1 Contents

1.1 Install MSL-LoadLib .................................................. 3
1.2 Load a library ......................................................... 6
1.3 Access a 32-bit library in 64-bit Python ................................. 12
1.4 MSL-LoadLib API Documentation ..................................... 28
1.5 Source code for the example libraries ................................. 67
1.6 Re-freezing the 32-bit server ......................................... 86
1.7 License ................................................................. 87
1.8 Developers .......................................................... 87
1.9 Changelog ........................................................... 88

2 Index

Python Module Index .................................................. 95

Index ................................................................. 97
This package is used to load a shared library in Python. It is basically just a thin wrapper around ctypes (for libraries that use the __cdecl or __stdcall calling convention), Python for .NET (for libraries that use Microsoft’s .NET Framework, CLR), Py4J (for Java .jar or .class files) and comtypes (for libraries that use the Component Object Model).

However, the primary advantage is that it is possible to communicate with a 32-bit shared library in 64-bit Python. For various reasons, mainly to do with the differences in pointer sizes, it is not possible to load a 32-bit shared library (e.g., .dll, .so, .dylib files) into a 64-bit process, and vice versa. This package contains a Server32 class that hosts a 32-bit library and a Client64 class that sends a request to the server to communicate with the 32-bit library as a form of inter-process communication.
1.1 Install MSL-LoadLib

To install MSL-LoadLib run:

```
pip install msl-loadlib
```

Alternatively, using the MSL Package Manager run:

```
msl install loadlib
```

1.1.1 Dependencies

- Python 2.7, 3.5+

Optional dependencies:
- Python for .NET
- Py4J
- comtypes

You can install MSL-LoadLib and Python for .NET using:

```
pip install msl-loadlib[clr]
```

MSL-LoadLib and Py4J:

```
pip install msl-loadlib[java]
```

MSL-LoadLib and comtypes:

```
pip install msl-loadlib[com]
```
or MSL-LoadLib and all optional dependencies:

```
pip install msl-loadlib[all]
```

## 1.1.2 Compatibility

- The `start_server32` module has been built into a frozen Python application for Windows and Linux and works with the Python versions listed above. The 32-bit server is running on Python 3.7 and therefore all modules that run on the server must use Python 3 syntax.
- You can create a new 32-bit server. See *Re-freezing the 32-bit server* for more details.

## 1.1.3 Prerequisites

### Windows

64-bit Windows already comes with WoW64 to run 32-bit software and therefore no prerequisites are required to load `__cdecl` or `__stdcall` libraries. However, the library might have its own dependencies, such as a particular Visual C++ Redistributable, that may need to be installed.

If you need to load a Microsoft .NET library then you must install Python for .NET

```
pip install pythonnet
```

If you need to load a Java library, a `.jar` or `.class` file, then you must install Py4J

```
pip install py4j
```

and a [Java Runtime Environment](https://www.oracle.com/java/technologies/javase-jre8-downloads.html) and ensure that the `java` executable is available on your PATH. For example, the following should return the version of Java that is installed

```
C:\>java --version
java 11.0.2 2019-01-15 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.2+9-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.2+9-LTS, mixed mode)
```

If you need to load a [Component Object Model](https://docs.microsoft.com/en-us/dotnet/framework_class/library/comtypes) library then you must install `comtypes`

```
pip install comtypes
```

When loading a shared library it is vital that all dependencies of the library are also available on your computer and that the directory of the dependency is also available on your PATH. A helpful utility to use to determine the dependencies of a shared library is [Dependencies](https://www.dependencywalker.com) (which is a modern [Dependency Walker](https://www.dependencywalker.com)). For finding the dependencies of a .NET library the [Dependency Walker for .NET](https://www.dependencywalker.com) is also useful.

### Linux

Before using **MSL-LoadLib** on Linux the following packages are required.

Install the packages that are needed to run a 32-bit binary on 64-bit Linux and to load C/C++ and FORTRAN libraries
The following ensures that the `netstat` command is available

```bash
sudo apt install net-tools
```

If you need to load a Microsoft .NET library then you must install Mono *(NOTE: v5.20 was used when embedding pythonnet in the 32-bit server for Linux)*

```bash
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys 3FA7E0328081BFF6A14DA29AA6A19B38D3D831EF
sudo apt install apt-transport-https ca-certificates
echo "deb https://download.mono-project.com/repo/ubuntu stable-bionic/" | sudo tee /etc/apt/sources.list.d/mono-official-stable.list
sudo apt update
sudo apt install mono-complete
```

the prerequisites to build Python for .NET from source

```bash
sudo apt install libglib2.0-dev clang python3-pip python3-dev
```

and Python for .NET

```bash
pip3 install pycparser
```

```
and Python for .NET
```

```
```

Installing Mono v5.20.1.34 and Python for .NET v2.4.0 on Ubuntu 18.04.2 has been confirmed to work

```
```

If you run in to problems installing Python for .NET then the best place to find help is on the issues page of Python for .NET’s repository.

If you need to load a Java library, a .jar or .class file, then you must install Py4J

### 1.1. Install MSL-LoadLib
pip3 install py4j

and a Java Runtime Environment

sudo apt install default-jre

and ensure that the java executable is available on your PATH. For example, the following should return the version of Java that is installed

```
joe@msl:$ java --version
openjdk 11.0.3 2019-04-16
OpenJDK Runtime Environment (build 11.0.3+7-Ubuntu-1ubuntu218.04.1)
OpenJDK 64-Bit Server VM (build 11.0.3+7-Ubuntu-1ubuntu218.04.1, mixed
˓→mode, sharing)
```

OSX

The 32-bit server has not been created for OSX nor have the example libraries been compiled in OSX.

1.2 Load a library

If you are loading a 32-bit library in 32-bit Python, or a 64-bit library in 64-bit Python, then you can directly load the library using `LoadLibrary`.

**Important:** If you want to load a 32-bit library in 64-bit Python then inter-process communication is used to communicate with the 32-bit library. See the examples for more details.

All of the shared libraries in the following examples are included with the MSL-LoadLib package. The C++ and FORTRAN libraries have been compiled in 32- and 64-bit Windows and Linux, using g++ and gfortran respectively. The .NET library was compiled to 32 and 64 bit using Microsoft Visual Studio. The kernel32 library is a 32-bit library and it is only valid on Windows, since it uses the __stdcall calling convention. The LabVIEW library was built using 32- and 64-bit LabVIEW on Windows. The Java libraries are platform and bitness independent since they run in the JVM.

The first step is to import the `LoadLibrary` class

```
>>> from msl.loadlib import LoadLibrary
```

and the directory where the example libraries are located

```
>>> from msl.examples.loadlib import EXAMPLES_DIR
```

**Tip:** If the file extension is not specified then a default extension, .dll (Windows) or .so (Linux), is used.
1.2.1 C++

Load a 64-bit C++ library in 64-bit Python, see [here](#) for the source code. To load the 32-bit version in 32-bit Python use 'cpp_lib32'.

```python
>>> cpp = LoadLibrary(EXAMPLES_DIR + '/cpp_lib64')
>>> cpp
<LoadLibrary libtype=CDLL path=D:\msl\examples\loadlib\cpp_lib64.dll>
```  

Call the `add` function that calculates the sum of two integers

```python
>>> cpp.lib.add(1, 2)
3
```

1.2.2 FORTRAN

Load a 64-bit FORTRAN library in 64-bit Python, see [here](#) for the source code. To load the 32-bit version in 32-bit Python use 'fortran_lib32'.

```python
>>> fortran = LoadLibrary(EXAMPLES_DIR + '/fortran_lib64')
>>> fortran
<LoadLibrary libtype=CDLL path=D:\msl\examples\loadlib\fortran_lib64.dll>
```  

Call the `factorial` function. With a FORTRAN library you must pass values by reference using `ctypes`, and, since the returned value is not of type `ctypes.c_int` we must configure `ctypes` for a value of type `ctypes.c_double` to be returned

```python
>>> from ctypes import byref, c_int, c_double
>>> fortran.lib.factorial.restype = c_double
>>> fortran.lib.factorial(byref(c_int(37)))
1.3763753091226343e+43
```

1.2.3 Microsoft .NET Framework

Load a 64-bit C# library (a .NET Framework) in 64-bit Python, see [here](#) for the source code. Include the 'net' argument to indicate that the .dll file is for the .NET Framework ('clr' is an alias for 'net' and can also be passed in as an argument). To load the 32-bit version in 32-bit Python use 'dotnet_lib32.dll'.

```python
>>> net = LoadLibrary(EXAMPLES_DIR + '/dotnet_lib64.dll', 'net')
>>> net
<LoadLibrary libtype=DotNet path=D:\msl\examples\loadlib\dotnet_lib64.dll>
```
The `dotnet_lib64` library contains a reference to the DotNetMSL module (which is a C# namespace), an instance of the StringManipulation class and a reference to the StaticClass class.

```python
>>> for item in dir(net.lib):
...     if not item.startswith('_'):
...         print(item, type(getattr(net.lib, item)))
...
DotNetMSL <class 'CLR.ModuleObject'>
StaticClass <class 'System.RuntimeType'>
StringManipulation <class '.StringManipulation'>
```

Create an instance of the BasicMath class in the DotNetMSL namespace and call the `multiply_doubles` method.

```python
>>> bm = net.lib.DotNetMSL.BasicMath()
>>> bm.multiply_doubles(2.3, 5.6)
12.879999999999999
```

Create an instance of the ArrayManipulation class in the DotNetMSL namespace and call the `scalar_multiply` method.

```python
>>> am = net.lib.DotNetMSL.ArrayManipulation()
>>> values = am.scalar_multiply(2., [1., 2., 3., 4., 5.])
>>> [val for val in values]
[2.0, 4.0, 6.0, 8.0, 10.0]
```

Use the `reverse_string` method in the StringManipulation class to reverse a string.

```python
>>> net.lib.StringManipulation.reverse_string('abcdefghijklmnopqrstuvwxyz')
'zyxwvutsrqponmlkjihgfedcba'
```

View the static methods in the StaticClass class.

```python
>>> for method in net.lib.StaticClass.GetMethods():
...     print(method)
...
Int32 add_multiple(Int32, Int32, Int32, Int32, Int32)
System.String ToString()
Boolean Equals(System.Object)
Int32 GetHashCode()
System.Type GetType()
```

Use the static `add_multiple` method in the StaticClass class to add five integers.

```python
>>> net.lib.StaticClass.GetMethod('add_multiple').Invoke(None, [1, 2, 3, 4, 5])
15
```

### 1.2.4 Windows __stdcall

Load a 32-bit Windows __stdcall library in 32-bit Python, see kernel32.dll. Include the 'windll' argument to specify that the calling convention is __stdcall.
See [here](#) for how to communicate with `kernel32.dll` from 64-bit Python.

### 1.2.5 LabVIEW

Load a 64-bit LabVIEW library in 64-bit Python, see [here](#) for the source code. To load the 32-bit version in 32-bit Python use `labview_lib32.dll`. Also, an appropriate LabVIEW Run-Time Engine must be installed. The LabVIEW example is only valid on Windows.

**Note**: A LabVIEW library can be built into a DLL using the `cdecl` or `stdcall` calling convention. Make sure that you specify the appropriate `libtype` when instantiating the `LoadLibrary` class.

```python
>>> labview = LoadLibrary(EXAMPLES_DIR + '/labview_lib64.dll')
>>> labview
<LoadLibrary libtype=CDLL path=D:\msl\examples\loadlib\labview_lib64.dll>
```

Create some data to calculate the mean, variance and standard deviation of

```python
>>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Convert `data` to a `ctypes` array and allocate memory for the returned values

```python
>>> from ctypes import c_double, byref
>>> x = (c_double * len(data))(*data)
>>> mean, variance, std = c_double(), c_double(), c_double()
```

Calculate the sample standard deviation (i.e., the third argument is set to 0) and variance

```python
>>> ret = labview.lib.stdev(x, len(data), 0, byref(mean), byref(variance),
                        byref(std))
>>> mean.value
5.0
>>> variance.value
7.5
>>> std.value
2.7386127875258306
```
Calculate the population standard deviation (i.e., the third argument is set to 1) and variance

```python
>>> ret = labview.lib.stdev(x, len(data), 1, byref(mean), byref(variance),
    → byref(std))
>>> mean.value
5.0
>>> variance.value
6.666666666666667
>>> std.value
2.581988897471611
```

1.2.6 Java

Since Java byte code is executed in the JVM it doesn’t matter whether it was built with a 32-bit or 64-bit Java Development Kit. The Python interpreter does not load the Java byte code but communicates with the JVM through a local network socket that is created by Py4J.

Load a Java archive, a .jar file, in a JVM, see here for the source code.

```python
>>> jar = LoadLibrary(EXAMPLES_DIR + '/java_lib.jar')
>>> jar
<LoadLibrary libtype=JVMView path=D:\msl\examples\loadlib\java_lib.jar>
>>> jar.gateway
<py4j.java_gateway.JavaGateway object at 0x000002061A4524E0>
```

The Java archive contains a nz.msl.examples package with two classes, MathUtils and Matrix

```python
>>> MathUtils = jar.lib.nz.msl.examples.MathUtils
>>> Matrix = jar.lib.nz.msl.examples.Matrix
```

Generate a random number and calculate the square root of a number using the MathUtils class

```python
>>> MathUtils.random()
0.17555846754602522
>>> MathUtils.sqrt(32.4)
5.692099788303083
```

Use the Matrix class to calculate the inverse of a 3x3 matrix that is filled with random numbers between 0 and 100

```python
>>> m = Matrix(3, 3, 0.0, 100.0)
>>> print(m.toString())
+5.937661e+01 +5.694407e+01 +5.132319e+01
+2.443462e+01 +9.051636e+00 +5.500980e+01
+6.183735e+01 +9.492954e+01 +4.519221e+01
>>> m_inverse = m.getInverse()
>>> print(m_inverse.toString())
+7.446422e-02 -3.556370e-02 -4.127679e-02
-3.554433e-02 +7.586144e-03 +3.113227e-02
-2.722735e-02 +3.272723e-02 +1.321192e-02
>>> identity = Matrix.multiply(m, m_inverse)
>>> print(identity.toString())
+1.000000e+00 +0.000000e+00 +2.220446e-16
+0.000000e+00 +1.000000e+00 +1.110223e-16
+0.000000e+00 -4.440892e-16 +1.000000e+00
```
Solve a linear system of equations, $Ax=b$

\[
\begin{align*}
A &= \text{jar.gateway.new_array(jar.lib.Double, 3, 3)} \\
\text{coeff} &= \begin{bmatrix} 3, 2, -1 \, | \, 7, -2, 4 \, | \, -1, 5, 1 \end{bmatrix} \\
\text{for } i \text{ in range}(3): \\
&\quad \text{for } j \text{ in range}(3): \\
&\quad \quad A[i][j] = \text{float(coeff}[i][j]) \\
\text{b} &= \text{jar.gateway.new_array(jar.lib.Double, 3)} \\
\text{b}[0] &= 1.6 \\
\text{b}[1] &= -12.3 \\
\text{b}[2] &= 3.4 \\
\text{x} &= \text{Matrix.solve(Matrix}(A), \text{Matrix}(b)) \\
\text{print}(\text{x.toString()})
\end{align*}
\]

Show that $x$ is a solution by getting $b$ back

\[
\begin{align*}
\text{for } i \text{ in range}(3): \\
&\quad \text{val} = 0.0 \\
&\quad \text{for } j \text{ in range}(3): \\
&\quad \quad \text{val} += \text{coeff}[i][j] \times \text{x.getValue}(j, 0) \\
&\quad \text{print}(\text{val})
\end{align*}
\]

Shutdown the connection to the JVM when you are finished

\[
\text{jar.gateway.shutdown()}
\]

Load Java byte code, a `.class` file, in a JVM, see here for the source code.

\[
\begin{align*}
\text{cls} &= \text{LoadLibrary(EXAMPLES_DIR + '/Trig.class')} \\
\text{Trig} &= \text{cls.lib.Trig}
\end{align*}
\]

The Java library contains a `Trig` class, which calculates various trigonometric quantities

\[
\begin{align*}
\text{Trig.cos}(1.2) &= 0.3623577544766736 \\
\text{Trig.asin}(0.6) &= 0.6435011087932844 \\
\text{Trig.tanh}(1.3) &= 0.8617231593133063
\end{align*}
\]

Once again, shutdown the connection to the JVM when you are finished
1.2.7 COM

To load a Component Object Model (COM) library pass in the library’s Program ID. To view the COM libraries that are available on your computer you can run the `get_com_info()` function.

**Attention:** This example will only work on Windows.

Here we load the `FileSystemObject` library and include the `com` argument to indicate that it is a COM library.

```python
>>> com = LoadLibrary('Scripting.FileSystemObject', 'com')
>>> com
<LoadLibrary libtype=POINTER(IFileSystem3) path=Scripting.FileSystemObject>
```

We can then use the library to create, edit and close a text file.

```python
>>> fp = com.lib.CreateTextFile('a_new_file.txt')
>>> fp.WriteLine('This is a test')
0
>>> fp.Close()
0
```

**Tip:** If you are importing `comtypes` and you get the following error

```python
OSError: [WinError -2147417850] Cannot change thread mode after it is set
```

then you can eliminate this error by setting `sys.coinit_flags = 0` before importing `comtypes`.

For example,

```python
import sys
sys.coinit_flags = 0
import comtypes
```

1.3 Access a 32-bit library in 64-bit Python

This section of the documentation shows examples for how a module running within a 64-bit Python interpreter can communicate with a 32-bit shared library by using inter-process communication. The method that is used to allow a 32-bit and a 64-bit process to exchange information is by use of a file. The `pickle` module is used to (de)serialize Python objects.

The following table summarizes the example modules that are available.

Modules that end in `32` contain a class that is a subclass of `Server32`. This subclass is a wrapper around a 32-bit library and is hosted on a 32-bit server.
Modules that end in 64 contain a class that is a subclass of Client64. This subclass sends a request to the corresponding Server32 subclass to communicate with the 32-bit library.

<table>
<thead>
<tr>
<th>64</th>
<th>32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>echo64</td>
<td>echo32</td>
<td>An example of a 64-bit echo server.</td>
</tr>
<tr>
<td>cpp64</td>
<td>cpp32</td>
<td>A wrapper around a 32-bit C++ library.</td>
</tr>
<tr>
<td>fortran64</td>
<td>fortran32</td>
<td>Communicates with fortran32 via the Fortran32 class.</td>
</tr>
<tr>
<td>dotnet64</td>
<td>dotnet32</td>
<td>A wrapper around a 32-bit .NET library via the DotNet32 class.</td>
</tr>
<tr>
<td>kernel64</td>
<td>kernel32</td>
<td>A wrapper around the 32-bit Windows kernel32.dll library.</td>
</tr>
<tr>
<td>labview64</td>
<td>labview32</td>
<td>Communicates with labview32 via the Labview32 class.</td>
</tr>
</tbody>
</table>

The following illustrates a minimal usage example. The cpp_lib32.dll file is a 32-bit C++ library that cannot be loaded from a module that is running within a 64-bit Python interpreter. This library gets loaded by the MyServer class (that is a subclass of Server32) which is running within a 32-bit executable, see start_server32. MyServer hosts the library at the specified host address and port number. Any class that is a subclass of Server32 must provide three arguments in its constructor: host, port and quiet (in that order) and **kwargs. Otherwise the 32-bit executable cannot create an instance of the subclass.

