Abstract

This describes how one can call R functions from Python.

1 Examples

```
[]
 import RS

  RS.call("rnorm", 10)

  RS.call("objects")
  RS.call("objects",3)

  RS.call("sin", 2)
  RS.call("sin", 2.0)

  RS.call("plot", [1,2,3,4])

  RS.call("plot", RS.call("rnorm", 10))
```

2 Named Arguments

What if we want to do something with named arguments in R? Note the argument METH˙KEYWORDS when declaring C-level entry points.

```
[]
 RS.call("plot", x,y, xaxis="foo")
```

As with the Java interface, it is essential that we be able to deal with non-primitive objects and send Python objects to R and vice-versa. We do this with references. Non-primitive Python objects are stored internally at the C level and a reference identifying them is sent to R as a *PythonReference*. (See the Perl interface.) Then, the R side can call methods on those objects via the .Python (yet to be done!).

```
[]
  RS.call("foo", pyObj, .convert = FALSE)
```

Need an RS.get function in Python to retrieve objects, not just methods. Like Java's fields.
When we have the OOP in R and S, we will want to be able to call methods on objects.
Also, when method dispatching in R/S, want to pass "signature".
Anonymous References
Also, want R to know what the $ operator is for Python object references.

3 References

The key development in this style of interface is that complex objects (i.e. non-primitive values) defined in one language remain in that language, by default, and are not serialized to the other language. For example, if we create a linear model fit in R, we do not attempt to represent its contents in Python, but instead we export its functionality to Python by providing it as a reference to an R object. But Python needs it to be a Python object. Therefore, we create the a Python object of class **RPython** that refers to this R object. Given multiple inheritance, we can create a class that is derived from **RPython** and also from another class.

```
[]
setenv PYTHONPATH ‘pwd‘/tests:‘pwd‘/PySrc
```

Here we create an instance of an R reference and call its *¨call˙R()* method. Note that this will then pass control to a C routine that will carry out the call to the R function, additionally passing the name/identifier of the referenced R object implicit in the Python object.

```
[]
>>> from RReference import  *
>>> r = RForeignReference("duncan")
>>> r.__callS__("plot", x=1, y=2, xlab="A string")
```

The RForeignReference objects are rarely useful by themselves. Instead, we want a Python object that is both a reference to an R object and also a class that does something. Suppose

```
[]
```

Consider the ftplib module. Can we register an R function as the callback in the retrlines method call? We can use several different approaches. We incrementally evolve to using an R closure to handle reading the lines/entries. We start by using our own Python function as follows:

```
[]
def myline(x):
  print('****'+x)

ftp = FTP('franz.stat.wisc.edu')
ftp.login()
ftp.retrlines('LIST',myline)
```

We should be able to specify an anonymous function, but this seems to have syntax problems!

```
[]
ftp.retrlines('LIST', labmda x: print('***'+x))
```

Next, we use a method from a Python class

```
[]
class lineCumulator:
    "Cumulates lines"
    def __init__(self):
       self.lines = []
    def add(self, x):
        self.lines.append(x)
    def clear(self):
        self.lines.clear();
    def getLines(self):
        return(self.lines)


k = lineCumulator()
ftp.retrlines('LIST', k.add)
k.getLines()
```

Now, let's invoke an R function.

```
[]

import RS
def rline(x):
    RS.call("print", x)

ftp.retrlines('LIST', rline)
```

Now lets do the aggregation or cumulation in R. We define a closure

```
[]
lineCumulator <-
function()
{
   lines <- character(0)
   add <- function(x) {
     lines
<<- c(lines, x)
   }

   x <- list(add=add, lines=function() {lines})
   class(x) <- "LineCumulator"
   return(x)
}
```

Now, from R this can be used in the following manner:

```
[]
> k <- lineCumulator()
> k$add("123")
> k$add("a b c")
> k$lines()
[1] "123"   "a b c"


@%$
 How do we call this from Python?  We have seen how to call the
\SFunction{print} function and we can use the same approach.


[]
k <- lineCumulator()
add <- k$add
```

```
getLines <- k$lines
```

Now, save this session and start the Python interpreter.

```
[]
python

import RS
def rline(x):
    RS.call("add", x)

ftp = FTP('franz.stat.wisc.edu')
ftp.login()
ftp.retrlines('LIST', rline)
RS.call("getLines")
```

This approach works, but requires that we have "instance" methods of a closure as global functions. This prohibits us from having two instances working simultaneously or being called in an interleaved order. A cleaner, more robust and more maintainable approach is to get the closure instance's *add()* function and have Python invoke this directly. In Python, we create the instance of the closure by calling the R function *lineCumulator()*.

```
[]
klosure =  RS.call("lineCumulator")
 def rline(x):
    RS.call("add", x, .ref=klosure)
```

or alternatively, we can fetch the add function once and write a Python function to call it directly.

```
[]
addFunction = RS.get(klosure, "add")
def rline(x):
    RS.call(addFunction, x)

ftp.retrlines('LIST', rline)
```

The *RS.call()* can operate on both function names and function references. Additionally, note the `.ref` argument which allows us to invoke a method contained in that reference, identifying the method by the name given as the first argument.

It would be nice to be able to create a Python object that was both a function object and a reference to an R function. In our example, the Python object `addFunction` would then be a callable Python function. In this way, we avoid having to write the wrapper function.

```
[]
import RS;
klosure =  RS.call("lineCumulator")
RS.call("add", "abc", ref=klosure)
RS.call("add", "abc", ref=klosure)
RS.call("add", "abc", ref=klosure)

RS.call("lines", ref=klosure)
```

Here we fetch a reference to an R object, the function *sum()* and assign this reference to a Python variable, `s`. Then we invoke the R function by passing the reference as the funtion identifier (“name”) in the call to *RS.call()*.

```
[]
>>> import RS
>>> s = RS.get("sum")
>>> RS.call(s, [1,2,3])
6
```

4 References from S

We us