RPy Reference Manual

(version 0.3.3)

Walter Moreira and Gregory R. Warnes

Table of Contents

1 Overview

RPy is a very simple, yet robust, Python interface to the R Programming Language. It can manage all kinds of R objects and can execute arbitrary R functions (including the graphic functions). All the errors from the R language are converted to Python exceptions. Any module that later were installed on the R system, can easily be used from within Python, without introducing any changes. Starting from version 0.3, RPy works on any POSIX system and Windows.

The RPy code is initially inspired in RSPython, which is part of the Omegahat project. The main goals of RPy are:

1. It should provide a very robust Python interface to R (segfaults shouldn't happen [tm]).
2. It should be as transparent and easy to use as possible.
3. It should be usable for real scientific and statistical computations.

Currently, RPy has a good degree of stability (in spite of the low version number). It provides a very customizable conversion system between Python and R types (see Chapter 4 [Conversion system], page 10), user defined I/O functions and a complete handling of the R errors via Python exceptions.

Since version 0.2, RPy uses the Numeric extension module for the conversion of arrays. However, if it is not available, RPy converts R arrays to Python lists.

Many things are still to be done (see Section 1.1 [To do], page 1), but priority one is the porting to Windows.

1.1 To do

- Possibility to pass Python functions to R functions (and, perhaps, to make RPy a bidirectional Python-R interface).
- More builtin classes for conversion of R classes.
- Add real examples and applications.

1.2 Contact info and contributing

Please, submit any bug, comment or suggestion to the address below. When submitting bugs, it would be preferable to fill the Sourceforge form, because it can be read by many people.

If you have used RPy in a real world application or have some interesting examples of use, please, drop me a line. They can be linked from the RPy website or included in the distribution, in order to make easier to grasp the RPy capabilities.

Original author:
Walter Moreira

Current maintainer
Gregory Warnes

Web: http://rpy.sourceforge.net

Email: gregory.r.warnes@pfizer.com

2 Starting up

After installation, you are able to execute, almost verbatim, most of the code from the R manuals. This section should be enough to start playing. See Chapter 3 [Interface description], page 4, for a detailed description; Chapter 4 [Conversion system], page 10, for details on the conversion of types and names; and Chapter 5 [Input/Output functions], page 18 for the customization of I/O routines.

2.1 Installation

See the installation procedure in the file 'README', which is provided with the RPy distribution. (Should the 'README' instructions appear in this place?)

2.2 Invocation

Once installed, the module can be imported with:

```
>>> from rpy import *
```

If an error occurs, refer to the section TROUBLESHOOTING on the 'README' file.

The module `rpy` imports a Python object named `r`, from which all the R functions and objects can be retrieved, see Section 3.1 [R objects look up], page 4. This module also implements a new Python type: `Robj`, which represents an arbitrary R object, see Section 3.2 [Robj type], page 5. For example:

```
>>> r.wilcox_test
<Robj object at 0x8a9e120>
```

is the R function `wilcox.test` which computes the Wilcoxon statistical test. An object of type `Robj` is always callable as long as the corresponding R object is:

```
>>> r.wilcox_test([1,2,3], [4,5,6])
{'p.value': 0.10000000000000001, 'statistic': {'W': 0.0},
 'null.value': {'mu': 0.0}, 'data.name': 'c(1, 2, 3) and c(4, 5, 6)',
 'alternative': 'two.sided', 'parameter': None, 'method':
 'Wilcoxon rank sum test'}
```

The arguments are translated automatically to R objects and the return value is translated back to Python, when this is possible (see Chapter 4 [Conversion system], page 10). This autoconversion can be customized at several levels or disabled at all.

Objects of type `Robj` also support keyword arguments, in the same way as normal Python functions:

```
>>> r.seq(1, 3, by=0.5)
[1.0, 1.5, 2.0, 2.5, 3]
```

The module `rpy` defines a new exception type derived from the base class `Exception`, called `RException`, see Section 3.3 [R exceptions], page 8. When any kind of error in the R interpreter occurs, an exception of this type is raised:

```
>>> r.plot()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
rpy.RException: Error in function (x, ...)  : Argument "x" is missing,
with no default
```

2.3 Small example

Here we present a very small example. Some other bigger examples can be found in the '`examples`' directory of the RPy distribution. See also Section 4.3 [Useful examples], page 14. If you want to contribute some code that you find interesting as an example of use, please, write to me (see Section 1.2 [Contact info and contributing], page 1).

