xmlrpc.client — Client Library for XML-RPC

Purpose: Client-side library for XML-RPC communication.

XML-RPC is a lightweight remote procedure call protocol built on top of HTTP and XML. The `xmlrpclib` module lets a Python program communicate with an XML-RPC server written in any language.

All of the examples in this section use the server defined in `xmlrpc_server.py`, available in the source distribution and included here for reference.

```
# xmlrpc_server.py

from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.client import Binary
import datetime


class ExampleService:

    def ping(self):
        """Simple function to respond when called
        to demonstrate connectivity.
        """
        return True

    def now(self):
        """Returns the server current date and time."""
        return datetime.datetime.now()

    def show_type(self, arg):
        """Illustrates how types are passed in and out of
        server methods.

        Accepts one argument of any type.

        Returns a tuple with string representation of the value,
        the name of the type, and the value itself.

        """
        return (str(arg), str(type(arg)), arg)

    def raises_exception(self, msg):
        "Always raises a RuntimeError with the message passed in"
        raise RuntimeError(msg)

    def send_back_binary(self, bin):
        """Accepts single Binary argument, and unpacks and
        repacks it to return it."""
        data = bin.data
        print('send_back_binary({!r})'.format(data))
        response = Binary(data)
        return response


if __name__ == '__main__':
    server = SimpleXMLRPCServer(('localhost', 9000),
                                logRequests=True,
                                allow_none=True)
    server.register_introspection_functions()
    server.register_multicall_functions()

    server.register_instance(ExampleService())

    try:
        print('Use Control-C to exit')
        print('Use Control-C to exit')
        server.serve_forever()
    except KeyboardInterrupt:
        print('Exiting')
```

Connecting to a Server

The simplest way to connect a client to a server is to instantiate a ServerProxy object, giving it the URI of the server. For example, the demo server runs on port 9000 of localhost.

```
# xmlrpc_ServerProxy.py

import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000')
print('Ping:', server.ping())
```

In this case, the ping() method of the service takes no arguments and returns a single Boolean value.

```
$ python3 xmlrpc_ServerProxy.py

Ping: True
```

Other options are available to support alternate transport. Both HTTP and HTTPS are supported out of the box, both with basic authentication. To implement a new communication channel, only a new transport class is needed. It could be an interesting exercise, for example, to implement XML-RPC over SMTP.

```
# xmlrpc_ServerProxy_verbose.py

import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   verbose=True)
print('Ping:', server.ping())
```

The verbose option gives debugging information useful for resolving communication errors.

```
$ python3 xmlrpc_ServerProxy_verbose.py

send: b'POST /RPC2 HTTP/1.1\r\nHost: localhost:9000\r\n
Accept-Encoding: gzip\r\nContent-Type: text/xml\r\n
User-Agent: Python-xmlrpc/3.5\r\nContent-Length: 98\r\n\r\n'
send: b"<?xml version='1.0'?>\n<methodCall>\n<methodName>
ping</methodName>\n<params>\n</params>\n</methodCall>\n"
reply: 'HTTP/1.0 200 OK\r\n'
header: Server header: Date header: Content-type header:
Content-length body: b"<?xml version='1.0'?>\n<methodResponse>\n
<params>\n<param>\n<value><boolean>1</boolean></value>\n</param>
\n</params>\n</methodResponse>\n"
Ping: True
```

The default encoding can be changed from UTF-8 if an alternate system is needed.

```
# xmlrpc_ServerProxy_encoding.py

import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   encoding='ISO-8859-1')
print('Ping:', server.ping())
```

The server automatically detects the correct encoding.

```
$ python3 xmlrpc_ServerProxy_encoding.py

Ping: True
```

The allow_none option controls whether Python's None value is automatically translated to a nil value or if it causes an error.

```
# xmlrpc_ServerProxy_allow_none.py

import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   allow_none=False)
try:
    server.show_type(None)
except TypeError as err:
    print('ERROR:', err)

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   allow_none=True)
print('Allowed:', server.show_type(None))
```

