Chapter 14

An Interface to Python

14.1 R and Python

The XRPython package implements an interface to Python according to the XR interface structure described in Chapter 13. This chapter describes the use of the interface in an application extending R. The application will likely implement an R package using the interface to carry out some computations in Python and incorporating those computations into software for the users of the application. The XRPython package, the XR package it imports and the shakespeare package example are available from `github.com/johnmchambers`.

Python computations consist mainly of function calls and the use of methods and fields in objects from Python classes. These are available directly through methods for an interface evaluator or equivalent function calls (Section 14.2).

Extending R using the interface to Python usually includes providing functions in an application package in R. These can incorporate such direct computations in Python.

Additional methods and function calls provide interfaces to other Python programming that is likely to be needed, such as importing modules (Section 14.3)

If specific functions or classes in Python are useful, *proxy* functions and classes in R (Sections 14.4 and 14.5) make a much simpler interface. Calls to the function and computations on the objects appear to the user as ordinary R expressions, but carry out the corresponding computation in Python.

The XR interface structure creates and manages proxy objects for the results of computations in Python, avoiding conversion of data between the languages when not required. Data conversion facilities are available when needed, including the representation of arbitrary R objects in Python and the return of arbitrary Python objects to R (Section 14.6).

Python is a valuable resource for interface programming. It is a popular language for implementing a wide range of computational techniques. Its design emphasizes readability and simplicity. From our perspective, it has many modules covering all sorts of techniques. Programming in Python, particularly to extend existing software, is convenient and much of the programming structure meshes smoothly with R and with the XR structure for interfaces.

Applications using the interface typically will do some Python programming of their own to supplement or specialize existing software. The R package structure facilitates this by having a directory of Python code installed in the R library. The code will typically consist of one or more Python modules that are imported by the package. When changing and testing the modules, it may be more convenient to work directly in Python, through an interactive environment such as supplied by the Jupyter application[1].

The XRPython package is designed for computing with R, with help from an interface to Python as a server language. Projects starting from Python could do the opposite. There is a widely used interface *from* Python to R, rpy2.[2] If you're working from a Python project, take a look at that. However, our target is to contribute to a project with some substantial programming and interaction in R; programming in the large, in the sense of Chapter 8. The $\boxed{\text{INTERFACE}}$ principle encourages such projects to reach out to helpful computations, in this case written in Python.

14.2 Python Computations

Computations in Python using the XRPython interface are carried out by a Python interface evaluator, an object from class `"PythonInterface"`. The current Python evaluator is obtained by:

```
ev <- RPython()
```

If there is no such evaluator, one will be created, starting the embedded version of Python. Although it is possible to have multiple active evaluator objects, there is usually no reason to do so with an embedded interface.

All the interface computations use methods and fields of an evaluator object. R programmers may feel more comfortable with a functional computation rather than methods. All the methods have functional equivalents as we will describe below. There is no operational difference for the computations in this section, but some differences for the programming operations of Section 14.3.

[1]http://jupyter.org

[2]`rpy.sourceforge.net`

The computations in this section are the base for all other computations such as those using proxy functions and classes. They provide maximum flexibility for implementing interface calculations, but the proxy techniques are simpler when they apply. For the user of the application package, in particular, computations are expected to appear as regular R with no need to refer explicitly to the interface. An application package can hide the interface through any combination of functions in R using direct interface computations and proxies to software in Python incorporated into the package.

The main interface methods for direct computation are `$Eval()` for expressions returning a value and `$Command()` for other Python statements and directives:

```
ev$Eval(expr, ...)
ev$Command(expr, ...)
```

with functional equivalents:

```
pythonEval(expr, ...)
pythonCommand(expr, ...)
```

These and the other functional equivalents for computations obtain the current evaluator object and call the corresponding method.

In these computations `expr` is a character string to be parsed and evaluated in Python. Remaining arguments are objects, either proxy objects from previous expressions or R objects to be converted to Python. The objects are inserted into `expr` to replace C-style `"%s"` fields in the string.

