MSL-LoadLib Documentation

Release 1.0.0.dev0

Measurement Standards Laboratory of New Zealand

Apr 29, 2024

CONTENTS

This package loads 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 .NET, 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) in 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.

CHAPTER

ONE

CONTENTS

1.1 Install

To install msl-loadlib run

```
pip install msl-loadlib
```

Alternatively, using the MSL Package Manager run

```
msl install loadlib
```

1.1.1 Dependencies

- Python 3.8+

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 32-bit server has been built into a frozen Python executable for Windows and Linux.
- You may create a new 32-bit server. See *Create a custom 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 32-bit 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, you must install Python for .NET

```
pip install pythonnet
```

If you need to load a Java library (i.e., a `.jar` or `.class` file), you must install Py4J,

```
pip install py4j
```

a Java Runtime Environment, and, ensure that the `java` executable is available on the PATH variable. For example, the following should return the version of Java that is installed

```
C:\Users\username>java --version
java 22 2024-03-19
Java(TM) SE Runtime Environment (build 22+36-2370)
Java HotSpot(TM) 64-Bit Server VM (build 22+36-2370, mixed mode, sharing)
```

If you need to load a Component Object Model (or ActiveX) library, you must install comtypes

```
pip install comtypes
```

Tip: When loading a shared library it is vital that all dependencies of the library are also on the computer and that the directory that the dependencies are located in is available on the PATH variable (and possibly you may need to add a directory with `os.add_dll_directory()`). A helpful utility to determine the dependencies of a shared library on Windows is Dependencies (which is a modern Dependency Walker). Microsoft also provides the DUMPBIN tool. For finding the dependencies of a .NET library the Dependency Walker for .NET may also be helpful.

Linux

Before using msl-loadlib on Debian-based Linux distributions, the following packages are required. For other distributions, use the appropriate system package manager (e.g., *yum*) and the equivalent command.

Attention: The following packages are required to run the examples that are included with msl-loadlib when it is installed. The dependencies for the C/C++ or FORTRAN library that you want to load may be different.

Install the packages that are required to load 32-bit and 64-bit C/C++ and FORTRAN libraries

```
sudo dpkg --add-architecture i386
sudo apt update
sudo apt install g++ gfortran libgfortran5 zlib1g:i386 libstdc++6:i386␣
↪libgfortran5:i386
```

The following ensures that the ss command is available

```
sudo apt install iproute2
```

If you need to load a Microsoft .NET library then you must install Mono (see here for instructions) and Python for .NET

```
pip3 install pythonnet
```

Important: As of version 0.10.0 of msl-loadlib, pythonnet is no longer installed on the 32-bit server for Linux. Mono can load both 32-bit and 64-bit libraries on 64-bit Linux and therefore a 32-bit .NET library can be loaded directly via *LoadLibrary* on 64-bit Linux.

If you need to load a Java library (i.e., a `.jar` or `.class` file), you must install Py4J,

```
pip3 install py4j
```

and a Java Runtime Environment

```
sudo apt install default-jre
```

Tip: When loading a shared library it is vital that all dependencies of the library are also on the computer and that the directory that the dependency is located in is available on the PATH variable. A helpful utility to determine the dependencies of a shared library on Unix is ldd.

macOS

The 32-bit server has not been created for macOS; however, the *LoadLibrary* class can be used to load a library that uses the `__cdecl` calling convention that is the same bitness as the Python interpreter, a .NET library or a Java library.

The following assumes that you are using Homebrew as your package manager.

It is recommended to update Homebrew before installing packages

```
brew update
```

To load a C/C++ or FORTRAN library install gcc (which includes gfortran)

```
brew install gcc
```

If you need to load a Microsoft .NET library, you must install Mono,

```
brew install mono
```

and Python for .NET

```
pip3 install pythonnet
```

If you need to load a Java library (i.e., a `.jar` or `.class` file), you must install Py4J,

```
pip3 install py4j
```

and a Java Runtime Environment

```
brew cask install java
```

1.2 Load a library

If you are loading a 64-bit library in 64-bit Python (or a 32-bit library in 32-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 *Access a 32-bit library in 64-bit Python* 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 and in 64-bit macOS. The *.NET* library was complied in 32- and 64-bit using Microsoft Visual Studio 2017. 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.

Tip: If the file extension is not specified then a default extension, `.dll` (Windows), `.so` (Linux) or `.dylib` (macOS) is used.

