Callbacks

Although R has been tightly bound to its console, the R-core development team has great progress in letting front-end developpers customize R's interactive behavior to their needs.

rpy2 is offering to customize R's interactive behavior through callback functions.

Console I/O

During an interactive session, much of the communication between R and the user happen happen through the console. How the console reads input and writes output, can be defined through callback functions.

Read console

The ‘read console’ function is called whenever console input is expected.

The default callback for inputing data is `rinterface.consoleRead()`

rpy2.rinterface.**consoleRead**(*prompt*)

```
Parameters:  prompt  -  str
Return type: str
```

A suitable callback function will be such as it accepts one parameter of class `str`, that is the prompt, and returns the user input as a `str`.

The pair of functions `rpy2.rinterface.set_readconsole()` and `rpy2.rinterface.get_readconsole()` can be used to set and retrieve the callback function respectively.

rpy2.rinterface.**set_readconsole**(*f*)

set_readconsole(f)

Set how to handle input to R with either None or a function f such as f(prompt) returns the string message to be passed to R

rpy2.rinterface.**get_readconsole**()

get_readconsole()

Retrieve the current R alert message handler (see set_readconsole)

```
Return type: a callable
```

Write console

The ‘write console’ function is called whenever output is sent to the R console.

A suitable callback function will be such as it accepts one parameter of class `str` and only has side-effects (does not return anything).

The pair of functions `rpy2.rinterface.set_writeconsole()` and `rpy2.rinterface.get_writeconsole()` can be used to set and retrieve the callback function respectively.

The default callback function, called `rinterface.consolePrint()` is a simple write to `sys.stdout`

rpy2.rinterface.**consolePrint**(*x*)

Parameters:	x – str
Return type:	None

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.set_writeconsole(f)

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

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


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

rpy2.rinterface.**set_writeconsole**(*f*)

set_writeconsole(f)

Set how to handle output from the R console with either None or a function f such as f(output) returns None (f only has side effects).

py2.rinterface.**get_writeconsole()**

get_writeconsole()

Retrieve the current R console output handler (see set_writeconsole)

`Return type:` a callable

Flush console

The 'write console' function is called whenever output is sent to the R console.

A suitable callback function will be such as it accepts no parameter and only has side-effects (does not return anything).

The pair of functions `rpy2.rinterface.set_flushconsole()` and `rpy2.rinterface.get_flushconsole()` can be used to set and retrieve the callback function respectively.

Files

Showing files

Choosing files

File choosing a on basic R console has very little bells and whistles.

```
def choose_csv(prompt):
print(prompt)
return(filename)
```

Other

Clean up

When asked to terminate, through either its terminal console win32 or quartz GUI front-end, *R* will perform a cleanup operation at the begining of which whether the user wants to save the workspace or not.

What is happening during that cleaning step can be specified through a callback function that will take three parameters *saveact*, *status*, and *runlast*, return of 1 (save the workspace), 0 (do not save the workspace), and None (cancel the exit/cleanup, raising an `RRuntimeError`).

```
import rpy2.rinterface

rpy2.rinterface.initr()

rquit = rpy2.rinterface.baseenv['q']

def cleanup(saveact, status, runlast):
# cancel all attempts to quit R programmatically
print("One can't escape...")
return None
>>>  orig_cleanup = rpy2.rinterface.get_cleanup()
>>>  rpy2.rinterface.set_cleanup(cleanup)
>>> rquit()
```

Restore the original cleanup:

```
>>> rpy2.rinterface.set_cleanup(orig_cleanup)
```

Client-Server

Rserve is currently the default solution when looking for a server solution for R, but `rpy2` can be used to develop very easily one's own server, tailored to answer specific requirements. Such requirements can include for example access restriction, a security model, access to subsets of the R engine, distribution of jobs to other servers, all of which are currently difficult or impossible to achieve with R serve.

The `pyRserve` package addresses the connection to Rserve from Python, and although it frees one from handling the R server is also constrains one to use Rserve.

Simple socket-based server and client

Server

An implementation of a simplistic socket server listening on a given port for a string to evaluate as R code is straightforward with Python's SocketServer module.