All expressions evaluated by the interface are computed by Python function `value_for_R()`. If the type of the result is one of the standard scalar types (numeric, logical, string or `None`), the value is converted to a string that R will evaluate to get the equivalent R object. Otherwise, the result is assigned in Python. The string returned will be interpreted in R as a proxy object containing the name used for the assignment and a description of the result (the Python class, the length and optional module). The proxy object will turn into its name when used in a subsequent calculation. Thus the result of any interface computation can always be passed back to Python for further use.

To force a computed result to be converted, the `$Eval()` method and all proxy functions take an optional argument `.get=`; supply `.get = TRUE` to force conversion.

Objects may also be converted explicitly. The `$Send()` method converts its argument to Python and returns a proxy object for the result. The `$Get()` method converts a proxy object to the equivalent R object, as far as possible. Both methods will do something for arbitrary objects; see Section 14.6.

To illustrate these methods, first we create a Python list of 3 numbers:

```
> xx <- ev$Eval("[1, %s, 5]", pi)
> xx
R Object of class "list_Python", for Python proxy object
Server Class: list; size: 3
```

The value returned is a proxy for the list. The first line of the example could equally have been:

```
xx <- ev$Send(c(1, pi, 5))
```

We can print any result in Python:

```
> ev$Command("print %s", xx)
[1, 3.14159265358979, 5]
```

In Python, unlike in R, printing is not an expression.

Python lists are one of the built-in proxy classes, as the first example showed. The proxy object `xx` has all the usual list methods:

```
> xx$append(4.5)
NULL
```

To get the Python list as an R object:

```
> as.numeric(ev$Get(xx))
[1] 1.000000 3.141593 5.000000 4.500000
```

Python has no typed arrays, so we convert the list of numbers to a numeric vector on the R side. An alternative is to construct a special vector object in Python and convert that:

```
> ev$Call("vectorR",xx, "numeric",.get = TRUE)
[1] 1.000000 3.141593 5.000000 4.500000
```

(See Section 14.6, page 319, for the technique and another example).

Objects in the interface evaluator are valid only while that evaluator is running; similarly, R proxy objects will no longer be valid afterwards. To preserve proxy objects over sessions, serialize them to a file:

```
> ev$Serialize(xx, "./xxPython.txt")
```

Then, at some later time, the `$Unserialize()` method will bring the object back, to a new evaluator:

```
> xxx <- ev$Unserialize("./xxPython.txt")
> xxx
Python proxy object
Server Class: list; size: 4
> as.numeric(ev$Get(xxx))
[1] 1.000000 3.141593 5.000000 4.500000
```

The serialization is done Python-style, with `pickle()`, so issues of conversion do not arise. The methods have optional arguments as described in Section 13.6, page 278.

All these have functional equivalents with the same arguments as the methods:

```
pythonGet(object); pythonSend(object)
pythonSerialize(object, file); pythonUnserialize(file)
```

14.3 Python Programming

In addition to the general methods in the previous section and to the proxy objects, functions and classes, the XRPython interface provides some methods to assist in using Python software and in extending the software. The software available can be extended by adding to the search path for Python modules and importing modules. Python source can be included directly from source. An interactive Python shell can access the objects computed in the interface.

Python's modules are files of source, possibly compiled. The Python evaluator looks for these in a system path list of directories, similar to R's `.libPaths()`. XRPython has two functions for appending a directory to the system path and for importing from modules found on the path.

To add a directory to the system path in Python:

```
pythonAddToPath()
```

The expected use of the call is to make a directory of Python code in an application package available. Application packages will usually need to write some of their own functions, classes and other computations in Python. The XR structure for interfaces has a convention that the Python code in an application is in the `"python"` subdirectory of the installed package,

The call with no arguments, as above, from the R source in an application package adds the directory named `"python"` in the installed version of the package from which the call originates.

If the application wants to use a different source directory, perhaps in order to have different versions or optional additional code, supply the directory as an argument; for example, `"pythonA"` for source directory `"inst/pythonA"`.

The `package=` argument should be given explicitly if the code to be added is from a different R package:

```
pythonAddToPath("plays", package = "shakespeare")
```

This would add directory `"plays"` in the R package shakespeare, which would correspond to directory `"inst/plays"` in the source for that package. To add a directory that is not part of any R package, give an empty string, `""`, as the `package` argument. For more details on the XR structure for path management see Section 13.4, page 270.