This lines of code draw the density of a chi-squared distribution:

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

3 Interface description

In general, R functions can be accessed transparently via attributes of the `r` object. Parameters and return values are converted to and from R automatically. However, some R functions, such us `$` or `[[`, need special syntax, and sometimes objects of type `Robj` need to be manipulated.

3.1 R objects look up

There are two ways of retrieving a R object via the Python `r` object:

- as attributes
- as keywords

The two ways are completely equivalent, the only difference is that some names cannot be used as attributes, so in some cases you must use the second way. The first time that a R object is required, it is looked up in the R global environment and it is cached in a dictionary in the `r` object. After then, retrieving the same object is only a look up in a Python dictionary.

The first way of retrieving a R object is as attributes of the `r` object. For example:

```
r.seq
r.as_data_frame
r.print_
```

refer to the R functions `seq`, `as.data.frame` and `print` respectively. Note that some kind of name conversion is required in order to make the attributes valid Python identifiers. But the rules of name conversions are pretty simple; namely, the following conversions are applied to Python identifiers

Python name	R name
Underscore (‘_’)	dot (‘.’)
Double underscore (‘__’)	arrow (`<-`)
Final underscore (preceded by a letter)	is removed

The final rule is used to allow the retrieving of R objects whose names are Python keywords. Some additional examples:

Python name	R name
`t_test`	`t.test`
`attr__`	`attr<-`
`parent_env__`	`parent.env<-`
`class_`	`class`

The second way of retrieving a R object is as keywords of the `r` object. In this form, no name conversion is required. The string used as keyword must be, exactly, the R name of the object. For example:

```
r['as.data.frame']
r['print']
r['$']
```

refer to the corresponding R functions. Note that with this syntax you can retrieve functions such as `$`, `$<-` or `[[`, which are impossible to express with the attribute syntax. However, the attributes are more appealing to the eyes.

Due to the dynamic nature of the look up, when installing additional modules in the R system, it is **not** necessary to make changes in the interface. All you have to do is to load the module in the same way as in R:

```
>>> r.library('splines')
['splines', 'ctest', 'base']
```

3.2 Robj type

The new type `Robj` represents an arbitrary R object. All the R functions retrieved via the `r` object (see Section 3.1 [R objects look up], page 4) are of type `Robj`:

```
>>> type(r.seq)
<type 'Robj'>
```

If you use the standard conversion of types, you'll probably never find another object of type `Robj`. However, there are reasons for, sometimes, manipulating these objects (see Chapter 4 [Conversion system], page 10).

3.2.1 Calling R objects

An object of type `Robj` is always callable. When it represents a R function, that function is invoked; if it is not a R function, an exception is raised (see Section 3.3 [R exceptions], page 8):

```
>>> callable(r.seq)
1
>>> callable(r.pi)
1
>>> r.pi()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
rpy.RException: Error: attempt to apply non-function
```

The arguments passed to the `Robj` object can be any Python object, including another `Robj` object. When an object of a standard Python type is passed, it is converted to a R type according to the rules described in Section 4.2 [Python to R], page 14. When a `Robj` object is passed, the R function receives the corresponding R object unchanged. Usually, you don't need to think on these things, because the conversion works as one expects.

A `Robj` also supports keyword arguments, if the corresponding R function does it. The names of the keyword arguments are also under the name conversion rules described in Section 3.1 [R objects look up], page 4.

For example:

```
>>> r.seq(1, 3)
[1, 2, 3]
>>> r.seq(1, 3, by=0.5)
[1.0, 1.5, 2.0, 2.5, 3.0]
>>> r['options'](show_coef_Pvalues=0)
```

```
{'show.coef.Pvalues': 1}
>>> r.print_(r.seq)
function (...)
UseMethod("seq")
<Robj object at 0x8acb010>
```

Return values of the call to a `Robj` object are also converted from R to Python. When no conversion is possible, an `Robj` object is returned, which holds a reference to the R object. This behavior is completely customizable (see Section 4.1 [R to Python], page 10).

