rpy2 Documentation

Release 2.1.0

Laurent Gautier

April 15, 2011

CONTENTS

CHAPTER

ONE

OVERVIEW

1.1 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 the two.

Having an interface between both languages, and benefit from the respective libraries of one language while working in the other language, appeared desirable and 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.

1.2 Installation

1.2.1 Requirements

Python version 2.4 or higher, and R-2.7.0 or higher are required.

Although the development was first done with R-2.7.2 (now with R-2.8.0) and Python-2.5.2, it has also been tested with:

- Python-2.6.0 (numpy-support not tested)

gcc-4.2.3, then *gcc-4.2.4* were used for compiling the C parts.

1.2.2 Download

In theory we could have available for download:

- Source packages.
- Pre-compiled binary packages for
  - Microsoft's Windows
  - Apple's MacOS X
  - 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.

Check on the Sourceforge download page what is available.

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

1.2.3 Microsoft's Windows precompiled binaries

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

At the time of writing, Microsoft Windows binaries are contributed by Laurent Oget (from Predictix) since version 2.0.0b1.

1.2.4 Install from source

To install from a source package <*rpy_package*> do in a shell:

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

1.2.5 Test an installation

At any time, an installation can be tested as follows:

```
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: At the time of writing, 2 unit tests will fail. Their failure is forced, because terminating then starting again an embbeded R is causing problems.

Warning: Win32 versions are still lacking some of the functionalities in the UNIX-alike versions, most notably the callback function for console input and output.

1.3 Contents

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

1.3.1 `rpy2.rpy_classic`

Higher-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.

1.3.2 `rpy2.robjects`

Higher-level interface, when ease-of-use matters most.

1.3.3 `rpy2.rinterface`

Low-level interface to R, when speed and flexibility matter most. Here the programmer gets close(r) to R's C-level API.

1.3.4 `rpy2.rlike`

Data structures and functions to mimic some of R's features and specificities

1.4 Design notes

When designing ryp2, attention was given to make:

- 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.
- the possibility 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 interexchangeable 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.RFunction 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.

1.5 Acknowledgements

Acknowledgements go to:

Walter Moreira, and Gregory Warnes For the original RPy and its maintainance through the years.

Alexander Belopolsky. His code contribution of an alternative RPy is acknowledged. I have found great inspiration in reading that code.

JRI The Java-R Interface, and its authors, as answers to some of the implementation questions were found there.

Contributors The help of people, donating time, ideas or software patches is much appreciated. Their names can be found in this documentation (mostly around the section Changes).

CHAPTER

TWO

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.

2.1 Getting started

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

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

2.2 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.

2.2.1 Getting R objects

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

With `rpy2.robjects`, the method `__getitem__()` functions like calling a variable from the R console.

Example in R:

```
pi
```

With `rpy2`:

```
>>> robjects.r['pi']
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 the *globalEnv*. The Section *Environments* tells more about that.

2.2.2 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
```

With `rpy2`:

```
>>> robjects.r('pi')
3.14159265358979
```

Warning: The result is an R vector. Reading Section *R vectors* is recommended as it will provide explanations for the following behavior:

```
>>> robjects.r('pi') + 2
c(3.14159265358979, 2)
>>> robjects.r('pi')[0] + 2
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 will be "located" in that *Global Environment* by default.

Example:

```
robjects.r('''
        f <- function(r) { 2 * pi * r }
        f(3)
        ''')
```

The expression above will return the value 18.85, but first also 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:

```
>>> robjects.globalEnv['f']
function (r)
{
    2 * pi * r
}
```

or

```
>>> robjects.r['f']
function (r)
{
    2 * pi * r
}
```

2.2.3 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())
>>> robjects.r(rcode)
"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"
```

2.3 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 will have to be stated explicitly:

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

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

2.3.1 Creating rpy2 vectors

Creating R vectors can be achieved simply:

```
>>> robjects.StrVector(['abc', 'def'])
c("abc", "def")
>>> robjects.IntVector([1, 2, 3])
1:3
```

```
>>> robjects.FloatVector([1.1, 2.2, 3.3])
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
```

2.4 Calling R functions

Calling R functions will be disappointingly similar to calling Python functions:

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

Keywords can be used with the same ease:

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

Note: By default, calling R functions will return R objects.

More information on functions is in Section *Functions*.

2.5 Examples

This section demonstrates some of the features of rpy2 by the example.

2.5.1 Function calls and plotting

```
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

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

Note: Since the named parameters are a Python `dict`, the order of the parameters is lost for ***kwargs* arguments.

2.5.2 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

```
import rpy2.robjects as robjects

r = robjects.r

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

robjects.globalEnv["weight"] = weight
robjects.globalEnv["group"] = group
lm_D9 = r.lm("weight ~ group")
print(r.anova(lm_D9))

lm_D90 = r.lm("weight ~ group - 1")
print(r.summary(lm_D90))
```

Q: Now how extract data from the resulting objects ?

A: The same, never it is. On the R object all depends.

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.r['coefficients'])
$coefficients
(Intercept)    groupTrt
      5.032      -0.371
```

More about extracting elements from vectors is available at *Indexing*.

2.5.3 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 is

```
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")
```

CHAPTER

THREE

HIGH-LEVEL INTERFACE

Platforms: Unix, Windows

3.1 Overview

This module is intended for casual and general use. Its aim is to abstracts some of the details and provide an intuitive interface to R.

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

`rpy2.robjects` is written on the top of `rpy2.rinterface`, and one not satisfied with it could easily build one's own flavor of a Python-R interface by modifying it (`rpy2.rpy_classic` is an other example of a Python interface built on the top of `rpy2.rinterface`).

Visible differences with RPy-1.x are:

- no `CONVERSION` mode in `rpy2`, the design has made this unnecessary
- easy to modify or rewrite with an all-Python implementation

3.2 *r*: the instance of *R*

This class is currently a singleton, with its one representation instanciated when the module is loaded:

```
>>> robjects.r
>>> print(robjects.r)
```

The instance can be seen as the entry point to an embedded R process.

The elements that would be accessible from an equivalent R environment are accessible as attributes of the instance. Readers familiar with the `ctypes` module for Python will note the similarity with it.

R vectors:

```
>>> pi = robjects.r.pi
>>> letters = robjects.r.letters
```

R functions:

```
>>> plot = robjects.r.plot
>>> dir = robjects.r.dir
```

This approach has limitation as:

- The actual Python attributes for the object masks the R elements
- **'.' (dot) is syntactically valid in names for R objects, but not for** python objects.

That last limitation can partly be removed by using `rpy2.rpy_classic` if this feature matters most to you.

```
>>> robjects.r.as_null
# AttributeError raised
>>> import rpy2.rpy_classic as rpy
>>> rpy.set_default_mode(NO_CONVERSION)
>>> rpy.r.as_null
# R function as.null() returned
```

Note: The section *Partial use of rpy_classic* outlines how to integrate `rpy2.rpy_classic` code.

Behind the scene, the steps for getting an attribute of *r* are rather straightforward:

1. Check if the attribute is defined as such in the python definition for *r*
2. Check if the attribute is can be accessed in R, starting from *globalEnv*

When safety matters most, we recommend using `__getitem__()` to get a given R object.

```
>>> as_null = robjects.r['as.null']
```

Storing the object in a python variable will protect it from garbage collection, even if deleted from the objects visible to an R user.