1.2.1 C++

Load a 64-bit C++ library in 64-bit Python (view the *C++ source code*). *To load the 32-bit library in 32-bit Python use* `'/cpp_lib32'`.

```
>>> from msl.loadlib import LoadLibrary
>>> from msl.examples.loadlib import EXAMPLES_DIR
>>> cpp = LoadLibrary(EXAMPLES_DIR + '/cpp_lib64')
```

Call the `add` function to calculate the sum of two integers

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

1.2.2 FORTRAN

Load a 64-bit FORTRAN library in 64-bit Python (view the *FORTRAN source code*). *To load the 32-bit library in 32-bit Python use* `'/fortran_lib32'`.

```
>>> from msl.loadlib import LoadLibrary
>>> from msl.examples.loadlib import EXAMPLES_DIR
>>> fortran = LoadLibrary(EXAMPLES_DIR + '/fortran_lib64')
```

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

```
>>> 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

Load a 64-bit C# library (a .NET Framework) in 64-bit Python (view the *C# source code*). Include the `'net'` argument to indicate that the `.dll` file is for the .NET Framework. *To load the 32-bit library in 32-bit Python use* `'/dotnet_lib32.dll'`.

```
>>> from msl.loadlib import LoadLibrary
>>> from msl.examples.loadlib import EXAMPLES_DIR
>>> net = LoadLibrary(EXAMPLES_DIR + '/dotnet_lib64.dll', 'net')
```

Tip: `'clr'` is an alias for `'net'` and can also be used as the *libtype*

Attention: To configure *pythonnet* to use the .NET Core runtime, you must either run

```
from pythonnet import load
load("coreclr")
```

or define a PYTHONNET_RUNTIME=coreclr environment variable

```
import os
os.environ["PYTHONNET_RUNTIME"] = "coreclr"
```

before *LoadLibrary* is called. To use the Mono runtime, replace "coreclr" with "mono".

The *dotnet_lib64* library contains a reference to the DotNetMSL module (which is a C# namespace), the StaticClass class, the StringManipulation class and the System namespace

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

```
>>> bm = net.lib.DotNetMSL.BasicMath()
>>> bm.multiply_doubles(2.3, 5.6)
12.879999...
```

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

```
>>> am = net.lib.DotNetMSL.ArrayManipulation()
>>> values = am.scalar_multiply(2., [1., 2., 3., 4., 5.])
>>> values
<System.Double[] object at ...>
>>> [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

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

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

```
>>> net.lib.StaticClass.add_multiple(1, 2, 3, 4, 5)
15
```

One can create objects from the System namespace,

```
>>> System = net.lib.System
```

for example, to create a 32-bit signed integer,

```
>>> System.Int32(9)
<System.Int32 object at ...>
```

or, a one-dimensional Array of the specified Type

```
>>> array = System.Array[int](list(range(10)))
>>> array
<System.Int32[] object at ...>
>>> list(array)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(continues on next page)

(continued from previous page)

```
>>> array[0] = -1
>>> list(array)
[-1, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

1.2.4 Java

Since Java byte code is executed in a JVM it doesn't matter whether it was built with a 32- 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 (view the *.jar source code*)

```
>>> from msl.loadlib import LoadLibrary
>>> from msl.examples.loadlib import EXAMPLES_DIR
>>> jar = LoadLibrary(EXAMPLES_DIR + '/java_lib.jar')
>>> jar
<LoadLibrary libtype=JVMView path=...java_lib.jar>
>>> jar.gateway
<py4j.java_gateway.JavaGateway object at ...>
```

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

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

Calculate the square root of a number using the `MathUtils` class

```
>>> MathUtils.sqrt(32.4)
5.692099788303...
```

Solve a linear system of equations, Ax=b

```
>>> A = jar.gateway.new_array(jar.lib.Double, 3, 3)
>>> coeff = [[3, 2, -1], [7, -2, 4], [-1, 5, 1]]
>>> for i in range(3):
...     for j in range(3):
...         A[i][j] = float(coeff[i][j])
...
>>> b = jar.gateway.new_array(jar.lib.Double, 3)
>>> b[0] = 4.0
>>> b[1] = 15.0
>>> b[2] = 12.0
>>> x = Matrix.solve(Matrix(A), Matrix(b))
>>> print(x.toString())
+1.000000e+00
+2.000000e+00
+3.000000e+00
```

Verify that x is the solution

```
>>> for i in range(3):
...     x_i = 0.0
...     for j in range(3):
...         x_i += coeff[i][j] * x.getValue(j,0)
...     assert abs(x_i - b[i]) < 1e-12
...
```

Shutdown the connection to the JVM when you are finished

```
>>> jar.gateway.shutdown()
```

Load Java byte code (view the *.class source code*)

```
>>> cls = LoadLibrary(EXAMPLES_DIR + '/Trig.class')
>>> cls
<LoadLibrary libtype=JVMView path=...Trig.class>
>>> cls.lib
<py4j.java_gateway.JVMView object at ...>
```

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

```
>>> Trig = cls.lib.Trig
>>> Trig
<py4j.java_gateway.JavaClass object at ...>
>>> Trig.cos(1.2)
0.3623577544766...
>>> Trig.asin(0.6)
0.6435011087932...
>>> Trig.tanh(1.3)
0.8617231593133...
```

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

```
>>> cls.gateway.shutdown()
```

1.2.5 COM

To load a Component Object Model (COM library) you 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 is only valid on Windows.

