20.23. xmlrpclib — XML-RPC client access

**Note:** The xmlrpclib module has been renamed to xmlrpc.client in Python 3.0. The 2to3 tool will automatically adapt imports when converting your sources to 3.0.

*New in version 2.2.*

XML-RPC is a Remote Procedure Call method that uses XML passed via HTTP as a transport. With it, a client can call methods with parameters on a remote server (the server is named by a URI) and get back structured data. This module supports writing XML-RPC client code; it handles all the details of translating between conformable Python objects and XML on the wire.

```python
class xmlrpclib.ServerProxy(uri[, transport[, encoding[, verbose[, allow_none[, use_datetime]]]]]])
```

A `ServerProxy` instance is an object that manages communication with a remote XML-RPC server. The required first argument is a URI (Uniform Resource Indicator), and will normally be the URL of the server. The optional second argument is a transport factory instance; by default it is an internal `SafeTransport` instance for https: URLs and an internal HTTP `Transport` instance otherwise. The optional third argument is an encoding, by default UTF-8. The optional fourth argument is a debugging flag. If `allow_none` is true, the Python constant `None` will be translated into XML; the default behaviour is for `None` to raise a `TypeError`. This is a commonly-used extension to the XML-RPC specification, but isn't supported by all clients and servers; see http://ontosys.com/xml-rpc/extensions.php for a description. The `use_datetime` flag can be used to cause date/time values to be presented as `datetime.datetime` objects; this is false by default. `datetime.datetime` objects may be passed to calls.

Both the HTTP and HTTPS transports support the URL syntax extension for HTTP Basic Authentication: `http://user:pass@host:port/path`. The `user:pass` portion will be base64-encoded as an HTTP ‘Authorization’ header, and sent to the remote server as part of the connection process when invoking an XML-RPC method. You only need to use this if the remote server requires a Basic Authentication user and password.

The returned instance is a proxy object with methods that can be used to invoke corresponding RPC calls on the remote server. If the remote server supports the introspection API, the proxy can also be used to query the remote server for the methods it supports (service discovery) and fetch other server-associated metadata.

`ServerProxy` instance methods take Python basic types and objects as arguments and return Python basic types and classes. Types that are conformable (e.g. that can be marshalled through XML), include the following (and except where noted, they are unmarshalled as the same Python type):
### Data Types Supported by XML-RPC

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>The True and False constants</td>
</tr>
<tr>
<td>integers</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>floating-point numbers</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>strings</td>
<td>Pass in directly</td>
</tr>
<tr>
<td>arrays</td>
<td>Any Python sequence type containing conformable elements. Arrays are returned as lists</td>
</tr>
<tr>
<td>structures</td>
<td>A Python dictionary. Keys must be strings, values may be any conformable type. Objects of user-defined classes can be passed in; only their <strong>dict</strong> attribute is transmitted.</td>
</tr>
<tr>
<td>dates</td>
<td>in seconds since the epoch (pass in an instance of the DateTime class) or a datetime.datetime instance.</td>
</tr>
<tr>
<td>binary data</td>
<td>pass in an instance of the Binary wrapper class</td>
</tr>
</tbody>
</table>

This is the full set of data types supported by XML-RPC. Method calls may also raise a special Fault instance, used to signal XML-RPC server errors, or ProtocolError used to signal an error in the HTTP/HTTPS transport layer. Both Fault and ProtocolError derive from a base class called Error. Note that even though starting with Python 2.2 you can subclass built-in types, the xmlrpclib module currently does not marshal instances of such subclasses.

When passing strings, characters special to XML such as <, >, and & will be automatically escaped. However, it’s the caller’s responsibility to ensure that the string is free of characters that aren’t allowed in XML, such as the control characters with ASCII values between 0 and 31 (except, of course, tab, newline and carriage return); failing to do this will result in an XML-RPC request that isn’t well-formed XML. If you have to pass arbitrary strings via XML-RPC, use the Binary wrapper class described below.

Server is retained as an alias for ServerProxy for backwards compatibility. New code should use ServerProxy.

**Changed in version 2.5:** The use_datetime flag was added.

**Changed in version 2.6:** Instances of new-style classes can be passed in if they have an __dict__ attribute and don’t have a base class that is marshalled in a special way.

#### See also:

**XML-RPC HOWTO**

A good description of XML-RPC operation and client software in several languages. Contains pretty much everything an XML-RPC client developer needs to know.
20.23.1. ServerProxy Objects

A `ServerProxy` instance has a method corresponding to each remote procedure call accepted by the XML-RPC server. Calling the method performs an RPC, dispatched by both name and argument signature (e.g. the same method name can be overloaded with multiple argument signatures). The RPC finishes by returning a value, which may be either returned data in a conformant type or a `Fault` or `ProtocolError` object indicating an error.

Servers that support the XML introspection API support some common methods grouped under the reserved `system` member:

- **ServerProxy.system.listMethods()**

  This method returns a list of strings, one for each (non-system) method supported by the XML-RPC server.

