```
env ARCHFLAGS="-arch i386 -arch x86_64" pip install rpy2
```

or

```
env ARCHFLAGS="-arch i386 -arch x86_64" python setup.py build install
```

Some people have reported trouble with OS X “Lion”. Please check the bug tracker if you are in that situation.

Using rpy2 with other versions of R or Python

Warning

When building rpy2, it is checked that this is against a recommended version of R. Building against a different version is possible, although not supported at all, through the flag *–ignore-check-rversion*

```
python setup.py build_ext --ignore-check-rversion install
```

Since recently, development R is no longer returning an R version and the check ends with an error “Error: R >= <some version> required (and R told ‘development.’).”. The flag *–ignore-check-rversion* is then required in order to build.

Note

When compiling R from source, do not forget to specify *–enable-R-shlib* at the *./configure* step.

Test an installation

An installation can be tested for functionalities, and whenever necessary the different layers constituting the packages can be tested independently.

```
python -m 'rpy2.tests'
```

On Python 2.6, this should return that all tests were successful.

Whenever more details are needed, one can consider running explicit tests.

```
import rpy2.tests
import unittest
```

```
# the verbosity level can be increased if needed
tr = unittest.TextTestRunner(verbosity = 1)
suite = rpy2.tests.suite()
tr.run(suite)
```

Note

Running the tests in an interactive session appears to trigger spurious exceptions when testing callback functions raising exceptions. If unsure, simply use the former way to test (in a shell).

Warning

For reasons that remain to be elucidated, running the test suites used to leave the Python interpreter in a fragile state, soon crashing after the tests have been run.

It is not clear whether this is still the case, but is recommended to terminate the Python process after the tests and start working with a fresh new session.

To test the `rpy2.robjects` high-level interface:

```
python -m 'rpy2.robjects.tests.__init__'
```

or for a full control of options

```
import rpy2.robjects.tests
import unittest

# the verbosity level can be increased if needed
tr = unittest.TextTestRunner(verbosity = 1)
suite = rpy2.robjects.tests.suite()
tr.run(suite)
```

If interested in the lower-level interface, the tests can be run with:

```
python -m 'rpy2.rinterface.tests.__init__'
```

or for a full control of options

```
import rpy2.rinterface.tests
import unittest
```

```
# the verbosity level can be increased if needed
tr = unittest.TextTestRunner(verbosity = 1)
suite = rpy2.rinterface.tests.suite()
tr.run(suite)
```

Contents

The package is made of several sub-packages or modules:

rpy2.rinterface

Low-level interface to R, when speed and flexibility matter most. Close to R's C-level API.

rpy2.robjects

High-level interface, when ease-of-use matters most. Should be the right pick for casual and general use. Based on the previous one.

rpy2.interactive

High-level interface, with an eye for interactive work. Largely based on `rpy2.robjects`.

rpy2.rpy_classic

High-level interface similar to the one in RPy-1.x. This is provided for compatibility reasons, as well as to facilitate the migration to RPy2.

rpy2.rlike

Data structures and functions to mimic some of R's features and specificities in pure Python (no embedded R process).

Design notes

When designing rpy2, attention was given to:

- render the use of the module simple from both a Python or R user's perspective,

- minimize the need for knowledge about R, and the need for tricks and workarounds,
- allow to customize a lot while remaining at the Python level (without having to go down to C-level).

`rpy2.robjects` implements an extension to the interface in `rpy2.rinterface` by extending the classes for R objects defined there with child classes.

The choice of inheritance was made to facilitate the implementation of mostly inter-exchangeable classes between `rpy2.rinterface` and`rpy2.robjects`. For example, an `rpy2.rinterface.SexpClosure` can be given any `rpy2.robjects.RObject` as a parameter while any `rpy2.robjects.Function`can be given any `rpy2.rinterface.Sexp`. Because of R's functional basis, a container-like extension is also present.

The module `rpy2.rpy_classic` is using delegation, letting us demonstrate how to extend `rpy2.rinterface` with an alternative to inheritance.

Acknowledgements

Acknowledgements for contributions, support, and early testing go to (alphabetical order):

Alexander Belopolsky, Brad Chapman, Peter Cock, Dirk Eddelbuettel, Thomas Kluyver, Walter Moreira, Laurent Oget, John Owens, Nicolas Rapin, Grzegorz Slodkowicz, Nathaniel Smith, Gregory Warnes, as well as the JRI author(s), the R authors, R-help list responders, Numpy list responders, and other contributors.