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

```
>>> from msl.loadlib import LoadLibrary
>>> 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 by using the CreateTextFile method

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

Verify that the file exists and that the text is correct

```
>>> com.lib.FileExists('a_new_file.txt')
True
>>> file = com.lib.OpenTextFile('a_new_file.txt')
>>> file.ReadAll()
'This is a test.'
>>> file.Close()
0
```

1.2.6 Windows __stdcall

Load a 32-bit Windows `__stdcall` library in 32-bit Python, see kernel32. Include the `'windll'` argument to specify that the calling convention is `__stdcall`.

```
>>> from msl.loadlib import LoadLibrary
>>> kernel = LoadLibrary(r'C:\Windows\SysWOW64\kernel32.dll', 'windll')
>>> kernel
<LoadLibrary libtype=WinDLL path=C:\Windows\SysWOW64\kernel32.dll>
>>> kernel.lib
<WinDLL 'C:\Windows\SysWOW64\kernel32.dll', handle ... at ...>
>>> from ctypes import pointer
>>> from msl.examples.loadlib.kernel32 import SystemTime
>>> st = SystemTime()
>>> time = kernel.lib.GetLocalTime(pointer(st))
```

Now that we have a SYSTEMTIME structure we can access its attributes

```
>>> from datetime import datetime
>>> today = datetime.today()
>>> st.wYear == today.year
True
>>> st.wMonth == today.month
True
>>> st.wDay == today.day
True
```

See *here* for an example on how to communicate with kernel32 from 64-bit Python.

1.2.7 LabVIEW

Load a 64-bit LabVIEW library in 64-bit Python (view the *LabVIEW source code*). *To load the 32-bit library 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.

```
>>> from msl.loadlib import LoadLibrary
>>> from msl.examples.loadlib import EXAMPLES_DIR
>>> labview = LoadLibrary(EXAMPLES_DIR + '/labview_lib64.dll')
>>> labview
<LoadLibrary libtype=CDLL path=...labview_lib64.dll>
>>> labview.lib
<CDLL '...labview_lib64.dll', handle ... at ...>
```

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

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

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

```
>>> 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

```
>>> 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

```
>>> 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
```

1.3 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.

`echo32`	An example of a 32-bit *echo* server.
`echo64`	An example of a 64-bit *echo* client.
`cpp32`	A wrapper around a 32-bit C++ library, *cpp_lib32*.
`cpp64`	Communicates with *cpp_lib32* via the *`Cpp32`* class.
`fortran32`	A wrapper around a 32-bit FORTRAN library, *fortran_lib32*.
`fortran64`	Communicates with *fortran_lib32* via the *`Fortran32`* class.
`dotnet32`	A wrapper around a 32-bit .NET library, *dotnet_lib32*.
`dotnet64`	Communicates with a 32-bit .NET library via the *`DotNet32`* class.
`kernel32`	A wrapper around the 32-bit Windows kernel32.dll library.
`kernel64`	Communicates with kernel32.dll via the *`Kernel32`* class.
`labview32`	A wrapper around a 32-bit LabVIEW library, *labview_lib32*.
`labview64`	Communicates with *labview_lib32* via the *`Labview32`* class.

The following illustrates a minimal usage example. The *my_lib.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 which is running within a 32-bit process. *MyServer* hosts the library at a specified host address and port number. Any class that is a subclass of *`Server32`* *must* provide two arguments in its constructor: `host` and `port`. Including keyword arguments in the constructor is optional.

```
# my_server.py

from msl.loadlib import Server32

class MyServer(Server32):
```

(continues on next page)

(continued from previous page)

```
    """Wrapper around a 32-bit C++ library 'my_lib.dll' that has an 'add' and
↪'version' function."""

    def __init__(self, host: str, port: int, **kwargs: str) -> None:
        # Load the 'my_lib' shared-library file using ctypes.CDLL
        super(MyServer, self).__init__('my_lib.dll', 'cdll', host, port)

        # The Server32 class has a 'lib' property that is a reference to the␣
↪ctypes.CDLL object

        # Call the version function from the library
        self.version: str = self.lib.version()

    def add(self, a: int, b: int) -> int:
        # The shared library's 'add' function takes two integers as inputs and␣
↪returns the sum
        return self.lib.add(a, b)
```

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

```
# my_client.py

from msl.loadlib import Client64

class MyClient(Client64):
    """Call a function in 'my_lib.dll' via the 'MyServer' wrapper."""

    def __init__(self) -> None:
        # 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: int, b: int) -> int:
        # 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)

    def version(self) -> str:
        # Get the version
        return self.request32('version')
```

The *MyClient* class would then be used as follows

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

(continues on next page)

(continued from previous page)