```
>>> robjects.globalEnv['foo'] = 1.2
>>> foo = robjects.r['foo']
>>> foo[0]
1.2
```

Here we *remove* the symbol *foo* from the R Global Environment.

```
>>> robjects.r['rm']('foo')
>>> robjects.r['foo']
LookupError: 'foo' not found
```

The object itself remains available, and protected from R's garbage collection until *foo* is deleted from Python

```
>>> foo[0]
1.2
```

3.2.1 Strings as R code

Just like it is the case with RPy-1.x, on-the-fly evaluation of R code contained in a string can be performed by calling the *r* instance:

```
>>> robjects.r('1+2')
3
>>> sqr = ro.r('function(x) x^2)

>>> sqr
function (x)
x^2
>>> sqr(2)
4
```

The astute reader will quickly realize that R objects named by python variables can be plugged into code through their *R* representation:

```
>>> x = robjects.r.rnorm(100)
>>> robjects.r('hist(%s, xlab="x", main="hist(x)")' %x.r_repr())
```

Warning: Doing this with large objects might not be the best use of your computing power.

3.3 R objects

The class `rpy2.robjects.RObject` represents an arbitray R object, meaning than object cannot be represented by any of the classes `RVector`, `RFunction`, `REnvironment`.

The class inherits from the class `rpy2.rinterface.Sexp`.

3.4 Vectors

Beside functions, and environemnts, most of the objects an R user is interacting with are vector-like. For example, this means that any scalar is in fact a vector of length one.

The class `RVector` has a constructor:

```
>>> x = robjects.RVector(3)
```

The class inherits from the class `rpy2.rinterface.VectorSexp`.

3.4.1 Creating vectors

Creating vectors can be achieved either from R or from Python.

When the vectors are created from R, one should not worry much as they will be exposed as they should by `rpy2.robjects`.

When one wants to create a vector from Python, either the class `RVector` or the convenience classes `IntVector`, `FloatVector`, `BoolVector`, `StrVector` can used.

class `rpy2.robjects.`**`BoolVector`**(*obj*)
Bases: `rpy2.robjects.RVector`

Vector of boolean (logical) elements

class `rpy2.robjects.`**`IntVector`**(*obj*)
Bases: `rpy2.robjects.RVector`

Vector of integer elements

class `rpy2.robjects.`**`FloatVector`**(*obj*)
Bases: `rpy2.robjects.RVector`

Vector of float (double) elements

class `rpy2.robjects.`**`StrVector`**(*obj*)
Bases: `rpy2.robjects.RVector`

Vector of string elements

3.4.2 Indexing

Indexing can become a thorny issue, since Python indexing starts at zero and R indexing starts at one.

The python `__getitem__()` method behaves like a Python user would expect it for a vector (and indexing starts at zero), while the method `subset()` behaves like a R user would expect subsetting to happen that is:

- indexing starts at one
- the parameter to subset on can be a vector of
  - integers (negative integers meaning exlusion of the element)
  - booleans
  - strings

```
>>> x = robjects.r.seq(1, 10)
>>> x[0]
1
>>> x.subset(0)
integer(0)
>>> x.subset(1)
1L
```

Rather than calling `subset()`, and to still have the conveniently short *[* operator available, a syntactic sugar is available in the form of delegating-like attribute `r`.

```
>>> x.r[0]
integer(0)
>>> x.r[1]
1L
```

The two next examples demonstrate some of *R*'s features regarding indexing, respectively element exclusion and recycling rule:

```
>>> x.r[-1]
2:10
>>> x.r[True]
1:10
```

This class is extending the class `rinterface.SexpVector`, and its documentation can be referred to for details of what is happenening at the low-level.

3.4.3 Operators

Mathematical operations on two vectors: the following operations are performed element-wise in R, recycling the shortest vector if, and as much as, necessary.

The delegating attribute mentioned in the Indexing section can also be used with the following operators:

operator	R (.r)
+	Add
-	Subtract
*	Multiply
/	Divide
**	Power
or	Or
and	And

```
>>> x = robjects.r.seq(1, 10)
>>> x.r + 1
2:11
```

Note: In Python, the operator + concatenate sequence object, and this behavior has been conserved.

Note: The boolean operator `not` cannot be redefined in Python (at least up to version 2.5), and its behavior could not be made to mimic R's behavior

---

3.4.4 Names

`R` vectors can have a name given to all or some of the items. The method `getnames()` retrieve those names.

3.4.5 `RArray`

In *R*, arrays are simply vectors with a dimension attribute. That fact was reflected in the class hierarchy with `robjects.RArray` inheriting from `robjects.RVector`.

3.4.6 `RMatrix`

A `RMatrix` is a special case of `RArray`.

3.4.7 Data frames

Data frames are important data structures in R, as they are used to represent a data to analyze in a study in a relatively large nunmber of cases.

A data frame can be thought of as a tabular representation of data, with one variable per column, and one data point per row. Each column is an R vector, which implies one type for all elements in one given column, and which allows for possibly different types across different columns.

In `rpy2.robjects`, `RDataFrame` represents the *R* class *data.frame*.

Creating an `RDataFrame` can be done by:

- Using the constructor for the class
- Create the data.frame through R

The constructor for `RDataFrame` accepts either a `rinterface.SexpVector` (with `typeof` equal to *VECSXP*, that is an R *list*) or an instance of class `rpy2.rlike.container.TaggedList`.

```
>>> robjects.RDataFrame()
```

Creating the data.frame in R can be achieved in numerous ways, as many R functions do return a data.frame. In this example, will use the R function *data.frame()*, that constructs a data.frame from named arguments

```
>>> d = {'value': robjects.IntVector((1,2,3)),
         'letter': robjects.StrVector(('x', 'y', 'z'))}
>>> dataf = robjects.r['data.frame'](**d)
>>> dataf.colnames()
c("letter", "value")
```

Note: The order of the columns *value* and *letter* cannot be conserved, since we are using a Python dictionnary. This difference between R and Python can be resolved by using TaggedList instances (XXX add material about that).

---

class `rpy2.robjects.`**`RDataFrame`**(*tlist*)

Bases: `rpy2.robjects.RVector`

R 'data.frame'.

`colnames`()

Column names

Return type SexpVector

`ncol`()

Number of columns. :rtype: integer

`nrow`()

Number of rows. :rtype: integer

`rownames`()

Row names

Return type SexpVector

3.5 Environments

R environments can be described to the Python user as an hybrid of a dictionary and a scope.

The first of all environments is called the Global Environment, that can also be referred to as the R workspace.