```python
# my_server.py
from msl.loadlib import Server32
class MyServer(Server32):
    """A wrapper around a 32-bit C++ library, 'cpp_lib32.dll', that has an 'add' function."""
    def __init__(self, host, port, quiet, **kwargs):
        # Load the 'cpp_lib32' shared-library file using ctypes.CDLL.
        super(MyServer, self).__init__('cpp_lib32.dll', 'cdll', host, port, quiet)

    def add(self, a, b):
        # The Server32 class has a 'lib' property that is a reference to
        # the ctypes.CDLL object.
        # The shared library’s 'add' function takes two integers as inputs
        # and returns the sum.
```

(continues on next page)
Keyword arguments, **kwargs, can be passed to the server either from the client (see, Client64) or by manually starting the server from the command line (see, start_server32). However, the data types for the values of the **kwargs are not preserved (since they are ultimately parsed from the command line). Therefore, all data types for the kwargs values will be of type str at the constructor of the Server32 subclass. You must convert each value to the appropriate data type. This **kwargs variable is the only variable that the data type is not preserved for the client-server protocol (see, the “Echo” example that shows that data types are preserved between client-server method calls).

MyClient is a subclass of Client64 which sends a request to MyServer to call the add function in the shared library. MyServer processes the request and sends the response back to MyClient.

```python
# my_client.py
from msl.loadlib import Client64
class MyClient(Client64):
    """Send a request to 'MyServer' to execute the 'add' method and get the response."""
    def __init__(self):
        # Specify the name of the Python module to execute on the 32-bit server (i.e., 'my_server')
        super(MyClient, self).__init__(module32='my_server')
    def add(self, a, b):
        # The Client64 class has a 'request32' method to send a request to the 32-bit server.
        # Send the 'a' and 'b' arguments to the 'add' method in MyServer.
        return self.request32('add', a, b)
```

The MyClient class would then be used as follows

```python
>>> from my_client import MyClient
>>> c = MyClient()
>>> c.add(1, 2)
3
```

The following examples are provided for communicating with different libraries that were compiled in different programming languages or using different calling conventions:

### 1.3.1 An Echo Example

This example does not actually communicate with a 32-bit shared library but shows how Python data types are preserved when they are passed from the Echo64 client to the Echo32 server and back. The Echo32 server just returns a tuple of the (**args, **kwargs) that it received back to the Echo64 client.

The following is a script that illustrates that the data types are preserved:

```python
from msl.examples.loadlib import Echo64
```
echo = Echo64()
echo.send_data()
echo.send_data(True)

echo.send_data([1, 2, 3, 4, 5, 6])
echo.send_data(data='my string')

Running this script would create the following output. The black text corresponds to the Echo64 print() statements and the red text to the Echo32 print() statements.

Observant readers will notice that the 32-bit server indicated that it is Serving cpp_lib32.dll on http://127.0.0.1:2521. Even though this is an echo example, a library must still be loaded even though it is not being called. The cpp_lib32.dll library is loaded to satisfy this requirement.

Or, by using an interactive console, create a Echo64 object:

```python
>>> from msl.examples.loadlib import Echo64
>>> echo = Echo64()
Client running on 3.5.2 |Continuum Analytics, Inc.| (default, Jul 5 2016, 11:41:13) [MSC v.1900 64 bit (AMD64)]

Send a boolean as an argument, see send_data():

```python
>>> result = echo.send_data(True)
Are the 64- and 32-bit arguments equal? True
<class 'bool'> True
```
>>> result = echo.send_data(1.2, {'my list':[1, 2, 3]}, 0.2j, range(10),
  x=True, y='hello world!')
Are the 64- and 32-bit arguments equal? True
  <class 'float'> 1.2
  <class 'dict'> {'my list': [1, 2, 3]}
  <class 'complex'> 0.2j
  <class 'range'> range(0, 10)
Are the 64- and 32-bit keyword arguments equal? True
  x: <class 'bool'> True
  y: <class 'str'> hello world!

Shutdown the server when you are done communicating with the 32-bit library, see `shutdown_server32()`:

```python
>>> echo.shutdown_server32()
```

The 32-bit server received these args:
  <class 'bool'> True
The 32-bit server received these args:
  <class 'list'> [1, 2, 3, 4, 5, 6]
The 32-bit server received these kwargs:
  data: <class 'str'> my string
  x: <class 'list'> [1.2, 3.4, 6.1]
  y: <class 'list'> [43.2, 23.6, 12.7]
The 32-bit server received these args:
  <class 'float'> 1.12345
  <class 'dict'> {'my list': [1, 2, 3, 4]}
  <class 'complex'> 0.2j
  <class 'range'> range(0, 10)
The 32-bit server received these kwargs:
  x: <class 'bool'> True
  y: <class 'str'> hello world!
Stopped http://127.0.0.1:25521

**Note:** The server will automatically shutdown when the `Echo64` object gets destroyed (as it did in the example script above). When using a subclass of `Client64` in a script, the `__del__` method gets called automatically when the instance is about to be destroyed (and the reference count reaches 0) and therefore you do not have to call the `shutdown_server32()` method to shutdown the server. If the `Client64` subclass does not get destroyed properly, for example if you are using an interactive console and then exit the console abruptly, then the server will still be running and therefore you must manually terminate the server processes.

### 1.3.2 Load a 32-bit C++ library in 64-bit Python

**Note:** If you have issues running the example please make sure that you have the `prerequisites` installed.

This example shows how to access a 32-bit C++ library from a module that is run by a 64-bit Python interpreter by using inter-process communication. `Cpp32` is the 32-bit server and `Cpp64` is the 64-bit
client. The source code of the C++ program is available [here](#).

**Important:** By default `ctypes` expects that a `ctypes.c_int` data type is returned from the library call. If the returned value from the library is not a `ctypes.c_int` then you MUST redefine the `ctypes` `restype` value to be the appropriate data type. The `Cpp32` class shows various examples of redefining the `restype` value.

The following shows that the `cpp_lib32` library cannot be loaded in a 64-bit Python interpreter:

```python
>>> from msl.loadlib import LoadLibrary, IS_PYTHON_64BIT
>>> from msl.examples.loadlib import EXAMPLES_DIR

>>> IS_PYTHON_64BIT
True

>>> cpp = LoadLibrary(EXAMPLES_DIR + '/cpp_lib32')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "D:\msl\loadlib\load_library.py", line 109, in __init__
    self._lib = ctypes.CDLL(self._path)
  File "C:\Miniconda3\lib\ctypes\__init__.py", line 348, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: [WinError 193] %1 is not a valid Win32 application
```

However, the 64-bit version of the C++ library can be directly loaded in 64-bit Python:

```python
>>> cpp64 = LoadLibrary(EXAMPLES_DIR + '/cpp_lib64')
>>> cpp64
<LoadLibrary libtype=CDLL path=D:\msl\examples\loadlib\cpp_lib64.dll>
>>> cpp64.lib.add(3, 14)
17
```

Instead, create a `Cpp64` client to communicate with the 32-bit `cpp_lib32` library from 64-bit Python:

```python
>>> from msl.examples.loadlib import Cpp64
>>> cpp = Cpp64()
>>> cpp
<Cpp64 lib=cpp_lib32.dll address=127.0.0.1:63238>
>>> cpp.lib32_path
'D:\msl\examples\loadlib\cpp_lib32.dll'
```

Add two integers, see `add()`:

```python
>>> cpp.add(3, 14)
17
```

Subtract two C++ floating-point numbers, see `subtract()`:

```python
>>> cpp.subtract(43.2, 3.2)
40.0
```

Add or subtract two C++ double-precision numbers, see `add_or_subtract()`:

```python
>>> cpp.add_or_subtract(1.1, 2.2, True)
3.3000000000000003
>>> cpp.add_or_subtract(1.1, 2.2, False)
-1.1
```

### 1.3. Access a 32-bit library in 64-bit Python
Arrays

Multiply a 1D array by a number, see `scalar_multiply()`:

```python
a = [float(val) for val in range(10)]
cpp.scalar_multiply(2.0, a)
>>> [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0]
```

If you have a numpy ndarray in 64-bit Python then you cannot pass the ndarray object to `Server32` because the 32-bit server would need to load the ndarray in a 32-bit version of numpy (which is not bundled with the 32-bit server, but could be bundled if you ran the `freeze_server32` module and included a 32-bit version of numpy in the frozen executable). To simplify the procedure we can convert the ndarray to a Python list using the `numpy.ndarray.tolist()` method.

```python
import numpy as np
a = np.arange(9.)
cpp.scalar_multiply(3.1, a.tolist())
>>> [0.0, 3.1, 6.2, 9.3, 12.4, 15.5, 18.6, 21.7, 24.8]
```

If you want the returned value from `scalar_multiply` to be a numpy ndarray then use:

```python
np.array(cpp.scalar_multiply(3.1, a.tolist()))
array([ 0. , 3.1, 6.2, 9.3, 12.4, 15.5, 18.6, 21.7, 24.8])
```

Strings

Reverse a string. The memory for the reversed string is allocated in Python, see `reverse_string_v1()`:

```python
cpp.reverse_string_v1('hello world!')
!'olleh world!
```

Reverse a string. The memory for the reversed string is allocated in C++, see `reverse_string_v2()`:

```python
cpp.reverse_string_v2('uncertainty')
'yniatrecnuy'
```

Structs

It is possible to pickle a `ctypes.Structure` and pass the `struct` object between `Cpp64` and `Cpp32` provided that the `struct` is a fixed size in memory (i.e., the `struct` does not contain any pointers). If the `struct` contains pointers then you must create the `struct` within `Cpp32` and you can only pass the values of the `struct` between `Cpp64` and `Cpp32`. 
Attention: The following will only work if Cpp64 is run using Python 3 because Cpp32 is running on Python 3 and there are issues with ctypes and pickle when mixing Python 2 and Python 3.

The `cpp_lib32` library contains the following structs:

```c
struct Point {
    double x;
    double y;
};

struct FourPoints {
    Point points[4];
};

struct NPoints {
    int n;
    Point *points;
};
```

The `distance_4_points()` method uses the `FourPoints` struct to calculate the total distance connecting 4 `Point` structs. Since the `FourPoints` struct is a fixed size it can be created in 64-bit Python, pickled and then unpickled in Cpp32.

```python
>>> from msl.examples.loadlib import FourPoints
>>> fp = FourPoints((0, 0), (0, 1), (1, 1), (1, 0))
>>> cpp.distance_4_points(fp)
4.0
```

The `Cpp32.circumference` method uses the `NPoints` struct to calculate the circumference of a circle using `n` `Point` structs. Since the `NPoints` struct is not a fixed size it must be created in the `Cpp32.circumference` method. The `Cpp64.circumference` method takes the values of the radius and `n` as input arguments to pass to the `Cpp32.circumference` method.

```python
>>> for i in range(16):
...     print(cpp.circumference(0.5, 2**i))
...0.0
2.0
2.82842712474619
3.061467458920718
3.121445152258052
3.1365484905459406
3.1403311569547543
3.1412772509327787
3.141513801144288
3.1415729403671087
3.14158772509327787
3.1415914215111314
3.1415923455699404
3.141592576584724
3.1415926343379557
3.1415926487759718
```

Shutdown the server, see `shutdown_server32()`.

1.3. Access a 32-bit library in 64-bit Python
Note: When using a subclass of Client64 in a script, the shutdown_server32() method gets called automatically when the instance of the subclass is about to be destroyed and therefore you do not have to call the shutdown_server32() method to shutdown the server.

## 1.3.3 Load a 32-bit FORTRAN library in 64-bit Python

Note: If you have issues running the example please make sure that you have the prerequisites installed.

This example shows how to access a 32-bit FORTRAN library from a module that is run by a 64-bit Python interpreter by using inter-process communication. Fortran32 is the 32-bit server and Fortran64 is the 64-bit client. The source code of the FORTRAN program is available here.

**Important:** By default ctypes expects that a ctypes.c_int data type is returned from the library call. If the returned value from the library is not a ctypes.c_int then you MUST redefine the ctypes restype value to be the appropriate data type. The Fortran32 class shows various examples of redefining the restype value.

The following shows that the fortran_lib32 library cannot be loaded in a 64-bit Python interpreter:

```python
>>> from msl.loadlib import LoadLibrary, IS_PYTHON_64BIT
>>> from msl.examples.loadlib import EXAMPLES_DIR
>>> IS_PYTHON_64BIT
True
>>> f = LoadLibrary(EXAMPLES_DIR + '/fortran_lib32')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "D:\msl\loadlib\load_library.py", line 109, in __init__
    self._lib = ctypes.CDLL(self._path)
  File "C:\Miniconda3\lib\ctypes\__init__.py", line 348, in __init__
    self._handle = _dlopen(self._name, mode)
OSError: [WinError 193] %1 is not a valid Win32 application
```

However, the 64-bit version of the FORTRAN library can be directly loaded in 64-bit Python:

```python
>>> f64 = LoadLibrary(EXAMPLES_DIR + '/fortran_lib64')
>>> f64
<LoadLibrary libtype=CDLL path=D:\msl\examples\loadlib\fortran_lib64.dll>
>>> from ctypes import byref, c_int8
>>> f64.lib.sum_8bit(byref(c_int8(-50)), byref(c_int8(110)))
60
```

Instead, create a Fortran64 client to communicate with the 32-bit fortran_lib32 library:

```python
>>> from msl.examples.loadlib import Fortran64
>>> f = Fortran64()
>>> f
```
Add two int8 values, see `sum_8bit()`:

```python
>>> f.sum_8bit(-50, 110)
60
```

Add two int16 values, see `sum_16bit()`:

```python
>>> f.sum_16bit(2**15-1, -1)
32766
```

Add two int32 values, see `sum_32bit()`:

```python
>>> f.sum_32bit(123456788, 1)
123456789
```

Add two int64 values, see `sum_64bit()`:

```python
>>> f.sum_64bit(-2**63, 1)
-9223372036854775807
```

Multiply two float32 values, see `multiply_float32()`:

```python
>>> f.multiply_float32(1e30, 2e3)
1.9999999889914546e+33
```

Multiply two float64 values, see `multiply_float64()`:

```python
>>> f.multiply_float64(1e30, 2e3)
2.0000000000000002e+33
```

Check if a value is positive, see `is_positive()`:

```python
>>> f.is_positive(1e-100)
True
>>> f.is_positive(-1e-100)
False
```

Add or subtract two integers, see `add_or_subtract()`:

```python
>>> f.add_or_subtract(1000, 2000, True)
3000
>>> f.add_or_subtract(1000, 2000, False)
-1000
```

Calculate the n’th factorial, see `factorial()`:

```python
>>> f.factorial(0)
1.0
>>> f.factorial(127)
3.012660018457658e+213
```

Calculate the standard deviation of an list of values, see `standard_deviation()`:

```python
1.3. Access a 32-bit library in 64-bit Python
```
>>> f.standard_deviation([float(val) for val in range(1,10)])
2.7386127875258306

Compute the Bessel function of the first kind of order 0 at $x$, see $besselJ0()$:

>>> f.besselJ0(8.6)
0.014622991278741278

Reverse a string, see $reverse_string()$:

>>> f.reverse_string('hello world!')
'!dlrow olleh'

Add two 1D arrays, see $add_1D_arrays()$:

```python
>>> a = [float(val) for val in range(1, 10)]
>>> b = [0.5*val for val in range(1, 10)]
>>> a
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
>>> b
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]
>>> f.add_1D_arrays(a, b)
[1.5, 3.0, 4.5, 6.0, 7.5, 9.0, 10.5, 12.0, 13.5]
```

Multiply two matrices, see $matrix_multiply()$:

```python
>>> m1 = [[1, 2, 3], [4, 5, 6]]
>>> m2 = [[1, 2], [3, 4], [5, 6]]
>>> f.matrix_multiply(m1, m2)
[[22.0, 28.0], [49.0, 64.0]]
```

Shutdown the server, see $shutdown_server32()$:

```python
>>> f.shutdown_server32()
```

**Note:** When using a subclass of $Client64$ in a script, the $shutdown_server32()$ method gets called automatically when the instance of the subclass is about to be destroyed and therefore you do not have to call the $shutdown_server32()$ method to shutdown the server.

### 1.3.4 Load a 32-bit .NET library in 64-bit Python

**Note:** If you have issues running the example please make sure that you have the prerequisites installed.

This example shows how to access a 32-bit .NET library from a module that is run by a 64-bit Python interpreter by using inter-process communication. $DotNet32$ is the 32-bit server and $DotNet64$ is the 64-bit client.

**Tip:** The JetBrains dotPeek program can be used to reliably decompile any .NET assembly into the equivalent C# source code. For example, peeking inside the $dotnet_lib32.dll$ library, that the $DotNet32$ class is a wrapper around, gives
The following shows that the 32-bit `dotnet_lib32.dll` library cannot be loaded in a 64-bit Python interpreter:

```python
>>> from msl.loadlib import LoadLibrary, IS_PYTHON_64BIT
>>> from msl.examples.loadlib import EXAMPLES_DIR

>>> IS_PYTHON_64BIT
True

>>> net = LoadLibrary(EXAMPLES_DIR + '/dotnet_lib32.dll', 'net')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "D:\msl\loadlib\load_library.py", line 130, in __init_
System.BadImageFormatException: Could not load file or assembly 'dotnet_lib32.dll' or one of its dependencies. is not a valid Win32 application.
          at System.Reflection.RuntimeAssembly.nLoadFile(String path, Evidence evidence)
          at System.Reflection.Assembly.LoadFile(String path)
```

However, the 64-bit version of the .NET library can be directly loaded in 64-bit Python:

```python
>>> net = LoadLibrary(EXAMPLES_DIR + '/dotnet_lib64.dll', 'net')
>>> net.lib.StringManipulation.reverse_string('Hello World!')
'!dlroW olleH'
```

Instead, create a `DotNet64` client to communicate with the 32-bit `dotnet_lib32.dll` library:

1.3. Access a 32-bit library in 64-bit Python
Get the names of the classes in the .NET library module, see `get_class_names()`:

```python
>>> dn.get_class_names()
['StringManipulation', 'StaticClass', 'DotNetMSL.BasicMath', 'DotNetMSL.ArrayManipulation']
```

Add two integers, see `add_integers()`:

```python
>>> dn.add_integers(8, 2)
10
```

Divide two C# floating-point numbers, see `divide_floats()`:

```python
>>> dn.divide_floats(4., 5.)
0.8
```

Multiple two C# double-precision numbers, see `multiply_doubles()`:

```python
>>> dn.multiply_doubles(872.24, 525.525)
458383.926
```

Add or subtract two C# double-precision numbers, see `add_or_subtract()`:

```python
>>> dn.add_or_subtract(99., 9., True)
108.0
>>> dn.add_or_subtract(99., 9., False)
90.0
```

Multiply a 1D array by a number, see `scalar_multiply()`:

```python
>>> a = [float(val) for val in range(10)]
>>> a
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
>>> dn.scalar_multiply(2.0, a)
[0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0]
```

Multiply two matrices, see `multiply_matrices()`:

```python
>>> m1 = [[1., 2., 3.], [4., 5., 6.]]
>>> m2 = [[1., 2.], [3., 4.], [5., 6.]]
>>> dn.multiply_matrices(m1, m2)
[[22.0, 28.0], [49.0, 64.0]]
```

Reverse a string, see `reverse_string()`:

```python
>>> dn.reverse_string('New Zealand')
'dnalaeZ weN'
```

Call the static methods in the `StaticClass` class
>>> dn.add_multiple(1, 2, 3, 4, 5)
15
>>> dn.concatenate('the ', 'experiment ', 'worked ', False, 'temporarily')
'the experiment worked '  
'false temporarily'
>>> dn.concatenate('the ', 'experiment ', 'worked ', True, 'temporarily')
'the experiment worked temporarily'

Shutdown the server, see `shutdown_server32()`:

```python
>>> dn.shutdown_server32()
```

Note: When using a subclass of `Client64` in a script, the `shutdown_server32()` method gets called automatically when the instance of the subclass is about to be destroyed and therefore you do not have to call the `shutdown_server32()` method to shutdown the server.