Our example server will be in a file *rpyserve.py*, containing the following code.

```
import SocketServer
import sys
import rpy2.robjects as robjects

class MyTCPHandler(SocketServer.StreamRequestHandler):

    def handle(self):
        # self.rfile is a file-like object created by the handler;
        # we can now use e.g. readline() instead of raw recv() calls
        self.data = self.rfile.readline().strip()

        # verbose server
        print "%s wrote:" % self.client_address[0]
        print self.data

        # evaluate the data passed as a string of R code
        results = robjects.r(self.data)

        # return the result of the evaluation as a string
        # to the client
        self.wfile.write(str(results))


if __name__ == "__main__":
    HOST, PORT = "localhost", int(sys.argv[1])

    # Create the server, binding to localhost on port 9999
    server = SocketServer.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

Running a server listening on port 9090 is then:

```
python rpyserve.py 9090
```

Client

Using Python's socket module, implementing a client is just as easy. We write the code for ours into a file *rpyclient.py*:

```
import socket
import sys

HOST, PORT = sys.argv[1].split(":")
PORT = int(PORT)
data = " ".join(sys.argv[2:])

# Create a socket (SOCK_STREAM means a TCP socket)
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to server and send data
sock.connect((HOST, PORT))
sock.send(data + "\n")

# Receive data from the server and shut down
received = sock.recv(1024)
sock.close()

print "Sent:     %s" % data
print "Received: %s" % received
```

Evaluating R code on a local server as defined in the previous section, listening on port 9090 is then:

```
python rpyclient.py localhost:9090 'R.version'
```

In this example, the client is querying the R version.

rpy_classic

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

Warning

The implementation of the RPy-1.x characteristics is incomplete. This is likely not due to limitations in the low-level interface `rpy2.rinterface` but due to limited time from this author, and from limited contributions to get it improved.

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

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

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)
```

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)
```

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")
>>>
```

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)
```

Related projects

Bioinformatics

Cloud computing

rpy2 is among the many bioinformatics-oriented packages provided with CloudBioLinux. Check it out if you are considering a project involving cloud computing.

Bioconductor

Bioconductor is a popular set of R packages for bioinformatics. A number of classes defined within that project are exposed as Python classes through rpy2, in the project rpy2-bioconductor-extensions. The bioconductor project is evolving quite rapidely the mapping might not longer be working.

The blog of Brad Chapman also has good examples about how to use *rpy2* for bioinformatics tasks (or Python for bioinformatics in general).

Interactive consoles

Data analysts often like to work interactively, that is going through short cycles like:

- write a bit of code, which can be mostly involving a call to an existing function
- run that code
- inspect the results, often using plots and figures

R users will be particularly familiar with this sort of approach, and will likely want it when working with `rpy2`.

Obviously the Python console can be used, but there exist improvements to it, making the user experience more pleasant with features such as history and autocompletion.

ipython: interactive-python shell

Developed under of the *scipy <http://scipy.org>* (Scientific Python) umbrella, ipython has an “R magic” mode since its release 0.13.

Used with Ipython system of noteboks and a server backend, it is possible to have a lab notebook tracking code and figures in interactive session, as well as broadcast or share a session. Check the documentation of iPython for further details.

Note

The “R magic” seems to be a continuation of the extension to rpy2 `rnumpy`, developped for rpy2-2.0.x series and unfortunately seemingly unmaintained since then. It is truly great to see no one else but the iPython developers themselves find interest in having R accessible from iPython.

Code historians can find details on the rnumpy page

Other interactive environments

- bpython: curse-based enhancement to the Python console
- emacs: the Emacs text editor can be used to host a python session, or an ipython session

Embeddeding an R console

Python can be used to develop full-fledged applications, including applications with a graphical user interface.

`rpy2` can be used to provide an R console embedded in such applications, or build an alternative R GUI.

When offering an R console, the developer(s) may want to retain control on the the way interaction with R is handled, at the level of the console and for the base R functions targetting interactivity (see Section *Callbacks*).

The RPyGTK project demonstrates how `rpy2` can be used to implement a full-blown GUI for R using python.

Alternative interfaces

The rtools package proposes additions / customizations of the higher-level interface in rpy2.

The *pandas* <*http://pypi.python.org/pypi/pandas*> package proposes an interpretation of data frames in Python, tied to numpy structures. A custom interfacing with rpy2 is mentioned, but it appears not as much developed as the rest of the project.

R-like data strucutures

R's data frames are extremely convient when manipulating data. In `rpy2` the original R *data.frame* is represented by `rpy2.robjects.vectors.DataFrame`, but thepydataframe project has a pure Python implementation of them (with a compatibility layer with `rpy2` providing a seamless transition whenever needed.

Performances

Optimizing for performances

Memory usage

R objects live in the R memory space, their size unbeknown to Python, and because of that it seems that Python does not always garbage collect often enough when large objects are involved. This is sometimes leading to transient increased memory usage when large objects are overwritten in loops, and although reaching a system's memory limit appears to trigger garbage collection, one may wish to explicitly trigger the collection.