Note that python uses a python dictionary to store named arguments. As a consequence the order named arguments will be lost in calls to R functions, and hence may not produce what an R programmer expects:

```
>>> set_default_mode(NO_CONVERSION)
>>> r.print_(r.c(0,a=1,b=2,c=3))
  a c b
0 1 3 2
<Robj object at 0xbc89d0>
>>> set_default_mode(BASIC_CONVERSION)
```

To work around this problem, `RObj` provides the `lcall` method which expects a list containing 2 element (name, value) tuples instead of a list of named and unnamed arguments. Unnamed arguments are indicated by using None or '' as for the name element of the tuple. While this form is unwieldy, it is functional and is occasionally necessary:

```
>>> set_default_mode(NO_CONVERSION)
>>> r.print_(r.c.lcall( (('',0),('a',1),('b',2),('c',3)) ) )
  a b c
0 1 2 3
<Robj object at 0xbc89b8>
>>> set_default_mode(BASIC_CONVERSION)
```

[See see Chapter 4 [Conversion system], page 10 for the meaning of `set_default_mode`. It is used here to prevent python from translating the output of `c` into a python dictionary (which loses element order) before `print_` displays it.]

3.2.2 Methods of Robj type

An object of type `Robj` has three methods:

`as_py([mode])`
: This method forces the conversion of a `Robj` object to a classical Python object, whenever possible. If it is not possible, the same object is returned. The optional parameter is the mode from which to apply the conversion, see Section 4.1.1 [Modes], page 10. The default value for this parameter is the *global mode* (see Section 4.1 [R to Python], page 10).

`autoconvert([mode])`
`local_mode([mode])`
: This method sets the local conversion mode for each object, which is used when the default mode is set to '`NO_DEFAULT`', (see Section 4.2 [Python to R], page 14). When no argument is passed to this method, it displays the

current local conversion mode of the object. (The two names are synonyms for compatibility with version 0.1.)

`lcall([argument list])`
: This method calls the R object (if callable) using the parameters provided as a single list containing a 2 element (name, value) tuple for each arguments. Unnamed arguments may have None or '' as the name element.

For example:

```
>>> r.seq.local_mode(NO_CONVERSION)
>>> a = r.seq(3, 5)
>>> a
<Robj object at 0x814c2e8>
>>> a.as_py()
[3, 4, 5]
>>> set_default_mode(NO_CONVERSION)
>>> r.print_(r.c.lcall( (('',0),('a',1),('b',2),('c',3)) ) )
  a c b
0 1 3 2
<Robj object at 0xbc89d0>
>>> set_default_mode(BASIC_CONVERSION)
```

3.2.3 Sequence protocol

An object of type `Robj` supports (partially at the moment, slices are not supported yet) the sequence protocol. You can retrieve or set the n-th item of a `Robj` object, and you can take its length with the usual Python function `len`.

Every R object is a vector, so this protocol can be applied to any `Robj` object; although it can raise an exception when an index is out of bounds. Note that in this case, the exception is `IndexError` instead of `RException`; this is done to allow a `for` loop to iterate over a `Robj` object.

The return values of the sequence functions are converted to Python according to the default mode. If the default mode is set to '`NO_DEFAULT`', the sequence functions use the '`PROC_MODE`' conversion mode.

```
>>> r.seq.local_mode(NO_CONVERSION)
>>> a = r.seq(3, 5)
>>> a[0]
3
>>> a[2]
5
>>> a[-1]
5
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: R object index out of range
>>>
>>> for i in a:
```

```
...     print i
...
3
4
5
```

The behavior of the setting of items is different from that of Python, mainly, when you try to set an item out of bounds. Remember, in these cases, that the setting is done via R functions, which have different semantic from the Python sequence functions.

```
(following the previous example)
>>> b = r.seq(1, 3)
>>> dummy = r.print_(b)
[1] 1 2 3
>>> b[0] = -1
>>> dummy = r.print_(b)
[1] -1  2  3
>>> b[6] = 4
>>> dummy = r.print_(b)
[1] -1  2  3 NA NA NA  4
```

Also, be careful with the different index convention between Python and R: in Python, indices start at 0; in R, they start at 1.

```
(following the previous example)
>>> a[0]
3
>>> r['['](a, 1)
3
```

Function `len` can also be applied to any `Robj` object:

```
(following the previous example)
>>> len(a)
3
>>> len(r.seq)
1
>>> len(r.pi)
1
```

3.3 R exceptions

RPy implements a new exception type, called `RException`, which is derived from the base class of all exceptions `Exception`. This exception is raised when an error in the R interpreter occurred. The error message included in the exception is the message given by the R interpreter. For example:

```
>>> RException
<class rpy.RException at 0x8a72b44>
>>>
>>> r.t_test(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
rpy.RException: Error in t.test.default(1) : not enough x observations
```

Note that not all operations with `Robj` objects raise this exception. When using the sequence protocol on `Robj` objects, the exceptions raised are `IndexError`, in order to properly use the `for` loop (see Section 3.2.3 [Sequence protocol], page 7). Other functions, such as mode functions, raise `ValueError` (see Section 4.1.1 [Modes], page 10).