1.3.5 Load a 32-bit Windows __stdcall library in 64-bit Python

This example shows how to access the 32-bit Windows `kernel32` library, from a module that is run by a 64-bit Python interpreter by using inter-process communication. `Kernel32` is the 32-bit server and `Kernel64` is the 64-bit client.

The following shows that the `kernel32` library cannot be loaded in a 64-bit Python interpreter:

```python
>>> from msl.loadlib import LoadLibrary, IS_PYTHON_64BIT
>>> IS_PYTHON_64BIT
True
>>> k = LoadLibrary('C:/Windows/SysWOW64/kernel32.dll', 'windll')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "D:\msl\loadlib\load_library.py", line 111, in __init__
    self._lib = ctypes.WinDLL(self._path)
  File "C:\Miniconda3\lib\ctypes\_init__.py", line 348, in _init_
    self._handle = _dlopen(self._name, mode)
OSError: [WinError 193] %1 is not a valid Win32 application
```

Instead, create a `Kernel64` client to communicate with the 32-bit `kernel32` library:

```python
>>> from msl.examples.loadlib import Kernel64
>>> k = Kernel64()
>>> k
<Kernel64 lib=kernel32.dll address=127.0.0.1:59481>
>>> k.lib32_path
'C:\\Windows\\SysWOW64\\kernel32.dll'
```

Call the library to get the current date and time, see `get_local_time()`:

```python
>>> k.get_local_time()
datetime.datetime(2017, 2, 3, 16, 37, 5, 351000)
```

Shutdown the server, see `shutdown_server32()`:

```python
>>> k.shutdown_server32()
```
Note: When using a subclass of Client64 in a script, the shutdown_server32() method gets called automatically when the instance of the subclass is about to be destroyed and therefore you do not have to call the shutdown_server32() method to shutdown the server.

1.3.6 Load a 32-bit LabVIEW library in 64-bit Python

Attention: This example requires that the appropriate LabVIEW Run-Time Engine is installed and that the operating system is Windows.

This example shows how to access a 32-bit LabVIEW library from a module that is run by a 64-bit Python interpreter by using inter-process communication. Labview32 is the 32-bit server and Labview64 is the 64-bit client. The source code of the LabVIEW program is available here.

Create a Labview64 client to communicate with the 32-bit labview_lib32 library from 64-bit Python:

```python
>>> from msl.examples.loadlib import Labview64
>>> labview = Labview64()
>>> labview
<Labview64 lib=labview_lib32.dll address=127.0.0.1:49952>
>>> labview.lib32_path
'D:\msl\examples\loadlib\labview_lib32.dll'
```

Calculate the mean and the sample variance and standard deviation of some data, see stdev():

```python
>>> data = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> labview.stdev(data)
(5.0, 7.5, 2.7386127875258306)
```

Calculate the mean and the population variance and standard deviation of data:

```python
>>> labview.stdev(data, 1)
(5.0, 6.666666666666667, 2.58198897471611)
```

Shutdown the server, see shutdown_server32():

```python
>>> labview.shutdown_server32()
```

Note: When using a subclass of Client64 in a script, the shutdown_server32() method gets called automatically when the instance of the subclass is about to be destroyed and therefore you do not have to call the shutdown_server32() method to shutdown the server.

Tip: If you find yourself repeatedly implementing each method in your Client64 subclass in the following way (i.e., you are essentially duplicating the code for each method)

```python
from msl.loadlib import Client64
class LinearAlgebra(Client64):
```

(continues on next page)
def __init__(self):
    super(LinearAlgebra, self).__init__(module32='linear_algebra_32.py')

def solve(self, matrix, vector):
    return self.request32('solve', matrix, vector)

def eigenvalues(self, matrix):
    return self.request32('eigenvalues', matrix)

def stdev(self, data, as_population=True):
    return self.request32('stdev', data, as_population=as_population)

def determinant(self, matrix):
    return self.request32('determinant', matrix)

def cross_product(self, vector1, vector2):
    return self.request32('cross_product', vector1, vector2)

Then you can simplify the implementation by defining your `Client64` subclass as

```python
from msl.loadlib import Client64
class LinearAlgebra(Client64):
    def __init__(self):
        super(LinearAlgebra, self).__init__(module32='linear_algebra_32.py')

    def __getattr__(self, method32):
        def send(*args, **kwargs):
            return self.request32(method32, *args, **kwargs)
        return send
```

and you will get the same behaviour. If you call a method that does not exist on the `Server32` subclass or if you specify the wrong number of arguments or keyword arguments then a `Server32Error` will be raised.

There are situations where you may want to explicitly write some (or all) of the methods in the `Client64` subclass in addition to (or instead of) implementing the `__getattr__` method, e.g.,

- you are writing an API for others to use and you want features like autocomplete or docstrings to be available in the IDE that the person using your API is using
- you want the `Client64` subclass to do error checking on the `*args, **kwargs` and/or on the result from the `Server32` subclass (this allows you to have control over the type of `Exception` that is raised because if the `Server32` subclass raises an exception then it is a `Server32Error`)
- you want to modify the returned object from a particular `Server32` method, for example, a `list` is returned but you want the corresponding `Client64` method to return a `numpy.ndarray`

1.3. Access a 32-bit library in 64-bit Python
1.4 MSL-LoadLib API Documentation

The root package is

```
msl.loadlib
```

which has the following class for directly loading a shared library,

```
LoadLibrary(path[, libtype])
```

the following client-server classes for communicating with a 32-bit library from 64-bit Python,

```
Client64(module32[, host, port, timeout, ...])
Server32(path, libtype, host, port, quiet, ...)
```

the following modules for creating a frozen 32-bit server for hosting a 32-bit library

```
freeze_server32
start_server32
```

and a utilities module with general helper functions

```
utils
```

1.4.1 Package Structure

**msl.loadlib package**

Load a shared library.

The following constants are provided in the MSL-LoadLib package.

```
msl.loadlib.IS_LINUX = True
Type bool
```

Whether the Operating System is Linux.

```
msl.loadlib.IS_MAC = False
Type bool
```

Whether the Operating System is Mac OS X.

```
msl.loadlib.IS_PYTHON2 = False
Type bool
```

Whether Python 2.x is being used.
msl.loadlib.IS_PYTHON3 = True
    Whether Python 3.x is being used.
    Type bool

msl.loadlib.IS_PYTHON_64BIT = True
    Whether the Python interpreter is 64-bits.
    Type bool

msl.loadlib.IS_WINDOWS = False
    Whether the Operating System is Windows.
    Type bool

msl.loadlib.version_info = version_info(major=0, minor=6, micro=1, releaselevel='dev0')
    Contains the version information as a (major, minor, micro, releaselevel) tuple.
    Type namedtuple

**msl.loadlib.client64 module**

Contains the base class for communicating with a 32-bit library from 64-bit Python.

The `Server32` class is used in combination with the `Client64` class to communicate with a 32-bit shared library from 64-bit Python.

```python
class msl.loadlib.client64.Client64(
    module32,  # str
    host='127.0.0.1',  # str, optional
    port=None,  # int, optional
    timeout=10.0,  # float, optional
    quiet=True,  # bool, optional
    append_sys_path=None,  # None, **kwargs
    append_environ_path=None,  # None, **kwargs
    rpc_timeout=None,  # None, **kwargs
    **kwargs
)
```

Bases: object

Base class for communicating with a 32-bit library from 64-bit Python.

Starts a 32-bit server, `Server32`, to host a Python module that is a wrapper around a 32-bit library. The `client64` module runs within a 64-bit Python interpreter and it sends a request to the server which calls the 32-bit library to execute the request. The server then provides a response back to the client.

Changed in version 0.6: Added the `rpc_timeout` parameter

**Parameters**

- `module32 (str)` – The name of the Python module that is to be imported by the 32-bit server.
- `host (str, optional)` – The address of the 32-bit server. Default is '127.0.0.1'.
- `port (int, optional)` – The port to open on the 32-bit server. Default is None, which means to automatically find a port that is available.
- `timeout (float, optional)` – The maximum number of seconds to wait to establish a connection to the 32-bit server. Default is 10 seconds.
- `quiet (bool, optional)` – Whether to hide `sys.stdout` messages on the 32-bit server. Default is True.
• **append_sys_path** *(str or list of str, optional)* – Append path(s) to the 32-bit server’s `sys.path` variable. The value of `sys.path` from the 64-bit process is automatically included, i.e., `sys.path(32bit) = sys.path(64bit) + append_sys_path`.

• **append_environ_path** *(str or list of str, optional)* – Append path(s) to the 32-bit server’s `os.environ['PATH']` variable. This can be useful if the library that is being loaded requires additional libraries that must be available on `PATH`.

• **rpc_timeout** *(float, optional)* – The maximum number of seconds to wait for a response from the 32-bit server. The RPC timeout value is used for all requests from the server. If you want different requests to have different timeout values then you will need to implement custom timeout handling for each method on the server. Default is `None`, which means to use the default timeout value used by the `socket` module (which is to `wait forever`).

• **kwargs** – Keyword arguments that will be passed to the `Server32` subclass. The data type of each value is not preserved. It will be a string at the constructor of the `Server32` subclass.

**Note:** If `module32` is not located in the current working directory then you must either specify the full path to `module32` or you can specify the folder where `module32` is located by passing a value to the `append_sys_path` parameter. Using the `append_sys_path` option also allows for any other modules that `module32` may depend on to also be included in `sys.path` so that those modules can be imported when `module32` is imported.

**Raises**

- `ConnectionTimeoutError` – If the connection to the 32-bit server cannot be established.
- `IOError` – If the frozen executable cannot be found.
- `TypeError` – If the data type of `append_sys_path` or `append_environ_path` is invalid.

**connection**

The reference to the connection to the 32-bit server.

Type `HTTPConnection`

**host**

The address of the host for the `connection`.

Type `str`

**lib32_path**

The path to the 32-bit library.

Returns `str` – The path to the 32-bit shared-library file.

**port**

The port number of the `connection`.

Type `int`
request32 (method32, *args, **kwargs)
Send a request to the 32-bit server.

Parameters

- **method32** (*str*) – The name of the method to call in the Server32 subclass.
- **args** – The arguments that the method in the Server32 subclass requires.
- **kwargs** – The keyword arguments that the method in the Server32 subclass requires.

Returns Whatever is returned by the method of the Server32 subclass.

Raises

- **Server32Error** – If there was an error processing the request on the 32-bit server.
- **ResponseTimeoutError** – If a timeout occurs while waiting for the response from the 32-bit server.

shutdown_server32 (kill_timeout=10)
Shutdown the 32-bit server.

This method shuts down the 32-bit server, closes the client connection and it deletes the temporary file that is used to save the serialized pickle’d data.

Changed in version 0.6: Added the kill_timeout parameter

Parameters kill_timeout (*float*, optional) – If the 32-bit server is still running after kill_timeout seconds then the server will be killed using brute force. A warning will be issued if the server is killed in this manner.

---

**Note:** This method gets called automatically when the reference count to the Client64 object reaches 0.

---

**msl.loadlib.exceptions module**

Exception classes.

exception msl.loadlib.exceptions.ConnectionTimeoutError
Bases: OSError
Raised when the connection to the 32-bit server cannot be established.

exception msl.loadlib.exceptions.ResponseTimeoutError
Bases: OSError
Raised when a timeout occurs while waiting for a response from the 32-bit server.
New in version 0.6.

exception msl.loadlib.exceptions.Server32Error (value, name=None, traceback=None)
Bases: http.client.HTTPException
Raised when an exception occurs on the 32-bit server.

New in version 0.5.

Parameters

- **value** *(str)* – The error message.
- **name** *(str, optional)* – The name of the exception.
- **traceback** *(str, optional)* – The exception traceback.

**name**
The name of the exception from the 32-bit server.

Type **str**

**traceback**
The exception traceback from the 32-bit server.

Type **str**

**value**
The error message from the 32-bit server.

Type **str**

---

### msl.loadlib.freeze_server32 module

Creates a 32-bit server to use for inter-process communication.

This module must be run from a 32-bit Python interpreter with PyInstaller installed.

If you want to re-freeze the 32-bit server, for example, if you want a 32-bit version of *numpy* to be available on the server, then run the following with a 32-bit Python interpreter that has the packages that you want to be available on the server installed:

```python
>>> from msl.loadlib import freeze_server32
>>> freeze_server32.main()
```

#### msl.loadlib.freeze_server32.main(*spec=None, requires_pythonnet=True, requires_comtypes=True*)

Creates a 32-bit Python server.

Uses PyInstaller to create a frozen 32-bit Python executable. This executable starts a 32-bit server, *Server32*, which hosts a Python module that can load a 32-bit library.

Changed in version 0.5: Added the `requires_pythonnet` and `requires_comtypes` arguments.

Parameters

- **spec** *(str, optional)* – If you want to freeze using a PyInstaller .spec file then you can specify the path to the .spec file.
- **requires_pythonnet** *(bool, optional)* – Whether Python for .NET must be available on the 32-bit server.
- **requires_comtypes** *(bool, optional)* – Whether comtypes must be available on the 32-bit server. If you using a non-Windows operating system then this argument is ignored.
msl.loadlib.load_library module

Load a shared library.

class msl.loadlib.load_library.LoadLibrary (path, libtype=None, **kwargs)
    Bases: object

    Load a shared library.

    For example, a C/C++, FORTRAN, C#, Java, Delphi, LabVIEW, ActiveX, ... library.

    Based on the value of `libtype` this class will load the shared library as a:

    - CDLL if `libtype` is 'cdll',
    - WinDLL if `libtype` is 'windll',
    - OleDLL if `libtype` is 'oledll',
    - System.Reflection.Assembly if `libtype` is 'net' or 'clr',
    - JavaGateway if `libtype` is 'java', or
    - comtypes.CreateObject if `libtype` is 'com'.

    Changed in version 0.4: Added support for Java archives
    Changed in version 0.5: Added support for COM libraries

    Parameters

    - `path` (str) – The path to the shared library.
      The search order for finding the shared library is:
      1. assume that a full or a relative (to the current working directory) path is specified,
      2. use ctypes.util.find_library() to find the shared library file,
      3. search sys.path, then
      4. search os.environ['PATH'] to find the shared library.

      If loading a COM library then `path` represents the progid argument.

    - `libtype` (str, optional) – The library type. The following values are currently supported:
      - 'cdll' – for a library that uses the __cdecl calling convention
      - 'windll' or 'oledll' – for a __stdcall calling convention
      - 'net' or 'clr' – for Microsoft’s .NET Framework (Common Language Runtime)
      - 'java' – for a Java archive, .jar, or Java byte code, .class, file
      - 'com' – for a COM library.
      Default is 'cdll'.

1.4. MSL-LoadLib API Documentation
Tip: Since the .jar or .class extension uniquely defines a Java library, the libtype will be automatically set to 'java' if path ends with .jar or .class. If path starts with '{' and ends with '}' then this uniquely defines the Class ID for a COM library and so libtype will be automatically set to 'com'.

**kwargs – Keyword arguments that are passed to the object that loads the library.

Raises

- IOError – If the shared library cannot be loaded.
- TypeError – If libtype is not a supported library type.

assembly

Returns a reference to the .NET Runtime Assembly object, only if the shared library is a .NET Framework, otherwise returns None.

Tip: The JetBrains dotPeek program can be used to reliably decompile any .NET Assembly in to the equivalent source code.

gateway

Returns the JavaGateway object, only if the shared library is a Java archive, otherwise returns None.

lib

Returns the reference to the loaded library object.

For example, if libtype is

- 'cdll' then a CDLL object
- 'windll' then a WinDLL object
- 'oledll' then a OleDLL object
- 'net' or 'clr' then a DotNet object
- 'java' then a JVMView object
- 'com' then the interface pointer returned by comtypes.CreateObject

path

The path to the shared library file.

Type str

class msl.loadlib.load_library.DotNet(dot_net_dict, path)

Bases: object

Contains the namespace modules, classes and System.Type objects of a .NET Assembly.

Do not instantiate this class directly.
**msl.loadlib.server32 module**

Contains the base class for loading a 32-bit shared library in 32-bit Python.

The `Server32` class is used in combination with the `Client64` class to communicate with a 32-bit shared library from 64-bit Python.

```python
class msl.loadlib.server32.Server32(path, libtype, host, port, quiet, **kwargs)
```

Bases: `http.server.HTTPServer`

Base class for loading a 32-bit library in 32-bit Python.

All modules that are to be run on the 32-bit server must contain a class that is inherited from this class and the module can import any of the standard python modules except for `distutils`, `ensurepip`, `tkinter` and `turtle`.

All modules that are run on the 32-bit server must be able to run on the Python interpreter that the server is running on, see `version()` for how to determine the version of the Python interpreter.

**Parameters**

- `path` *(str)* – The path to the 32-bit library.
- `libtype` *(str)* – The library type to use for the calling convention. One of the following:
  - `'cdll'` – for a __cdecl library
  - `'windll'` or `'oledll'` – for a __stdcall library (Windows only)
  - `'net'` or `'clr'` – for Microsoft’s .NET Framework (Common Language Runtime)
  - `'com'` – for a COM library.