```
import gc
gc.collect()
```

As a concrete example, consider the code below. This has been used somewhere a unique benchmark Python-to-R bridge, unfortunately without considering specificities of the Python and R respective garbage collection mechanisms. The outcome of the benchmark changes dramatically, probably putting back rpy2 as the fastest, most memory efficient, and most versatile Python-to-R bridge.

```
import rpy2.robjects
import gc

r = rpy2.robjects.r

r("a <- NULL")
for i in range(20):
    rcode = "a <- rbind(a, seq(1000000) * 1.0 * %d)" % i
    r(rcode)
    print r("sum(a)")
    # explicit garbage collection
    gc.collect()
```

Low-level interface

The high-level layer `rpy2.robjects` brings a lot of convenience, such a class mappings and interfaces, but obviously with a cost in term of performances. This cost is neglibible for common usage, but compute-intensive programms traversing the Python-to-R bridge way and back a very large number of time will notice it.

For those cases, the `rpy2.rinterface` low-level layer gets the programmer closer to R's C-level interface, bring rpy2 faster than R code itself, as shown below.

A simple benchmark

As a simple benchmark, we took a function that would sum up all elements in a numerical vector.

In pure R, the code is like:

```
function(x)
{
  total = 0;
  for (elt in x) {
    total <- total + elt
  }
}
```

while in pure Python this is like:

```
def python_sum(x):
    total = 0.0
    for elt in x:
        total += elt
    return total
```

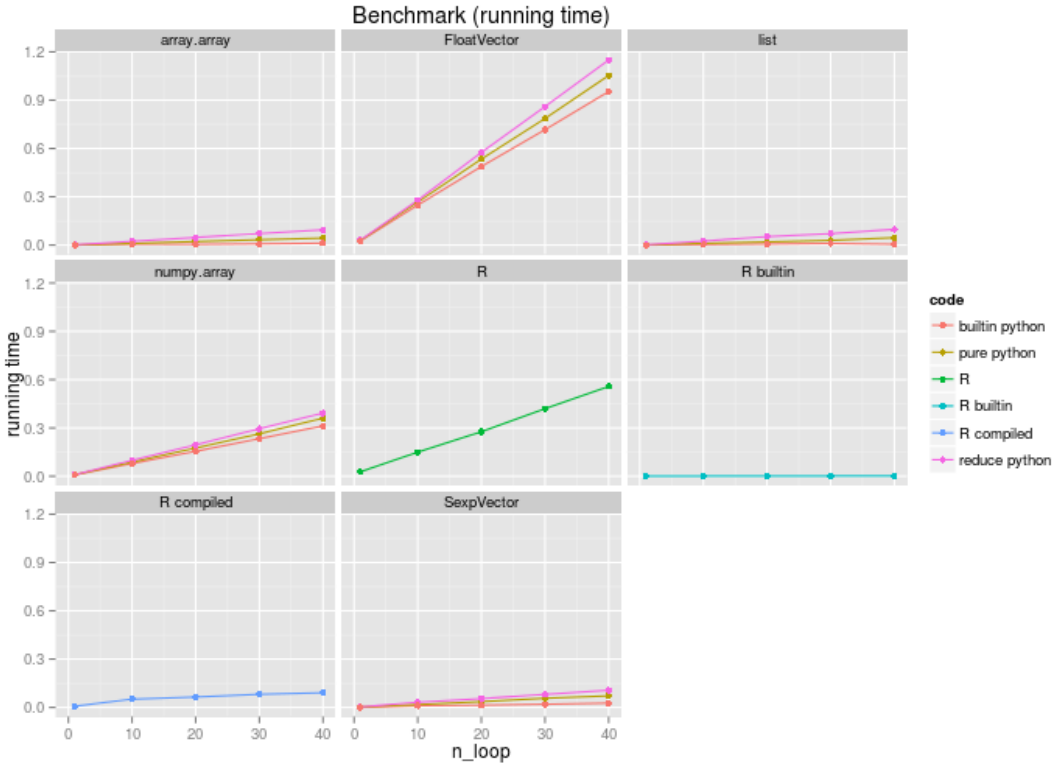
R has obviously a vectorized function *sum()* calling underlying C code, but the purpose of the benchmark is to measure the running time of pure R code.

We ran this function over different types of sequences (of the same length)