3.4 R boolean objects

The RPy module provides, as a convenience, the `TRUE` and `FALSE` R objects, as attributes to the `r` Python object (i.e.: `r.TRUE` and `r.FALSE`). For example:

```
>>> r.TRUE
<Robj object at 0x8b3a498>
>>> r.typeof(r.TRUE)
'logical'
```

Note that the `T` and `F` names from R are variables bounded to the logical objects; however, they can be rebound. So, `r.T` and `r.F` may not be the objects you expect. Use `r.TRUE` and `r.FALSE` instead.

4 Conversion system

Usually, automatic conversion from and to R objects works as expected. However, the system is very customizable; you can define your own conversion rules. Some useful examples are shown.

4.1 R to Python

RPy has four different *conversion modes*. A mode can be *global* or *local* to every `Robj` object. When a global mode is active, the output from every `Robj` is converted according to that mode (see Section 4.1.1 [Modes], page 10). (We'll use *default* as a synonym of global.)

The local mode is an attribute of the `Robj` objects, which can be retrieved with the `local_mode` method (see Section 3.2.2 [Methods of Robj type], page 6). When the global mode is not active, the local mode is looked up to convert the output of a given `Robj` object.

4.1.1 Modes

There are five different conversion modes, identified by the following constants (provided by the `rpy` module) and another constant to indicate the absence of a global mode:

- `PROC_CONVERSION`
- `CLASS_CONVERSION`
- `BASIC_CONVERSION`
- `VECTOR_CONVERSION`
- `NO_CONVERSION`
- `NO_DEFAULT`

The `rpy` module provides three functions for manipulating the conversion modes:

`get_default_mode()`
: Get the default conversion mode. It returns some of the previous constants (actually, an integer from the set {-1,0,1,2}, but you should use the literal constant rather than the numeric value).

`set_default_mode(m)`
: Set the default conversion mode to `m`.

`with_mode(m, fun)`
: Wrap the function `fun` in the conversion mode `m`. It returns a new function which accepts the same parameters as `fun` but, when called, it is evaluated in the conversion mode `m`. For example:

```
>>> set_default_mode(BASIC_CONVERSION)
>>> r.seq(1,3)
[1, 2, 3]
>>> with_mode(NO_CONVERSION, r.seq)(1,3)
<Robj object at 0x8acb2a0>
```

The result of a call to a `Robj` object is converted according to the following rules:

1. If the default mode has a value in the set {`PROC_CONVERSION`, `CLASS_CONVERSION`, `BASIC_CONVERSION`, `NO_CONVERSION`}, that mode is used.

2. If the default mode has the value `NO_DEFAULT`, then the object's local mode is used.
3. When an object cannot be converted in some mode (global or local), the object fall through to the next mode. The `NO_CONVERSION` mode always succeed returning a "pure" `Robj` object.

At startup the default mode is set to `NO_DEFAULT`, which means that each object has its own conversion mode, and every `Robj` object is retrieved with a local mode set to `PROC_CONVERSION`.

4.1.2 Proc conversion

This mode converts a `Robj` object according to a Python dictionary, named `proc_table`, whose keys and values are functions of one parameter. The keys are applied sequentially to the `Robj` object:

- if no function returns a true value, then the conversion mode fails;
- if some function returns a true value, then the corresponding value of the dictionary is applied to the `Robj` object, and the result is returned as the converted object.

For example:

```
>>> set_default_mode(PROC_CONVERSION)
>>> def check_str(o):
...     return r.is_character(o)
...
>>> def f(o):
...     o_py = o.as_py(BASIC_CONVERSION)
...     if o_py == 'foo':
...          return 'cannot return "foo"'
...     return o_py
...
>>> proc_table[check_str] = f
>>> r.c('bar')
'bar'
>>> r.c('foo')
'cannot return "foo"'
>>> r.c(['bar','foo'])
['bar', 'foo']
```

Note that the conversion is not applied recursively. This mode is applied only before returning the final result to Python.