Once the necessary directories are on the system search path, Python has a variety of commands to import modules (files) and make their contents available. These are expressed with an R-style syntax through `pythonImport()`. The first argument in the call is a module name, the remaining arguments are the names of objects to be explicitly imported from the module. If `"getPlays.py"` is a file of Python code in a directory that has been added to the path, then

```
pythonImport("getPlays", "byTitle")
```

will import the object `byTitle` created by the code in that file. For those familiar with Python, this is in the style of the Python command

```
from module import ...
```

All the arguments should be character strings, but the `"..."` arguments can be vectors with multiple object names. The evaluator keeps track of imports, so repeated calls for the same imports only go to Python once. The call can be used to import any objects, not just functions.

Giving only the module argument is equivalent to the Python `import` directive. This is different from a simple `import` call in an R namespace, in that it does not make the simple object names available. So

```
pythonImport("getPlays")
```

would not make `"byTitle"` available by simple name, but would recognize the fully qualified version, `"getPlays.byTitle"`. The Python convention to import all objects is invoked by providing the second argument as `"*"`.

If functions are handled by defining proxy functions in R as described in Section 14.4, there is no need to import them or their module explicitly; the proxy function takes care of that.

The functions `pythonAddToPath()` and `pythonImport()` have corresponding evaluator methods. These differ in that they only affect the particular evaluator on which they are invoked. The function versions will apply to any evaluator, and therefore can be called before an evaluator exists. An application package can call them from its source code: the actions will take place when the package is loaded. For any likely situation, the function version is more appropriate.

The remaining programming methods to be described can be called equivalently as method or function, with identical results.

To evaluate a file of Python source, use `pythonSource()` or the method:

```
ev$Source(filename)
```

The file can contain arbitrary expressions and commands. They will be evaluated in the same namespace as other evaluator expressions, so assignments will become available for later computations. To refer to an explicitly assigned object as an argument in an interface expression, the name should be supplied as an R object of class `"name"`, *not* as a character string. So, if our source file had the line

```
pi = 3.14159
```

then the object assigned as `pi` should be referred to by `as.name("pi")`:

```
> ev$Eval("%s/2", as.name("pi"))
[1] 1.570795
```

Proxy objects are nearly always a simpler form of reference, however.

For the definition of a single function XRPython also has a `$Define()` method which can take as an argument text containing a Python function definition. It sends the function definition to Python as a command and returns a proxy for the newly defined function. For a trivial example, consider the function definition:

```
def repx(x):
    return [x, x]
```

For the `$Define()` method this is equivalent to a character vector with the lines of the definition as elements:

```
> text <- c("def repx(x):", "    return [x, x]")
> repxP <- ev$Define(text)
```

This creates the R proxy function, assigned as `repxP`:

```
> twice <- repxP(1:3)
> unlist(pythonGet(twice))
[1] 1 2 3 1 2 3
```

The definition can also be taken from a file, via the `file=` argument.

```
>repxP <- ev$Define(file = "./repx.py")
```

The effect is to read all the lines from the file and collapse them into a string, separated by new lines. If one does not need to create the proxy function in the same call, `$Source()` is more efficient.

To do some interactive computations in the same workspace used by the evaluator, through the `$Shell()` method or the function version:

```
pythonShell()
```

This runs a simple interaction, reading expressions and evaluating them through the command method. To see results, you need to use the Python `print` directive, or call some other output-generating function. To exit the evaluator, give the usual Python command, `"exit"`.

```
> pythonShell()
Py>: print repx([1,2,3])
[[1, 2, 3], [1, 2, 3]]
Py>: exit
```

The interactive environment is definitely not competitive for Python programming; in particular, to continue an expression over multiple lines, you need to end all the lines except the last with an unescaped backslash. The advantage of `$Shell()` is that expressions use the same working environment as other interface methods. Another way to define a Python function directly would be to enter the definition to the shell, escaping all but the last line (and being careful to follow Python's use of spaces to define blocks of code).

```
> pythonShell()
Py>: def rep3(x):\
Py+:   return [x, x, x]
Py>: print rep3(pi)
Python error: Evaluation error in command "print rep3(pi)":
      name 'pi' is not defined
Py>: pi = 3.14159
Py>: print rep3(pi)
[3.14159, 3.14159, 3.14159]
Py>: exit
```

As the example shows, errors in Python are reported but the shell continues until a line matches `"exit"`. The interactive expressions are evaluated in the same context as other interface method calls, meaning that you can query various system parameters, such as the search path, to better understand what is happening to your interface computations.