```
>>> globalEnv = robjects.globalEnv
```

An R environment in RPy2 can be seen as a kind of Python dictionnary.

Assigning a value to a symbol in an environment has been made as simple as assigning a value to a key in a Python dictionary:

```
>>> robjects.r.ls(globalEnv)
>>> globalEnv["a"] = 123
>>> robjects.r.ls(globalEnv)
```

Care must be taken when assigning objects into an environment such as the Global Environment, as this can hide other objects with an identical name. The following example should make one measure that this can mean trouble if no care is taken:

```
>>> globalEnv["pi"] = 123
>>> robjects.r.pi
123L
>>>
>>> robjects.r.rm("pi")
```

```
>>> robjects.r.pi
3.1415926535897931
```

The class inherits from the class `rpy2.rinterface.SexpEnvironment`.

An environment is also iter-able, returning all the symbols (keys) it contains:

```
>>> env = robjects.r.baseenv()
>>> len([x for x in env])
<a long list returned>
```

For further information, read the documentation for the class `rpy2.rinterface.SexpEnvironment`.

3.6 Functions

R functions are callable objects, and be called almost like any regular Python function:

```
>>> plot = robjects.r.plot
>>> rnorm = robjects.r.rnorm
>>> plot(rnorm(100), ylab="random")
```

This is all looking fine and simple until R parameters with names such as *na.rm* are encountered. In those cases, using the special syntax ***kwargs* is one way to go.

Let's take an example in R:

```
sum(0, na.rm = TRUE)
```

In Python it can then write:

```
from rpy2 import robjects

myparams = {'na.rm': True}
robjects.r.sum(0, **myparams)
```

Things are also not always that simple, as the use of dictionary does ensure that the order in which the parameters are passed is conserved.

The R functions as defined in `rpy2.robjects` inherit from the class `rpy2.rinterface.SexpClosure`, and further documentation on the behavior of function can be found in Section *Functions*.

3.7 Formulae

For tasks such as modelling and plotting, an R formula can be a terse, yet readable, way of expressing what is wanted.

In R, it generally looks like:

```
x <- 1:10
y <- x + rnorm(10, sd=0.2)

fit <- lm(y ~ x)
```

In the call to *lm*, the argument is a *formula*, and it can read like *model y using x*. A formula is a *R* language object, and the terms in the formula are evaluated in the environment it was defined in. Without further specification, that environment is the environment in which the the formula is created.

The class `robjects.RFormula` is representing an *R* formula.

```
x = robjects.RVector(array.array('i', range(1, 11)))
y = x.r + robjects.r.rnorm(10, sd=0.2)

fmla = robjects.RFormula('y ~ x')
env = fmla.getenvironment()
env['x'] = x
env['y'] = y

fit = robjects.r.lm(fmla)
```

One drawback with that approach is that pretty printing of the *fit* object is note quite as clear as what one would expect when working in *R*. However, by evaluating R code on the fly, we can obtain a *fit* object that will display nicely:

```
fit = robjects.r('lm(%s)' %fmla.r_repr())
```

CHAPTER

FOUR

NUMPY

A popular solution for scientific computing with Python is `numpy` (previous instances were `Numpy` and `numarray`).

`rpy2` has features for facilitating the integration with code using `numpy` in both directions: from *rpy2* to *numpy*, and from *numpy* to *rpy2*.

4.1 High-level interface

4.1.1 From *rpy2* to *numpy*:

Vectors can be converted to `numpy` arrays using `array()` or `asarray()`:

```
import numpy

ltr = robjects.r.letters
ltr_np = numpy.array(ltr)
```

This behavior is inherited from the low-level interface, and is means that the objects presents an interface recognized by *numpy*, and that interface used to know the structure of the object.

4.1.2 From *numpy* to *rpy2*:

The conversion of *numpy* objects to *rpy2* objects can be activated by importing the module `numpy2ri`:

```
import rpy2.robjects.numpy2ri
```

That import alone is sufficient to switch an automatic conversion of *numpy* objects into *rpy2* objects.

Note: Why make this an optional import, while it could have been included in the function `py2ri()` (as done in the original patch submitted for that function) ?

Although both are valid and reasonable options, the design decision was taken in order to decouple *rpy2* from *numpy* the most, and do not assume that having *numpy* installed automatically meant that a programmer wanted to use it.

```
import rpy2.robjects as ro
import rpy2.rinterface as rinterface
import numpy

def numpy2ri(o):
    if isinstance(o, numpy.ndarray):
        if not o.dtype.isnative:
            raise(ValueError("Cannot pass numpy arrays with non-native byte orde

        # The possible kind codes are listed at
        #   http://numpy.scipy.org/array_interface.shtml
        kinds = {
            # "t" -> not really supported by numpy
            "b": rinterface.LGLSXP,
            "i": rinterface.INTSXP,
            # "u" -> special-cased below
            "f": rinterface.REALSXP,
            "c": rinterface.CPLXSXP,
            # "O" -> special-cased below
            "S": rinterface.STRSXP,
            "U": rinterface.STRSXP,
            # "V" -> special-cased below
            }
        # Most types map onto R arrays:
        if o.dtype.kind in kinds:
            # "F" means "use column-major order"
            vec = rinterface.SexpVector(o.ravel("F"), kinds[o.dtype.kind])
            dim = rinterface.SexpVector(o.shape, rinterface.INTSXP)
            res = ro.r.array(vec, dim=dim)
        # R does not support unsigned types:
        elif o.dtype.kind == "u":
            raise(ValueError("Cannot convert numpy array of unsigned values -- R
        # Array-of-PyObject is treated like a Python list:
        elif o.dtype.kind == "O":
            res = ro.conversion.py2ri(list(o))
        # Record arrays map onto R data frames:
        elif o.dtype.kind == "V":
            if o.dtype.names is None:
                raise(ValueError("Nothing can be done for this numpy array type
            df_args = []
            for field_name in o.dtype.names:
                df_args.append((field_name,
                                ro.conversion.py2ri(o[field_name])))
            res = ro.baseNameSpaceEnv["data.frame"].rcall(tuple(df_args))
        # It should be impossible to get here:
        else:
```

```
            raise(ValueError("Unknown numpy array type."))
    else:
        res = ro.default_py2ri(o)
    return res


ro.conversion.py2ri = numpy2ri
```

4.2 Low-level interface

The `rpy2.rinterface.SexpVector` objects are made to behave like arrays, as defined in the Python package `numpy`.

The functions `numpy.array()` and `numpy.asarray()` can be used construct *numpy* arrays:

```
>>> import numpy
>>> rx = rinterface.SexpVector([1,2,3,4], rinterface.INTSXP)
>>> nx = numpy.array(rx)
>>> nx_nc = numpy.asarray(rx)
```

Note: when using `asarray()`, the data are not copied.

```
>>> rx[2]
3
>>> nx_nc[2] = 42
>>> rx[2]
42
>>>
```

CHAPTER

FIVE

NUMPY

A popular solution for scientific computing with Python is `numpy` (previous instances were `Numpy` and `numarray`).

`rpy2` has features for facilitating the integration with code using `numpy` in both directions: from *rpy2* to *numpy*, and from *numpy* to *rpy2*.

5.1 High-level interface

5.1.1 From *rpy2* to *numpy*:

Vectors can be converted to `numpy` arrays using `array()` or `asarray()`:

```
import numpy

ltr = robjects.r.letters
ltr_np = numpy.array(ltr)
```

This behavior is inherited from the low-level interface, and is means that the objects presents an interface recognized by *numpy*, and that interface used to know the structure of the object.

5.1.2 From *numpy* to *rpy2*:

The conversion of *numpy* objects to *rpy2* objects can be activated by importing the module `numpy2ri`:

```
import rpy2.robjects.numpy2ri
```

That import alone is sufficient to switch an automatic conversion of *numpy* objects into *rpy2* objects.

Note: Why make this an optional import, while it could have been included in the function `py2ri()` (as done in the original patch submitted for that function) ?

Although both are valid and reasonable options, the design decision was taken in order to decouple *rpy2* from *numpy* the most, and do not assume that having *numpy* installed automatically meant that a programmer wanted to use it.

Note: The module `numpy2ri` is an example of how custom conversion to and from `rpy2.robjects` can be performed.

5.2 Low-level interface

The `rpy2.rinterface.SexpVector` objects are made to behave like arrays, as defined in the Python package `numpy`.

The functions `numpy.array()` and `numpy.asarray()` can be used construct *numpy* arrays:

```
>>> import numpy
>>> rx = rinterface.SexpVector([1,2,3,4], rinterface.INTSXP)
>>> nx = numpy.array(rx)
>>> nx_nc = numpy.asarray(rx)
```

Note: when using `asarray()`, the data are not copied.