This conversion mode can be used for many purposes (see Section 4.3 [Useful examples], page 14); but, mainly, it is used to test whether a R object has some attribute, and to act in consequence.

Note that this conversion mode is not efficient if the `proc_table` dictionary has many keys, because, usually, all of them must be checked. On the other hand, with only one key which always returns true, it can be used to completely intercept the conversion system (see Section 4.3.1 [Enhanced Robj], page 15).

4.1.3 Class conversion

This mode converts a `Robj` object according to a Python dictionary, named `class_table`, whose keys are strings or tuples of strings and its values are functions of one parameter. If a `Robj` object matches this table (see below), the corresponding value of the dictionary is applied to the `Robj` object and the result is returned as the converted object. If the `Robj` object has no class attribute or the class attribute does not match in `class_table`, then this conversion mode fails.

In order to a `Robj` object match the `class_table` dictionary, one of the following cases must be satisfied:

1. the `class` R attribute of the object is a string and it is found in the `class_table` dictionary; or
2. the `class` R attribute of the object is a vector of strings and it is found in the `class_table` dictionary; or
3. the `class` R attribute of the object is a tuple of strings and one of the tuple's elements is found in the `class_table` dictionary.

For example:

```
>>> set_default_mode(CLASS_CONVERSION)
>>> def f(o):
...     return 5
...
>>> class_table['data.frame'] = f
>>> r.as_data_frame([1,2,3])
5
```

This table is used, mainly, to translate R objects of some class, to Python objects of a class which mimics the R original class behavior. See Section 4.3.2 [DataFrame class], page 16.

Note that this mode is far more efficient than the `PROC_CONVERSION` mode. It only needs a look up in a Python dictionary.

4.1.4 Basic conversion

This mode tries to convert a `Robj` object to a basic Python object. It can convert most of the R types to an adequate Python type; but, sometimes, some information is lost.

The following table shows the conversion of types. When converting lists of objects, the rules are applied recursively.

R object	Python object	Notes
NULL	None	
Logical	Boolean	(1)(2)
Integer	Plain integer	(1)(2)
Real	Float	(1)(3)
Complex	Complex (1)	
String	String (1)	
Vector	list or dictionary	(1)(4)
List	list or dictionary	(1)(4)

Array	Array or list	(5)
Other	(fails)	

Notes:

(1) In the R system there are no true scalar types. All values are vectors, with scalars represented by vectors of length one. In Python, however, there is a representational and conceptual difference between scalars immutable lists (tuples), and mutable lists. Thus, An R vector of length one could potentially be translated into any of three Python forms :

```
r("as.integer(1)")  -->  int(1)
                    -->  [int(1),]
                    -->  (int(1),)
```

It is impossible to tell which of these is best from the R object itself. With `BASIC_CONVERSION`, Rassumes that a vector of length one should be translated as scalar, and that vectors with other lengths (including 0) should be translated into Python [] lists.

RPy 0.4.3 introduced the new `VECTOR_CONVERSION` mode (see Section 4.1.5 [Vector conversion], page 14), which always returns a python list regardless of the length of the R vector.

(2) The R programming language has an integer value represented by '`NA`' (not applicable) which is converted to and from Python as the minimum integer (which is the actual value in R). Be careful, because the semantic is completely different:

Python: `NA/100 -> (-sys.maxint-1)/100 != NA`

R: `NA/100 -> NA`

(3) The IEEE float values `NaN` (not a number) and `Inf` (infinite) are also converted between Python and R.

(4) Vectors and lists from R may have an attribute `names`, which are the names of the elements of the sequence. In those cases, the sequence is translated to a Python dictionary whose keys are the names, and the values are the corresponding values of the sequence. When there are no names, the vector or list is translated to a normal Python list.

(5) When Numeric is installed, a R array is converted to a Numeric array. Otherwise, a list of nested lists is returned.

When converting R arrays, the column and row names are discarded. Also, for R data frames, row names are discarded while column names are kept. And many other R objects with complex attribute structure may loose some of its attributes when converted to Python objects. When it is necessary to keep all the information of an R object, it is better to use the `CLASS_CONVERSION` mode with proper classes (see Section 4.3 [Useful examples], page 14), or to use the `NO_CONVERSION` mode (see Section 4.1.6 [No conversion], page 14).