In order to examine proxy objects in Python, you need to get the name under which the Python object was assigned, by calling `pythonName()` (outside the shell):

```
> pythonName(xx)
[1] "R_1_1"
> pythonShell()
Py>: print R_1_1
[1, 3.14159265358979, 5, 4.5]
Py>: exit
```

14.4 Python Functions

Functions are central to Python; it operates largely following its own version of the $\boxed{\text{FUNCTION}}$ principle. Low-level computations such as arithmetic are not function calls, at least not when they apply to basic objects. Other than that, pretty much everything is, including methods in Python's encapsulated OOP implementation. This is functional *computing* as opposed to functional *programming*, as distinguished in Section 1.5. Objects are references and side effects from function calls are to be expected.

Calling Python functions from R is made simpler by the use of proxy functions; that is, functions in R, calls to which map directly to corresponding Python function calls. For most functional computing, proxy functions eliminate the need to deal directly with the interface evaluator.

A proxy to a Python function is created by calling `PythonFunction()` in the XRPython package:

```
PythonFunction(name, module)
```

This returns a proxy function object given the name and, optionally, the module from which the function should be imported. A call to this function object in R evaluates a corresponding call to the named Python function, returning (usually) a proxy object for the result of that call. `PythonFunction()` is actually the generator function for the `"PythonFunction"` class, a subclass of `"function"` with extra information describing the Python function.

An example: The `"xml"` module in standard Python has a function `parse()` down a few levels, in module `"xml.etree.ElementTree"`. To create a proxy function for `parse()`:

```
> parseXML <- PythonFunction("parse", "xml.etree.ElementTree")
```

A call to the R function imports the Python module if necessary and calls the function:

```
> hamlet <- parseXML("./plays/hamlet.xml")
> hamlet
Python proxy object
Server Class: ElementTree; size: NA
```

The proxy object returned may be from any relevant Python class, in this example "ElementTree". The Python function can also be written specially to return the representation of a general R object as discussed in Section 13.8, page 288, in which case the interface will construct an object from the target R class.

For extending R, proxy functions should be defined in the source of an application package. Users of that package need not be concerned with the proxy nature of the function: parseXML() can be treated by users as a regular R function.

Proxy functions mesh with proxy classes. If the Python methods or fields for the "ElementTree" object are useful (as they are, in fact), the application package will define a proxy for the Python class. As with the proxy function, users of the application package can then invoke methods or access fields just as they would for an R reference class, without worrying about the interface. We'll look at this example in the next section.

Let's examine the actual R function generated in the example above:

```
> parseXML
Proxy for Python function "parse", from module "xml....
function (..., .ev = XRPython::RPython(), .get = NA)
{
    nPyArgs <- nargs() - (!missing(.ev))
    if (nPyArgs < 1)
        stop("Python function parse() requires at least....
            nPyArgs)
    if (nPyArgs > 2)
        stop("Python function parse() only allows 2 ....
            nPyArgs)
    .ev$Import("xml.etree.ElementTree", "parse")
    .ev$Call("parse", ..., .get = .get)
}
```

In addition to arguments that will be passed to the Python function, all proxy functions have two optional arguments: .ev= and .get=: the first of these specifies the evaluator to be used and the second the strategy for converting the result to an R object. The evaluator is by default, and nearly always, the current Python interface evaluator.

The default strategy for conversion corresponding to `.get=NA` is to convert scalars and return everything else as a proxy object. To force all results to be converted, include the argument `.get = TRUE`.

In constructing the proxy function, XRPython uses metadata in Python about the function object. In this example, the Python function has two arguments, one required and one optional. After checking for too few or too many arguments, the body of the proxy function does two things: it ensures that the Python module is imported in the form needed and it evaluates a call to the Python function using the `$Call()` evaluator method.