The error is raised locally if the client does not allow None, but can also be raised from within the server if it is not configured to allow None.

```
$ python3 xmlrpc_ServerProxy_allow_none.py

ERROR: cannot marshal None unless allow_none is enabled
Allowed: ['None', "<class 'NoneType'>", None]
```

Data Types

The XML-RPC protocol recognizes a limited set of common data types. The types can be passed as arguments or return values and combined to create more complex data structures.

```
# xmlrpc_types.py

import xmlrpc.client
import datetime

server = xmlrpc.client.ServerProxy('http://localhost:9000')

data = [
    ('boolean', True),
    ('integer', 1),
    ('float', 2.5),
    ('string', 'some text'),
    ('datetime', datetime.datetime.now()),
    ('array', ['a', 'list']),
    ('array', ('a', 'tuple')),
    ('structure', {'a': 'dictionary'}),
]

for t, v in data:
    as_string, type_name, value = server.show_type(v)
    print('{:<12}: {}'.format(t, as_string))
    print('{:12}  {}'.format('', type_name))
    print('{:12}  {}'.format('', value))
```

The simple types are

```
$ python3 xmlrpc_types.py

boolean     : True
              <class 'bool'>
              True
integer     : 1
              <class 'int'>
              1
float       : 2.5
              <class 'float'>
              2.5
string      : some text
              <class 'str'>
```

```
                some text
datetime      : 20160618T19:31:47
                <class 'xmlrpc.client.DateTime'>
                20160618T19:31:47
array         : ['a', 'list']
                <class 'list'>
                ['a', 'list']
array         : ['a', 'tuple']
                <class 'list'>
                ['a', 'tuple']
structure     : {'a': 'dictionary'}
                <class 'dict'>
                {'a': 'dictionary'}
```

The supported types can be nested to create values of arbitrary complexity.

```
# xmlrpc_types_nested.py

import xmlrpc.client
import datetime
import pprint

server = xmlrpc.client.ServerProxy('http://localhost:9000')

data = {
    'boolean': True,
    'integer': 1,
    'floating-point number': 2.5,
    'string': 'some text',
    'datetime': datetime.datetime.now(),
    'array1': ['a', 'list'],
    'array2': ('a', 'tuple'),
    'structure': {'a': 'dictionary'},
}
arg = []
for i in range(3):
    d = {}
    d.update(data)
    d['integer'] = i
    arg.append(d)

print('Before:')
pprint.pprint(arg, width=40)

print('\nAfter:')
pprint.pprint(server.show_type(arg)[-1], width=40)
```

This program passes a list of dictionaries containing all of the supported types to the sample server, which returns the data. Tuples are converted to lists and `datetime` instances are converted to `DateTime` objects, but otherwise the data is unchanged.

```
$ python3 xmlrpc_types_nested.py

Before:
[{'array': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2016, 6, 18, 19, 27, 30, 45333),
  'floating-point number': 2.5,
  'integer': 0,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2016, 6, 18, 19, 27, 30, 45333),
  'floating-point number': 2.5,
  'integer': 1,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array': ('a', 'tuple'),
  'boolean': True,
  'datetime': datetime.datetime(2016, 6, 18, 19, 27, 30, 45333),
  'floating-point number': 2.5,
  'integer': 2,
  'string': 'some text',
  'structure': {'a': 'dictionary'}}]

After:
[{'array': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20160618T19:27:30' at 0x101ecfac8>,
  'floating-point number': 2.5,
  'integer': 0,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20160618T19:27:30' at 0x101ecfcc0>,
  'floating-point number': 2.5,
  'integer': 1,
  'string': 'some text',
  'structure': {'a': 'dictionary'}},
 {'array': ['a', 'tuple'],
  'boolean': True,
  'datetime': <DateTime '20160618T19:27:30' at 0x101ecfe10>,
  'floating-point number': 2.5,
  'integer': 2,
  'string': 'some text',
  'structure': {'a': 'dictionary'}}]
```