**Note:** Since Java byte code is executed on the JVM it does not make sense to use `Server32` for a Java `.jar` or `.class` file.

- `host` *(str)* – The IP address of the server.
- `port` *(int)* – The port to open on the server.
- `quiet` *(bool)* – Whether to hide `sys.stdout` messages on the server.
- `**kwargs` – Keyword arguments that are passed to `LoadLibrary`.

**Raises**

- `IOError` – If the shared library cannot be loaded.
- `TypeError` – If the value of `libtype` is not supported.

**assembly**

Returns a reference to the `.NET Runtime Assembly` object, only if the shared library is a .NET Framework, otherwise returns `None`.

**Tip:** The JetBrains dotPeek program can be used to reliably decompile any .NET Assembly in to the equivalent source code.
lib

Returns the reference to the 32-bit, loaded library object.

For example, if libtype is

- 'cdll' then a CDLL object
- 'windll' then a WinDLL object
- 'oledll' then a OleDLL object
- 'net' or 'clr' then a DotNet object
- 'com' then the interface pointer returned by comtypes.CreateObject

path

The path to the shared library file.

Type str

static version()

Gets the version of the Python interpreter that the 32-bit server is running on.

Returns str – The result of executing 'Python ' + sys.version on the 32-bit server.

Examples

```python
>>> from msl.loadlib import Server32
>>> Server32.version()
'Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)]'
```

Note: This method takes about 1 second to finish because the 32-bit server needs to start in order to determine the version of the Python interpreter.

static interactive_console()

Start an interactive console.

This method starts an interactive console, in a new terminal, with the Python interpreter on the 32-bit server.

Examples

```python
>>> from msl.loadlib import Server32
>>> Server32.interactive_console()
```

quiet

Whether sys.stdout messages are hidden on the server.

Type bool

shutdown_handler()

Proxy function that is called immediately prior to the server shutting down.
The intended use case is for the server to do any necessary cleanup, such as stopping locally started threads, closing file-handles, etc...

New in version 0.6.

**msl.loadlib.start_server32 module**

This module is built in to a 32-bit executable by running `freeze_server32`.

The executable is used to host a 32-bit library, `Server32`, so that a module running in a 64-bit Python interpreter, `Client64`, can communicate with the library. This client-server exchange of information is a form of inter-process communication.

**msl.loadlib.start_server32.main()**

Starts a 32-bit server (which is a subclass of `Server32`).

Parses the command-line arguments to run a Python module on a 32-bit server to host a 32-bit library. To see the list of command-line arguments that are allowed, run the executable with the `--help` flag (or click here to view the source code of the `argparse.ArgumentParser` implementation).

**msl.loadlib.utils module**

Common functions used by the MSL-LoadLib package.

**msl.loadlib.utils.check_dot_net_config(py_exe_path)**

Check if the `useLegacyV2RuntimeActivationPolicy` property is enabled.

By default, Python for .NET only works with .NET 4.0+ and therefore it cannot automatically load a shared library that was compiled with .NET <4.0. This method ensures that the `useLegacyV2RuntimeActivationPolicy` property exists in the `<python-executable>.config` file and that it is enabled.

This link provides an overview explaining why the `useLegacyV2RuntimeActivationPolicy` property is required.

The `<python-executable>.config` file should look like

```xml
<?xml version ="1.0"?>
<configuration>
  <startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0" />
    <supportedRuntime version="v2.0.50727" />
  </startup>
</configuration>
```

**Parameters** `py_exe_path` (str) – The path to the Python executable.

**Returns**

- int –

  One of the following values:

  - -1 – if there was a problem
  - 0 – if the .NET property was already enabled, or
msl.loadlib.utils.get_available_port()  
**int**: Returns a port number that is available.

msl.loadlib.utils.get_com_info(*additional_keys)  
Reads the registry for the COM libraries that are available.  
This function is only supported on Windows.

New in version 0.5.

**Parameters**  
*additional_keys* (str, optional) – The Program ID (ProgID) key is returned automatically. You can include additional keys (e.g., Version, InprocHandler32, ToolboxBitmap32, VersionIndependentProgID, ...) if you also want this additional information to be returned for each Class ID.

**Returns**  
dict – The keys are the Class ID’s and each value is a dict of the information that was requested.

**Examples**

```python  
>>> from msl.loadlib import utils  
>>> info = utils.get_com_info()  
>>> info = utils.get_com_info('Version', 'ToolboxBitmap32')  
```

msl.loadlib.utils.is_comtypes_installed()  
Checks if comtypes is installed.

New in version 0.5.

**Returns**  
bool – Whether comtypes is installed.

msl.loadlib.utils.is_py4j_installed()  
Checks if Py4J is installed.

New in version 0.4.

**Returns**  
bool – Whether Py4J is installed.

msl.loadlib.utils.is_pythonnet_installed()  
Checks if Python for .NET is installed.

**Returns**  
bool – Whether Python for .NET is installed.

**Note:** For help getting Python for .NET installed on a non-Windows operating system look at the prerequisites, the Mono project and the Python for .NET documentation.

msl.loadlib.utils.port_in_use(port)  
Uses netstat to determine if the network port is in use.

**Parameters**  
port (int) – The port number to test.

**Returns**  
bool – Whether the port is in use.
msl.loadlib.utils.wait_for_server (host, port, timeout)
Wait for the 32-bit server to start.

Parameters

- host (str) – The host address of the server, e.g., '127.0.0.1'.
- port (int) – The port number of the server.
- timeout (float) – The maximum number of seconds to wait to establish a connection to the server.

Raises ConnectionTimeoutError – If a timeout occurred.

1.4.2 Example modules for communicating with a 32-bit library from 64-bit Python

msl.examples.loadlib package

Example modules showing how to load a 32-bit shared library in 64-bit Python.

Modules that end in 32 contain a class that is a subclass of Server32. This subclass is a wrapper around a 32-bit library and is hosted on a 32-bit server.

Modules that end in 64 contain a class that is a subclass of Client64. This subclass sends a request to the corresponding Server32 subclass to communicate with the 32-bit library.

msl.examples.loadlib.cpp32 module

A wrapper around a 32-bit C++ library, cpp_lib32.

Example of a server that loads a 32-bit shared library, cpp_lib, in a 32-bit Python interpreter to host the library. The corresponding cpp64 module can be executed by a 64-bit Python interpreter and the Cpp64 class can send a request to the Cpp32 class which calls the 32-bit library to execute the request and then return the response from the library.

class msl.examples.loadlib.cpp32.Cpp32 (host, port, quiet, **kwargs)
Bases: msl.loadlib.server32.Server32
A wrapper around the 32-bit C++ library, cpp_lib32.

This class demonstrates how to send/receive various data types to/from a 32-bit C++ library via ctypes.

Parameters

- host (str) – The IP address of the server.
- port (int) – The port to open on the server.
- quiet (bool) – Whether to hide sys.stdout messages from the server.

Note: Any class that is a subclass of Server32 MUST provide three arguments in its constructor: host, port and quiet (in that order) and **kwargs. Otherwise the server32 executable, see start_server32, cannot create an instance of the Server32 subclass.
add \((a, b)\)
Add two integers.

The corresponding C++ code is

```cpp
int add(int a, int b) {
    return a + b;
}
```

See the corresponding 64-bit `add()` method.

Parameters
- \(a\) (int) – The first integer.
- \(b\) (int) – The second integer.

Returns int – The sum of \(a\) and \(b\).

add_or_subtract \((a, b, \text{do_addition})\)
Add or subtract two double-precision numbers (‘double’ refers to the C++ data type).

The corresponding C++ code is

```cpp
double add_or_subtract(double a, double b, bool do_addition) {
    if (do_addition) {
        return a + b;
    } else {
        return a - b;
    }
}
```

See the corresponding 64-bit `add_or_subtract()` method.

Parameters
- \(a\) (float) – The first double-precision number.
- \(b\) (float) – The second double-precision number.
- \(\text{do_addition}\) (bool) – Whether to add the numbers.

Returns float – Either \(a + b\) if \(\text{do_addition}\) is True else \(a - b\).

circumference \((\text{radius}, n)\)
Estimates the circumference of a circle.

This method calls the `distance_n_points` function in `cpp_lib32`.

See the corresponding 64-bit `circumference()` method.

The corresponding C++ code uses the `NPoints` struct as the input parameter to sum the distance between adjacent points on the circle.

```cpp
double distance_n_points(NPoints p) {
    if (p.n < 2) {
        return 0.0;
    }
    double d = distance(p.points[0], p.points[p.n-1]);
    for (int i = 1; i < p.n; i++) {
        d += distance(p.points[i], p.points[i-1]);
    }
    return d;
}
```


```cpp
global double distance_4_points(FourPoints p) {
    double d = distance(p.points[0], p.points[3]);
    for (int i = 1; i < 4; i++) {
        d += distance(p.points[i], p.points[i-1]);
    }
    return d;
}
```

See the corresponding 64-bit `distance_4_points()` method.

**Parameters**

- `points` ([FourPoints]) – The points to use to calculate the total distance.

**Returns**

- `float` – The total distance connecting the 4 Point's.

### reverse_string_v1 (original)
Reverse a string (version 1).

In this method Python allocates the memory for the reversed string and passes the string to C++.

The corresponding C++ code is

```cpp
void reverse_string_v1(const char* original, int n, char* reversed) {
    for (int i = 0; i < n; i++) {
        reversed[i] = original[n-i-1];
    }
}
```

See the corresponding 64-bit `reverse_string_v1()` method.

**Parameters**

- `original` ([str]) – The original string.

**Returns**

- `str` – The string reversed.

### reverse_string_v2 (original)
Reverse a string (version 2).

In this method C++ allocates the memory for the reversed string and passes the string to Python.
The corresponding C++ code is

```cpp
char* reverse_string_v2(char* original, int n) {
    char* reversed = new char[n];
    for (int i = 0; i < n; i++) {
        reversed[i] = original[n - i - 1];
    }
    return reversed;
}
```
See the corresponding 64-bit `reverse_string_v2()` method.

**Parameters**
- `original` (str) – The original string.

**Returns**
- `str` – The string reversed.

**scalar_multiply** *(a, xin)*
Multiply each element in an array by a number.

The corresponding C++ code is

```cpp
void scalar_multiply(double a, double* xin, int n, double* xout) {
    for (int i = 0; i < n; i++) {
        xout[i] = a * xin[i];
    }
}
```
See the corresponding 64-bit `scalar_multiply()` method.

**Parameters**
- `a` (float) – The scalar value.
- `xin` (list of float) – The array to modify.

**Returns**
- `list of float` – A new array with each element in `xin` multiplied by `a`.

**subtract** *(a, b)*
Subtract two floating-point numbers (`float` refers to the C++ data type).

The corresponding floating-point code is

```cpp
float subtract(float a, float b) {
    return a - b;
}
```
See the corresponding 64-bit `subtract()` method.

**Parameters**
- `a` (float) – The first floating-point number.
- `b` (float) – The second floating-point number.

**Returns**
- `float` – The difference between `a` and `b`.

**class** `msl.examples.loadlib.cpp32.FourPoints(point1, point2, point3, point4)`
Bases: `_ctypes.Structure`

C++ struct that is a fixed size in memory.

This object can be `pickle`’d.
```c
struct FourPoints {
    Point points[4];
};
```

**Parameters**

- **point1** (tuple of int) – The first point as an (x, y) ordered pair.
- **point2** (tuple of int) – The second point as an (x, y) ordered pair.
- **point3** (tuple of int) – The third point as an (x, y) ordered pair.
- **point4** (tuple of int) – The fourth point as an (x, y) ordered pair.

```c
class msl.examples.loadlib.cpp32.NPoints
    Bases: _ctypes.Structure

C++ struct that is not a fixed size in memory.

This object cannot be *pickle*’d because it contains a pointer. A 32-bit process and a 64-bit process cannot share a pointer.

```c
struct NPoints {
    int n;
    Point *points;
};
```

- **n**
  Structure/Union member

- **points**
  Structure/Union member

```c
class msl.examples.loadlib.cpp32.Point
    Bases: _ctypes.Structure

C++ struct that is a fixed size in memory.

This object can be *pickle*’d.

```c
struct Point {
    double x;
    double y;
};
```

- **x**
  Structure/Union member

- **y**
  Structure/Union member

**msl.examples.loadlib.cpp64 module**

Communicates with *cpp_lib32* via the *Cpp32* class.
Example of a module that can be executed within a 64-bit Python interpreter which can communicate with a 32-bit library, `cpp_lib32`, that is hosted by a 32-bit Python server, `cpp32`. A 64-bit process cannot load a 32-bit library and therefore inter-process communication is used to interact with a 32-bit library from a 64-bit process.

`Cpp64` is the 64-bit client and `Cpp32` is the 32-bit server for inter-process communication.

```python
class msl.examples.loadlib.cpp64.Cpp64
    Bases: msl.loadlib.client64.Client64

    Communicates with a 32-bit C++ library, `cpp_lib32`.

    This class demonstrates how to communicate with a 32-bit C++ library if an instance of this class is created within a 64-bit Python interpreter.

    add(a, b)
    Add two integers.

    See the corresponding 32-bit `add()` method.

    Parameters
    • a (int) – The first integer.
    • b (int) – The second integer.

    Returns int – The sum of a and b.

    add_or_subtract(a, b, do_addition)
    Add or subtract two floating-point numbers ('double' refers to the C++ data type).

    See the corresponding 32-bit `add_or_subtract()` method.

    Parameters
    • a (float) – The first floating-point number.
    • b (float) – The second floating-point number.
    • do_addition (bool) – Whether to add the numbers.

    Returns float – Either a + b if do_addition is True else a - b.

    circumference(radius, n)
    Estimates the circumference of a circle.

    This method calls the distance_n_points function in `cpp_lib32`.

    See the corresponding 32-bit `circumference()` method.

    Parameters
    • radius (float) – The radius of the circle.
    • n (int) – The number of points to use to estimate the circumference.

    Returns float – The estimated circumference of the circle.

    distance_4_points(points)
    Calculates the total distance connecting 4 Point’s.

    See the corresponding 32-bit `distance_4_points()` method.
```

44 Chapter 1. Contents
Attention: This method does not work with if Cpp64 is running in Python 2. You would have to create the `FourPoints` object in the 32-bit version of `distance_4_points()` because there are issues using the `pickle` module between different major version numbers of Python for `ctypes` objects.

**Parameters** `points (FourPoints)` – Since `points` is a struct that is a fixed size we can pass the `ctypes.Structure` object directly from 64-bit Python to the 32-bit Python. The `ctypes` module on the 32-bit server can load the pickle’d `ctypes.Structure`.

**Returns** `float` – The total distance connecting the 4 Point’s.

**reverse_string_v1 (original)**
Reverse a string (version 1).

In this method Python allocates the memory for the reversed string and passes the string to C++.

See the corresponding 32-bit `reverse_string_v1()` method.

**Parameters** `original (str)` – The original string.

**Returns** `str` – The string reversed.

**reverse_string_v2 (original)**
Reverse a string (version 2).

In this method C++ allocates the memory for the reversed string and passes the string to Python.

See the corresponding 32-bit `reverse_string_v2()` method.

**Parameters** `original (str)` – The original string.

**Returns** `str` – The string reversed.

**scalar_multiply (a, xin)**
Multiply each element in an array by a number.

See the corresponding 32-bit `scalar_multiply()` method.

**Parameters**
- `a (float)` – The scalar value.
- `xin (list of float)` – The array to modify.

**Returns** `list of float` – A new array with each element in `xin` multiplied by `a`.

**subtract (a, b)**
Subtract two floating-point numbers (‘float’ refers to the C++ data type).

See the corresponding 32-bit `subtract()` method.

**Parameters**
- `a (float)` – The first floating-point number.
- `b (float)` – The second floating-point number.

**Returns** `float` – The difference between `a` and `b`.
msl.examples.loadlib.dotnet32 module

A wrapper around a 32-bit .NET library, *dotnet_lib32*.

Example of a server that loads a 32-bit .NET library, *dotnet_lib32.dll* in a 32-bit Python interpreter to host the library. The corresponding *dotnet64* module can be executed by a 64-bit Python interpreter and the *DotNet64* class can send a request to the *DotNet32* class which calls the 32-bit library to execute the request and then return the response from the library.

```python
class msl.examples.loadlib.dotnet32.DotNet32 (host, port, quiet, **kwargs)
    Bases: msl.loadlib.server32.Server32

    Example of a class that is a wrapper around a 32-bit .NET Framework library, *dotnet_lib32.dll*. Python for .NET can handle many native Python data types as input arguments.

    Parameters
    • host (str) – The IP address of the server.
    • port (int) – The port to open on the server.
    • quiet (bool) – Whether to hide *sys.stdout* messages from the server.

    Note: Any class that is a subclass of *Server32* **MUST** provide three arguments in its constructor: *host*, *port* and *quiet* (in that order) and **kwargs. Otherwise the *server32* executable, see *start_server32*, cannot create an instance of the *Server32* subclass.
```

```python
add_integers (a, b)
    Add two integers.
    The corresponding C# code is

    ```csharp
    public int add_integers(int a, int b)
    {
        return a + b;
    }
    ```

    See the corresponding 64-bit *add_integers()* method.

    Parameters
    • a (int) – The first integer.
    • b (int) – The second integer.

    Returns int – The sum of *a* and *b*.

add_multiple (a, b, c, d, e)
    Add multiple integers. *Calls a static method in a static class.*
    The corresponding C# code is

    ```csharp
    public static int add_multiple(int a, int b, int c, int d, int e)
    {
        return a + b + c + d + e;
    }
    ```

    See the corresponding 64-bit *add_multiple()* method.
```
Parameters

- **a (int)** – An integer.
- **b (int)** – An integer.
- **c (int)** – An integer.
- **d (int)** – An integer.
- **e (int)** – An integer.

Returns **int** – The sum of the input arguments.

**add_or_subtract (a, b, do_addition)**
Add or subtract two C# double-precision numbers.

The corresponding C# code is

```csharp
public double add_or_subtract(double a, double b, bool do_addition)
{
    if (do_addition)
    {
        return a + b;
    }
    else
    {
        return a - b;
    }
}
```

See the corresponding 64-bit **add_or_subtract ()** method.

Parameters

- **a (float)** – The first double-precision number.
- **b (float)** – The second double-precision number.
- **do_addition (bool)** – Whether to add the numbers.

Returns **float** – Either **a + b** if **do_addition** is True else **a - b**.

**concatenate (a, b, c, d, e)**
Concatenate strings. *Calls a static method in a static class.*

The corresponding C# code is

```csharp
public static string concatenate(string a, string b, string c, string d, string e)
{
    string res = a + b + c;
    if (d)
    {
        res += e;
    }
    return res;
}
```

See the corresponding 64-bit **concatenate ()** method.
Parameters

- **a** *(str)* – A string.
- **b** *(str)* – A string.
- **c** *(str)* – A string.
- **d** *(bool)* – Whether to include e in the concatenation.
- **e** *(str)* – A string.