```
>>> c.version()
1
```

Keyword arguments, *kwargs*, that the *Server32* subclass requires can be passed to the server from the client (see, *Client64*). 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 values of the *kwargs* will be of type `str` at the constructor of the *Server32* subclass. These *kwargs* are the only arguments where the data type is not preserved for the client-server protocol. See the "Echo" example which shows that data types are preserved between client-server method calls.

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 illustrates that 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.

Create an *Echo64* object

```
>>> from msl.examples.loadlib import Echo64
>>> echo = Echo64()
```

send a boolean as an argument, see *send_data()*

```
>>> echo.send_data(True)
((True,), {})
```

send a boolean as a keyword argument

```
>>> echo.send_data(boolean=True)
((), {'boolean': True})
```

send multiple data types as arguments and as keyword arguments

```
>>> echo.send_data(1.2, {'my_list':[1, 2, 3]}, 0.2j, range(10), x=True, y=
↪'hello world!')
((1.2, {'my_list': [1, 2, 3]}, 0.2j, range(0, 10)), {'x': True, 'y': 'hello␣
↪world!'})
```

Shutdown the 32-bit server when you are done

```
>>> stdout, stderr = echo.shutdown_server32()
```

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

This example shows how to access a 32-bit C++ library from 64-bit Python. *Cpp32* is the 32-bit server and *Cpp64* is the 64-bit client. The source code of the C++ program is available *here*.

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

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.

Create a *Cpp64* client to communicate with the 32-bit *cpp_lib32* library from 64-bit Python

```
>>> from msl.examples.loadlib import Cpp64
>>> cpp = Cpp64()
```

Add two integers, see *add()*

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

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

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

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

```
>>> cpp.add_or_subtract(1.0, 2.0, True)
3.0
>>> cpp.add_or_subtract(1.0, 2.0, False)
-1.0
```

Arrays

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

Attention: The *scalar_multiply()* function takes a pointer to an array as an input argument, see *cpp_lib.h*. One cannot pass pointers from *Client64* to *Server32* because a 64-bit process cannot share the same memory space as a 32-bit process. All 32-bit pointers must be created (using `ctypes`) in the class that is a subclass of *Server32* and only the **value** that is stored at that address can be returned to *Client64* for use in the 64-bit program.

```
>>> 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 included by default in the 32-bit server, but could be – see *Create a custom 32-bit server* for more details). To simplify the procedure you could convert the ndarray to a `list` using the `numpy.ndarray.tolist()` method

```
>>> 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]
```

or you could use the builtin `array.array` class

```
>>> from array import array
>>> b = array('d', a.tobytes())
>>> cpp.scalar_multiply(3.1, b)
[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

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

Strings

In this example the memory for the reversed string is allocated in Python, see *reverse_string_v1()*

```
>>> cpp.reverse_string_v1('hello world!')
'!dlrow olleh'
```

In this example the memory for the reversed string is allocated in C++, see *reverse_string_v2()*

```
>>> cpp.reverse_string_v2('uncertainty')
'ytniatrecnu'
```

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* back to *Cpp64*.

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 (client) and Python 3 (server).

The *cpp_lib32* library contains the following structs

```
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*

```
>>> 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.

```
>>> for i in range(16):
...     print(cpp.circumference(0.5, 2**i))
...
0.0
2.0
2.828427124746...
3.061467458920...
3.121445152258...
3.136548490545...
3.140331156954...
3.141277250932...
3.141513801144...
3.141572940367...
3.141587725277...
3.141591421511...
3.141592345569...
```

(continues on next page)

(continued from previous page)

```
3.141592576584...
3.141592634337...
3.141592648775...
```

Shutdown the 32-bit server when you are done communicating with the 32-bit library

```
>>> stdout, stderr = cpp.shutdown_server32()
```

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

This example shows how to access a 32-bit FORTRAN library from 64-bit Python. *Fortran32* is the 32-bit server and *Fortran64* is the 64-bit client. The source code of the FORTRAN program is available *here*.

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

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.

Create a *Fortran64* client to communicate with the 32-bit *fortran_lib32* library

```
>>> from msl.examples.loadlib import Fortran64
>>> f = Fortran64()
```

Add two `int8` values, see *sum_8bit()*

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

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

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

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

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

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

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

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

```
>>> f.multiply_float32(1e30, 2e3)
1.99999998899...e+33
```

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

```
>>> f.multiply_float64(1e30, 2e3)
2.00000000000...e+33
```

Check if a value is positive, see *is_positive()*

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

Add or subtract two integers, see *add_or_subtract()*

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

Calculate the n'th factorial, see *factorial()*

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

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

```
>>> f.standard_deviation([float(val) for val in range(1,10)])
2.73861278752583...
```

Compute the Bessel function of the first kind of order 0, see *besselJ0()*