XML-RPC supports dates as a native type, and xmlrpclib can use one of two classes to represent the date values in the outgoing proxy or when they are received from the server.

```
# xmlrpc_ServerProxy_use_datetime.py

import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   use_datetime=True)
now = server.now()
print('With:', now, type(now), now.__class__.__name__)

server = xmlrpc.client.ServerProxy('http://localhost:9000',
                                   use_datetime=False)
now = server.now()
print('Without:', now, type(now), now.__class__.__name__)
```

By default an internal version of DateTime is used, but the use_datetime option turns on support for using the classes in the datetime module.

```
$ python3 source/xmlrpc.client/xmlrpc_ServerProxy_use_datetime.py

With: 2016-06-18 19:18:31 <class 'datetime.datetime'> datetime
Without: 20160618T19:18:31 <class 'xmlrpc.client.DateTime'> DateTime
```

Passing Objects

Instances of Python classes are treated as structures and passed as a dictionary, with the attributes of the object as values in the dictionary.

```
# xmlrpc_types_object.py

import xmlrpc.client
import pprint


class MyObj:

    def __init__(self, a, b):
        self.a = a
        self.b = b
    def __repr__(self):
        return 'MyObj({!r}, {!r})'.format(self.a, self.b)


server = xmlrpc.client.ServerProxy('http://localhost:9000')

o = MyObj(1, 'b goes here')
print('o  :', o)
pprint.pprint(server.show_type(o))

o2 = MyObj(2, o)
print('\no2 :', o2)
pprint.pprint(server.show_type(o2))
```

When the value is sent back to the client from the server the result is a dictionary on the client, since there is nothing encoded in the values to tell the server (or client) that it should be instantiated as part of a class.

```
$ python3 xmlrpc_types_object.py

o  : MyObj(1, 'b goes here')
["{'b': 'b goes here', 'a': 1}", "<class 'dict'>",
{'a': 1, 'b': 'b goes here'}]

o2 : MyObj(2, MyObj(1, 'b goes here'))
["{'b': {'b': 'b goes here', 'a': 1}, 'a': 2}",
 "<class 'dict'>",
 {'a': 2, 'b': {'a': 1, 'b': 'b goes here'}}]
```

Binary Data

All values passed to the server are encoded and escaped automatically. However, some data types may contain characters that are not valid XML. For example, binary image data may include byte values in the ASCII control range 0 to 31. To pass binary data, it is best to use the Binary class to encode it for transport.

```
# xmlrpc_Binary.py

import xmlrpc.client
import xml.parsers.expat

server = xmlrpc.client.ServerProxy('http://localhost:9000')

s = b'This is a string with control characters\x00'
print('Local string:', s)

data = xmlrpc.client.Binary(s)
response = server.send_back_binary(data)
print('As binary:', response.data)

try:
    print('As string:', server.show_type(s))
except xml.parsers.expat.ExpatError as err:
    print('\nERROR:', err)
```

If the string containing a NULL byte is passed to show_type(), an exception is raised in the XML parser as it processes the response.

```
$ python3 xmlrpc_Binary.py

Local string: b'This is a string with control characters\x00'
As binary: b'This is a string with control characters\x00'

ERROR: not well-formed (invalid token): line 6, column 55
```

Binary objects can also be used to send objects using pickle. The normal security issues related to sending what amounts to executable code over the wire apply here (i.e., do not do this unless the communication channel is secure).

```
import xmlrpc.client
import pickle
import pprint


class MyObj:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return 'MyObj({!r}, {!r})'.format(self.a, self.b)


server = xmlrpc.client.ServerProxy('http://localhost:9000')

o = MyObj(1, 'b goes here')
print('Local:', id(o))
print(o)

print('\nAs object:')
pprint.pprint(server.show_type(o))

p = pickle.dumps(o)
b = xmlrpc.client.Binary(p)
r = server.send_back_binary(b)

o2 = pickle.loads(r.data)
print('\nFrom pickle:', id(o2))
pprint.pprint(o2)
```