- **ServerProxy.system.methodSignature(name)**

  This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns an array of possible signatures for this method. A signature is an array of types. The first of these types is the return type of the method, the rest are parameters.

  Because multiple signatures (ie. overloading) is permitted, this method returns a list of signatures rather than a singleton.

  Signatures themselves are restricted to the top level parameters expected by a method. For instance if a method expects one array of structs as a parameter, and it returns a string, its signature is simply “string, array”. If it expects three integers and returns a string, its signature is “string, int, int, int”.

  If no signature is defined for the method, a non-array value is returned. In Python this means that the type of the returned value will be something other than list.

- **ServerProxy.system.methodHelp(name)**

  This method takes one parameter, the name of a method implemented by the XML-RPC server. It returns a documentation string describing the use of that method. If no such string is available, an empty string is returned. The documentation string may contain HTML markup.
20.23.2. Boolean Objects

This class may be initialized from any Python value; the instance returned depends only on its truth value. It supports various Python operators through __cmp__(), __repr__(), __int__() and __nonzero__() methods, all implemented in the obvious ways.

It also has the following method, supported mainly for internal use by the unmarshalling code:

Boolean. encode(out)
Write the XML-RPC encoding of this Boolean item to the out stream object.

A working example follows. The server code:

```python
import xmlrpclib
from SimpleXMLRPCServer import SimpleXMLRPCServer

def is_even(n):
    return n%2 == 0

server = SimpleXMLRPCServer(('localhost', 8000))
print "Listening on port 8000..."
server.register_function(is_even, "is_even")
server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
print "3 is even: %s" % str(proxy.is_even(3))
print "100 is even: %s" % str(proxy.is_even(100))
```

20.23.3. DateTime Objects

This class may be initialized with seconds since the epoch, a time tuple, an ISO 8601 time/date string, or a datetime.datetime instance. It has the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

DateTime. decode(string)
Accept a string as the instance’s new time value.

DateTime. encode(out)
Write the XML-RPC encoding of this DateTime item to the out stream object.

It also supports certain of Python’s built-in operators through __cmp__() and __repr__() methods.

A working example follows. The server code:
import datetime
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def today():
    today = datetime.datetime.today()
    return xmlrpclib.DateTime(today)

server = SimpleXMLRPCServer(('localhost', 8000))
print "Listening on port 8000..."
server.register_function(today, "today")
server.serve_forever()

The client code for the preceding server:

import xmlrpclib
import datetime

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")

today = proxy.today()
# convert the ISO8601 string to a datetime object
converted = datetime.datetime.strptime(today.value, "%Y%m%dT%H:%M:%S")
print "Today: %s" % converted.strftime("%d.%m.%Y, %H:%M")

20.23.4. Binary Objects

This class may be initialized from string data (which may include NULs). The primary access to the content of a Binary object is provided by an attribute:

Binary.data
The binary data encapsulated by the Binary instance. The data is provided as an 8-bit string.

Binary objects have the following methods, supported mainly for internal use by the marshalling/unmarshalling code:

Binary.decode(string)
Accept a base64 string and decode it as the instance's new data.

Binary.encode(out)
Write the XML-RPC base 64 encoding of this binary item to the out stream object.

The encoded data will have newlines every 76 characters as per RFC 2045 section 6.8, which was the de facto standard base64 specification when the XML-RPC spec was written.

It also supports certain of Python's built-in operators through a __cmp__() method.

Example usage of the binary objects. We're going to transfer an image over
from SimpleXMLRPCServer import SimpleXMLRPCServer
import xmlrpclib

def python_logo():
    with open("python_logo.jpg", "rb") as handle:
        return xmlrpclib.Binary(handle.read())

server = SimpleXMLRPCServer(("localhost", 8000))
print "Listening on port 8000..."
server.register_function(python_logo, 'python_logo')
s
server.serve_forever()

The client gets the image and saves it to a file:

import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
with open("fetched_python_logo.jpg", "wb") as handle:
    handle.write(proxy.python_logo().data)

20.23.5. Fault Objects

A Fault object encapsulates the content of an XML-RPC fault tag. Fault objects have the following members:

Fault.faultCode
    A string indicating the fault type.

Fault.faultString
    A string containing a diagnostic message associated with the fault.

In the following example we’re going to intentionally cause a Fault by returning a complex type object. The server code:

from SimpleXMLRPCServer import SimpleXMLRPCServer

# A marshalling error is going to occur because we're returning a
# complex number
def add(x,y):
    return x+y+0j

server = SimpleXMLRPCServer("localhost", 8000))
print "Listening on port 8000..."
s
server.register_function(add, 'add')
s
server.serve_forever()

The client code for the preceding server:
```python
import xmlrpclib

proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
try:
    proxy.add(2, 5)
except xmlrpclib.Fault, err:
    print "A fault occurred"
    print "Fault code: %d" % err.faultCode
    print "Fault string: %s" % err.faultString
```

### 20.23.6. ProtocolError Objects

A **ProtocolError** object describes a protocol error in the underlying transport layer (such as a 404 ‘not found’ error if the server named by the URI does not exist). It has the following members:

- **ProtocolError.url**
  - The URI or URL that triggered the error.