**Returns**  *str* – The strings concatenated together.

`divide_floats(a, b)`

Divide two C# floating-point numbers.

The corresponding C# code is

```csharp
public float divide_floats(float a, float b)
{
    return a / b;
}
```

See the corresponding 64-bit `divide_floats()` method.

**Parameters**

- **a** *(float)* – The first number.
- **b** *(float)* – The second number.

**Returns**  *float* – The quotient of \(a / b\).

`get_class_names()`

Returns the class names in the library.

See the corresponding 64-bit `get_class_names()` method.

**Returns**  *list of str* – The names of the classes that are available in `dot-net_lib32.dll`.

`multiply_doubles(a, b)`

Multiply two C# double-precision numbers.

The corresponding C# code is

```csharp
public double multiply_doubles(double a, double b)
{
    return a * b;
}
```

See the corresponding 64-bit `multiply_doubles()` method.

**Parameters**

- **a** *(float)* – The first number.
- **b** *(float)* – The second number.

**Returns**  *float* – The product of \(a \times b\).
**multiply_matrices**(*a1*, *a2*)

Multiply two matrices.

The corresponding C# code is

```csharp
public double[,] multiply_matrices(double[,] A, double[,] B)
{
    int rA = A.GetLength(0);
    int cA = A.GetLength(1);
    int rB = B.GetLength(0);
    int cB = B.GetLength(1);
    double temp = 0;
    double[,] C = new double[rA, cB];
    if (cA != rB)
    {
        Console.WriteLine("matrices can't be multiplied!");
        return new double[0, 0];
    }
    else
    {
        for (int i = 0; i < rA; i++)
        {
            for (int j = 0; j < cB; j++)
            {
                temp = 0;
                for (int k = 0; k < cA; k++)
                {
                    temp += A[i, k] * B[k, j];
                }
                C[i, j] = temp;
            }
        }
        return C;
    }
}
```

See the corresponding 64-bit `multiply_matrices()` method.

**Note:** The CLR package from Python for .NET contains the System namespace from the .NET Framework that is required to create and initialize a 2D matrix.

**Parameters**

- **a1** (list of list of float) – The first matrix.
- **a2** (list of list of float) – The second matrix.

**Returns** list of list of float – The result of *a1* * *a2*.

**reverse_string**(original)

Reverse a string.

The corresponding C# code is

```csharp
public string reverse_string(string original)
{
    // (continues on next page)
}
```
char[] charArray = original.ToCharArray();
Array.Reverse(charArray);
return new string(charArray);

See the corresponding 64-bit reverse_string() method.

Parameters original (str) – The original string.

Returns str – The string reversed.

scalar_multiply (a, xin)
Multiply each element in an array by a number.

The corresponding C# code is

```csharp
public double[] scalar_multiply(double a, double[] xin)
{
    int n = xin.GetLength(0);
    double[] xout = new double[n];
    for (int i = 0; i < n; i++)
    {
        xout[i] = a * xin[i];
    }
    return xout;
}
```

See the corresponding 64-bit scalar_multiply() method.

Parameters

- a (float) – The scalar value.
- xin (list of float) – The array to modify.

Returns list of float – A new array with each element in xin multiplied by a.

msl.examples.loadlib.dotnet64 module

Communicates with a 32-bit .NET library via the DotNet32 class.

Example of a module that can be executed within a 64-bit Python interpreter which can communicate with a 32-bit .NET library, dotnet_lib32.dll that is hosted by a 32-bit Python server, dotnet32. A 64-bit process cannot load a 32-bit library and therefore inter-process communication is used to interact with a 32-bit library from a 64-bit process.

DotNet64 is the 64-bit client and DotNet32 is the 32-bit server for inter-process communication.

class msl.examples.loadlib.dotnet64.DotNet64
    Bases: msl.loadlib.client64.Client64

    Communicates with the 32-bit C# dotnet_lib32.dll library.

    This class demonstrates how to communicate with a 32-bit .NET library if an instance of this class is created within a 64-bit Python interpreter.

    add_integers (a, b)
        Add two integers.
See the corresponding 32-bit `add_integers()` method.

**Parameters**
- `a` (int) – The first integer.
- `b` (int) – The second integer.

**Returns** int – The sum of `a` and `b`.

### add_multiple(a, b, c, d, e)
Add multiple integers. *Calls a static method in a static class.*

See the corresponding 32-bit `add_multiple()` method.

**Parameters**
- `a` (int) – An integer.
- `b` (int) – An integer.
- `c` (int) – An integer.
- `d` (int) – An integer.
- `e` (int) – An integer.

**Returns** int – The sum of the input arguments.

### add_or_subtract(a, b, do_addition)
Add or subtract two C# double-precision numbers.

See the corresponding 32-bit `add_or_subtract()` method.

**Parameters**
- `a` (float) – The first double-precision number.
- `b` (float) – The second double-precision number.
- `do_addition` (bool) – Whether to add the numbers.

**Returns** float – Either `a + b` if `do_addition` is True else `a - b`.

### concatenate(a, b, c, d, e)
Concatenate strings. *Calls a static method in a static class.*

See the corresponding 32-bit `concatenate()` method.

**Parameters**
- `a` (str) – A string.
- `b` (str) – A string.
- `c` (str) – A string.
- `d` (bool) – Whether to include `e` in the concatenation.
- `e` (str) – A string.

**Returns** str – The strings concatenated together.

### divide_floats(a, b)
Divide two C# floating-point numbers.

See the corresponding 32-bit `divide_floats()` method.
Parameters

- a (float) – The first number.
- b (float) – The second number.

Returns float – The quotient of $a / b$.

get_class_names()
Returns list of str – The names of the classes that are available in dot-net_lib32.dll.

multiply_doubles(a, b)
Multiply two C# double-precision numbers.

Parameters

- a (float) – The first number.
- b (float) – The second number.

Returns float – The product of $a * b$.

multiply_matrices(al, a2)
Multiply two matrices.

Parameters

- al (list of list of float) – The first matrix.
- a2 (list of list of float) – The second matrix.

Returns list of list of float – The result of $a1 * a2$.

reverse_string(original)
Reverse a string.

Parameters original (str) – The original string.

Returns str – The string reversed.

scalar_multiply(a, xin)
Multiply each element in an array by a number.

Parameters

- a (float) – The scalar value.
- xin (list of float) – The array to modify.

Returns list of float – A new array with each element in xin multiplied by a.
msl.examples.loadlib.echo32 module

An example of a 32-bit echo server.

Example of a server that is executed by a 32-bit Python interpreter that receives requests from the corresponding echo64 module which can be run by a 64-bit Python interpreter.

Echo32 is the 32-bit server class and Echo64 is the 64-bit client class. These echo classes do not actually communicate with a shared library. The point of these echo classes is to show that a Python data type in a 64-bit process appears as the same data type in the 32-bit process and vice versa.

class msl.examples.loadlib.echo32.Echo32 (host, port, quiet, **kwargs)

Bases: msl.loadlib.server32.Server32

Example of a server class that illustrates that Python data types are preserved when they are sent from the Echo64 client to the server.

Parameters

• **host (str)** – The IP address of the server.
• **port (int)** – The port to open on the server.
• **quiet (bool)** – Whether to hide sys.stdout messages from the server.

Note: Any class that is a subclass of Server32 MUST provide three arguments in its constructor: host, port and quiet (in that order) and **kwargs. Otherwise the server32 executable, see start_server32, cannot create an instance of the Server32 subclass.

received_data (*args, **kwargs)

Process a request from the send_data() method from the 64-bit client.

Parameters

• **args** – The arguments.
• **kwargs** – The keyword arguments.

Returns tuple – The args and kwargs that were received.

msl.examples.loadlib.echo64 module

An example of a 64-bit echo client.

Example of a client that can be executed by a 64-bit Python interpreter that sends requests to the corresponding echo32 module which is executed by a 32-bit Python interpreter.

Echo32 is the 32-bit server class and Echo64 is the 64-bit client class. These echo classes do not actually communicate with a shared library. The point of these echo classes is to show that a Python data type in a 64-bit process appears as the same data type in the 32-bit process and vice versa.

class msl.examples.loadlib.echo64.Echo64 (quiet=False)

Bases: msl.loadlib.client64.Client64

Example of a client class that illustrates that Python data types are preserved when they are sent to the Echo32 server and back again.
Parameters quiet (bool, optional) – Whether to hide sys.stdout messages from the client and from the server.

send_data (*args, **kwargs)
Send a request to execute the received_data() method on the 32-bit server.

Parameters

• *args – The arguments that the received_data() method requires.

• **kwargs – The keyword arguments that the received_data() method requires.

Returns tuple – The args and kwargs that were returned from received_data().

msl.examples.loadlib.fortran32 module

A wrapper around a 32-bit FORTRAN library, fortran_lib32.

Example of a server that loads a 32-bit FORTRAN library, fortran_lib32, in a 32-bit Python interpreter to host the library. The corresponding fortran64 module can be executed by a 64-bit Python interpreter and the Fortran64 class can send a request to the Fortran32 class which calls the 32-bit library to execute the request and then return the response from the library.

class msl.examples.loadlib.fortran32.Fortran32 (host, port, quiet, **kwargs)
Bases: msl.loadlib.server32.Server32

A wrapper around a 32-bit FORTRAN library, fortran_lib32.

This class demonstrates how to send/receive various data types to/from a 32-bit FORTRAN library via ctypes. For a summary of the FORTRAN data types see here.

Parameters

• host (str) – The IP address of the server.

• port (int) – The port to open on the server.

• quiet (bool) – Whether to hide sys.stdout messages from the server.

Note: Any class that is a subclass of Server32 MUST provide three arguments in its constructor: host, port and quiet (in that order) and **kwargs. Otherwise the server32 executable, see start_server32, cannot create an instance of the Server32 subclass.

add_1D_arrays (a1, a2)
Perform an element-wise addition of two 1D double-precision arrays.

The corresponding FORTRAN code is

```fortran
subroutine add_1d_arrays(a, in1, in2, n)
!DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'add_1d_arrays' :: add_1d_arrays
  implicit none
  integer(4) :: n ! the length of the input arrays
  double precision :: in1(n), in2(n) ! the arrays to add
```
See the corresponding 64-bit `add_1D_arrays()` method.

**Parameters**

- `a1` ([list of float]) – The first array.
- `a2` ([list of float]) – The second array.

**Returns** list of float – The element-wise addition of `a1 + a2`.

### add_or_subtract `(a, b, do_addition)`

Add or subtract two integers.

The corresponding FORTRAN code is

```fortran
function add_or_subtract(a, b, do_addition) result(value)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'add_or_subtract' :: add_or_subtract
  implicit none
  logical :: do_addition
  integer(4) :: a, b, value
  if (do_addition) then
    value = a + b
  else
    value = a - b
  endif
end function add_or_subtract
```

See the corresponding 64-bit `add_or_subtract()` method.

**Parameters**

- `a` (int) – The first integer.
- `b` (int) – The second integer.
- `do_addition` (bool) – Whether to add the numbers.

**Returns** int – Either `a + b` if `do_addition` is True else `a - b`.

### besselJ0 `(x)`

Compute the Bessel function of the first kind of order 0 of `x`.

The corresponding FORTRAN code is

```fortran
function besselj0(x) result(val)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'besselj0' :: besselj0
  double precision :: x, val
  val = BESSEL_J0(x)
end function besselj0
```

See the corresponding 64-bit `besselJ0()` method.

**Parameters** `x` (float) – The value to compute BESSEL_J0 of.
Returns float – The value of BESSEL_J0(x).

factorial(n)
Compute the n’th factorial.

The corresponding FORTRAN code is

```fortran
function factorial(n) result(value)
    !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'factorial' :: factorial
    implicit none
    integer(1) :: n
    integer(4) :: i
    double precision value
    if (n < 0) then
        value = 0.d0
        print *, "Cannot compute the factorial of a negative number", n
    else
        value = 1.d0
        do i = 2, n
            value = value * i
        enddo
    endif
end function factorial
```

See the corresponding 64-bit factorial() method.

Parameters n (int) – The integer to computer the factorial of. The maximum allowed value is 127.

Returns float – The factorial of n.

is_positive(a)
Returns whether the value of the input argument is > 0.

The corresponding FORTRAN code is

```fortran
function is_positive(a) result(value)
    !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'is_positive' :: is_positive
    implicit none
    logical :: value
    real(8) :: a
    value = a > 0.d0
end function is_positive
```

See the corresponding 64-bit is_positive() method.

Parameters a (float) – A double-precision number.

Returns bool – Whether the value of a is > 0.

matrix_multiply(a1, a2)
Multiply two matrices.

The corresponding FORTRAN code is

```fortran
subroutine matrix_multiply(a, a1, r1, c1, a2, r2, c2)
    !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'matrix_multiply' :: matrix_multiply
```

(continues on next page)
**Note:** FORTRAN stores multi-dimensional arrays in column-major order, as opposed to row-major order for C (Python) arrays. Therefore the input matrices need to be transposed before sending the matrices to FORTRAN and also the result needs to be transposed before returning the result.

See the corresponding 64-bit `matrix_multiply() method.

### Parameters

- **a1** (list of list of float) – The first matrix.
- **a2** (list of list of float) – The second matrix.

### Returns

list of list of float – The result of \(a1 \times a2\).

### multiply_float32 \((a, b)\)

Multiply two FORTRAN floating-point numbers.

The corresponding FORTRAN code is

```fortran
function multiply_float32(a, b) result(value)
   !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'multiply_float32' ::
   implicit none
   real(4) :: a, b, value
   value = a * b
end function multiply_float32
```

See the corresponding 64-bit `multiply_float32()` method.

### Parameters

- **a** (float) – The first floating-point number.
- **b** (float) – The second floating-point number.

### Returns

float – The product of \(a\) and \(b\).

### multiply_float64 \((a, b)\)

Multiply two FORTRAN double-precision numbers.

The corresponding FORTRAN code is

```fortran
function multiply_float64(a, b) result(value)
   !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'multiply_float64' ::
   implicit none
   real(8) :: a, b, value
   value = a * b
end function multiply_float64
```
See the corresponding 64-bit `multiply_float64()` method.

**Parameters**

- `a` (*float*) – The first double-precision number.
- `b` (*float*) – The second double-precision number.

**Returns** `float` – The product of `a` and `b`.

### reverse_string(*original*)

Reverse a string.

The corresponding FORTRAN code is

```fortran
subroutine reverse_string(original, n, reversed) !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'reverse_string' :: reverse_string
  !DEC$ ATTRIBUTES REFERENCE :: original, reversed
  implicit none
  integer :: i, n
  character(len=n) :: original, reversed
  do i = 1, n
    reversed(i:i) = original(n-i+1:n-i+1)
  end do
end subroutine reverse_string
```

See the corresponding 64-bit `reverse_string()` method.

**Parameters** `original` (*str*) – The original string.

**Returns** `str` – The string reversed.

### standard_deviation(*data*)

Compute the standard deviation.

The corresponding FORTRAN code is

```fortran
function standard_deviation(a, n) result(var) !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'standard_deviation' :: standard_deviation
  integer :: n ! the length of the array
  double precision :: var, a(n)
  var = SUM(a)/SIZE(a) ! SUM is a built-in fortran function
  var = SQRT(SUM((a-var)**2)/(SIZE(a)-1.0))
end function standard_deviation
```

See the corresponding 64-bit `standard_deviation()` method.

**Parameters** `data` (*list of float*) – The data to compute the standard deviation of.

**Returns** `float` – The standard deviation of `data`.
sum_16bit \((a, b)\)
Add two 16-bit signed integers

Python only has one \texttt{int} data type to represent integer values. The \texttt{sum\_16bit()} method converts the data types of \(a\) and \(b\) to be \texttt{ctypes.c\_int16}.

The corresponding FORTRAN code is

\begin{verbatim}
function sum_16bit(a, b) result(value)
  !DEC$ ATTIBUTES DLLEXPORT, ALIAS:'sum_16bit' :: sum_16bit
  implicit none
  integer(2) :: a, b, value
  value = a + b
end function sum_16bit
\end{verbatim}

See the corresponding 64-bit \texttt{sum\_16bit()} method.

Parameters

- \texttt{a (int)} – The first 16-bit signed integer.
- \texttt{b (int)} – The second 16-bit signed integer.

Returns \texttt{int} – The sum of \(a\) and \(b\).

sum_32bit \((a, b)\)
Add two 32-bit signed integers.

Python only has one \texttt{int} data type to represent integer values. The \texttt{sum\_32bit()} method converts the data types of \(a\) and \(b\) to be \texttt{ctypes.c\_int32}.

The corresponding FORTRAN code is

\begin{verbatim}
function sum_32bit(a, b) result(value)
  !DEC$ ATTIBUTES DLLEXPORT, ALIAS:'sum_32bit' :: sum_32bit
  implicit none
  integer(4) :: a, b, value
  value = a + b
end function sum_32bit
\end{verbatim}

See the corresponding 64-bit \texttt{sum\_32bit()} method.

Parameters

- \texttt{a (int)} – The first 32-bit signed integer.
- \texttt{b (int)} – The second 32-bit signed integer.

Returns \texttt{int} – The sum of \(a\) and \(b\).

sum_64bit \((a, b)\)
Add two 64-bit signed integers.

Python only has one \texttt{int} data type to represent integer values. The \texttt{sum\_64bit()} method converts the data types of \(a\) and \(b\) to be \texttt{ctypes.c\_int64}.

The corresponding FORTRAN code is

\begin{verbatim}
function sum_64bit(a, b) result(value)
  !DEC$ ATTIBUTES DLLEXPORT, ALIAS:'sum_64bit' :: sum_64bit
  implicit none
end function sum_64bit
\end{verbatim}
integer(8) :: a, b, value
value = a + b
end function sum_64bit

See the corresponding 64-bit `sum_64bit()` method.

**Parameters**

- **a** (int) – The first 64-bit signed integer.
- **b** (int) – The second 64-bit signed integer.

**Returns** int – The sum of `a` and `b`.

`sum_8bit(a, b)`

Add two 8-bit signed integers.

Python only has one `int` data type to represent integer values. The `sum_8bit()` method converts the data types of `a` and `b` to be `ctypes.c_int8`.

The corresponding FORTRAN code is

```fortran
function sum_8bit(a, b) result(value)
!DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'sum_8bit' :: sum_8bit
implicit none
integer(1) :: a, b, value
value = a + b
end function sum_8bit
```

See the corresponding 64-bit `sum_8bit()` method.

**Parameters**

- **a** (int) – The first 8-bit signed integer.
- **b** (int) – The second 8-bit signed integer.

**Returns** int – The sum of `a` and `b`.

**msl.examples.loadlib.fortran64 module**

Communicates with `fortran_lib32` via the `Fortran32` class.