Python has several versions of importing, presenting options for proxy functions. The call to `PythonFunction()` above implied that the object named `"parse"` should be imported from the specified module and called as `parse()`. The `$Import()` method call in the proxy function does this, effectively generating the Python directive

```
from xml.etree.ElementTree import parse
```

The alternative is to import only the module and then to call a fully qualified reference to the function. This would be done in the example by giving the fully qualified reference to the function entirely in the function name:

```
> parse2 <- PythonFunction("xml.etree.ElementTree.parse")
```

The resulting function will generate similar Python calls, but to the fully qualified version of the function, and will generate the simple import statement:

```
import xml.etree.ElementTree
```

The two versions will do the same computations but only the first will define a local variable `"parse"` in the namespace. The distinction is similar to that in R between importing a function into a package and referring to it by an expression like `shakespeare::parseXML`. The tradeoff is similar too. Importing the object explicitly leads to more readable code but could mask a relevant object of the same name in some standard module, just as having `parse()` in the R application package could in some situations mask the `parse()` function in base.

The argument list of the proxy function is passed directly to Python, aside from checking the number of arguments. Python allows much of the flexibility of R in argument calls, such as named arguments ("keyword" in Python terminology), and missing arguments to some extent, provided the arguments have default values (precomputed, not expressions as in R). The XRPython proxy functions pass along the pattern of positional and named arguments in the R call. The proxy calls the Python function with the arguments supplied, including names.

In addition to regular arguments, required or optional, Python has a mechanism (two, actually) for an arbitrary number of arguments, similar to `"..."` in R. A formal argument in Python of the forms `"*name"` and `"**name"` match arbitrarily many unnamed and named ("keyword") arguments. If either of these patterns is present, the interface allows arbitrarily many arguments to the Python function.

The specific list of formal arguments is available from the Python metadata, and is retained in the proxy function for documentation as slot `"pyArgs"`:

```
> parseXML@pyArgs
[1] "source"   "parser ="
```

The trailing `"="` indicates optional arguments. If the function took arbitrarily many arguments, the last name in the list would be `"..."`. Python functions can have inline documentation, like reference methods in R. If they do, this is kept in the R proxy as slot `"pyDocs"` (`parseXML()` doesn't have any).

14.5 Python Classes

Python implements encapsulated OOP by `class` definitions in the language, containing a set of method definitions. Evaluating the definition creates a corresponding class object. The class definition also creates automatically a generator function of the same name for instances of the class, defined by a special initialization method. As in R, methods are slightly specialized Python functions stored as members of the class object.

Typically for Python, the implementation is simple to use and relatively simple internally. There are very few restrictions, which makes for flexibility. But the definition says nothing about fields, preventing validation of their types, or even identifying the fields through the class definition. With a few such limitations, an interface from R to classes in Python is straightforward and helpful.

A call to `setPythonClass()` in the XRPython package produces a proxy class definition in R for a Python class, incorporating the Python metadata for the class within the XR structure. When the call comes from the source code for an application package, the metadata is needed during installation. To avoid this requirement, a load action or a setup step may be used to define the class when the application package is loaded or by a separate script (see Section 13.7, page 282).

Knowing the Python class and module is enough to obtain metadata that defines the methods for the R class. The module must be available from the Python search path when `setPythonClass()` is called, which may require an earlier call to `pythonAddToPath()`.

Because the fields are not formally defined, more effort may be required for them. If possible, an object from the class is used as an example. If the default object generated from the class has the necessary fields, this will be used automatically. Otherwise, an appropriate `example=` argument in the call will be needed, as shown below.

In the use of XML data in our shakespeare package, the object returned from the `parseXML()` is of Python class `"ElementTree"`, defined in the Python module `"xml.etree.ElementTree"`. A proxy R class is created by:

```
ElementTree <- setPythonClass("ElementTree",
                       module = "xml.etree.ElementTree")
```

The value returned by a call to `setPythonClass()` is a generator function for the class, as usual. It's more likely, however, that a suitable object is created by a call to a Python function, such as the example on page 312. If a proxy class has been defined for the server language class of the proxy object returned, the return value is automatically promoted to this class, making the methods and fields available.