```
>>> rx[2]
3
>>> nx_nc[2] = 42
>>> rx[2]
42
>>>
```

CHAPTER

SIX

MAPPING RPY2 OBJECTS TO ARBITRARY PYTHON OBJECTS

Switching between a conversion and a no conversion mode, an operation often present when working with RPy-1.x, is no longer necessary as the R objects can be either passed on to R functions or used in Python.

However, there is a low-level mapping between *R* and *Python* objects performed behind the (Python-level) scene, done by the `rpy2.rinterface`, while an higher-level mapping is done between low-level objects and higher-level objects using the functions:

`conversion.ri2py()` `rpy2.rinterface` to Python. By default, this function is just an alias for the function `default_ri2py()`.

`conversion.py2ri()` Python to `rpy2.rinterface`. By default, this function is just an alias for the function `default_py2ri()`.

`conversion.py2ro()` Python to `rpy2.robjects`. By default, that one function is merely a call to `conversion.py2ri()` followed by a call to `conversion.ri2py()`.

Those functions can be re-routed to satisfy all requirements, with the easiest option being to write a custom function calling itself the default function when the custom conversion should not apply.

6.1 A simple example

As an example, let's assume that one want to return atomic values whenever an R numerical vector is of length one. This is only a matter of writing a new function *ri2py* that handles this, as shown below:

```
import rpy2.robjects as robjects

def my_ri2py(obj):
    res = robjects.default_ri2py(obj)
    if isinstance(res, robjects.RVector) and (len(res) == 1):
        res = res[0]
    return res
```

```
robjects.conversion.ri2py = my_ri2py
```

Once this is done, we can verify immediately that this is working with:

```
>>> pi = robjects.r.pi
>>>  type(pi)
<type 'float'>
>>>
```

The default behavior can be restored with:

```
>>> robjects.conversion.ri2py = default_ri2py
```

6.2 Default functions

The docstrings for `default_ri2py()`, `default_py2ri()`, and `py2ro()` are:

`rpy2.robjects.`**`default_ri2py`**`(`*o*`)`
Convert `rpy2.rinterface.Sexp` to higher-level objects, without copying the R objects.

Parameters **o** – object

Return type `rpy2.robjects.RObject (and subclasses)`

`rpy2.robjects.`**`default_py2ri`**`(`*o*`)`
Convert arbitrary Python object to `rpy2.rinterface.Sexp` to objects, creating an R object with the content of the Python object in the process (wichi means data copying).

Parameters **o** – object

Return type `rpy2.rinterface.Sexp` (and subclasses)

`rpy2.robjects.`**`default_py2ro`**`(`*o*`)`
Convert any Python object into an robject. :param o: object :rtype: `rpy2.robjects.RObject (and subclasses)`

Platforms: Unix, Windows

CHAPTER

SEVEN

LOW-LEVEL INTERFACE

7.1 Overview

The package `rinterface` is provided as a lower-level interface, for situations where either the use-cases addressed by `robjects` are not covered, or for the cases where the layer in `robjects` has an excessive cost in terms of performances.

The package can be imported with:

```
>>> import rpy2.rinterface as rinterface
```

7.1.1 Initialization

One has to initialize R before much can be done. The function `initr()` lets one initialize the embedded R.

This is done with the function `initr()`.

`rpy2.rinterface.`**`initr`**`()`
Initialize an embedded R.

```
>>> rinterface.initr()
```

Initialization should only be performed once. To avoid unpredictable results when using the embedded R, subsequent calls to `initr()` will not have any effect.

The functions `get_initoptions()` and `set_initoptions()` can be used to modify the options. Default parameters for the initialization are otherwise in the module variable *initoptions*.

Note: If calling `initr()` returns an error stating that

`R_HOME` is not defined, you should either have the **R** executable in your path (`PATH` on unix-alikes, or `Path` on Microsoft Windows) or have the environment variable `R_HOME` defined.

Ending R

Ending the R process is possible, but starting it again with `initr()` does appear to lead to an R process that is hardly usable. For that reason, the use of `endEmbeddedR()` should be considered carefully.

7.1.2 R space and Python space

When using the RPy2 package, two realms are co-existing: R and Python.

The `Sexp_Type` objects can be considered as Python enveloppes pointing to data stored and administered in the R space.

R variables are existing within an embedded R workspace, and can be accessed from Python through their python object representations.

We distinguish two kind of R objects: named objects and anonymous objects. Named objects have an associated symbol in the R workspace.

Named objects

For example, the following R code is creating two objects, named *x* and *hyp* respectively, in the *global environment*. Those two objects could be accessed from Python using their names.

```
x <- c(1,2,3)

hyp <- function(x, y) sqrt(x^2 + y^2)
```

Two environments are provided as `rpy2.rinterface.SexpEnvironment`

globalEnv

The global environment can be seen as the root (or topmost) environment, and is in fact a list, that is a sequence, of environments.

When an R library (package in R's terminology) is loaded, is it added to the existing sequence of environments. Unless specified, it is inserted in second position. The first position always remains attributed to the global environment (FIXME: there is a bit of circulariry in this definition - check how to present it a clear(er) way). The library is said to be attached to the current search path.

baseNamespaceEnv

The base package has a namespace, that can be accessed as an environment.

Note: Depending on what is in *globalEnv* and on the attached packages, base objects can be masked when starting the search from *globalEnv*. Use *baseNamespaceEnv* when you want to be sure to access a function you know to be in the base namespace.

Anonymous objects

Anonymous R objects do not have an associated symbol, yet are protected from garbage collection.

Such objects can be created when using the constructor for an *Sexp** class.

7.1.3 Interacting with the R console

Two functions can be used to set callbacks.

`rpy2.rinterface.`**`setWriteConsole`**(*function*)
Use the function to handle R console output.

Parameters **function** – function

`rpy2.rinterface.`**`setReadConsole`**(*function*)
Use the function to handle R console input.

Parameters **function** – function

Output from the console

The function `setWriteConsole()` let one specify what do with output from the R console with a callback function.

The callback function should accept one argument of type string (that is the string output to the console)

An example should make it obvious:

```
buf = []
def f(x):
    # function that append its argument to the list 'buf'
    buf.append(x)

# output from the R console will now be appended to the list 'buf'
rinterface.setWriteConsole(f)

date = rinterface.baseNamespaceEnv['date']
rprint = rinterface.baseNamespaceEnv['print']
rprint(date())

# the output is in our list (as defined in the function f above)
print(buf)
```

```
# restore default function
rinterface.setWriteConsole(rinterface.consolePrint)
```

Input to the console

User input to the console can be can be customized the very same way.

The callback function should accept one argument of type string (that is the prompt string), and return a string (what was returned by the user).

7.2 Classes

7.2.1 Sexp

The class `Sexp` is the base class for all R objects.

class `rpy2.rinterface.`**Sexp**

__sexp__
Opaque C pointer to the underlying R object

named
R does not count references for its object. This method returns the *NAMED* value (an integer). See the R-extensions manual for further details.

typeof
Internal R type for the underlying R object

