The S-Python interface, like its Java, Perl and CORBA counterparts provide very generic and dynamic facilities. They understand very little of the data structures that are being passed between the different languages. One of the reasons that the interface provide very limited built-in conversion mechanisms is due to the generality. Basically, in the absence of strong typing in a language (e.g. Java, C) a user may desire to convert an instance of a particular type in the remote system to one of several classes in the local system. A simple example illustrates this one-to-many mapping. Suppose we return the value of the S expression

```
[]
 cbind(1:10, rnorm(10))
```

To what Python type should this be converted? A matrix? and if so what particular class should be used? A tuple of tuples? and if so, by which dimension do we order these inner tuples, i.e. 2 tuples of size 10 or 10 tuples of size 2? And what if the user wants to immediately use the values to create and populate an new Python object by simply treating the values as a collection of 20 integers?

Because of the numerous possibilities for conversion that can only be determined by the author of the command, we employ a very general system which can be customized at run-time to provide the appropriate conversions. There will be several default converters that get registered in our system, but the user can replace these. To this end, and at this early stage in the exploration of these inter-system interfaces, we allow the user to specify converters using functions written in either language, i.e. S and Python, along with the low-level conversion facilities using C. As the usage and needs become better identified and more stable, then we can translate these to C-level routines in the interest of efficiency.

1 Default Mechanism

Before we discuss how to customize the conversion process, it is probably a good thing to go through the steps of the process. Let's consider the case where we are calling an S function from within Python. (The following also applies in the other direction.) We consider a very simple case where we call an S function, *foo()*, with no arguments.

```
[]
 RS.call("foo")
```

This involves finding the function *foo()* in the S environment and then evaluating a call to it. Now, this returns an S value and the conversion to Python starts (via a call to `toPython()` from `PyR_call()`). If the object is an S vector without dimensions, then it is converted to a Python tuple or dictionary. The result is a dictionary if the vector has names, and a tuple otherwise.

But now what happens if it the object is not a primitive S object? In other words, how is an S object that

- has dimensions,
- is a list, or
- has a class attribute

returned to Python?

The answer is that we search through a list of registered converters and check whether any of them can handle this particular S object. The first of these converters that says it can is then asked to do so and is expected to return a Python object. If no converter is able to handle the object, then we return what we call an *RForeignReference* object. An *RForeignReference* object is a Python object which stores an identifier for an S object that is stored in a table within S (R in this particular case). We can use this Python object as a regular object and pass it to other calls. It has no real utility until we access its contents, which is done by calling S functions that work on it.

We should note that the S object associated with a *RForeignReference* does not go away (i.e. is not garbage collected) until we explicitly say so.

Let's return to the list of converters and see how they work. While these are maintained within the C code and one can register converters which are C routines, it is more usual that Python and S users will register functions written in those languages, and this works perfectly well. Basically, each converter that is registered has two components:

- a predicate function,
- a converter function, and

The predicate function is used to determine if the converter function is capable of handling the particular type of object that is to be converted. This is given a reference to S object being converted and a tuple containing its class names (i.e. the value of a call to *class()*). It returns a logical value indicating whether the converter will operate on it.

The file **tests/PythonFunctionConverters.py** contains an example of of a predicate function – *isMatrix()* – that handles matrices in and it is used in **tests/PythonFunctionConvertersTest.py**. So let's consider a different example. Suppose we have a class named *longitudinal* which represents multiple measurements on a patient at different times. The S object may be created something like

```
[]
 x <- list(times=c(1,3,8), values=c(96.8, 98.6, 99.3))
 class(x) <- "longitudinal"
```

or via a *longitudinal()* constructor function. We want to register a converter for these types of objects and so we write a Python predicate function such as

```
[longitudinal predicate]
def longitudinal_p(ref, klasses):
     "Checks if the RForeignReference object is a 'longitudinal' object."
     x = 'longitudinal' in klasses
     return(x)

@use longitudinal converters
```

This suffices for the predicate. Now, let's focus on writing the converter function. Let's suppose that we develop a corresponding Python class.

```
[longitudinal class]
class Longitudinal:
    times = ()
    vals = ();
    def __init__(self, Times, Vals):
      ""
      self.times = Times
      self.vals = Vals
      print "Creating Longitudinal object"
      return(None)

    def length(self):
      ""
      return(len(self.times))
```

Then, the converter function may look something like

```
[longitudinal converters]
import RS
def longitudinalConverter(ref, klasses):
    ""
    return(Longitudinal(RS.call("$", ref,"times"),  RS.call("$", ref,"values")))
```

We can then put these functions and class definition into the file named **longitudinal.py**. This can be imported in the usual manner.

```
[longitudinal.py]
@use longitudinal class

@use longitudinal predicate

@use longitudinal converters
```

The next step is to register the predicate and converter functions. We do this using the Python *setConverter()* function. Note the order in which we specify the functions.

```
[]
RS.setConverter(longitudinal.longitudinalConverter, longitudinal.longitudinal_p, "Converte
```

So now lets use this converter. We will write a short S function *longitudinal()* that creates a longitudinal object and returns it.

```
[longitudinal.S]
 longitudinal <-
 function(values, times=1:length(values)) {
   x <- list(times=times, values=values)
   class(x) <- "longitudinal"
   x
 }
```

1.1 Ordering the Converters

We mentioned that we look through the list of converters until we find one that claims to be able to handle the object. This means that the order is important.

We will add an argument to control the position when adding a converter. Also, we will hook up the remove converter routine to a Python a function.

One can determine what converters are currently registered by retrieving a list of the descriptions. This is done by a call to *RS.getConverterDescriptions()*.

In the following, we illustrate how an S object with a class is transferred as a reference to an **RListReference**. The elements of this object can then be accessed as if they were Python attributes.

```
[]

import RS
RS.call("source", RS.call("system.file","tests/myClass.S"))
a = RS.call("myClass")
a.a
```

2 Python Functions as Converters

```
[]
def dimensionalInteger(x):
   ncol = dim(x)[1]
   val = []
   for i in range(ncol):
     val[i] = RS.call("[",x,,i+1)
   return val
```