The data attribute of the `Binary` instance contains the pickled version of the object, so it has to be unpickled before it can be used. That results in a different object (with a new id value).

```
$ python3 xmlrpc_Binary_pickle.py

Local: 4327262304
MyObj(1, 'b goes here')

As object:
["{'a': 1, 'b': 'b goes here'}", "<class 'dict'>",
{'a': 1, 'b': 'b goes here'}]

From pickle: 4327262472
MyObj(1, 'b goes here')
```

Exception Handling

Since the XML-RPC server might be written in any language, exception classes cannot be transmitted directly. Instead, exceptions raised in the server are converted to `Fault` objects and raised as exceptions locally in the client.

```
# xmlrpc_exception.py

import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000')
try:
    server.raises_exception('A message')
except Exception as err:
    print('Fault code:', err.faultCode)
    print('Message   :', err.faultString)
```

The original error message is saved in the `faultString` attribute, and `faultCode` is set to an XML-RPC error number.

```
$ python3 xmlrpc_exception.py

Fault code: 1
Message   : <class 'RuntimeError'>:A message
```

Combining Calls Into One Message

Multicall is an extension to the XML-RPC protocol that allows more than one call to be sent at the same time, with the responses collected and returned to the caller.

```
# xmlrpc_MultiCall.py

import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000')

multicall = xmlrpc.client.MultiCall(server)
multicall.ping()
multicall.show_type(1)
multicall.show_type('string')

for i, r in enumerate(multicall()):
    print(i, r)
```

To use a `MultiCall` instance, invoke the methods on it as with a `ServerProxy`, then call the object with no arguments to actually run the remote functions. The return value is an iterator that yields the results from all of the calls.

```
$ python3 xmlrpc_MultiCall.py

0 True
1 ['1', "<class 'int'>", 1]
2 ['string', "<class 'str'>", 'string']
```

If one of the calls causes a `Fault`, the exception is raised when the result is produced from the iterator and no more results are available.

```
# xmlrpc_MultiCall_exception.py

import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://localhost:9000')

multicall = xmlrpc.client.MultiCall(server)
multicall.ping()
multicall.show_type(1)
multicall.raises_exception('Next to last call stops execution')
multicall.show_type('string')

try:
    for i, r in enumerate(multicall()):
        print(i, r)
except xmlrpc.client.Fault as err:
    print('ERROR:', err)
```

Since the third response, from `raises_exception()`, generates an exception, the response from `show_type()` is not accessible.

```
$ python3 xmlrpc_MultiCall_exception.py

0 True
1 ['1', "<class 'int'>", 1]
ERROR: <Fault 1: "<class 'RuntimeError'>:Next to last call stops execution">
```

See also

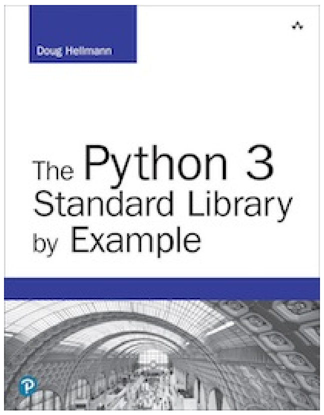
- Standard library documentation for xmlrpc.client
- `xmlrpc.server` – An XML-RPC server implementation.
- `http.server` – An HTTP server implementation.
- XML-RPC How To – Describes how to use XML-RPC to implement clients and servers in a variety of languages.

Quick Links

This page was last updated 2016-12-29.

Navigation

json — JavaScript Object Notation
xmlrpc.server — An XML-RPC server



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?