```
>>> letters.typeof
16
```

do_slot (*name*)
R objects can be given attributes. In R, the function *attr* lets one access an object's attribute; it is called `do_slot()` in the C interface to R.

Parameters **name** – string

Return type instance of `Sexp`

```
>>> matrix = rinterface.globalEnv.get("matrix")
>>> letters = rinterface.globalEnv.get("letters")
>>> ncol = rinterface.SexpVector([2, ], rinterface.INTSXP)
>>> m = matrix(letters, ncol = ncol)
>>> [x for x in m.do_slot("dim")]
[13, 2]
>>>
```

do_slot_assign (*name*, *value*)
Assign value to the slot with the given name

Parameters

- **name** – string
- **value** – instance of `Sexp`

rsame(*sexp_obj*)

Tell whether the underlying R object for sexp_obj is the same or not.

Return type boolean

7.2.2 SexpVector

Overview

In R all scalars are in fact vectors. Anything like a one-value variable is a vector of length 1.

To use again the constant *pi*:

```
>>> pi = rinterface.globalEnv.get("pi")
>>> len(pi)
1
>>> pi
<rinterface.SexpVector - Python:0x2b20325d2660 / R:0x16d5248>
>>> pi[0]
3.1415926535897931
>>>
```

The letters of the (western) alphabet are:

```
>>> letters = rinterface.globalEnv.get("letters")
>>> len(letters)
26
>>> LETTERS = rinterface.globalEnv.get("LETTERS")
```

R types

R vectors all have a *type*, sometimes referred to in R as a *mode*. This information is encoded as an integer by R, but it can sometimes be better for human reader to be able to provide a string.

rpy2.rinterface.**str_typeint**(*typeint*)

Return a string corresponding to a integer-encoded R type.

Parameters **typeint** – integer (as returned by `Sexp.typeof`)

Return type string

Indexing

The indexing is working like it would on regular *Python* tuples or lists. The indexing starts at 0 (zero), which differs from *R*, where indexing start at 1 (one).

Note: The *__getitem__* operator *[* is returning a Python scalar. Casting an *SexpVector* into a list is only a matter of either iterating through it, or simply calling the constructor `list()`.

Common attributes

Names

In R, vectors can be named, that is each value in the vector can be given a name (that is be associated a string). The names are added to the other as an attribute (conveniently called *names*), and can be accessed as such:

```
>>> options = rinterface.globalEnv.get("options")()
>>> option_names = options.do_slot("names")
>>> [x for x in options_names]
```

Note: Elements in a name vector do not have to be unique. A Python counterpart is provided as `rpy2.rlike.container.TaggedList`.

Dim and dimnames

In the case of an *array*, the names across the respective dimensions of the object are accessible through the slot named *dimnames*.

Constructors

class `rpy2.rinterface.`**`SexpVector`**(*obj*, *sexptype*, *copy*)
Bases: `rinterface.Sexp`

R object that is a vector. R vectors start their indexing at one, while Python lists or arrays start indexing at zero. In the hope to avoid confusion, the indexing in Python (e.g., `__getitem__()` / `__setitem__()`) starts at zero.

Convenience classes are provided to create vectors of a given type:

class `rpy2.rinterface.`**`StrSexpVector`**(*v*)
Bases: `rinterface.SexpVector`

Vector of strings.

class `rpy2.rinterface.`**`IntSexpVector`**(*v*)
Bases: `rinterface.SexpVector`

Vector of integers.

class rpy2.rinterface.**FloatSexpVector**(*v*)
Bases: rinterface.SexpVector

Vector of floats.

class rpy2.rinterface.**BoolSexpVector**(*v*)
Bases: rinterface.SexpVector

Vector of booleans (logical in R terminology).

7.2.3 SexpEnvironment

__getitem__() / __setitem__()

The *[* operator will only look for a symbol in the environment without looking further in the path of enclosing environments.

The following will return an exception LookupError:

```
>>> rinterface.globalEnv["pi"]
```

The constant *pi* is defined in R's *base* package, and therefore cannot be found in the Global Environment.

The assignment of a value to a symbol in an environment is as simple as assigning a value to a key in a Python dictionary:

```
>>> x = rinterface.Sexp_Vector([123, ], rinterface.INTSXP)
>>> rinterface.globalEnv["x"] = x
```

Note: Not all R environment are hash tables, and this may influence performances when doing repeated lookups

---

Note: a copy of the R object is made in the R space.

__iter__()

The object is made iter-able.

For example, we take the base name space (that is the environment that contains R's base objects:

```
>>> base = rinterface.baseNameSpace
>>> basetypes = [x.typeof for x in base]
```

Warning: In the current implementation the content of the environment is evaluated only once, when the iterator is created. Adding or removing elements to the environment will not update the iterator (this is a problem, that will be solved in the near future).

get()

Whenever a search for a symbol is performed, the whole search path is considered: the environments in the list are inspected in sequence and the value for the first symbol found matching is returned.

Let's start with an example:

```
>>> rinterface.globalEnv.get("pi")[0]
3.1415926535897931
```

The constant *pi* is defined in the package *base*, that is always in the search path (and in the last position, as it is attached first). The call to `get()` will look for *pi* first in *globalEnv*, then in the next environment in the search path and repeat this until an object is found or the sequence of environments to explore is exhausted.

We know that *pi* is in the base namespace and we could have gotten here directly from there:

```
>>> ri.baseNameSpaceEnv.get("pi")[0]
3.1415926535897931
>>> ri.baseNameSpaceEnv["pi"][0]
3.1415926535897931
>>> ri.globalEnv["pi"][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LookupError: 'pi' not found
```

R can look specifically for functions, which is happening when a parsed function call is evaluated. The following example of an *R* interactive session should demonstrate it:

```
> mydate <- "hohoho"
> mydate()
Error: could not find function "mydate"
>
> date <- "hohoho"
> date()
[1] "Sat Aug  9 15:27:40 2008"
```

The base function *date* is still found, although a non-function object is present earlier on the search path.

The same behavior can be obtained from `rpy2` with the optional parameter *wantFun* (specify that `get()` should return an R function).

```
>>> ri.globalEnv["date"] = ri.StrSexpVector(["hohoho", ])
>>> ri.globalEnv.get("date")[0]
'hohoho'
>>> ri.globalEnv.get("date", wantFun=True)
<rinterface.SexpClosure - Python:0x7f142aa96198 / R:0x16e9500>
>>> date = ri.globalEnv.get("date", wantFun=True)
>>> date()[0]
'Sat Aug  9 15:48:42 2008'
```

R packages as environments

In a *Python* programmer's perspective, it would be nice to map loaded *R* packages as modules and provide access to *R* objects in packages the same way than *Python* object in modules are accessed.

This is unfortunately not possible in a robust way: the dot character . can be used for symbol names in R (like pretty much any character), and this can make an exact correspondance between *R* and *Python* names rather difficult. `rpy` uses transformation functions that translates '.' to '_' and back, but this can lead to complications since '_' can also be used for R symbols.

There is a way to provide explict access to object in R packages, since loaded packages can be considered as environments.

For example, we can reimplement in *Python* the *R* function returning the search path (*search*).