Example of a module that can be executed within a 64-bit Python interpreter which can communicate with a 32-bit library, `fortran_lib32`, that is hosted by a 32-bit Python server, `fortran32`. A 64-bit process cannot load a 32-bit library and therefore inter-process communication is used to interact with a 32-bit library from a 64-bit process.

`Fortran64` is the 64-bit client and `Fortran32` is the 32-bit server for inter-process communication.

**class msl.examples.loadlib.fortran64.Fortran64**

**Bases:** `msl.loadlib.client64.Client64`

Communicates with the 32-bit FORTRAN `fortran_lib32` library.

This class demonstrates how to communicate with a 32-bit FORTRAN library if an instance of this class is created within a 64-bit Python interpreter.
add_1D_arrays \((a1, a2)\)
Perform an element-wise addition of two 1D double-precision arrays.
See the corresponding 32-bit \texttt{add\_1D\_arrays()} method.

Parameters

\begin{itemize}
  \item \texttt{a1 (list of float)} – The first array.
  \item \texttt{a2 (list of float)} – The second array.
\end{itemize}

Returns \texttt{list of float} – The element-wise addition of \(a1 + a2\).

add_or_subtract \((a, b, do\_addition)\)
Add or subtract two integers.
See the corresponding 32-bit \texttt{add\_or\_subtract()} method.

Parameters

\begin{itemize}
  \item \texttt{a (int)} – The first integer.
  \item \texttt{b (int)} – The second integer.
  \item \texttt{do\_addition (bool)} – Whether to add the numbers.
\end{itemize}

Returns \texttt{int} – Either \(a + b\) if \texttt{do\_addition} is \texttt{True} else \(a - b\).

besselJ0 \((x)\)
Compute the Bessel function of the first kind of order 0 of \(x\).
See the corresponding 32-bit \texttt{besselJ0()} method.

Parameters \texttt{x (float)} – The value to compute BESSEL\_J0 of.

Returns \texttt{float} – The value of BESSEL\_J0 \((x)\).

factorial \((n)\)
Compute the \(n\)'th factorial.
See the corresponding 32-bit \texttt{factorial()} method.

Parameters \texttt{n (int)} – The integer to compute the factorial of. The maximum allowed value is 127.

Returns \texttt{float} – The factorial of \(n\).

is_positive \((a)\)
Returns whether the value of the input argument is > 0.
See the corresponding 32-bit \texttt{is\_positive()} method.

Parameters \texttt{a (float)} – A double-precision number.

Returns \texttt{bool} – Whether the value of \(a\) is > 0.

matrix_multiply \((a1, a2)\)
Multiply two matrices.
See the corresponding 32-bit \texttt{matrix\_multiply()} method.

Parameters

\begin{itemize}
  \item \texttt{a1 (list of list of float)} – The first matrix.
  \item \texttt{a2 (list of list of float)} – The second matrix.
\end{itemize}
Returns list of list of float – The result of \( a1 \times a2 \).

\textbf{multiply\_float32}(a, b)
Send a request to multiply two FORTRAN floating-point numbers.
See the corresponding 32-bit \texttt{multiply\_float32()} method.

\begin{itemize}
  \item \texttt{a (float)} – The first floating-point number.
  \item \texttt{b (float)} – The second floating-point number.
\end{itemize}

Returns float – The product of \( a \) and \( b \).

\textbf{multiply\_float64}(a, b)
Send a request to multiply two FORTRAN double-precision numbers.
See the corresponding 32-bit \texttt{multiply\_float64()} method.

\begin{itemize}
  \item \texttt{a (float)} – The first double-precision number.
  \item \texttt{b (float)} – The second double-precision number.
\end{itemize}

Returns float – The product of \( a \) and \( b \).

\textbf{reverse\_string}(original)
Reverse a string.
See the corresponding 32-bit \texttt{reverse\_string()} method.

\begin{itemize}
  \item \texttt{original (str)} – The original string.
\end{itemize}

Returns str – The string reversed.

\textbf{standard\_deviation}(data)
Compute the standard deviation.
See the corresponding 32-bit \texttt{standard\_deviation()} method.

\begin{itemize}
  \item \texttt{data (list of float)} – The data to compute the standard deviation of.
\end{itemize}

Returns float – The standard deviation of \( data \).

\textbf{sum\_16bit}(a, b)
Send a request to add two 16-bit signed integers.
See the corresponding 32-bit \texttt{sum\_16bit()} method.

\begin{itemize}
  \item \texttt{a (int)} – The first 16-bit signed integer.
  \item \texttt{b (int)} – The second 16-bit signed integer.
\end{itemize}

Returns int – The sum of \( a \) and \( b \).

\textbf{sum\_32bit}(a, b)
Send a request to add two 32-bit signed integers.
See the corresponding 32-bit \texttt{sum\_32bit()} method.

\begin{itemize}
  \item \texttt{a (int)} – The first 32-bit signed integer.
  \item \texttt{b (int)} – The second 32-bit signed integer.
\end{itemize}

Returns int – The sum of \( a \) and \( b \).
• \( a \) (int) – The first 32-bit signed integer.
• \( b \) (int) – The second 32-bit signed integer.

Returns int – The sum of \( a \) and \( b \).

\textbf{sum\_64bit} \((a, b)\)

Send a request to add two 64-bit signed integers.

See the corresponding 32-bit \texttt{sum\_64bit()} method.

Parameters

• \( a \) (int) – The first 64-bit signed integer.
• \( b \) (int) – The second 64-bit signed integer.

Returns int – The sum of \( a \) and \( b \).

\textbf{sum\_8bit} \((a, b)\)

Send a request to add two 8-bit signed integers.

See the corresponding 32-bit \texttt{sum\_8bit()} method.

Parameters

• \( a \) (int) – The first 8-bit signed integer.
• \( b \) (int) – The second 8-bit signed integer.

Returns int – The sum of \( a \) and \( b \).

\texttt{msl.examples.loadlib.kernel32} module

A wrapper around the 32-bit Windows \texttt{kernel32.dll} library.

Example of a server that loads a 32-bit Windows library, \texttt{kernel32.dll}, in a 32-bit Python interpreter to host the library. The corresponding \texttt{kernel64} module can be executed by a 64-bit Python interpreter and the \texttt{Kernel64} class can send a request to the \texttt{Kernel32} class which calls the 32-bit library to execute the request and then return the response from the library.

\texttt{Kernel32} is the 32-bit server and \texttt{Kernel64} is the 64-bit client for inter-process communication.

\textbf{Note}: The \texttt{kernel32.dll} library is a standard Windows library and therefore this example is only valid on a Windows computer.

\texttt{msl.examples.loadlib.kernel32.Kernel32} \((host, port, quiet, \textit{**kwargs})\)

Bases: \texttt{msl.loadlib.server32.Server32}

Example of a class that is a wrapper around the Windows 32-bit \texttt{kernel32.dll} library.

Parameters

• \texttt{host} (str) – The IP address of the server.
• \texttt{port} (int) – The port to open on the server.
• \texttt{quiet} (bool) – Whether to hide \texttt{sys.stdout} messages from the server.
Note: Any class that is a subclass of Server32 MUST provide three arguments in its constructor: host, port and quiet (in that order) and **kwargs. Otherwise the server32 executable, see start_server32, cannot create an instance of the Server32 subclass.

get_time()

Calls the kernel32.GetLocalTime function to get the current date and time.

See the corresponding 64-bit get_local_time() method.

Returns datetime – The current date and time.

class msl.examples.loadlib.kernel32.SystemTime

Bases: _ctypes.Structure

Example of creating a ctypes.Structure.

See SYSTEMTIME for a description of the struct.

WORD

alias of ctypes.c_ushort

wDay

Structure/Union member

wDayOfWeek

Structure/Union member

wHour

Structure/Union member

wMilliseconds

Structure/Union member

wMinute

Structure/Union member

wMonth

Structure/Union member

wSecond

Structure/Union member

wYear

Structure/Union member

msl.examples.loadlib.kernel64 module

Communicates with kernel32.dll via the Kernel32 class.

Example of a module that can be executed by a 64-bit Python interpreter which can communicate with a Windows 32-bit library, kernel32.dll, that is hosted by the corresponding 32-bit Python server, kernel32.

Kernel64 is the 64-bit client and Kernel32 is the 32-bit server for inter-process communication.
**Note:** The `kernel32.dll` library is a standard Windows library and therefore this example is only valid on a computer running Windows.

```python
class msl.examples.loadlib.kernel64.Kernel64
    Bases: msl.loadlib.client64.Client64

    Example of a class that can communicate with the 32-bit `kernel32.dll` library.
    This class demonstrates how to communicate with a Windows 32-bit library if an instance of this class is created within a 64-bit Python interpreter.

    `get_local_time()`
    Sends a request to the 32-bit server, `Kernel32`, to execute the `kernel32.GetLocalTime` function to get the current date and time.
    See the corresponding 32-bit `get_time()` method.

    Returns `datetime` – The current date and time.
```

**msl.examples.loadlib.labview32 module**

A wrapper around a 32-bit LabVIEW library, `labview_lib32`.

**Attention:** This example requires that the appropriate LabVIEW Run-Time Engine is installed and that the operating system is Windows.

Example of a server that loads a 32-bit shared library, `labview_lib`, in a 32-bit Python interpreter to host the library. The corresponding `labview64` module can be executed by a 64-bit Python interpreter and the `Labview64` class can send a request to the `Labview32` class which calls the 32-bit library to execute the request and then return the response from the library.

```python
class msl.examples.loadlib.labview32.Labview32(host, port, quiet, **kwargs)
    Bases: msl.loadlib.server32.Server32

    A wrapper around the 32-bit LabVIEW library, `labview_lib32`.

    Parameters
    • `host` (str) – The IP address of the server.
    • `port` (int) – The port to open on the server.
    • `quiet` (bool) – Whether to hide `sys.stdout` messages from the server.
```

**stdev**
Calculates the mean, variance and standard deviation of the values in the input `x`.

See the corresponding 64-bit `stdev()` method.
Parameters

• **x** ([list](#) of [float](#)) – The data to calculate the mean, variance and standard deviation of.

• **weighting** ([int](#), [optional](#)) – Whether to calculate the [sample](#), `weighting = 0`, or the [population](#), `weighting = 1`, standard deviation and variance.

Returns

• [float](#) – The mean.

• [float](#) – The variance.

• [float](#) – The standard deviation.

**msl.examples.loadlib.labview64 module**

Communicates with `labview_lib32` via the `Labview32` class.

**Attention:** This example requires that the appropriate LabVIEW Run-Time Engine is installed and that the operating system is Windows.

Example of a module that can be executed within a 64-bit Python interpreter which can communicate with a 32-bit library, `labview_lib32`, that is hosted by a 32-bit Python server, `labview32`. A 64-bit process cannot load a 32-bit library and therefore inter-process communication is used to interact with a 32-bit library from a 64-bit process.

`Labview64` is the 64-bit client and `Labview32` is the 32-bit server for inter-process communication.

```python
class msl.examples.loadlib.labview64.Labview64
    Bases: msl.loadlib.client64.Client64

    Communicates with a 32-bit LabVIEW library, `labview_lib32`.

    This class demonstrates how to communicate with a 32-bit LabVIEW library if an instance of this
class is created within a 64-bit Python interpreter.

def stdev(x, weighting=0)
    Calculates the mean, variance and standard deviation of the values in the input x.

    See the corresponding 32-bit `stdev()` method.

    Parameters

    • **x** ([list](#) of [float](#)) – The data to calculate the mean, variance and standard deviation of.

    • **weighting** ([int](#), [optional](#)) – Whether to calculate the [sample](#), `weighting = 0`, or the [population](#), `weighting = 1`, standard deviation and variance.

    Returns

    • [float](#) – The mean.
```

66 Chapter 1. Contents
• float – The variance.
• float – The standard deviation.

1.5 Source code for the example libraries

The source code for the example shared libraries that are included with the MSL-LoadLib package can be found via the links below.

1.5.1 C++

Source code for the example C++ library.

cpp_lib.h

```c
// cpp_lib.h
// Contains the declaration of exported functions.
/
#ifdef _MSC_VER
    #define EXPORT __declspec(dllexport)
#elif defined(__GNUC__)
    #define EXPORT __attribute__((visibility("default")))
#else
    #error "Unknown EXPORT semantics"
#endif

struct Point {
    double x;
    double y;
};

struct FourPoints {
    Point points[4];
};

struct NPoints {
    int n;
    Point *points;
};

extern "C" {

    // a + b
    EXPORT int add(int a, int b);

    // a - b
    EXPORT float subtract(float a, float b);

    // IF do_addition IS TRUE THEN a + b ELSE a - b

    (continues on next page)
```
EXPORT double add_or_subtract(double a, double b, bool do_addition);

// multiply each element in 'x' by 'a'
EXPORT void scalar_multiply(double a, double* xin, int n, double* xout);

// reverse a string
EXPORT void reverse_string_v1(const char* original, int n, char* reversed);

// reverse a string and return it
EXPORT char* reverse_string_v2(char* original, int n);

// calculate the total distance connecting 4 Points
EXPORT double distance_4_points(FourPoints p);

// calculate the total distance connecting N Points
EXPORT double distance_n_points(NPoints p);

}
void reverse_string_v1(const char* original, int n, char* reversed) {
    for (int i = 0; i < n; i++) {
        reversed[i] = original[n-i-1];
    }
}

char* reverse_string_v2(char* original, int n) {
    char* reversed = new char[n];
    for (int i = 0; i < n; i++) {
        reversed[i] = original[n - i - 1];
    }
    return reversed;
}

// this function is not exported to the shared library
double distance(Point p1, Point p2) {
    double d = sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));
    return d;
}

double distance_4_points(FourPoints p) {
    double d = distance(p.points[0], p.points[3]);
    for (int i = 1; i < 4; i++) {
        d += distance(p.points[i], p.points[i-1]);
    }
    return d;
}

double distance_n_points(NPoints p) {
    if (p.n < 2) {
        return 0.0;
    }
    double d = distance(p.points[0], p.points[p.n-1]);
    for (int i = 1; i < p.n; i++) {
        d += distance(p.points[i], p.points[i-1]);
    }
    return d;
}

1.5.2 FORTRAN

Source code for the example FORTRAN library.

fortran_lib.f90

! fortran_lib.f90
!
! Basic examples of passing different data types to a FORTRAN function and _subroutine.
!
! Compiled in Windows using:
! gfortran -fno-underscoring -fPIC fortran_lib.f90 -static -shared -c -o fortran_lib64.dll
! return the sum of two 8-bit signed integers
function sum_8bit(a, b) result(value)
   !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'sum_8bit' :: sum_8bit
   implicit none
   integer(1) :: a, b, value
   value = a + b
end function sum_8bit

! return the sum of two 16-bit signed integers
function sum_16bit(a, b) result(value)
   !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'sum_16bit' :: sum_16bit
   implicit none
   integer(2) :: a, b, value
   value = a + b
end function sum_16bit

! return the sum of two 32-bit signed integers
function sum_32bit(a, b) result(value)
   !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'sum_32bit' :: sum_32bit
   implicit none
   integer(4) :: a, b, value
   value = a + b
end function sum_32bit

! return the sum of two 64-bit signed integers
function sum_64bit(a, b) result(value)
   !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'sum_64bit' :: sum_64bit
   implicit none
   integer(8) :: a, b, value
   value = a + b
end function sum_64bit

! return the product of two 32-bit floating point numbers
function multiply_float32(a, b) result(value)
   !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'multiply_float32' :: multiply_float32
   implicit none
   real(4) :: a, b, value
   value = a * b
end function multiply_float32

! return the product of two 64-bit floating point numbers
function multiply_float64(a, b) result(value)
   !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'multiply_float64' :: multiply_float64
   implicit none
   real(8) :: a, b, value
   value = a * b
end function multiply_float64
! return True if 'a' > 0 else False
function is_positive(a) result(value)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'is_positive' :: is_positive
  implicit none
  logical :: value
  real(8) :: a
  value = a > 0.d0
end function is_positive

! if do_addition is True return a+b otherwise return a-b
function add_or_subtract(a, b, do_addition) result(value)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'add_or_subtract' :: add_or_subtract
  implicit none
  logical :: do_addition
  integer(4) :: a, b, value
  if (do_addition) then
    value = a + b
  else
    value = a - b
  endif
end function add_or_subtract

! compute the n'th factorial of a 8-bit signed integer, return a double-precision number
function factorial(n) result(value)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'factorial' :: factorial
  implicit none
  integer(1) :: n
  integer(4) :: i
  double precision value
  if (n < 0) then
    value = 0.d0
    print *, "Cannot compute the factorial of a negative number", n
  else
    value = 1.d0
    do i = 2, n
      value = value * i
    enddo
  endif
end function factorial

! calculate the standard deviation of an array.
function standard_deviation(a, n) result(var)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'standard_deviation' :: standard_deviation
  integer :: n ! the length of the array
  double precision :: var, a(n)
  var = SUM(a)/SIZE(a) ! SUM is a built-in fortran function
  var = SQRT(SUM((a-var)**2)/SIZE(a)-1.0))
end function standard_deviation
! compute the Bessel function of the first kind of order 0 of x

function besselj0(x) result(val)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'besselj0' :: besselj0
  double precision :: x, val
  val = BESSEL_J0(x)
end function besselJ0

! reverse a string, 'n' is the length of the original string

subroutine reverse_string(original, n, reversed)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'reverse_string' :: reverse_string
  !DEC$ ATTRIBUTES REFERENCE :: original, reversed
  implicit none
  integer :: i, n
  character(len=n) :: original, reversed
  do i = 1, n
    reversed(i:i) = original(n-i+1:n-i+1)
  end do
end subroutine reverse_string

! element-wise addition of two 1D double-precision arrays

subroutine add_1d_arrays(a, in1, in2, n)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'add_1d_arrays' :: add_1d_arrays
  implicit none
  integer(4) :: n ! the length of the input arrays
  double precision :: in1(n), in2(n) ! the arrays to add (element-wise)
  double precision :: a(n) ! the array that will contain the element-
  !wise sum
  a(:) = in1(:) + in2(:)
end subroutine add_1D_arrays

! multiply two 2D, double-precision arrays.
! NOTE: multi-dimensional arrays are column-major order in FORTRAN,
! whereas C (Python) is row-major order.

subroutine matrix_multiply(a, a1, r1, c1, a2, r2, c2)
  !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'matrix_multiply' :: matrix_multiply
  implicit none
  integer(4) :: r1, c1, r2, c2 ! the dimensions of the input arrays
  double precision :: a1(r1,c1), a2(r2,c2) ! the arrays to multiply
  double precision :: a(r1,c2) ! resultant array
  a = MATMUL(a1, a2)
end subroutine matrix_multiply