4.1.5 Vector conversion

The `VECTOR_CONVERSION` differs from the `BASIC_CONVERSION` mode (see Section 4.1.4 [Basic conversion], page 12) in only one way. It always returns a Python list [] object regardless of the length of the original R vector.

4.1.6 No conversion

This mode simply returns a `Robj` object which is a reference to the R object under conversion. See Section 3.2 [Robj type], page 5.

This mode always succeed.

4.1.7 Notes

Warning: In order to avoid infinite recursion with the user conversion functions, the value functions in the `class_table` and the key and value functions in the `proc_table`, are evaluated in the `BASIC_CONVERSION` mode (see Section 4.1.4 [Basic conversion], page 12). This allows the R functions, called inside the conversion functions, to return Python values, without consulting the tables again. You may force other conversion modes with the `as_py()` method of the `Robj` type (see Section 3.2.2 [Methods of Robj type], page 6), but you should be careful.

4.2 Python to R

The conversion from Python objects to R objects is automatic. It is done when passing parameters in `Robj` objects. Normal Python objects are converted to R objects according to the table given in Section 4.1.4 [Basic conversion], page 12. A `Robj` object is converted to the R reference which it represents.

In addition, every Python object which defines a `as_r()` method, is converted to R as the result of calling that method.

If none of this apply, an exception is raised.

For example:

```
>>> class Foo:
...     def as_r(self):
...             return 5
...
>>> a = Foo()
>>> dummy = r.print_(a)
[1] 5
>>>
>>> r.print_(range)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
rpy.RException: cannot convert from type 'builtin_function_or_method'
```

4.3 Useful examples

We present two examples which can be useful in real applications. They also show the use of the `PROC_CONVERSION` and `CLASS_CONVERSION` modes. These examples are provided with the RPy distribution in the directory '`examples/useful`'.

4.3.1 Enhanced Robj

This example shows an extended `Robj` type which supports R attribute look up via normal Python attribute access. It also supports the representation given by the R interpreter, and it implements the `as_r()` method for converting itself to R.

In Python 2.2 you can take advantage of the possibility to subclass types, and the examples can be rewritten in a more powerful way. However, these examples are Python 2.1 and 2.2 compatible.

File '`erobj.py`':

```
from rpy import *

class ERobj:

    def __init__(self, robj):
        self.robj = robj

    def as_r(self):
        return self.robj

    def __str__(self):
        a = with_mode(NO_CONVERSION,
                      lambda: r.textConnection('tmpobj', 'w'))()
        r.sink(file=a, type='output')
        r.print_(self.robj)
        r.sink()
        r.close_connection(a)
        str = with_mode(BASIC_CONVERSION,
                        lambda: r('tmpobj'))()
        return '\n'.join(as_list(str))

    def __getattr__(self, attr):
        e = with_mode(BASIC_CONVERSION,
                      lambda: r['$'](self.robj, attr))()
        if e:
            return e
        return self.__dict__[attr]
```

The `__str__` method makes the R interpreter print to the `tmpobj` R variable. Then, it is retrieved and returned as the string representation of the object. Note the use of the `with_mode` function for not changing the mode in use. Note, also, the use of the utility functions `as_list` and `r` (see Chapter 6 [Miscellaneous], page 20).

An example of use:

```
>>> from rpy import *
>>> from erobj import *
>>> proc_table[lambda o: 1] = ERobj
>>> set_default_mode(PROC_CONVERSION)
>>>
```

```
>>> e = r.t_test([1,2,3])
>>> e
<erobj.ERobj instance at 0x8ad4ea4>
>>> print e

        One Sample t-test

data:  c(1, 2, 3)
t = 3.4641, df = 2, p-value = 0.07418
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -0.4841377  4.4841377
sample estimates:
mean of x
        2

>>>
>>> e.statistic
{'t': 3.4641016151377548}
```

4.3.2 DataFrame class

This example is a subclass of the Enhanced Robj (see Section 4.3.1 [Enhanced Robj], page 15), which can be used to mimic the '`Data Frame`' class of the R language.

It overrides the `__getattr__` method for retrieving the columns of the data frame object. It adds a method for accessing the rows and it inherits the representation and `as_r` method.