Introduction to rpy2

This introduction aims at making a gentle start to rpy2, either when coming from R to Python/rpy2, from Python to rpy2/R, or from elsewhere to Python/rpy2/R.

Getting started

It is assumed here that the rpy2 package has been properly installed. In python, making a package or module available is achieved by importing it:

```
import rpy2.robjects as robjects
```

The *r* instance

The object `r` in `rpy2.robjects` represents the running embedded *R* process.

If familiar with R and the R console, `r` is a little like a communication channel from Python to R.

Getting R objects

In Python the *[* operator is an alias for the ethod `__getitem__()`.

The `__getitem__()` method of `rpy2.robjects.r`, evaluates a variable from the R console.

Example in R:

```
> pi
[1] 3.141593
```

With `rpy2`:

```
>>> pi = robjects.r['pi']
>>> pi[0]
3.14159265358979
```

Note

Under the hood, the variable *pi* is gotten by default from the R *base* package, unless an other variable with the name *pi* was created in R's *.globalEnv*.

Whenever one wishes to be specific about where the symbol should be looked for (which should be most of the time), it possible to wrap R packages in Python namespace objects (see *R packages*).

For more details on environments, see Section *Environments*.

Also, note that *pi* is not a scalar but a vector of length 1

Evaluating R code

The `r` object is also callable, and the string passed to it evaluated as *R* code.

This can be used to *get* variables, and provide an alternative to the method presented above.

Example in R:

```
> pi
```

```
[1] 3.141593
```

With rpy2:

```
>>> pi = robjects.r('pi')
>>> pi[0]
3.14159265358979
```

Warning

The result is an R vector. The Section *R vectors* below will provide explanation for the following behavior:

```
>>> piplus2 = robjects.r('pi') + 2
>>> piplus2.r_repr()
c(3.14159265358979, 2)
>>> pi0plus2 = robjects.r('pi')[0] + 2
>>> print(pi0plus2)
5.1415926535897931
```

The evaluation is performed in what is known to R users as the *Global Environment*, that is the place one starts at when starting the R console. Whenever the *R* code creates variables, those variables are "located" in that *Global Environment* by default.

Example:

```
robjects.r('''
        f <- function(r, verbose=FALSE) {
            if (verbose) {
                cat("I am calling f().\n")
            }
            2 * pi * r
        }
        f(3)
        ''')
```

The expression above returns the value 18.85, but first creates an R function *f*. That function *f* is present in the R *Global Environement*, and can be accessed with the *__getitem__* mechanism outlined above:

```
>>> r_f = robjects.globalenv['f']
>>> print(r_f.r_repr())
```

```
function (r, verbose = FALSE)
{
    if (verbose) {
        cat("I am calling f().\n")
    }
    2 * pi * r
}
```

Note

As shown earlier, an alternative way to get the function is to get it from the `R` singleton

```
>>> r_f = robjects.r['f']
```

The function r_f is callable, and can be used like a regular Python function.

```
>>> res = r_f(3)
```

Jump to Section *Calling R functions* for more on calling functions.

Interpolating R objects into R code strings

Against the first impression one may get from the title of this section, simple and handy features of `rpy2` are presented here.

An R object has a string representation that can be used directly into R code to be evaluated.

Simple example:

```
>>> letters = robjects.r['letters']
>>> rcode = 'paste(%s, collapse="-")' %(letters.r_repr())
>>> res = robjects.r(rcode)
>>> print(res)
"a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z"
```

R vectors

In *R*, data are mostly represented by vectors, even when looking like scalars.

When looking closely at the R object *pi* used previously, we can observe that this is in fact a vector of length 1.

```
>>> len(robjects.r['pi'])
1
```

As such, the python method `add()` will result in a concatenation (function *c()* in R), as this is the case for regular python lists.

Accessing the one value in that vector has to be stated explicitly:

```
>>> robjects.r['pi'][0]
3.1415926535897931
```

There is much that can be achieved with vectors, having them to behave more like Python lists or R vectors. A comprehensive description of the behavior of vectors is found in `robjects.vector`.