1.5.3 Microsoft .NET Framework

Source code for the example C# library.
dotnet_lib.cs

```csharp
// dotnet_lib.cs
// Examples that show how to pass various data types between Python and a C
→ $# library.

using System;

// The DotNetMSL namespace contains two classes: BasicMath,
˓→ ArrayManipulation
namespace DotNetMSL
{
    // A class that is part of the DotNetMSL namespace
    public class BasicMath
    {
        public int add_integers(int a, int b)
        {
            return a + b;
        }

        public float divide_floats(float a, float b)
        {
            return a / b;
        }

        public double multiply_doubles(double a, double b)
        {
            return a * b;
        }

        public double add_or_subtract(double a, double b, bool do_addition)
        {
            if (do_addition)
            {
                return a + b;
            }
            else
            {
                return a - b;
            }
        }
    }

    // A class that is part of the DotNetMSL namespace
    public class ArrayManipulation
    {
        public double[] scalar_multiply(double a, double[] xin)
        {
            int n = xin.GetLength(0);
            double[] xout = new double[n];
            for (int i = 0; i < n; i++)
            {
                xout[i] = a * xin[i];
            }
        }
    }
}
```
return xout;
}

public double[,] multiply_matrices(double[,] A, double[,] B)
{
    int rA = A.GetLength(0);
    int cA = A.GetLength(1);
    int rB = B.GetLength(0);
    int cB = B.GetLength(1);
    double temp = 0;
    double[,] C = new double[rA, cB];
    if (cA != rB)
    {
        Console.WriteLine("matrices can't be multiplied!");
        return new double[0, 0];
    }
    else
    {
        for (int i = 0; i < rA; i++)
        {
            for (int j = 0; j < cB; j++)
            {
                temp = 0;
                for (int k = 0; k < cA; k++)
                {
                    temp += A[i, k] * B[k, j];
                }
                C[i, j] = temp;
            }
        }
        return C;
    }
}

// A class that is not part of the DotNetMSL namespace
public class StringManipulation
{
    public string reverse_string(string original)
    {
        char[] charArray = original.ToCharArray();
        Array.Reverse(charArray);
        return new string(charArray);
    }
}

// A static class cannot be initiated and therefore the object is available in
// Python as a System.RuntimeType object.
public static class StaticClass
{
public static int add_multiple(int a, int b, int c, int d, int e) {
    return a + b + c + d + e;
}

public static string concatenate(string a, string b, string c, bool d, ...
string e) {
    string res = a + b + c;
    if (d) {
        res += e;
    }
    return res;
}

1.5.4 LabVIEW

Source code for the example LabVIEW library.

labview_lib.vi

labview_lib.h

#include "extcode.h"
#pragma pack(push)
#pragma pack(1)
#ifdef __cplusplus
extern "C" {
#endif
typedef uint16_t Enum;
#define Enum_Sample 0
#define Enum_Population 1

/*

(continues on next page)
* stdev
*/
void __cdecl stdev(double X[], int32_t lenX, Enum WeightingSample,
        double *mean, double *variance, double *standardDeviation);
MgErr __cdecl LVDLLStatus(char *errStr, int errStrLen, void *module);

1.5.5 Java

Source code for the example .class and .jar libraries.

Example .class file

The following file is compiled to Trig.class byte code.

Trig.java

/*
 * Compile with JDK 6 for maximal compatibility with Py4J
 * javac Trig.java
 */

public class Trig {
    /** Returns the trigonometric cosine of an angle. */
    static public double cos(double x) {
        return Math.cos(x);
    }

    /** Returns the hyperbolic cosine of a value. */
    static public double cosh(double x) {
        return Math.cosh(x);
    }

    /** Returns the arc cosine of a value, [0.0, pi]. */
    static public double acos(double x) {
        return Math.acos(x);
    }

    /** Returns the trigonometric sine of an angle. */
    static public double sin(double x) {
        return Math.sin(x);
    }
}
/** Returns the hyperbolic sine of a value. */
static public double sinh(double x) {
    return Math.sinh(x);
}

/** Returns the arc sine of a value, [-pi/2, pi/2]. */
static public double asin(double x) {
    return Math.asin(x);
}

/** Returns the trigonometric tangent of an angle. */
static public double tan(double x) {
    return Math.tan(x);
}

/** Returns the hyperbolic tangent of a value. */
static public double tanh(double x) {
    return Math.tanh(x);
}

/** Returns the arc tangent of a value; [-pi/2, pi/2]. */
static public double atan(double x) {
    return Math.atan(x);
}

/** Returns the angle theta from the conversion of rectangular coordinates (x, y) to polar coordinates (r, theta). */
static public double atan2(double y, double x) {
    return Math.atan2(y, x);
}

Example .jar file
The following classes are included in the nz.msl.examples package in java_lib.jar.

MathUtils.java

```java
package nz.msl.examples;

public class MathUtils {
    /** Generate a random number between [0, 1) */
    static public double random() {
        return Math.random();
    }
}
```

1.5. Source code for the example libraries
/** Calculate the square root of {@code x} */
static public double sqrt(double x) {
    return Math.sqrt(x);
}

Matrix.java

package nz.msl.examples;

import java.util.Random;

public class Matrix {
    /** The matrix, M */
    private double[][] m;

    /** Lower-triangular matrix representation, M=LU, in LU Decomposition */
    private Matrix L;

    /** Upper-triangular matrix representation, M=LU, in LU Decomposition */
    private Matrix U;

    /** A NxM orthogonal matrix representation, M=QR, in QR Decomposition */
    private Matrix Q;

    /** Upper-triangular matrix representation, M=QR, in QR Decomposition */
    private Matrix R;

    /** When calculating the inverse we calculate the LU matrices once */
    static private boolean calculatingInverse = false;

    /** Create a Matrix that is a copy of another Matrix. */
    public Matrix(Matrix m) {
        this.m = new double[m.getNumberOfRows()][m.getNumberOfColumns()];
        for (int i=0; i<m.getNumberOfRows(); i++)
            for (int j=0; j<m.getNumberOfColumns(); j++)
                this.m[i][j] = m.getValue(i, j);
    }

    /** Create a {@code n} x {@code n} identity Matrix */
    public Matrix(int n) {
        m = new double[n][n];
        for (int i=0; i<n; i++)
            m[i][i] = 1.0;
/** Create a {rows} x {cols} Matrix filled with zeros. */
public Matrix(int rows, int cols) {
    m = new double[rows][cols];
}

/** Create a {rows} x {cols} Matrix filled with a value. */
public Matrix(int rows, int cols, double value) {
    m = new double[rows][cols];
    for (int i=0; i<rows; i++)
        for (int j=0; j<cols; j++)
            m[i][j] = value;
}

/** Create a {rows} x {cols} Matrix that is filled with uniformly-distributed random values that are within the range {min} to {max}. */
public Matrix(int rows, int cols, double min, double max) {
    Random rand = new Random();
    m = new double[rows][cols];
    for (int i=0; i<rows; i++)
        for (int j=0; j<cols; j++)
            m[i][j] = (max-min)*rand.nextDouble()+min;
}

/** Create a Matrix from {m}. */
public Matrix(Double[][] m) {
    this.m = new double[m.length][m[0].length];
    for (int i=0; i<m.length; i++)
        for (int j=0; j<m[0].length; j++)
            this.m[i][j] = m[i][j];
}

/** Create a Matrix from a vector. */
public Matrix(Double[] vector) {
    m = new double[1][vector.length];
    for (int i=0; i<vector.length; i++)
        m[0][i] = vector[i];
}

/*
* The public static methods.
*
*/

/** Returns the product of two Matrices as a new Matrix, C=AB. */
public static Matrix multiply(Matrix a, Matrix b) {
    if (a.getNumberOfColumns() != b.getNumberOfRows()) {
        throw new IllegalArgumentException(String.format("ERROR! Cannot multiply a %dx%d matrix "
            + "with a %dx%d matrix",
            a.getNumberOfRows(), a.getNumberOfColumns(),
            b.getNumberOfRows(), b.getNumberOfColumns()));
    }
    int rows = a.getNumberOfRows();
    int cols = b.getNumberOfColumns();
    double[][] result = new double[rows][cols];
    for (int i=0; i<rows; i++)
        for (int j=0; j<cols; j++)
            for (int k=0; k<a.getNumberOfColumns(); k++)
                result[i][j] += a[i][k] * b[k][j];
    return new Matrix(rows, cols, result);
}
a.getNumberOfRows(), a.getNumberOfColumns(),
b.getNumberOfRows(), b.getNumberOfColumns()));
} else {
    Matrix c = new Matrix(a.getNumberOfRows(), b.getNumberOfColumns());
    double sum = 0.0;
    for (int i = 0; i < a.getNumberOfRows(); i++) {
        for (int j = 0; j < b.getNumberOfColumns(); j++) {
            for (int k = 0; k < b.getNumberOfRows(); k++) {
                sum += a.getValue(i,k)*b.getValue(k,j);
            }
            c.setValue(i, j, sum);
            sum = 0.0;
        }
    }
    return c;
}

/**
 * Solves \(b = Ax\) for \(x\).
 * @param A - the coefficient matrix
 * @param b - the expected values
 * @return x - the solution to the system of equations
 */
public static Matrix solve(Matrix A, Matrix b) {
    // ensure that 'b' is a column vector
    if (b.getNumberOfColumns() > 1) b = b.transpose();

    // ensure that 'A' and 'b' have the correct dimensions
    if (b.getNumberOfRows() != A.getNumberOfRows()) {
        throw new IllegalArgumentException(
            String.format("ERROR! Dimension mismatch when solving the " + "system of equations using b=Ax, b has dimension " + "%dx%d and A is %dx%d.", b.getNumberOfRows(), b.getNumberOfColumns(), A.getNumberOfRows(), A.getNumberOfColumns()));
    }

    // if A is an under-determined system of equations then use the
    // matrix-multiplication expression to solve for x
    if (A.getNumberOfRows() < A.getNumberOfColumns()) {
        Matrix At = A.transpose();
        return Matrix.multiply(Matrix.multiply(At,
            Matrix.multiply(A, At).getInverse() ), b);
    }

    // If A is a square matrix then use LU Decomposition, if it is an
    // over-determined system of equations then use QR Decomposition
    Double[] x = new Double[A.getNumberOfColumns()];
    if (A.isSquare()) {
        // when using 'solve' to calculate the inverse of a matrix we
        // only need to generate the LU Decomposition matrices once
        if (!calculatingInverse) A.makeLU();

        // ensure that 'x' is a row vector
        if (x.length > 1) x = new Double[1];

        // inverse of a square matrix is calculated as follows:
        // inverse(A) = C^T * (det(A) / determinant(C))
        // where C is the matrix obtained by replacing the
        // diagonal of A with the identity matrix
        // To ensure that 'A' and 'x' have the correct dimensions
        // if A is an over-determined system of equations then use the
        // matrix-multiplication expression to solve for x
        if (A.getNumberOfRows() > A.getNumberOfColumns()) {
            Matrix x = Matrix.multiply(At,
                Matrix.multiply(A, At).inverse());
            x = x.toArray();
        }

        // solve for x using the LU Decomposition
        if (calculatingInverse)
            A.solve(x);
        else
            A.solve(x, Matrix.LUDecomposition.ALU);

        // if the inverse of 'A' is not calculated then we
        // only need to generate the LU Decomposition matrices once
        if (!calculatingInverse) A.makeLU();
    }

    // calculate the determinant of 'A'
    double det = A.determinant();

    // calculate the inverse of 'A'
    Matrix inverse = Matrix.createInverse(A);

    // copy the inverse of 'A' into the result matrix
    for (int i = 0; i < A.getNumberOfRows(); i++) {
        for (int j = 0; j < A.getNumberOfColumns(); j++) {
            inverse.setValue(i, j, inverse.getValue(i, j));
        }
    }

    return inverse;
}
// solve Ly=b for y using forward substitution
double[] y = new double[b.getNumberOfRows()];
y[0] = b.getValue(0,0);
for (int i=1; i<y.length; i++) {
    y[i] = b.getValue(i,0);
    for (int j=0; j<i; j++)
        y[i] -= A.getL().getValue(i,j)*y[j];
}

// solve Ux=y for x using backward substitution
for (int i=x.length-1; i>-1; i--) {
    x[i] = y[i];
    for (int j=i+1; j<x.length; j++)
        x[i] -= A.getU().getValue(i,j)*x[j];
    x[i] /= A.getU().getValue(i,i);
}

else {
    A.makeQR();
    Matrix d = Matrix.multiply(A.getQ().transpose(), b);

    // solve Rx=d for x using backward substitution
    for (int i=x.length-1; i>-1; i--) {
        x[i] = d.getValue(i, 0);
        for (int j=i+1; j<x.length; j++)
            x[i] -= A.getR().getValue(i,j)*x[j];
        x[i] /= A.getR().getValue(i,i);
    }
}

return new Matrix(x).transpose();

/** Returns the primitive data of the Matrix. */
public double[][] primitive() {
    return m;
}

/** Convert the Matrix to a string. */
@Override
public String toString() {
    StringBuffer sb = new StringBuffer();
    for (int i=0; i<m.length; i++) {
        for (int j=0; j<m[0].length; j++) {
            sb.append(String.format("%+.6e\t", m[i][j]));
        }
    }
    sb.append("\n");
    return sb.toString();
}
\begin{verbatim}
} return sb.toString();
}

/** Returns the number of rows in the Matrix. */
public int getNumberOfRows() {
    return m.length;
}

/** Returns the number of columns in the Matrix. */
public int getNumberOfColumns() {
    try {
        return m[0].length;
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}

/** Returns the value at \(row\) and \(col\). */
public double getValue(int row, int col) {
    return m[row][col];
}

/** Sets the value at \(row\) and \(col\) to be \(value\). */
public void setValue(int row, int col, double value) {
    m[row][col] = value;
}

/** Returns the transpose of the Matrix. */
public Matrix transpose() {
    Matrix mt = new Matrix(m[0].length, m.length);
    for (int i=0; i<m.length; i++)
        for (int j=0; j<m[0].length; j++)
            mt.setValue(j, i, m[i][j]);
    return mt;
}

/** Returns whether the Matrix is a square Matrix. */
public boolean isSquare() {
    return m.length == m[0].length;
}

/** Returns the determinant of the Matrix. */
public double getDeterminant() {
    if (isSquare()) {
        makeLU();
        double det = 1.0;
        for (int i=0; i<m.length; i++)
            det *= U.getValue(i,i);
        // 's' is the number of row and column exchanges in LU Decomposition
        // but we are currently not using pivoting
        int s = 0;
        return Math.pow(-1.0, s)*det;
    } else {
        return Double.NaN;
    }
\end{verbatim}
/** Returns the lower-triangular Matrix, L, from a LU Decomposition */
public Matrix getL() {
    if (L == null) makeLU();
    return L;
}

/** Returns the upper-triangular Matrix, U, from a LU Decomposition */
public Matrix getU() {
    if (U == null) makeLU();
    return U;
}

/** Returns the orthogonal Matrix, Q, from a QR Decomposition */
public Matrix getQ() {
    if (Q == null) makeQR();
    return Q;
}

/** Returns the upper-triangular Matrix, R, from a QR Decomposition */
public Matrix getR() {
    if (R == null) makeQR();
    return R;
}

/** Returns the inverse of the Matrix, if it exists. */
public Matrix getInverse() {
    if (isSquare()) {
        Matrix inv = new Matrix(m.length);
        Matrix bb = new Matrix(m.length);
        for (int i = 0; i < m.length; i++) {
            inv.setColumn(i, Matrix.solve(this, bb.getColumn(i)));
            calculatingInverse = true;
        }
        calculatingInverse = false;
        return inv;
    } else {
        throw new IllegalArgumentException( String.format("ERROR! Cannot calculate the inverse of a " + "%dx%d matrix, it must be a square Matrix", m.length, m[0].length));
    }
}

/*
 * Private methods.
 * *
 */

/**
 * Create the Lower, L, and Upper, U, triangular matrices, such that.
 */

(continues on next page)
private void makeLU() {
    L = new Matrix(m.length); // create an identity matrix
    U = new Matrix(this); // copy the values of this matrix
    double val;
    for (int k=0; k<m[0].length; k++) {
        for (int i=k+1; i<m.length; i++) {
            val = U.getValue(i,k)/U.getValue(k,k);
            L.setValue(i, k, val);
            for (int j=k; j<m[0].length; j++)
                U.setValue(i, j, U.getValue(i,j)-val*U.getValue(k,j));
        }
    }
}

/** Computes the QR Factorization matrices using a modified
 * Gram-Schmidt process.<p>
 * @see http://www.inf.ethz.ch/personal/gander/papers/qrneu.pdf
 */
private void makeQR() {
    Q = new Matrix(m.length, m[0].length);
    R = new Matrix(m[0].length, m[0].length);
    Matrix A = new Matrix(this);
    double s;
    for (int k=0; k<m[0].length; k++) {
        s = 0.0;
        for (int j=0; j<m.length; j++)
            s += Math.pow(A.getValue(j, k), 2);
        s = Math.sqrt(s);
        R.setValue(k, k, s);
        for (int j=0; j<m.length; j++)
            Q.setValue(j, k, A.getValue(j, k)/s);
        for (int i=k+1; i<m[0].length; i++) {
            s = 0.0;
            for (int j=0; j<m.length; j++)
                s += A.getValue(j, i)*Q.getValue(j, k);
            R.setValue(k, i, s);
            for (int j=0; j<m.length; j++)
                A.setValue(j, i, A.getValue(j,i)-R.getValue(k,i)*Q.getValue(j, k));
        }
    }
}

/** Returns a copy of the specified column. */
private Matrix getColumn(int column) {
    if (column < m[0].length) {
        Matrix c = new Matrix(m.length, 1);
        for (int i=0; i<m.length; i++)
            c.setValue(i, 0, m[i][column]);
        return c;
    }


```java
} else {
    throw new IllegalArgumentException(
        String.format("ERROR! Cannot get column %d in the Matrix " + "since it is > the number of columns in the " + "Matrix, %d.", column, m[0].length));
}
}

/**
 * Replace the values in the specified column of the matrix to the values in
 * @code vector.
 * * The @code vector must be a 1D vector, can have dimension 1xN or Nx1.
 */
private void setColumn(int column, Matrix vector) {

    // make sure that 'vector' is either a 1xN or Nx1 vector and not a NxM Matrix
    if ((vector.getNumberOfColumns() != 1) && (vector.getNumberOfRows() != 1)) {
        throw new IllegalArgumentException(
            String.format("ERROR! Require a 1D vector to replace the values " + "in a column of a matrix. Got a %dx%d vector.", vector.getNumberOfRows(), vector.getNumberOfColumns()));
    }

    // make sure we have a column vector
    if (vector.getNumberOfColumns() != 1) {
        vector = vector.transpose();
    }

    // make sure the 'vector' has the correct length
    if (vector.getNumberOfRows() != m.length) {
        throw new IllegalArgumentException(
            String.format("ERROR! Cannot replace a Matrix column of length " + "%d, with a column vector of length %d.", m.length, vector.getNumberOfRows()));
    }

    // make sure the column is valid
    if (column >= m[0].length) {
        throw new IllegalArgumentException(
            String.format("ERROR! Cannot replace column %d in the Matrix " + "since it is > the number of columns in the matrix.", column));
    }

    for (int i=0; i<m.length; i++)
        m[i][column] = vector.getValue(i,0);
}
```
1.6 Re-freezing the 32-bit server

If you want to make your own 32-bit server you will need

1) a 32-bit version of Python 2.7 or 3.5+ (whatever version you want)

2) PyInstaller

Using pip from the 32-bit Python interpreter run

```
pip install msl-loadlib pyinstaller
```

**Note:** If you want to include additional packages, for example, pythonnet, comtypes, numpy, etc. run

```
pip install pythonnet comtypes numpy
```

1.6.1 Using the API

Launch an Interactive Console using the 32-bit Python interpreter

```
python
```

and enter

```
>>> from msl.loadlib import freeze_server32
>>> freeze_server32.main()
... PyInstaller logging messages ...
Server saved to: ...
>>> exit()
```

Specify the appropriate keyword arguments to the `main()` function.