File '`dataframe.py`':

```
from rpy import *
import erobj

class DataFrame(erobj.ERobj):
    def __init__(self, robj):
        erobj.ERobj.__init__(self, robj)

    def rows(self):
        return r.attr(self.robj, 'row.names')

    def __getattr__(self, attr):
        o = self.__dict__['robj']
        if attr in as_list(r.colnames(o)):
            return r['$'](o, attr)
        return self.__dict__[attr]
```

An example of use:

```
>>> from rpy import *
>>> from dataframe import *
>>> class_table['data.frame'] = DataFrame
>>> set_default_mode(CLASS_CONVERSION)
```

```
>>>
>>> e = r.as_data_frame({'foo': [4,5,6], 'bar': ['X','Y','Z']})
>>> e
<dataframe.DataFrame instance at 0x8156e34>
>>> print e
  foo bar
1   4   X
2   5   Y
3   6   Z
>>>
>>> e.foo
[4, 5, 6]
>>> e.bar
['X', 'Y', 'Z']
>>> e.rows()
['1', '2', '3']
```

5 Input/Output functions

RPy provides three functions for customizing the input and output from the R interpreter.

In versions 0.1 and 0.2, the input/output from/to R was connected to the C stdin/stdout, which don't necessarily coincides with the Python sys.stdin/sys.stdout. These was noticeable if you run those versions over IDLE or other IDE (probably, you don't see the output of `r.print_(5)`). Now, the R input/output is connected, by default, to the Python streams. But you can insert your own functions for reading, writing and displaying files.

`get_rpy_input()`
`set_rpy_input(f)`

Get/set the function used by the R interpreter to require input.

The parameter for `set_rpy_input` must be a function with signature `f(prompt, size)`. The parameter *prompt* is a string to be displayed and *size* is an integer which denotes the maximum length of the input buffer.

`get_rpy_output()`
`set_rpy_output(f)`

Get/set the function used by the R interpreter to output data.

The parameter for `set_rpy_output` must be a function with signature `f(s)`, where *s* is the string to be displayed.

`get_rpy_showfiles()`
`set_rpy_showfiles(f)`

[**Not available on Windows**] Get/set the function used by the R interpreter to display files, including the output from the `help` command.

The parameter for `set_rpy_showfiles` must be a function with signature `f(files, headers, title, delete)`. Parameters *files* and *headers* are lists of filenames and strings, respectively, to be displayed sequentially. Parameter *title* is the overall title and parameter *delete* signals whether the files should be deleted after displaying.

The default values for the input/output/showfiles functions are in the '`io`' module. That is, when RPy is imported, the following instructions are executed:

```
import io
set_rpy_input(io.rpy_input)
set_rpy_output(io.rpy_output)
set_rpy_showfiles(io.rpy_showfiles)
```

For input and output, the functions `io.rpy_input` and `io.rpy_output` just use the sys.stdin and sys.stdout streams of Python. For displaying files, the '`io`' module provides two functions: `io.showfiles_common` and `io.showfiles_tty`, and the default `io.rpy_showfiles` is an alias for the former. Function `io.showfiles_common` displays the files using the `io.rpy_output` function, while function `io.showfiles_tty` displays the files using a pager (namely `less`, you may need to customize it).

Notes

- When an exception is raised inside I/O functions, the exception is ignored, although it is displayed normally.
- On Windows, the output of the `help` command is always displayed on a separate window. The R event loop (see Chapter 6 [Miscellaneous], page 20) must be running for the window to be functional.

6 Miscellaneous

The `rpy` module includes some utility functions:

`as_list(obj)`
: If `obj` is a list or an object which supports the list protocol, it returns `obj`. Otherwise, it returns the one element list `[obj]`.

  This function is useful when testing whether a `Robj` has some given attribute. For example:

  ```
  >>> 'class' in as_list(r.attributes(robj))
  ```

  The reason for not doing `'class' in r.attributes(robj)` is that `r.attributes` can return either `None` (when `robj` has no attributes), a string (when `robj` has only one attribute) or a list of strings (when it has several attributes). Function `as_list` unify these three cases to allow the `in` test.

`r(s)`
: Parameter `s` is a string containing arbitrary R code. Function `r` evaluates the string `s` in the R interpreter and returns its result.