```
>>> f.besselJ0(8.6)
0.0146229912787412...
```

Reverse a string, see *reverse_string()*

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

Add two 1D arrays, see *add_1d_arrays()*

```
>>> 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]
```

(continues on next page)

(continued from previous page)

```
>>> 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()*

```
>>> 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 32-bit server when you are done communicating with the 32-bit library

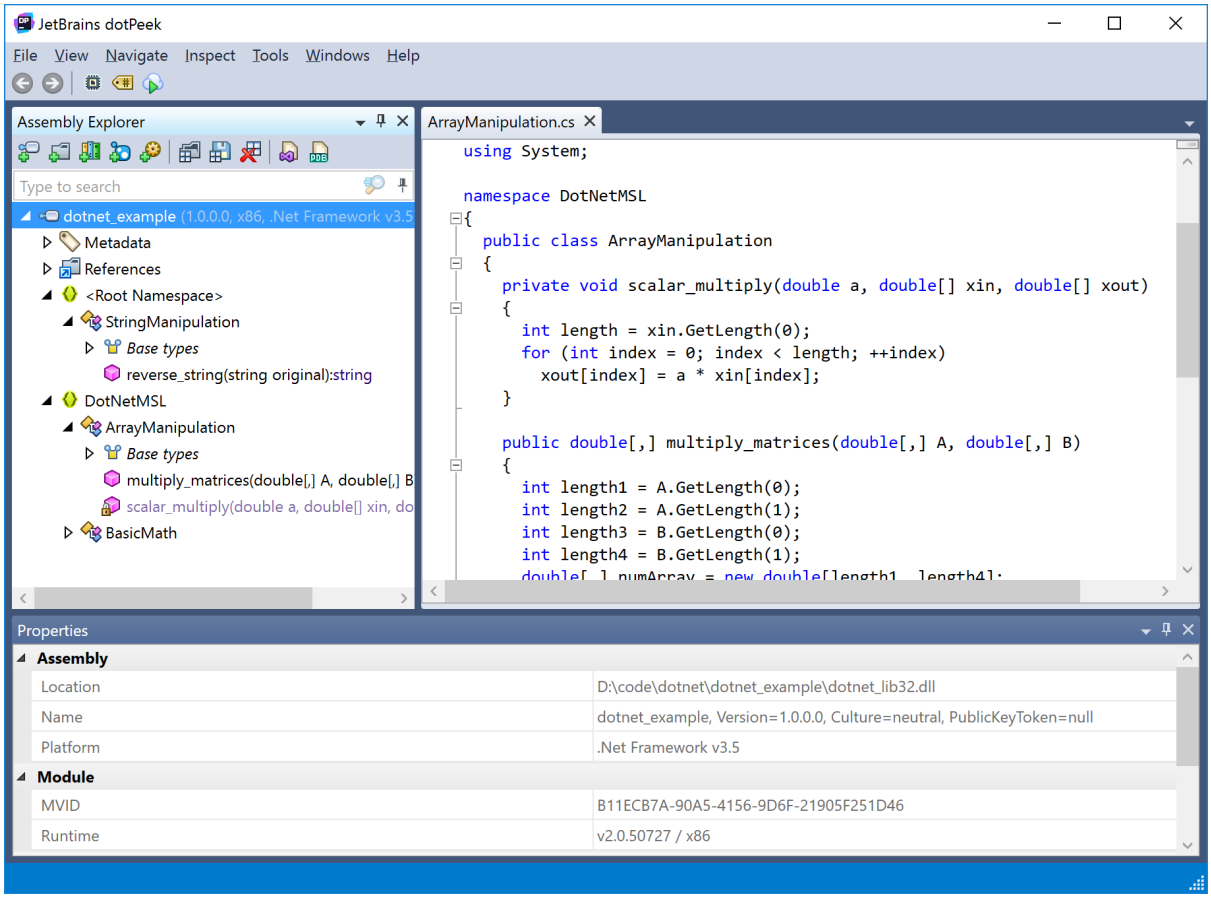
```
>>> stdout, stderr = f.shutdown_server32()
```

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

This example shows how to access a 32-bit .NET library from 64-bit Python. *DotNet32* is the 32-bit server and *DotNet64* is the 64-bit client. The source code of the C# program is available *here*.

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

Tip: The JetBrains dotPeek program can be used to decompile a .NET assembly into the equivalent source code. For example, *peeking* inside the *dotnet_lib32.dll* library, that the *DotNet32* class is a wrapper around, gives



Attention: To configure *pythonnet* to use the .NET Core runtime, you must either run

```
from pythonnet import load
load("coreclr")
```

or define a `PYTHONNET_RUNTIME=coreclr` environment variable