Creating rpy2 vectors

Creating R vectors can be achieved simply:

```
>>> res = robjects.StrVector(['abc', 'def'])
>>> print(res.r_repr())
c("abc", "def")
>>> res = robjects.IntVector([1, 2, 3])
>>> print(res.r_repr())
1:3
>>> res = robjects.FloatVector([1.1, 2.2, 3.3])
>>> print(res.r_repr())
c(1.1, 2.2, 3.3)
```

R matrixes and arrays are just vectors with a *dim* attribute.

The easiest way to create such objects is to do it through R functions:

```
>>> v = robjects.FloatVector([1.1, 2.2, 3.3, 4.4, 5.5, 6.6])
>>> m = robjects.r['matrix'](v, nrow = 2)
>>> print(m)
     [,1] [,2] [,3]
[1,]  1.1  3.3  5.5
[2,]  2.2  4.4  6.6
```

Calling R functions

Calling R functions is disappointingly similar to calling Python functions:

```
>>> rsum = robjects.r['sum']
>>> rsum(robjects.IntVector([1,2,3]))[0]
6L
```

Keywords are also working:

```
>>> rsort = robjects.r['sort']
>>> res = rsort(robjects.IntVector([1,2,3]), decreasing=True)
>>> print(res.r_repr())
c(3L, 2L, 1L)
```

Note

By default, calling R functions return R objects.

More information on functions is in Section *Functions*.

Getting help

R has a builtin help system that, just like the pydoc strings are used frequently in python during interactive sessions, is used very frequently by R programmmers. This help system is accessible from an R function, therefore accessible from rpy2.

Help on a topic within a given package, or currently loaded packages

```
>>> from rpy2.robjects.packages import importr
>>> utils = importr("utils")
>>> help_doc = utils.help("help")
>>> help_doc[0]
'/where/R/is/installed/library/utils/help/help'
```

Converting the object returned to a string produces the full help text on the topic:

```
>>> str(help_doc)
[...long output...]
```

Warning

The help message so produced is not a string returned to the console but is directly printed by R to the standard output. The call to `str()` only returns an empty string, and the reason for this is somewhat involved for an introductory documentation. This behaviour is rooted in **R** itself and in `rpy2` the string representation of R objects is the string representation as given by the **R**console, which in that case takes a singular route.

For a Python friendly help to the R help system, consider the module `rpy2.robjects.help`.

Locate topics among available packages

```
>>> help_where = utils.help_search("help")
```

As before with *help*, the result can be printed / converted to a string, giving a similar result to what is obtained from an R session.

Note

The data structure returned can otherwise be used to access the information returned in details.

```
>>> tuple(help_where)
(<StrVector - Python:0x1f9a968 / R:0x247f908>,
 <StrVector - Python:0x1f9a990 / R:0x25079d0>,
 <StrVector - Python:0x1f9a9b8 / R:0x247f928>,
 <Matrix - Python:0x1f9a850 / R:0x1ec0390>)
>>> tuple(help_where[3].colnames)
('topic', 'title', 'Package', 'LibPath')
```

However, this is beyond the scope of an introduction, and one should master the content of the module `robjects.vector` before anything else.

Examples

This section demonstrates some of the features of rpy2.

Graphics and plots

```
import rpy2.robjects as robjects

r = robjects.r

x = robjects.IntVector(range(10))
y = r.rnorm(10)

r.X11()

r.layout(r.matrix(robjects.IntVector([1,2,3,2]), nrow=2, ncol=2))
r.plot(r.runif(10), y, xlab="runif", ylab="foo/bar", col="red")
```

Setting dynamically the number of arguments in a function call can be done the usual way in python.

There are several ways to plot data in *R*, some of which are presented in this documentation:

```
import math, datetime
import rpy2.robjects.lib.ggplot2 as ggplot2
import rpy2.robjects as ro
from rpy2.robjects.packages import importr
base = importr('base')

mtcars = datasets.__rdata__.fetch('mtcars')['mtcars']
pp = ggplot2.ggplot(mtcars) + \
     ggplot2.aes_string(x='wt', y='mpg', col='factor(cyl)') + \
     ggplot2.geom_point() + \
     ggplot2.geom_smooth(ggplot2.aes_string(group = 'cyl'),
                         method = 'lm')
pp.plot()
```

More about plots and graphics in R, as well as more advanced plots are presented in Section *Graphics*.