In the example, the value returned is from Python class `"ElementTree"`, which has a method `findtext()`, among others. Once the proxy class is defined, the object returned can use all these methods directly:

```
> hamlet$findtext("TITLE")
[1] "The Tragedy of Hamlet, Prince of Denmark"
```

As it happens, this class has no interesting fields.

Because Python has no formal definition of the fields in a class (as with S3 classes in R), objects from the class will have fields with whatever name has been used in assigning them, usually via the `` `.` `` operator in Python. Fields must be inferred from an object or specified explicitly; there are some options in the call to `setPythonClass()` to handle different cases.

The assumption when only class and module are specified is that the default object from the Python class has the appropriate fields. The interface generates that object, by calling the Python generator function for the class with no arguments, and examines its fields. This works in the example since there are no useful fields anyway. It would also work if the class was specially designed for the R interface, as is the `"Speech"` class discussed later in the section.

More typical of Python programming is to generate a default object with few or no fields, and have various methods add the fields. In this case the application package should compute an object from the class that *does* have suitable fields, if possible, and supply a proxy for this object as the argument `example=` to `setPythonClass()`.

The elements of the parse tree in our example have Python class "Element". A suitable object from this class is returned by the getroot() method for class "ElementTree". A computation to define the "Element" proxy class could be:

```
> hamlet <- parse("./plays/hamlet.xml")
> Element <- setPythonClass("Element",
+                  module = "xml.etree.ElementTree",
+                  example = hamlet$getroot())
```

The output of

```
Element$fields()
```

will show the fields; a few of these are special fields for the interface; the rest, with class "activeBindingFunction", are proxies for the Python field of the same name. In this case, the object has proxy field "tag", among others. Once the proxy class for "Element" has been defined the fields are available:

```
> hamlet$getroot()$tag
[1] "PLAY"
```

Applications will often find it convenient to add their own Python code, includine class definitions. These can be designed for use by proxy and be valuable in structuring the data conveniently for an interface from R.

In this example, the XML structure is very hierarchical, suitable for tree-walking logic and recursive computations. In the Shakespeare data, plays are organized into subtrees by ACT and these in turn into subtrees by SCENE. Within a scene there will be multiple speeches, a third level of subtree, along with other nodes for stage directions, etc.

Both R and Python tend to prefer a more linearized organization; in R to be natural for vectorized computation, in Python to simplify iteration. The Python software in shakespeare includes functions and corresponding classes to “flatten” the structure. In particular, one would often like to analyze the contents of individual speeches; for example, to compare speakers within a play or across plays.

Package shakespeare defines proxy Python classes for each level of the hierarchy and corresponding methods that flatten the tree into the form of a list of all acts, all scenes or all speeches. In particular, the Python function getSpeeches() returns a list of all the speeches in the tree supplied as its argument. Each element is an object of class "Speech".

All the objects of this class have fields "play", "act", "scene", "speaker" and "lines". The first four fields are character strings. The last is a list of the lines of text in speech, extracted from the corresponding XML nodes.

The idea of the class is that objects are normally created from corresponding XML nodes of tag SPEECH. The initialization method expects to get such an object as its argument. It's a good rule, however, to make initialization methods (whether in Python or R) work sensibly when no argument is given. In this case, the class was designed to work with the R interface, so its initialization method in Python deliberately creates all the relevant fields:

```
def __init__(self, obj = None,
             act = '<Unspecified>', scene = '<Unspecified>'):
    self.act = act
    self.scene = scene
    ## to be a well-behaved class, we always set the 4 fields
    if obj is None:
        self.speaker = '<Unspecified>'
        self.lines = [ ]
    else:
        self.speaker = obj.findtext('SPEAKER')
        lines = obj.findall('.//LINE')
        linetext = []
        for line in lines:
            linetext.append(line.text)
        self.lines = linetext
```

To set up some analysis of speeches, the user calls the R proxy for the function getSpeeches() with the XML tree previously parsed from the file for the play:

```
> speeches <- getSpeeches(hamlet)
> speeches
R Object of class "list_Python", for Python proxy object
Server Class: list; size: 1138
```

The speeches object is a proxy for a standard Python list, whose elements are objects of class "Speech". Proxy classes for Python lists and dictionaries are built into the XRPython interface, so the methods for them can be used in R as they would be in Python. The shakespeare package has created a proxy class for "Speech".