```
    n = 20000
    x_list = [random.random() for i in xrange(n)]
    module = None
    if kind == "array.array":
        import array as module
        res = module.array('f', x_list)
    elif kind == "numpy.array":
        import numpy as module
        res = module.array(x_list, 'f')
    elif kind == "FloatVector":
        import rpy2.robjects as module
        res = module.FloatVector(x_list)
    elif kind == "SexpVector":
        import rpy2.rinterface as module
        module.initr()
        res = module.IntSexpVector(x_list)
    elif kind == "list":
```

```
        res = x_list
    elif kind == "R":
        import rpy2.robjects as module
        res = module.rinterface.IntSexpVector(x_list)
        module.globalenv['x'] = res
        res = None
```

The running times are summarized in the figure below.



Iterating through a `list` is likely the fastest, explaining why implementations of the sum in pure Python over this type are the fastest. Python is much faster than R for iterating through a vector/list.

Measuring the respective slopes, and using the slope for the R code as reference we obtain relative speedup, that is how many times faster code runs.

Function	Sequence	Speedup
builtin python	array.array	33.18
builtin python	FloatVector	0.70
builtin python	list	37.43
builtin python	numpy.array	1.46

Function	Sequence	Speedup
builtin python	SexpVector	21.12
pure python	array.array	10.13
pure python	FloatVector	0.65
pure python	list	11.53
pure python	numpy.array	1.32
pure python	SexpVector	7.03
reduce python	array.array	5.22
reduce python	FloatVector	0.60
reduce python	list	5.19
reduce python	numpy.array	1.37
reduce python	SexpVector	5.13
R	R	1.00

The object one iterates through matters much for the speed, and the poorest performer is `rpy2.robjects.vectors.FloatVector`, being almost twice slower than pure R. This is expected since the iteration relies on R-level mechanisms to which a penalty for using a higher-level interface must be added. On the other hand, using a `rpy2.rinterface.SexpVector` provides an impressive speedup, making the use of R through rpy2 faster that using R from R itself. This was not unexpected, as the lower-level interface is closer to the C API for R. Since casting back a `rpy2.robjects.vectors.FloatVector` to its parent class`rpy2.rinterface.SexpVector` is straightforward, we have a mechanism that allows rpy2 to run code over R objects faster than R can. It also means than `rpy2` is at faster than other Python-to-R bridges delegating all there code to be evaluated by R when considering the execution of code. Traversing from Python to R and back will also be faster with `rpy2` than with either pipes-based solutions or Rserve-based solutions.

What might seem more of a surprise is that iterating through a `numpy.array` is only slightly faster than pure R, and slower than when using`rpy2.rinterface.SexpVector`. This is happening because the subsetting mechanism for the later is kept much lighter weight, giving speed when needed. On the other hand, accessing `rpy2.robjects.vectors.FloatVector` is slower because the interface is currently implemented in pure Python, while it is in C for `numpy.array`.

Finally, and to put the earlier benchmarks in perspective, it would be fair to note that python and R have a builtin function *sum*, calling C-compiled code, and to compare their performances.

Custom graphical devices

Warning

This is still very experimental, and using this may result in crashing the Python interpreter.

The C-API to R allows extension writers to implement custom graphical devices (using C). This feature was used to implement drivers to SVG or Cairo, for example (Cairo support made it later to the R codebase).

Rpy2 is exposing the creation of custome graphical devies to Python programmer, without the need for C.

To demonstrate how to implement a graphical, we consider the following example: a device that counts the number of times graphical primitives are used. This is something of very limited practical use, but enough to explain the principles.

Such a device would be implemented as follows:

```
import rpy2.rinterface._rpy_device as rdevice
from collections import Counter

class BeancounterDevice(rdevice.GraphicalDevice):
    """ Graphical devive for R that counts the
    number of times primitives are called."""

    def __init__(self):
        super(BeancounterDevice, self).__init__()
        self._ct = Counter()

    def circle(self, x, y, radius):
        self._ct['circle'] += 1

    def clip(self, x0, x1, y0, y1):
        self._ct['clip'] += 1

    def line(self, x1, y1, x2, y2):
        self._ct['lines'] += 1

    def mode(self, mode):
        self._ct['mode'] += 1

    def rect(self, x0, x1, y0, y1):
        self._ct['rectangle'] += 1
```

```
    def strwidth(self, text):
        self._ct['strwidth'] += 1
        return float(0)

    def text(x, y, string, rot, hadj):
        self._ct['text'] += 1
```

The class `BeancounterDevice` can now be used as genuine R plotting device.

```
from rpy2.robjects.packages import importr

dev = BeancounterDevice()

graphics = importr("graphics")
# plot into our counting device
graphics.plot(0, 0)

# Print the counts
print(dev._ct)
```

To implement a new custom graphical device for R, one only has to extend the class `rpy2.rinterface._rpy_device.GraphicalDevice`. Error messages will be printed if that new device does not implement functionalities used by R.

The Python documentation strings for the class and its methods are:

class rpy2.rinterface._rpy_device.GraphicalDevice

Python-defined graphical device for R.

activate()

Callback to implement: activation of the graphical device.

bottom

Bottom coordinate.

canGenKeybd

Ability to generate keyboard events.

canGenMouseDown

Ability to generate mouse down events.

canGenMouseMove

Ability to generate mouse move events.

canGenMouseUp

Ability to generate mouse up events.

circle()

Callback to implement: draw a circle on the graphical device. The callback will receive the parameters x, y, radius

clip()

Callback to implement: clip the graphical device. The callback method will receive 4 arguments (Python floats) corresponding to the x0, x1, y0, y1 respectively.

close()

Callback to implement: close the device.

deactivate()

Callback to implement: deactivate the graphical device.

devnum

Device number.displayListOn

Status of the display list.

getevent()

Callback to implement: get event on the graphical device.

hasTextUTF8

UTF8 capabilities of the device.

left

Left coordinate.

line()

Callback to implement: draw a line on the graphical device. The callback will receive the arguments x1, y1, x2, y2.

locator()

Callback to implement: locator on the graphical device.

metricinfo()

Callback to implement: MetricInfo on the graphical device.

mode()

Callback to implement: mode of the graphical device.

newpage()

Callback to implement: create a new page for the graphical device. If the device can only handle one page, the callback will have to eventually terminate clean an existing page.

polygon()

Callback to implement: draw a polygon on the graphical device

polyline()

Callback to implement: draw a polyline on the graphical device.

rect()

Callback to implement: draw a rectangle on the graphical device. The callback will receive 4 parameters x0, x1, y0, y1.

right

Right coordinate.

size()

Callback to implement: set the size of the graphical device. The callback must return a tuple of 4 Python float (C double). These could be: left = 0 right= <WindowWidth> bottom = <WindowHeight> top=0

strwidth()

Callback to implement: strwidth(text) -> width

Width (in pixels) of a text when represented on the graphical device. The callback will return a Python float (C double).

text()

Callback to implement: display text on the device. The callback will receive the parameters: x, y (position), string, rot (angle in degrees), hadj (some horizontal spacing parameter ?)

top

Top coordinate.

wantSymbolUTF8

UTF8 capabilities of the device.