Warning

By default, the embedded R open an interactive plotting device, that is a window in which the plot is located. Processing interactive events on that devices, such as resizing or closing the window must be explicitly required (see Section *Processing interactive events*).

Linear models

The R code is:

```
ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)
trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
group <- gl(2, 10, 20, labels = c("Ctl","Trt"))
weight <- c(ctl, trt)

anova(lm.D9 <- lm(weight ~ group))

summary(lm.D90 <- lm(weight ~ group - 1))# omitting intercept
```

One way to achieve the same with `rpy2.robjects` is

```
from rpy2.robjects import FloatVector
from rpy2.robjects.packages import importr
stats = importr('stats')
base = importr('base')

ctl = FloatVector([4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14])
trt = FloatVector([4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69])
group = base.gl(2, 10, 20, labels = ["Ctl","Trt"])
weight = ctl + trt

robjects.globalenv["weight"] = weight
robjects.globalenv["group"] = group
lm_D9 = stats.lm("weight ~ group")
print(stats.anova(lm_D9))

# omitting the intercept
lm_D90 = stats.lm("weight ~ group - 1")
print(base.summary(lm_D90))
```

This way to perform a linear fit it matching precisely the way in R presented above, but there are other ways (see Section *Formulae* for storing the variables directly in the lookup environment of the formula).

Q: Now how to extract data from the resulting objects ?

A: Well, it all depends on the object. R is very much designed for interactive sessions, and users often inspect what a function is returning in order to know how to extract information.

When taking the results from the code above, one could go like:

```
>>> print(lm_D9.rclass)
[1] "lm"
```

Here the resulting object is a list structure, as either inspecting the data structure or reading the R man pages for *lm* would tell us. Checking its element names is then trivial:

```
>>> print(lm_D9.names)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "contrasts"     "xlevels"       "call"          "terms"
[13] "model"
```

And so is extracting a particular element:

```
>>> print(lm_D9.rx2('coefficients'))
(Intercept)    groupTrt
      5.032      -0.371
```

or

```
>>> print(lm_D9.rx('coefficients'))
$coefficients
(Intercept)    groupTrt
      5.032      -0.371
```

More about extracting elements from vectors is available at *Extracting items*.

Principal component analysis

The R code is

```
m <- matrix(rnorm(100), ncol=5)
pca <- princomp(m)
plot(pca, main="Eigen values")
biplot(pca, main="biplot")
```

The `rpy2.robjects` code can be as close to the R code as possible:

```
import rpy2.robjects as robjects

r = robjects.r

m = r.matrix(r.rnorm(100), ncol=5)
pca = r.princomp(m)
r.plot(pca, main="Eigen values")
r.biplot(pca, main="biplot")
```

However, the same example can be made a little tidier (with respect to being specific about R functions used)

```
from rpy2.robjects.packages import importr

base     = importr('base')
stats    = importr('stats')
graphics = importr('graphics')

m = base.matrix(stats.rnorm(100), ncol = 5)
pca = stats.princomp(m)
graphics.plot(pca, main = "Eigen values")
stats.biplot(pca, main = "biplot")
```

Creating an R vector or matrix, and filling its cells using Python code

```
from rpy2.robjects import NA_Real
from rpy2.rlike.container import TaggedList
from rpy2.robjects.packages import importr

base = importr('base')

# create a numerical matrix of size 100x10 filled with NAs
m = base.matrix(NA_Real, nrow=100, ncol=10)

# fill the matrix
for row_i in xrange(1, 100+1):
    for col_i in xrange(1, 10+1):
        m.rx[TaggedList((row_i, ), (col_i, ))] = row_i + col_i * 100
None
```

One more example

```
"""
short demo.

"""

from rpy2.robjects.packages import importr
graphics = importr('graphics')
grdevices = importr('grDevices')
base = importr('base')
stats = importr('stats')

import array

x = array.array('i', range(10))
y = stats.rnorm(10)

grdevices.X11()

graphics.par(mfrow = array.array('i', [2,2]))
graphics.plot(x, y, ylab = "foo/bar", col = "red")

kwargs = {'ylab':"foo/bar", 'type':"b", 'col':"blue", 'log':"x"}
graphics.plot(x, y, **kwargs)


m = base.matrix(stats.rnorm(100), ncol=5)
pca = stats.princomp(m)
graphics.plot(pca, main="Eigen values")
stats.biplot(pca, main="biplot")
```