Copy the `server32-*` file to the folder where you have MSL-LoadLib installed in your 64-bit version of Python and replace the existing server file.

**Tip:** You could also use pip to install MSL-LoadLib from the master branch of GitHub

```
pip install https://github.com/MSLNZ/msl-loadlib/archive/master.zip
```

This may require you to update your version of pip. Only install MSL-LoadLib in this way if you are creating a new server. The servers that are on GitHub are not necessarily synced with the rest of the code that is on GitHub. The updated servers are only uploaded to GitHub when a release is published.

1.6.2 Using the CLI

In this example we will clone the GitHub repository and create the server from the command line. Make sure that invoking `python` on your terminal uses the 32-bit Python interpreter or specify the full path to the 32-bit Python interpreter that you want to use.
Copy the server32-* file to the folder where you have MSL-LoadLib installed in your 64-bit version of Python and replace the existing server file.

To see the help for the freeze_server32.py module run

```bash
python freeze_server32.py --help
```

For example, if you wanted to bypass the error that pythonnet is not installed run

```bash
python freeze_server32.py --ignore-pythonnet
```

### 1.7 License

**MIT License**

Copyright (c) 2017 - 2019, Measurement Standards Laboratory of New Zealand

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### 1.8 Developers

- **Joseph Borbely** <joseph.borbely@measurement.govt.nz>
1.9 Changelog

1.9.1 Version 0.6.1.dev0

- Removed
  - Support for Python 3.4

1.9.2 Version 0.6.0 (2019.05.07)

- Added
  - a `shutdown_handler()` method to `Server32` (PR #19)
  - a section to the docs that explains how to re-freeze the 32-bit server
  - a `kill_timeout` keyword argument to `Client64.shutdown_server32()`
  - the `rpc_timeout` keyword argument to `Client64` (thanks to @fake-name)
  - search `HKEY_CLASSES_ROOT\Wow6432Node\CLSID` in the Windows Registry for additional ActiveX ProgID’s
  - `extras_require` parameter to `setup.py` with keys: `clr, java, com, all`

- Changed
  - the frozen server32 executable (for Windows/Linux) now uses Python 3.7.3 and Python.NET 2.4.0
  - rename the optional `-asp` and `-aep` command line arguments to be `-s` and `-e` respectively
  - the current working directory where the 64-bit Python interpreter was executed from is now automatically appended to `os.environ['PATH']` on the 32-bit server
  - `freeze_server32.py` uses an `ArgumentParser` instead of directly reading from `sys.argv`

- Fixed
  - use `sys.executable -m PyInstaller` to create the 32-bit server (part of PR #18)
  - the 32-bit server prints error messages to `sys.stderr` instead of `sys.stdout`
  - issue #15 - wait for the subprocess that starts the 32-bit server to terminate and set a value for the `returncode`
  - issue #14 - use `os.kill` to stop the 32-bit server if it won’t stop after `kill_timeout` seconds

1.9.3 Version 0.5.0 (2019.01.06)

- Added
  - support for loading a Component Object Model (COM) library on Windows
  - the `requires_pythonnet` and `requires_comtypes` kwargs to `freeze_server32.main()`
  - `clr` as an alias for `net` for the `libtype` parameter in `LoadLibrary`
  - the `utils.get_com_info()` function
- support for unicode paths in Python 2
- examples for working with numpy arrays and C++ structs

**Changed**
- the frozen server32 executable (for Windows/Linux) now runs on Python 3.6.8
- if loading a .NET assembly succeeds but calling GetTypes() fails then a detailed error message is logged rather than raising the exception - the value of \textit{lib} will be \texttt{None}
- the default timeout value when waiting for the 32-bit server to start is now 10 seconds
- the \texttt{Client64} class now raises \texttt{Server32Error} if the 32-bit server raises an exception
- the \texttt{Client64} class now inherits from \texttt{object} and the reference to \texttt{HTTPConnection} is now a property value
- the \texttt{\_\_repr\_\_} methods no longer include the id as a hex number

**Fixed**
- \texttt{set sys.stdout = io.StringIO() if quiet=\texttt{True} on the server}

**1.9.4 Version 0.4.1 (2018.08.24)**

**Added**
- the \texttt{version\_info} namedtuple now includes a \texttt{releaselevel}
- Support for Python 3.7

**Fixed**
- Issue #11
  - \texttt{utils.wait\_for\_server()} raised \texttt{NameError: name `TimeoutError` is not defined} for Python 2.7
  - \texttt{utils.port\_in\_use()} raised \texttt{UnicodeDecodeError} (PR #9)
  - \texttt{setup.py} is now also compatible with Sphinx 1.7+

**Changed**
- the frozen server32 executable (for Windows/Linux) now runs on Python 3.6.6
- \texttt{pythonnet} is now an optional dependency on Windows and \texttt{py4j} is now optional for all OS
- rename \texttt{Dummy} example to \texttt{Echo}

**Removed**
- Support for Python 3.3

**1.9.5 Version 0.4.0 (2018.02.28)**

**Added**
- \texttt{Py4J} wrapper for loading \texttt{.jar} and \texttt{.class} Java files
- example on how to load a library that was built with LabVIEW
• Fixed
  – Issue #8
  – Issue #7
  – AttributeError("'LoadLibrary' object has no attribute '_lib'") raised in repr()

• Changed
  – rename DotNetContainer to DotNet
  – use bind() to select an available port instead of checking if port_in_use()
  – moved the static methods to the msl.loadlib.utils module:
    * Client64.port_in_use -> utils.port_in_use
    * Client64.get_available_port -> utils.get_available_port
    * Client64.wait_for_server -> utils.wait_for_server
    * LoadLibrary.check_dot_net_config -> utils.check_dot_net_config
    * LoadLibrary.is_pythonnet_installed -> utils.is_pythonnet_installed

1.9.6 Version 0.3.2 (2017.10.18)

• Added
  – include os.environ['PATH'] as a search path when loading a shared library
  – the frozen server32 executable (for Windows/Linux) now runs on Python 3.6.3
  – support that the package can now be installed by pip install msl-loadlib

• Fixed
  – remove sys.getsitepackages() error for virtualenv (issue #5)
  – received RecursionError when freezing freeze_server32.py with PyInstaller 3.3
  – replaced FileNotFoundError with IOError (for Python 2.7 support)
  – recompile cpp_lib*.dll and fortran_lib*.dll to not depend on external dependencies

1.9.7 Version 0.3.1 (2017.05.15)

• fix ReadTheDocs build error – AttributeError: module ‘site’ has no attribute ‘getsitepackages’
• strip whitespace from append_sys_path and append_environ_path
• make pythonnet a required dependency only for Windows

1.9.8 Version 0.3.0 (2017.05.09)

NOTE: This release breaks backward compatibility

• can now pass **kwargs from the Client64 constructor to the Server32-subclass constructor
• new command line arguments for starting the 32-bit server: –kwargs, –append_environ_path
• renamed the –append_path command line argument to –append_sys_path
• Server32.interactive_console() works on Windows and Linux
• edit documentation (thanks to @karna48 for the pull request)

1.9.9 Version 0.2.3 (2017.04.11)

• the frozen server32 executable (for Windows/Linux) now uses Python v3.6.1 and Python.NET v2.3.0
• include ctypes.util.find_library and sys.path when searching for shared library

1.9.10 Version 0.2.2 (2017.03.03)

• refreeze server32 executables

1.9.11 Version 0.2.1 (2017.03.02)

• fix releaselevel bug

1.9.12 Version 0.2.0 (2017.03.02)

• examples now working in Linux
• fix MSL namespace
• include all C# modules, classes and System.Type objects in the .NET loaded-library object
• create a custom C# library for the examples
• edit docstrings and documentation
• many bug fixes

1.9.13 Version 0.1.0 (2017.02.15)

• Initial release
CHAPTER \textbf{2}

Index

• modindex
m
msl.examples.loadlib, 39
msl.examples.loadlib.cpp32, 39
msl.examples.loadlib.cpp64, 43
msl.examples.loadlib.dotnet32, 46
msl.examples.loadlib.dotnet64, 50
msl.examples.loadlib.echo32, 53
msl.examples.loadlib.echo64, 53
msl.examples.loadlib.fortran32, 54
msl.examples.loadlib.fortran64, 60
msl.examples.loadlib.kernel32, 63
msl.examples.loadlib.kernel64, 64
msl.examples.loadlib.labview32, 65
msl.examples.loadlib.labview64, 66
msl.loadlib, 28
msl.loadlib.client64, 29
msl.loadlib.exceptions, 31
msl.loadlib.freeze_server32, 32
msl.loadlib.load_library, 33
msl.loadlib.server32, 35
msl.loadlib.start_server32, 37
msl.loadlib.utils, 37
Index

A
add() (msl.examples.loadlib.cpp32.Cpp32 method), 39
add() (msl.examples.loadlib.cpp64.Cpp64 method), 44
add_1D_arrays() (msl.examples.loadlib.fortran32.Fortran32 method), 54
add_1D_arrays() (msl.examples.loadlib.fortran64.Fortran64 method), 60
add_integers() (msl.examples.loadlib.dotnet32.DotNet32 method), 46
add_integers() (msl.examples.loadlib.dotnet64.DotNet64 method), 50
add_multiple() (msl.examples.loadlib.dotnet32.DotNet32 method), 46
add_multiple() (msl.examples.loadlib.dotnet64.DotNet64 method), 51
add_or_subtract() (msl.examples.loadlib.cpp32.Cpp32 method), 40
add_or_subtract() (msl.examples.loadlib.cpp64.Cpp64 method), 44
add_or_subtract() (msl.examples.loadlib.dotnet32.DotNet32 method), 47
add_or_subtract() (msl.examples.loadlib.dotnet64.DotNet64 method), 51
add_or_subtract() (msl.examples.loadlib.fortran32.Fortran32 method), 55
add_or_subtract() (msl.examples.loadlib.fortran64.Fortran64 method), 61
assembly (msl.loadlib.load_library.LoadLibrary attribute), 34
assembly (msl.loadlib.server32.Server32 attribute), 35

B
besselJ0() (msl.examples.loadlib.fortran32.Fortran32 method), 55
besselJ0() (msl.examples.loadlib.fortran64.Fortran64 method), 61

C
check_dot_net_config() (in module msl.loadlib.utils), 37
circumference() (msl.examples.loadlib.cpp32.Cpp32 method), 40
circumference() (msl.examples.loadlib.cpp64.Cpp64 method), 44
Client64 (class in msl.loadlib.client64), 29
concatenate() (msl.examples.loadlib.dotnet32.DotNet32 method), 47
concatenate() (msl.examples.loadlib.dotnet64.DotNet64 method), 51
connection (msl.loadlib.client64.Client64 attribute), 30
ConnectionTimeoutError, 31
Cpp32 (class in msl.examples.loadlib.cpp32), 39
Cpp64 (class in msl.examples.loadlib.cpp64), 44

distance_4_points() (msl.examples.loadlib.cpp32.Cpp32 method), 40
method), 41

distance_4_points()
(msl.examples.loadlib.cpp64.Cpp64 method), 44

divide_floats()
(msl.examples.loadlib.dotnet32.DotNet32 method), 48

divide_floats()
(msl.examples.loadlib.dotnet64.DotNet64 method), 51

DotNet (class in msl.loadlib.load_library), 34

DotNet32 (class in msl.examples.loadlib.dotnet32), 46

DotNet64 (class in msl.examples.loadlib.dotnet64), 50

E

Echo32 (class in msl.examples.loadlib.echo32), 53

Echo64 (class in msl.examples.loadlib.echo64), 53

F

factorial() (msl.examples.loadlib.fortran32.Fortran32 method), 56

factorial() (msl.examples.loadlib.fortran64.Fortran64 method), 61

Fortran32 (class in msl.examples.loadlib.fortran32), 54

Fortran64 (class in msl.examples.loadlib.fortran64), 60

FourPoints (class in msl.examples.loadlib.cpp32), 42

G

gateway (msl.loadlib.load_library.LoadLibrary attribute), 34

get_available_port() (in module msl.loadlib.utils), 38

get_class_names()
(msl.examples.loadlib.dotnet32.DotNet32 method), 48

get_class_names()
(msl.examples.loadlib.dotnet64.DotNet64 method), 52

get_com_info() (in module msl.loadlib.utils), 38

get_local_time()
(msl.examples.loadlib.kernel64.Kernel64 method), 65

get_time() (msl.examples.loadlib.kernel32.Kernel32 method), 64

H

host (msl.loadlib.client64.Client64 attribute), 30

I

interactive_console()
(msl.loadlib.server32.Server32 static method), 36

is_comtypes_installed() (in module msl.loadlib.utils), 38

IS_LINUX (in module msl.loadlib), 28

IS_MAC (in module msl.loadlib), 28

is_positive()
(msl.examples.loadlib.fortran32.Fortran32 method), 56

is_positive()
(msl.examples.loadlib.fortran64.Fortran64 method), 61

is_py4j_installed() (in module msl.loadlib.utils), 38

IS_PYTHON2 (in module msl.loadlib), 28

IS_PYTHON3 (in module msl.loadlib), 28

IS_PYTHON_64BIT (in module msl.loadlib), 29

is_pythonnet_installed() (in module msl.loadlib.utils), 38

IS_WINDOWS (in module msl.loadlib), 29

K

Kernel32 (class in msl.examples.loadlib.kernel32), 63

Kernel64 (class in msl.examples.loadlib.kernel64), 65

L

Labview32 (class in msl.examples.loadlib.labview32), 65

Labview64 (class in msl.examples.loadlib.labview64), 66

lib (msl.loadlib.load_library.LoadLibrary attribute), 34

lib (msl.loadlib.server32.Server32 attribute), 35

lib32_path (msl.loadlib.client64.Client64 attribute), 30

LoadLibrary (class in msl.loadlib.load_library), 33

M

main() (in module msl.loadlib.freeze_server32), 32

main() (in module msl.loadlib.start_server32), 37
reverse_string()
(mlsl.examples.loadlib.dotnet64.DotNet64 method), 52
reverse_string()
(mlsl.examples.loadlib.fortran32.Fortran32 method), 58
reverse_string()
(mlsl.examples.loadlib.fortran64.Fortran64 method), 62
reverse_string_v1()
(mlsl.examples.loadlib.cpp32.Cpp32 method), 41
reverse_string_v1()
(mlsl.examples.loadlib.cpp64.Cpp64 method), 45
reverse_string_v2()
(mlsl.examples.loadlib.cpp32.Cpp32 method), 41
reverse_string_v2()
(mlsl.examples.loadlib.cpp64.Cpp64 method), 45

S
scalar_multiply()
(mlsl.examples.loadlib.cpp32.Cpp32 method), 42
scalar_multiply()
(mlsl.examples.loadlib.cpp64.Cpp64 method), 45
scalar_multiply()
(mlsl.examples.loadlib.dotnet32.DotNet32 method), 50
scalar_multiply()
(mlsl.examples.loadlib.dotnet64.DotNet64 method), 52
send_data()
(mlsl.examples.loadlib.echo64.Echo64 method), 54
Server32 (class in msl.loadlib.server32), 35
Server32Error, 31
shutdown_handler()
(mlsl.loadlib.server32.Server32 method), 36
shutdown_server32()
(mlsl.loadlib.client64.Client64 method), 31
standard_deviation()
(mlsl.examples.loadlib.fortran32.Fortran32 method), 58
standard_deviation()
(mlsl.examples.loadlib.fortran64.Fortran64 method), 62

stddev()
(mlsl.examples.loadlib.labview32.Labview32 method), 65
stddev()
(mlsl.examples.loadlib.labview64.Labview64 method), 66
subtract()
(mlsl.examples.loadlib.cpp32.Cpp32 method), 42
subtract()
(mlsl.examples.loadlib.cpp64.Cpp64 method), 45
sum_16bit()
(mlsl.examples.loadlib.fortran32.Fortran32 method), 58
sum_16bit()
(mlsl.examples.loadlib.fortran64.Fortran64 method), 62
sum_32bit()
(mlsl.examples.loadlib.fortran32.Fortran32 method), 59
sum_32bit()
(mlsl.examples.loadlib.fortran64.Fortran64 method), 62
sum_64bit()
(mlsl.examples.loadlib.fortran32.Fortran32 method), 63
sum_64bit()
(mlsl.examples.loadlib.fortran64.Fortran64 method), 63
sum_8bit()
(mlsl.examples.loadlib.fortran32.Fortran32 method), 60
sum_8bit()
(mlsl.examples.loadlib.fortran64.Fortran64 method), 63

SystemTime (class in msl.examples.loadlib.kernel32), 64

T
traceback (mlsl.loadlib.exceptions.Server32Error attribute), 32

V
value (mlsl.loadlib.exceptions.Server32Error attribute), 32
version()
(mlsl.loadlib.server32.Server32 static method), 36
version_info (in module msl.loadlib), 29

W
wait_for_server()
(in module msl.loadlib.utils), 38
wDay (mlsl.examples.loadlib.kernel32.SystemTime attribute), 64
wDayOfWeek (mlsl.examples.loadlib.kernel32.SystemTime attribute), 64
wHour (mlsl.examples.loadlib.kernel32.SystemTime attribute), 64
wMilliseconds
(mlsl.examples.loadlib.kernel32.SystemTime attribute), 64
wMinute (mlsl.examples.loadlib.kernel32.SystemTime attribute), 64
wMonth (msl.examples.loadlib.kernel32.SystemTime attribute), 64
WORD (msl.examples.loadlib.kernel32.SystemTime attribute), 64
wSecond (msl.examples.loadlib.kernel32.SystemTime attribute), 64
wYear (msl.examples.loadlib.kernel32.SystemTime attribute), 64

X
x (msl.examples.loadlib.cpp32.Point attribute), 43

Y
y (msl.examples.loadlib.cpp32.Point attribute), 43