- **ProtocolError.errcode**
  - The error code.

- **ProtocolError.errmsg**
  - The error message or diagnostic string.

- **ProtocolError.headers**
  - A string containing the headers of the HTTP/HTTPS request that triggered the error.

In the following example we’re going to intentionally cause a **ProtocolError** by providing an URI that doesn’t point to an XMLRPC server:

```python
import xmlrpclib

# create a ServerProxy with an URI that doesn't respond to XMLRPC requests
proxy = xmlrpclib.ServerProxy("http://www.google.com/")
try:
    proxy.some_method()
except xmlrpclib.ProtocolError, err:
    print "A protocol error occurred"
    print "URL: %s" % err.url
    print "HTTP/HTTPS headers: %s" % err.headers
    print "Error code: %d" % err.errcode
    print "Error message: %s" % err.errmsg
```

### 20.23.7. MultiCall Objects

*New in version 2.4.*
In http://www.xmlrpc.com/discuss/msgReader%241208, an approach is presented to encapsulate multiple calls to a remote server into a single request.

```python
class xmlrpclib.MultiCall(server)
    Create an object used to boxcar method calls. server is the eventual target of the call. Calls can be made to the result object, but they will immediately return None, and only store the call name and parameters in the MultiCall object. Calling the object itself causes all stored calls to be transmitted as a single system.multicall request. The result of this call is a generator; iterating over this generator yields the individual results.
```

A usage example of this class follows. The server code

```python
from SimpleXMLRPCServer import SimpleXMLRPCServer
def add(x, y):
    return x+y
def subtract(x, y):
    return x-y
def multiply(x, y):
    return x*y
def divide(x, y):
    return x/y

# A simple server with simple arithmetic functions
server = SimpleXMLRPCServer(('localhost', 8000))
print "Listening on port 8000..."
server.register_multicall_functions()
server.register_function(add, 'add')
server.register_function(subtract, 'subtract')
server.register_function(multiply, 'multiply')
server.register_function(divide, 'divide')
server.serve_forever()
```

The client code for the preceding server:

```python
import xmlrpclib
proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
m multicall = xmlrpclib.MultiCall(proxy)
m multicall.add(7,3)
m multicall.subtract(7,3)
m multicall.multiply(7,3)
m multicall.divide(7,3)
result = multicall()

print "7+3=%d, 7-3=%d, 7*3=%d, 7/3=%d" % tuple(result)
```

20.23.8. Convenience Functions
xmlrpclib.boolean(value)
Convert any Python value to one of the XML-RPC Boolean constants, True or False.

xmlrpclib.dumps(params[, methodname[, methodresponse[, encoding[, allow_none]]]])
Convert params into an XML-RPC request or into a response if methodresponse is true. params can be either a tuple of arguments or an instance of the Fault exception class. If methodresponse is true, only a single value can be returned, meaning that params must be of length 1. encoding, if supplied, is the encoding to use in the generated XML; the default is UTF-8. Python's None value cannot be used in standard XML-RPC; to allow using it via an extension, provide a true value for allow_none.

xmlrpclib.loads(data[, use_datetime])
Convert an XML-RPC request or response into Python objects, a (params, methodname). params is a tuple of argument; methodname is a string, or None if no method name is present in the packet. If the XML-RPC packet represents a fault condition, this function will raise a Fault exception. The use_datetime flag can be used to cause date/time values to be presented as datetime.datetime objects; this is false by default.

Changed in version 2.5: The use_datetime flag was added.

20.23.9. Example of Client Usage

```python
# simple test program (from the XML-RPC specification)
from xmlrpclib import ServerProxy, Error

# server = ServerProxy("http://localhost:8000")  # local server
server = ServerProxy("http://betty.userland.com")

print server

try:
    print server.examples.getStateName(41)
except Error, v:
    print "ERROR", v
```

To access an XML-RPC server through a proxy, you need to define a custom transport. The following example shows how:

```python
import xmlrpclib, httplib

class ProxiedTransport(xmlrpclib.Transport):
    def set_proxy(self, proxy):
        self.proxy = proxy
    def make_connection(self, host):
        self.realhost = host
        h = httplib.HTTP(self.proxy)
```

9 of 10 04/30/2012 04:49 PM
def send_request(self, connection, handler, request_body):
    connection.putrequest("POST", 'http://%s%s' % (self.realhost, l

def send_host(self, connection, host):
    connection.putheader('Host', self.realhost)

p = ProxiedTransport()
p.set_proxy('proxy-server:8080')
print server.currentTime.getCurrentTime()

20.23.10. Example of Client and Server Usage

See SimpleXMLRPCServer Example.