```
def rsearch():
    """ Return a list of package environments corresponding to the
    R search path. """
    spath = [ri.globalEnv, ]
    item = ri.globalEnv.enclos()
    while not item.rsame(ri.emptyEnv):
        spath.append(item)
        item = item.enclos()
    spath.append(ri.baseNameSpaceEnv)
    return spath
```

As an other example, one can implement simply a function that returns from which environment an object called by `get()` comes from.

```
def wherefrom(name, startenv=ri.globalEnv):
    """ when calling 'get', where the R object is coming from. """
    env = startenv
    obj = None
    retry = True
    while retry:
        try:
            obj = env[name]
            retry = False
        except LookupError, knf:
            env = env.enclos()
            if env.rsame(ri.emptyEnv):
                retry = False
            else:
                retry = True
    return env
```

```
>>> wherefrom('plot').do_slot('name')[0]
'package:graphics'
>>> wherefrom('help').do_slot('name')[0]
'package:utils'
```

Note: There is a gotcha: the base package does not have a name.

```
>>> wherefrom('get').do_slot('name')[0]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
LookupError: The object has no such attribute.
```

7.2.4 Functions

A function with a context

In R terminology, a closure is a function (with its enclosing environment). That enclosing environment can be thought of as a context to the function.

```
>>> sum = rinterface.globalEnv.get("sum")
>>> x = rinterface.SexpVector([1,2,3], rinterface.INTSXP)
>>> s = sum(x)
>>> s[0]
6
```

Named arguments

Named arguments to an R function can be specified just the way they would be with any other regular Python function.

```
>>> rnorm = rinterface.globalEnv.get("rnorm")
>>> rnorm(rinterface.SexpVector([1, ], rinterface.INTSXP),
          mean = rinterface.SexpVector([2, ], rinterface.INTSXP))[0]
0.32796768001636134
```

There are however frequent names for R parameters causing problems: all the names with a *dot*. using such parameters for an R function will either require to:

- use the special syntax ***kwargs* on a dictionary with the named parameters
- use the method `rcall()`.

Order for named parameters

One point where function calls in R can differ from the ones in Python is that all parameters in R are passed in the order they are in the call (no matter whether the parameter is named or not), while in Python only parameters without a name are passed in order. Using the class `ArgsDict` in the module `rpy2.rlike.container`, together with the method `rcall()`, permits calling a function the same way it would in R. For example:

```
import rpy2.rlike.container as rpc
args = rpc.ArgsDict()
```

```
args['x'] = rinterface.IntSexpVector([1,2,3], rinterface.INTSXP)
args[None] = rinterface.IntSexpVector([4,5], rinterface.INTSXP)
args['y'] = rinterface.IntSexpVector([6, ], rinterface.INTSXP)
rlist = rinterface.baseNameSpaceEnv['list']
rl = rlist.rcall(args.items())

>>> [x for x in rl.do_slot("names")]
['x', '', 'y']
```

closureEnv

In the example below, we inspect the environment for the function *plot*, that is the namespace for the package *graphics*.

```
>>> plot = rinterface.globalEnv.get("plot")
>>> ls = rinterface.globalEnv.get("ls")
>>> envplot_list = ls(plot.closureEnv())
>>> [x for x in envplot_ls]
>>>
```

7.2.5 SexpS4

Object-Oriented programming in R exists in several flavours, and one of those is called *S4*. It has its own type at R's C-API level, and because of that specificity we defined a class. Beside that, the class does not provide much specific features (see the pydoc for the class below).

An instance's attributes can be accessed through the parent class `Sexp` method `do_slot()`.

class `rpy2.rinterface.`**`SexpS4`**(*obj*)

Bases: `rinterface.Sexp`

R object that is an 'S4 object'.Attributes can be accessed using the method 'do_slot'.

7.3 Misc. variables

R_HOME R HOME

R_LEN_T_MAX largest usable integer for indexing R vectors

TRUE/FALSE R's TRUE and FALSE

7.3.1 Missing values

NA_INTEGER Missing value for integers

NA_LOGICAL Missing value for booleans

NA_REAL Missing value for numerical values (float / double)

7.3.2 R types

Vector types

CPLXSXP Complex

INTSXP Integer.

LGLSXP Boolean (logical in the R terminology)

REALSXP Numerical value (float / double)

STRSXP String

VECSXP List

LANGSXP Language object.

EXPRSXP Unevaluated expression.

Other types

CLOSXP Function with an enclosure. Represented by `rpy2.rinterface.SexpClosure`.

ENVSXP Environment. Represented by `rpy2.rinterface.SexpEnvironment`.

S4SXP Instance of class S4. Represented by `rpy2.rinterface.SexpS4`.

Types one should not meet

PROMSXP Promise.

CHAPTER

EIGHT

RPY_CLASSIC

Platforms: Unix, Windows

This module provides an API similar to the one in RPy-1.x (*rpy*).

To match examples and documentation for *rpy*, we load the module as:

```
>>> import rpy2.rpy_classic as rpy
```

8.1 Conversion

Although the proposed high-level interface in `rpy2.robjects` does not need explicit conversion settings, the conversion system existing in *rpy* is provided, and the default mode can be set with `set_default_mode()`:

```
>>> rpy.set_default_mode(rpy.NO_CONVERSION)
>>> rpy.set_default_mode(rpy.BASIC_CONVERSION)
```

8.2 R instance

The `r` instance of class `R` behaves like before:

```
>>> rpy.r.help
```

'dots' in the R name are translated to underscores:

```
>>> rpy.r.wilcox_test
```

```
>>> rpy.r.wilcox_test([1,2,3], [4,5,6])
```

```
>>> x = rpy.r.seq(1, 3, by=0.5)
>>> rpy.r.plot(x)
```

An example:

```
degrees = 4
grid = rpy.r.seq(0, 10, length=100)
values = [rpy.r.dchisq(x, degrees) for x in grid]
rpy.r.par(ann=0)
rpy.r.plot(grid, values, type='l')

rpy.r.library('splines')

type(rpy.r.seq)
```

8.3 Functions

As in RPy-1.x, all R objects are callable:

```
>>> callable(rpy.r.seq)
True
>>> callable(rpy.r.pi)
True
>>>
```

If an object is not a R function, a `RuntimeError` is thrown by R whenever called:

```
>>> rpy.r.pi()
```

The function are called like regular Python functions:

```
>>> rpy.r.seq(1, 3)
>>> rpy.r.seq(1, 3, by=0.5)
>>> rpy.r['options'](show_coef_Pvalues=0)
>>>

>>> m = rpy.r.matrix(r.rnorm(100), 20, 5)
>>> pca = rpy.r.princomp(m)
>>> rpy.r.plot(pca, main = "PCA")
>>>
```

8.4 Partial use of `rpy_classic`

The use of rpy_classic does not need to be exclusive of the other interface(s) proposed in rpy2.

Chaining code designed for either of the interfaces is rather easy and, among other possible use-cases, should make the inclusion of legacy rpy code into newly written rpy2 code a simple take.

The link between `rpy_classic` and the rest of `rpy2` is the property `RObj.sexp`, that give the representation of the underlying R object in the low-level `rpy2.rinterface` definition. This representation can then be used in function calls with

`rpy2.rinterface` and `rpy2.robjects`. With `rpy2.robjects`, a conversion using `rpy2.robjects.default_ri2py()` can be considered.

Note: Obviously, that property *sexp* is not part of the original *Robj* in rpy.

An example:

```
import rpy2.robjects as ro
import rpy2.rpy_classic as rpy
rpy.set_default_mode(rpy.NO_CONVERSION)


def legacy_paste(v):
    # legacy rpy code
    res = rpy.r.paste(v, collapse = '-')
    return res


rletters = ro.r['letters']

# the legaxy code is called using an rpy2.robjects object
alphabet_rpy = legacy_paste(rletters)

# convert the resulting rpy2.rpy_classic object to
# an rpy2.robjects object
alphabet = ro.default_ri2py(alphabet_rpy.sexp)
```

CHAPTER

NINE

RLIKE

Platforms: Unix, Windows

9.1 Overview

The package proposes R features for a pure Python context, that is without an embedded R running.

9.2 Containers

The module contains data collection-type data structures. `ArgsDict` and `TaggedList` are structures with which containeed items/elements can be tagged.

The module can be imported as follows:

```
>>> import rpy2.rlike.container as rlc
```

9.2.1 ArgsDict

The `ArgsDict` proposes an implementation of what is sometimes referred to in Python as an ordered dictionnary, with a particularity: a key `None` means that, although an item has a rank and can be retrieved from that rank, it has no "name".

In the hope of simplifying its usage, the API for an ordered dictionnary in **PEP 372** was implemented. An example of usage is:

```
>>> x = (('a', 123), ('b', 456), ('c', 789))
>>> nl = rlc.ArgsDict(x)

>>> nl['a']
123
>>> nl.index('a')
0
```

Not all elements have to be named, and specifying a key value equal to *None* indicates a value for which no name is associated.

```
>>> nl[None] = 'no name'
```

9.2.2 TaggedList

A `TaggedList` is a Python `list` in which each item has an associated *tag*. This is similar to *named* vectors in R.

```
>>> tl = rlc.TaggedList([1,2,3])
>>> tl
[1, 2, 3]
>>> tl.tags()
(None, None, None)
>>> tl.settag(0, 'a')
>>> tl.tags()
('a', None, None)

>>> tl = rlc.TaggedList([1,2,3], tags=('a', 'b', 'c'))
>>> tl
[1, 2, 3]
>>> tl.tags()
('a', 'b', 'c')
>>> tl.settag(2, 'a')
>>> tl.tags()
('a', 'b', 'a')
>>> it = tl.iterontag('a')
>>> [x for x in it]
[1, 3]

>>> [(t, sum([i for i in tl.iterontag(t)])) for t in set(tl.itertags())]
[('a', 4), ('b', 2)]
```

The Python docstring for the class is:

class `rpy2.rlike.container.`**`TaggedList`**(*l*, *tags=None*)

A list for which each item has a 'tag'.

Parameters

- **l** – list
- **tag** – optional sequence of tags

`append`(*obj*, *tag=None*)

Append an object to the list :param obj: object :param tag: object

`extend`(*iterable*)

Extend the list with an iterable object.

Parameters **iterable** – iterable object

insert(*index*, *obj*, *tag=None*)
Insert an object in the list

Parameters

- **index** – integer
- **obj** – object
- **tag** – object

items()
Return a tuple of all pairs (tag, item).

Return type tuple of 2-element tuples (tag, item)

iterontag(*tag*)
iterate on items marked with one given tag.

Parameters **tag** – object

itertags()
iterate on tags.

Return type iterator

pop(*index=None*)
Pop the item at a given index out of the list

Parameters **index** – integer

remove(*value*)
Remove a given value from the list.

Parameters **value** – object

reverse()
Reverse the order of the elements in the list.

settag(*i*, *t*)
Set tag 't' for item 'i'.

Parameters

- **i** – integer (index)
- **t** – object (tag)

sort(*reverse=False*)
Sort in place

tags()
Return a tuple of all tags

Return type tuple

9.3 Tools for working with sequences

Tools for working with objects implementing the the sequence protocol can be found here.

rpy2.rlike.functional.**tapply**(*seq*, *tag*, *fun*)
Apply the function *fun* to the items in *seq*, grouped by the tags defined in *tag*.

Parameters

- **seq** – sequence
- **tag** – any sequence of tags
- **fun** – function

Return type list

```
>>> import rpy2.rlike.functional as rlf
>>> rlf.tapply((1,2,3), ('a', 'b', 'a'), sum)
[('a', 4), ('b', 2)]
```

`TaggedList` objects can be used with their tags (although more flexibility can be achieved using their method `iterontags()`):

```
>>> import rpy2.rlike.container as rlc
>>> tl = rlc.TaggedList([1, 2, 3], tags = ('a', 'b', 'a'))
>>> rlf.tapply(tl, tl.tags(), sum)
[('a', 4), ('b', 2)]
```

9.4 Indexing

Much of the R-style indexing can be achieved with Python's list comprehension:

```
>>> l = ('a', 'b', 'c')
>>> l_i = (0, 2)
>>> [l[i] for i in l_i]
['a', 'c']
```

In *R*, negative indexes mean that values should be excluded. Again, list comprehension can be used (although this is not the most efficient way):

```
>>> l = ('a', 'b', 'c')
>>> l_i = (-1, -2)
>>> [x for i, x in enumerate(l) if -i not in l_i]
['a']
```

rpy2.rlike.indexing.**order**(*seq*, *cmp = default_cmp*, *reverse = False*)
Give the order in which to take the items in the sequence *seq* and have them sorted. The optional function cmp should return +1, -1, or 0.

Parameters

- **seq** – sequence

- **cmp** – function
- **reverse** – boolean

Return type list of integers