```
import os
os.environ["PYTHONNET_RUNTIME"] = "coreclr"
```

before `super()` is called in the *Server32* subclass. To use the Mono runtime, replace `"coreclr"` with `"mono"`.

Create a *DotNet64* client to communicate with the 32-bit *dotnet_lib32.dll* library

```
>>> from msl.examples.loadlib import DotNet64
>>> dn = DotNet64()
```

Get the names of the classes in the .NET library module, see *get_class_names()*

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

Add two integers, see *add_integers()*

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

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

```
>>> dn.divide_floats(3., 2.)
1.5
```

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

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

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

```
>>> 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()*

```
>>> 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()*

```
>>> 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()*

```
>>> 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 '
>>> dn.concatenate('the ', 'experiment ', 'worked ', True, 'temporarily')
'the experiment worked temporarily'
```

Shutdown the 32-bit server when you are done communicating with the 32-bit library

```
>>> stdout, stderr = dn.shutdown_server32()
```

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

This example shows how to access the 32-bit Windows kernel32 library from 64-bit Python. *Kernel32* is the 32-bit server and *Kernel64* is the 64-bit client.

Create a *Kernel64* client to communicate with the 32-bit kernel32 library

```
>>> from msl.examples.loadlib import Kernel64
>>> k = Kernel64()
>>> k.lib32_path
'C:\\Windows\\SysWOW64\\kernel32.dll'
```

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

```
>>> k.get_local_time()
datetime.datetime(2021, 1, 21, 15, 29, 8, 482000)
```

Shutdown the 32-bit server when you are done communicating with the 32-bit library

```
>>> stdout, stderr = k.shutdown_server32()
```

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

This example shows how to access a 32-bit LabVIEW library from 64-bit Python. *Labview32* is the 32-bit server and *Labview64* is the 64-bit client. The source code of the LabVIEW program is available *here*.

Attention: This example requires that a 32-bit LabVIEW Run-Time Engine is installed and that the operating system is Windows.

Create a *Labview64* client to communicate with the 32-bit *labview_lib32* library

```
>>> from msl.examples.loadlib import Labview64
>>> labview = Labview64()
```

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

```
>>> 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

```
>>> labview.stdev(data, 1)
(5.0, 6.666666666666667, 2.581988897471611)
```

Shutdown the 32-bit server when you are done communicating with the 32-bit library

```
>>> stdout, stderr = labview.shutdown_server32()
```

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)

```
from msl.loadlib import Client64

class LinearAlgebra(Client64):

    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

```
from msl.loadlib import Client64

class LinearAlgebra(Client64):

    def __init__(self):
        super(LinearAlgebra, self).__init__(module32='linear_algebra_32.py')

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

1.4 MSL-LoadLib API Documentation

The root package is

msl.loadlib	Load a shared library.

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

LoadLibrary(path[, libtype])	Load a shared library.

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

Client64(module32, *[, add_dll_directory, ...])	Base class for communicating with a 32-bit library from 64-bit Python.
Server32(path, libtype, host, port, **kwargs)	Base class for loading a 32-bit library in 32-bit Python.

a function to create a frozen 32-bit server

freeze_server32	Create a 32-bit server for inter-process communication.

some general helper functions

utils	Common functions used by the **MSL-LoadLib** package.

and a module for interacting with ActiveX libraries

activex	Helper module for loading an ActiveX library in an application window.

1.4.1 Package Structure

msl.loadlib package

Load a shared library.

msl.loadlib.**version_info = (1, 0, 0, 'dev0')**

Contains the version information as a (major, minor, micro, releaselevel) named tuple.

msl.loadlib.activex module

Helper module for loading an ActiveX library in an application window.

Example usage,

```
import sys

from msl.loadlib.activex import Application, Icon, MenuGroup, MenuItem

def letter_clicked(item: MenuItem):
    print(item, item.data)
    if item.text == 'C':
        item.checked = not item.checked

def group_clicked(item: MenuItem):
    group.checked = item
    print(item)

# Create an application window
app = Application(title='My ActiveX Control', icon=Icon(sys.executable))

# Create a new 'Letters' menu
letters = app.menu.create('Letters')

# append items to the 'Letters' menu
app.menu.append(letters, 'A')
app.menu.append(letters, 'B', callback=letter_clicked, data=1)
app.menu.append_separator(letters)
app.menu.append(letters, 'C', callback=letter_clicked, data=[1, 2, 3])

# Create a new menu group
group = MenuGroup()
group.append('Group 1', callback=group_clicked)
group.append('Group 2', callback=group_clicked)
group.append('Group 3', callback=group_clicked)

# Create a new 'Numbers' menu
numbers = app.menu.create('Numbers')
# add the group to the 'Numbers' menu
app.menu.append_group(numbers, group)
# add a separator then another item
```

(continues on next page)

(continued from previous page)

```
app.menu.append_separator(numbers)
app.menu.append(numbers, 'Not in Group')