  This function is useful when working with R constructions which have no parallel Python syntax, such as linear models, for example.

  ```
  >>> set_default_mode(NO_CONVERSION)
  >>> d = r.data_frame(x=[1,2,3], y=[4,5,6])
  >>>
  >>> model = r("y ~ x")
  >>> fitted_model = r.lm(model, data = d)
  ```

  Complete fragments of R code can also be evaluated (note that the value returned by the function `r` is the value of the last expression):

  ```
  >>> r("""
  ... print(5)
  ... x <- "foo"
  ... print(x)
  ... """)
  [1] 5
  [1] "foo"
  'foo'
  ```

  This function is useful, also, when a changing R object is required. Since the expression `r.foo` is cached in a Python dictionary, later changes in the object pointed by `r.foo` are not seen. In that case, the proper expression to use is `r('foo')`, which evaluates `foo` and returns its result every time it is called.

`start_r_eventloop()`
`stop_r_eventloop()`
: *(new in 0.3)* **[Not available on Windows]** These functions start and stop the R event loop. When RPy is imported, the `start_r_eventloop` function is executed. Normally, in interactive use, you needn't stop it.

  The R event loop keeps running in a daemonic thread. In case you need finer control over that loop, you can use the `r_events` function.

`r_events([usec])`

(new in 0.3) [**parameter** *usec* **not available on Windows**] This function processes a pending R event or blocks with a *usec* microseconds timeout. The default value for *usec* is 10000 microseconds.

If, for some reason, you don't want to use the threaded event loop and you want to manually use `r_events`, you can do some loop like the following:

```
>>> r.plot([1])
>>> while 'X11' in as_list(r('.Devices')):
...     r_events()
```

The `while` loop will run until the graphics window is closed.

7 Caveat and bugs

You should be warned about some "corners":

- In some situations, the R interpreter uses the name bound to an object for displaying or for taking some action. For example, if you eval in R the expression:

  ```
  var <- c(1,2,3)
  plot(var)
  ```

  you obtain a plot with the y-axis labeled by `var`. However, in Python there is no standard way to know a name bound to an object, so, even when an `Robj` has a Python name, the R interpreter doesn't see that information. The practical consequence is that the R plots made from Python are labeled with the *entire* vector of data, usually cluttering up the graphic. I really don't see an elegant way to solve this.

  Of course, the immediate and pragmatic solution is to write `r.par(ann=0)` for disabling the automatic label annotations, or to set the labels explicitly.

  Another solution (a bit trickier) is:

  ```
  >>> r.assign("var", [1,2,3])
  >>> r("plot(var)")
  ```

- The conversion between the IEEE values, `NaN` and `Inf`, depends highly on the operating system, since that is what happens with the Python interpreter (AFAIK). I have no access to a platform other than Linux, so I may be wrong.

- At the present moment, it is not possible to pass Python functions to those R functions which requires callables as arguments, such as `integrate`, for example. It would be necessary to embed the Python interpreter in R (and only a step forward: an R package for calling Python :-)

Now, some Windows' specific notes. Most of these are due to my ignorance on the Windows OS; if you have suggestions for solving some of these points, please, contact me (see Section 1.2 [Contact info and contributing], page 1).

- The interruption from keyboard (Ctrl-C) when RPy is imported in a Python instance running on a console, is completely non-operational. I tried to deal with the `SIGINT` and `SIGBREAK` signals in a similar fashion than with Linux, but I failed. So, the code you run in a console is uninterruptible.

- Running the R event loop in a separate thread doesn't work. This point, together with the previous one, means that if you want to use a console and you need to make a plot, you must do something like the following:

  ```
  >>> r.plot(data)
  >>> while 'windows' in r_events():
  ...     pass
  ```

  Then, the loop is unbreakable with Ctrl-C. The only way to stop it is to close the graphic window.

- The Windows version of R always displays the output from the `help` command in a separate window. This window must be refreshed by the R event loop. But, now, the

problem is that the device of that window doesn't appear listed on the return value of `r_events`; so, there is no way to make a loop like the previous one.

Suggestion for Windows users: the problems listed before *only* occur when Python runs on a console (a MS-DOS window). I strongly suggest that, for interactive sessions, you use the IDLE shell instead. With IDLE, the keyboard interruption works as expected and the graphics and help windows are completely functional without the need to use the `r_events` function.

8 Acknowledgements

I want to thank to these people for their feedback and help on the RPy project:

Tim Churches
: for his advocacy, suggestions and for contributing with the `faithful` demo.

Gregory Warnes
: for his many patches and help with the compilation and testing of RPy in the Solaris system.

Rene Hagendoorn
: for his patches and help for making the Windows version of RPy. Also, for the idea of the R event loop functions.

Duncan Temple Lang
: for his work on RSPython and for detecting bugs in the initial version of RPy.