For example, if we're curious about the last speech in the play:

```
> last <- speeches$pop() # the last speech
> speeches$append(last) # put it back
NULL
> last
```

```
R Object of class "Speech_Python", for Python proxy object
Server Class: Speech; size: NA
> last$speaker
[1] "PRINCE FORTINBRAS"
> last$lines
R Object of class "list_Python", for Python proxy object
Server Class: list; size: 9
```

We'll come back to this example in discussing data conversion, the next topic.

14.6 Data Conversion

Python objects come in a range of built-in and user-defined classes (or "types", both terms being used in Python). For applications, as opposed to programming, the important built-in types include a variety of scalars: numeric of several flavors, character string and boolean. Other than scalars, the built-in container types are `"list"` for (unnamed) list objects and `"dict"` for dictionaries.

The default XR strategy sends named lists and environments to Python as `"dict"` objects; all other R vectors (of any type) are sent as `"list"`, except that by default vectors of length 1 are sent as scalars. Some examples, in which the `"Server Class"` of the proxy object shows what was sent:

```
> pythonSend(1:3)
R Object of class "list_Python", for Python proxy object
Server Class: list; size: 3
> pythonSend(1)
Python proxy object
Server Class: float; size: NA
> pythonSend(1L)
Python proxy object
Server Class: int; size: NA
> pythonSend(list(first = 1, second = 2))
R Object of class "dict_Python", for Python proxy object
Server Class: dict; size: 2
```

As the first and last examples show, lists and dictionaries have proxy classes in R. All their usual Python methods should be available.

```
> a <- pythonSend(1:3)
> a$reverse()
NULL
```

```
> as.integer(pythonGet(a))
[1] 3 2 1
> b <- pythonSend(list(first = 1, second = 2))
> b$has_key("first")
[1] TRUE
```

We don't create proxy classes for the scalar types; they have few useful methods and will normally not be retained as proxy objects.

Vectors of length 1 by default will be sent as scalars. The `noScalar()` function turns off the scalar option (the contents of the object are unchanged otherwise):

```
> pythonSend("testing")
Python proxy object
Server Class: str; size: 7
> pythonSend(noScalar("testing"))
R Object of class "list_Python", for Python proxy object
Server Class: list; size: 1
```

Python has no typed arrays, unlike R or Julia. When an application needs to indicate the type of a vector in R, either when the object is being sent to Python or is being returned in the value of an expression, the technique is to use the R class, `"vector_R"` (Section 13.8, page 300). This class has a slot `"data"` containing a vector, but the type of the vector is explicitly provided as slot `"type"`, so that the intended type of the vector remains clear when the object is translated in JSON and also when it is sent to a language such as Python that will treat all basic R vectors as list objects.

When an object being converted is explicitly of class `"vector_R"`, a method for `asRObject()` will coerce it to the declared vector type. This can be used in Python functions to customize list objects when returned to R. The Python side of the XRPython interface has a function `vectorR()` with arguments

```
vectorR(obj, type, missing)
```

The last two arguments are optional, specifying the R class to which the converted vector should belong and a list of elements that should be interpreted as missing.

In our previous example with Python class `"Speech"`, the list in field `"lines"` is known to have only character strings as elements. The class has a method:

```
def getText(self):
    return RPython.vectorR(self.lines, "character")
```

that returns a character vector to R:

```
> pythonGet(last$getText())
[1] "Let four captains"
[2] "Bear Hamlet, like a soldier, to the stage;"
[3] "For he was likely, had he been put on,"
[4] "To have proved most royally: and, for his passage,"
[5] "The soldiers' music and the rites of war"
[6] "Speak loudly for him."
[7] "Take up the bodies: such a sight as this"
[8] "Becomes the field, but here shows much amiss."
[9] "Go, bid the soldiers shoot."
```

There are frequently a number of options in these situations. Computations on the R side could coerce the list to a specific type, if the desired type was known. The field in the Python object could itself have class `"vector_R"`, but this would be a complication for Python computations on the field.