# Load an ActiveX control in the main application window
# ocx = app.load('My.OCX.Application', width=300, height=300)

app.set_window_size(300, 300)

app.show()
app.run()
```

Classes

- *Application*
- *Icon*
- *Menu*
- *MenuItem*
- *MenuGroup*

class msl.loadlib.activex.**Application**(*, *background=Background.WHITE*, *class_style=<ClassStyle.NONE: 0>*, *icon=None*, *style=<WindowStyle.OVERLAPPEDWINDOW: 13565952>*, *title='ActiveX'*)

Bases: object

Create the main application window to display ActiveX controls.

Parameters

- **background** (*int*) – The background colour of the main window (a *Background* value).
- **class_style** (*int*) – The class style(s). Can be any combination (bitwise OR) of *ClassStyle* values.
- **icon** (Icon) – The application icon.
- **style** (*int*) – The window style(s). Can be any combination (bitwise OR) of *WindowStyle* values.
- **title** (*str*) – The text to display in the titlebar (if one is visible).

add_message_handler(*handler*)

Add a custom handler for processing window messages.

Messages correspond to events from the user and from the operating system.

Parameters

handler (*Callable[[int, int, int, int], None]*) – A function that processes messages sent to the main window. The function must accept four positional arguments (all integer values) and the returned object is ignored. See WindowProc for more details about the input arguments to the *handler*.

Return type
None

close()

Close the application.

Return type
None

handle_events(*source*, *sink=None*, *, *interface=None*)

Handle events from an ActiveX object.

Parameters

- **source** (*Any*) – An ActiveX object that emits events.
- **sink** (*Callable[[...], Any]*) – The object that handles the events. The *sink* must define methods with the same names as the ActiveX event names. If not specified, the *Application* instance is used as the *sink*.
- **interface** (*Any*) – The COM interface to use.

Returns
The advise-connection object.

property hwnd: int

Returns the handle to the main application window.

load(*activex_id*, *, *parent=None*, *x=0*, *y=0*, *width=0*, *height=0*, *style=<WindowStyle.CHILD|VISIBLE: 1342177280>*, *ex_style=<ExtendedWindowStyle.NONE: 0>*)

Load an ActiveX library.

Parameters

- **activex_id** (*str*) – ProgID or CLSID of the ActiveX object.
- **parent** (*int*) – The handle to the parent window that the ActiveX object will belong to. Default is the main application window.
- **x** (*int*) – Horizontal position of the ActiveX object in the parent window.
- **y** (*int*) – Vertical position of the ActiveX object in the parent window.
- **width** (*int*) – Width (in pixels) of the ActiveX object.
- **height** (*int*) – Height (in pixels) of the ActiveX object.
- **style** (*int*) – Style of the window that is created to contain the ActiveX object. Can be any combination (bitwise OR) of *WindowStyle* values.
- **ex_style** (*int*) – Extended style of the window that is created to contain the ActiveX object. Can be any combination (bitwise OR) of *ExtendedWindowStyle* values.

Returns
The interface pointer to the ActiveX library.

Return type
POINTER

property menu: *Menu*

Returns the menu instance.

static message_box(*, *hwnd=None*, *language_id=0*, *options=<MessageBoxOption.OK: 0>*, *text=''*, *title=''*)

Displays a modal dialog box.

Parameters

- **hwnd** (*int*) – A handle to the owner window of the message box to be created.
- **language_id** (*int*) – The language for the text displayed in the message box button(s).
- **options** (*int*) – The contents and behavior of the dialog box. Can be any combination (bitwise OR) of *MessageBoxOption* values.
- **text** (*str*) – The message to be displayed.
- **title** (*str*) – The dialog box title.

Returns

An indication of how the message box was closed.

Return type

int

static run()

Run the application.

This is a blocking call. Create and run the application in a separate thread if you want to execute other code while the application is running.

Return type

None

set_window_position(*x*, *y*, *width*, *height*, *, *flags=<PositionFlag.NONE: 0>*)

Set the position of the main window.

Parameters

- **x** (*int*) – The new position of the left side of the window.
- **y** (*int*) – The new position of the top of the window.
- **width** (*int*) – The new width of the window (in pixels).
- **height** (*int*) – The new height of the window (in pixels).
- **flags** (*int*) – The window sizing and positioning flags. Can be any combination (bitwise OR) of *PositionFlag* values.

Return type

None

set_window_size(*width*, *height*)

Set the size of the main window.

Parameters

- **width** (*int*) – The new width of the window (in pixels).

- **height** (*int*) – The new height of the window (in pixels).

Return type
None

set_window_title(*title*)

Set the text to display in the window's title bar.

Parameters
title (*str*) – The title bar text.

Return type
None

show(*command=ShowWindow.SHOWNORMAL*)