```
>>> import rpy2.rlike.indexing as rli
>>> x = ('a', 'c', 'b')
>>> o = rli.order(x)
>>> o
[0, 2, 1]
>>> [x[i] for i in o]
['a', 'b', 'c']
```

CHAPTER

TEN

CHANGES IN RPY2

10.1 Release 2.0.1

10.1.1 New features

`rpy2.robjects`:

- Property *names* for the `RVector` methods `getnames()` and `setnames()` (this was likely forgotten for Release 2.0.0).
- Property *rclass* for `RObjectMixin`

10.1.2 Changes

`rpy2.robjects`:

- `rclass()` becomes `getrclass()`

10.1.3 Bugs fixed

- Having the environment variable R_HOME specified resulted in an error when importing `rpy2.rinterface` # root of the problem spotted by Peter
- Setup.py has no longer a (possibly outdated) static hardcoded version number for rpy2
- Testing no longer stops with an error in the absence of the third-party module `numpy`
- `rpy2.rlike.container.TaggedList.pop()` is now returning the element matching the given index

10.2 Release 2.0.0

10.2.1 New features

- New module `rpy2.robjects.conversion`.
- New module `rpy2.robjects.numpy2ri` to convert `numpy` objects into `rpy2` objects. # adapted from a patch contributed by Nathaniel Smith

10.2.2 Changes

- `RObject.__repr__()` moved to `RObject.r_repr()`

10.2.3 Bugs fixed

- Informative message returned as RuntimeError when failing to find R's HOME
- Use the registry to find the R's HOME on win32 # snatched from Peter's earlier contribution to rpy-1.x

10.3 Release 2.0.0rc1

`rpy2.rpy_classic`:

- `rpy_classic.RObj.getSexp()` moved to a property `rpy_classic.Robj.sexp`.

`rpy2.robjects`:

- `RObject.__repr__()` moved to `RObject.r_repr()`
- `ri2py()`, `ro2py()`, and `py2ri()` moved to the new module `conversion`. Adding the prefix *conversion.* to calls to those functions will be enough to update existing code

10.3.1 Bugs fixed

- Informative message returned as RuntimeError when failing to find R's HOME
- Use the registry to find the R's HOME on win32 # snatched from Peter's earlier contribution to rpy-1.x

10.4 Release 2.0.0rc1

10.4.1 New features

- added `__version__` to rpy2/__init__.py

`rpy2.robjects`:

- added classes `StrVector`, `IntVector`, `FloatVector`, `BoolVector`

`rpy2.rinterface`:

- added missing class `BoolSexpVector`.

10.4.2 Changes

`rpy2.robjects`:

- does not alias `rinterface.StrSexpVector`, `rinterface.IntSexpVector`, `rinterface.FloatSexpVector` anymore
- Constructor for `rpy2.robjects.RDataFrame` checks that R lists are data.frames (not all lists are data.frame)
- Formerly new attribute `_dotter` for `R` is now gone. The documentaion now points to `rpy2.rpy_classic` for this sort of things.

10.4.3 Bugs fixed

- conditional typedef in rinterface.c to compile under win32 # reported and initial proposed fix from Paul Harrington
- __pow__ was missing from the delegator object for robjects.RVector (while the documentation was claiming it was there) # bug report by Robert Nuske
- Earlier change from Sexp.typeof() to getter Sexp.typeof was not reflected in `rpy2.rpy_classic` # bug report by Robert Denham

10.5 Release 2.0.0b1

10.5.1 New features

`rpy2.robjects`:

- added `setenvironment()` for `RFormula`, and defined *environment* as a property
- defined *names* as a property for `RVector`

`rpy2.rinterface`:

- added functions `get_initoptions()` and `set_initoptions()`.
- new attribute `_dotter` for `R` singleton. Setting it to True will translate '_' into '.' if the attribute is not found

10.5.2 Changes

`rpy2.robjects`:

- constructor for RDataFrame now now accepts either `rlike.container.TaggedList` or `rinterface.SexpVector`

`rpy2.rinterface`:

- `sexpTypeEmbeddedR()` is now called `str_typeint()`.
- `initOptions` is now called `initoptions`. Changes of options can only be done through `set_initoptions()`.

10.5.3 Bugs fixed

- crash of `Sexp.enclos()` when R not yet initialized (bug report #2078176)
- potential crash of `Sexp.frame()` when R not yet initialized
- proper reference counting when handling, and deleting, `Sexp.__sexp__` generated CObjects
- setup.py: get properly the include directories (no matter where they are) #bug report and fix adapted from Robert Nuske
- setup.py: link to external lapack or blas library when relevant
- added a MANIFEST.in ensuring that headers get included in the source distribution #missing headers reported by Nicholas Lewin-Koh
- `rinterface.str_typeint()` was causing segfault when called with 99
- fixed subsetting for LANGSXP objects

10.6 Release 2.0.0a3

10.6.1 New features

`rpy2.rinterface`:

- `setReadConsole()`: specify Python callback for console input
- *R* string vectors can now be built from Python unicode objects
- getter `__sexp__` to return an opaque C pointer to the underlying R object

- method `rsame()` to test if the underlying R objects for two `Sexp` are the same.
- added *emptyEnv* (R's C-level *R_EmptyEnv*)
- added method `Sexp.do_slot_assign()`

`rpy2.robjects`:

- R string vectors can now be built from Python unicode objects

`rpy2.rlike`:

- module `functional` with the functions `tapply()`, `listify()`, `iterify()`.
- module `indexing` with the function `order()`
- method `TaggedList.sort()` now implemented

10.6.2 Changes

`rpy2.rinterface`:

- `initEmbeddedR()` is only initializing if R is not started (no effect otherwise, and no exception thrown anymore)
- the method `Sexp.typeof()` was replaced by a Python *getter* `typeof`.
- the method `Sexp.named()` was replaced by a Python *getter* `named`.
- R objects of type LANGSXP are now one kind of vector (... but this may change again)
- R objects of type EXPRSXP are now handled as vectors (... but this may change again)
- `initEmbeddedR()` renamed to `initr()`
- `endEmbeddedR()` renamed to `endr()`

`rpy2.robjects`:

- `R` remains a singleton, but does not throw an exception when multiple instances are requested

10.6.3 Bugs fixed

- unable to compile on Python2.4 (definition of aliases to Python2.5-specific were not where they should be).
- overflow issues on Python 2.4/64 bits when indexing R vector with very large integers.
- handling of negative indexes for `SexpVector`'s `__getitem__()` and `__setitem__()` was missing
- trying to create an instance of `SexpVector` before initializing R raises a RuntimeException (used to segfault)
- experimental method `enclos()` was not properly exported

- setup.py was exiting prematurely when R was compiled against an existing BLAS library
- complex vectors should now be handled properly by `rpy2.rinterface.robjects`.
- methods `rownames()` and `colnames()` for `RDataFrame` were incorrect.

10.7 Release 2.0.0a2

10.7.1 New features

`rpy2.rlike`:

- package for R-like features in Python
- module `rpy2.rlike.container`
- class `ArgsDict` in `rpy2.rlike.container`
- class `TaggedList` in `rpy2.rlike.container`

`rpy2.rinterface`:

- method `named()`, corresponding to R's C-level NAMED
- experimental methods `frame()` and `enclos()` for SexpEnvironment corresponding to R's C-level FRAME and ENCLOS
- method `rcall()` for `ClosureSexp`
- new experimental class `SexpLang` for R language objects.

10.7.2 Bugs fixed

- R stack checking is disabled (no longer crashes when multithreading)
- fixed missing R_PreserveObject for vectors (causing R part of the object to sometimes vanish during garbage collection)
- prevents calling an R function when R has been ended (raise `RuntimeException`).

10.8 Release 2.0.0a1

10.8.1 New features

`rpy2.robjects`:

- method `getnames()` for `RVector`
- experimental methods `__setitem__()` and `setnames()` for `RVector`

- method 'getnames' for `RArray`
- new class `RFormula`
- new helper class `RVectorDelegator` (see below)
- indexing RVector the "R way" with subset is now possible through a delegating attribute (e.g., myvec.r[True] rather than myvec.subset(True)). #suggested by Michael Sorich
- new class `RDataFrame`. The constructor `__init__()` is still experimental (need for an ordered dictionnary, that will be in before the beta
- filled documentation about mapping between objects

10.8.2 Changes

- many fixes and additions to the documentation
- improved GTK console in the demos
- changed the major version number to 2 in order to avoid confusion with rpy 1.x # Suggested by Peter and Gregory Warnes
- moved test.py to demos/example01.py

`rpy2.robjects`:

- changed method name *getNames* to *getnames* where available (all lower-case names for methods seems to be the accepted norm in Python).

10.8.3 Bugs fixed

`rpy2.robjects`:

- fixed string representation of R object on Microsoft Windows (using fifo, not available on win32)
- `__getattr__()` for `RS4` is now using `ri2py()`

`rpy2.rinterface`:

- fixed context of evaluation for R functions (now R_GlobalEnv)

10.9 Release 1.0a0

- first public release

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T