Show the main application window.

Parameters
command (*int*) – Controls how the window is shown (a *ShowWindow* value).

Return type
None

property thread_id: int

Returns the identifier of the thread that created the main application window.

class msl.loadlib.activex.**Icon**(*file*, *, *index=0*, *hinstance=None*)

Bases: object

Extract an icon from an executable file, DLL or icon file.

Parameters

- **file** (*str*) – The path to an executable file, DLL or icon file.
- **index** (*int*) – The zero-based index of the icon to extract.
- **hinstance** (*int*) – Handle to the instance of the calling application.

property hicon: int | None

Returns the handle to the icon or None if no icon was found.

destroy()

Destroys the icon and frees any memory the icon occupied.

Return type
None

class msl.loadlib.activex.**Menu**

Bases: object

A menu associated with the main application window.

Do not instantiate directly. Use the *Application.menu* property to access the menu instance.

append(*hmenu*, *text*, *, *callback=None*, *data=None*, *flags=<MenuFlag.ENABLED: 0>*)

Create a new *MenuItem* and append it to a popup menu.

Parameters

- **hmenu** (*int*) – The handle of a popup menu to append the new menu item to.
- **text** (*str*) – The content of the new menu item.
- **callback** (*Callable[[MenuItem], None] | None*) – A callable object that will be called when this menu item is selected. The callable object will receive the *MenuItem* instance as an argument and the returned object is ignored.
- **data** (*Any*) – User data associated with the menu item.
- **flags** (*int*) – Controls the appearance and behaviour of the new menu item. Can be any combination (bitwise OR) of *MenuFlag* values.

Returns
The menu item that was appended.

Return type
MenuItem

append_group(*hmenu*, *menu_group*)

Append a group of menu items to a popup menu.

Parameters

- **hmenu** (*int*) – The handle of a popup menu to append the group to.
- **menu_group** (MenuGroup) – A group of menu items.

Return type
None

append_separator(*hmenu*)

Append a horizontal dividing line to a popup menu.

Parameters
hmenu (*int*) – The handle to a popup menu.

Return type
None

create(*text*)

Create a new popup menu and append it to the main menu.

Parameters
text (*str*) – The text to display for the popup menu.

Returns
The handle to the popup menu that was created.

Return type
int

property hmenu: int

Returns the handle to the main menu.

class msl.loadlib.activex.**MenuItem**(***kwargs*)

Bases: object

A menu item that belongs to a popup menu.

Do not instantiate this class directly. Use *MenuGroup.append()* or *Menu.append()* to create a new menu item.

data: Any

User data associated with the menu item.

property callback: Callable[[*MenuItem*], None] | None

The callback function to call when the menu item is clicked.

property checked: bool

Whether the menu item's check mark is shown.

property flags: int

The flags that were used to create the menu item.

property hmenu: int

The handle to the popup menu that the menu item belongs to.

property id: int

The identifier of the menu item.

property text: str

The content of the menu item.

class msl.loadlib.activex.**MenuGroup**(*name=''*)

Bases: object

A group of *MenuItem*'s where only one item in the group may have a check mark to indicate that a particular item is selected.

Parameters

name (*str*) – A name to associate with the group.

append(*text*, *, *callback=None*, *data=None*, *flags=<MenuFlag.ENABLED: 0>*)

Create a new *MenuItem* and append it to the group.

Parameters

- **text** (*str*) – The content of the new menu item.
- **callback** (*Callable[[MenuItem], None] | None*) – A callable object that will be called when this menu item is selected. The callable object will receive the *MenuItem* instance as an argument and the returned object is ignored.
- **data** (*Any*) – User data associated with the menu item.
- **flags** (*int*) – Controls the appearance and behaviour of the new menu item. Can be any combination (bitwise OR) of *MenuFlag* values.

Returns

The menu item that was appended to the group.

Return type

MenuItem

append_separator()

Append a horizontal dividing line to the group.

Return type
None

property checked: *MenuItem* | None

Returns the menu item that is currently checked in the group.

property name: str

Returns the name of the menu group.

Enumerations

- *Background*
- *ClassStyle*
- *ExtendedWindowStyle*
- *MenuFlag*
- *MessageBoxOption*
- *PositionFlag*
- *ShowWindow*
- *WindowStyle*

class msl.loadlib.activex.**Background**(*value*)

Bases: IntEnum

Background colours.

WHITE = 0

LIGHT_GREY = 1

GREY = 2

DARK_GREY = 3

BLACK = 4

class msl.loadlib.activex.**ClassStyle**(*value*)

Bases: IntFlag

Window class style flags. See window-class-styles for more details.

BYTEALIGNCLIENT = 0x01000

BYTEALIGNWINDOW = 0x02000

CLASSDC = 0x00040

DBLCLKS = 0x00008

DROPSHADOW = 0x20000

GLOBALCLASS = 0x04000

HREDRAW = 0x00002

NOCLOSE = 0x00200

OWNDC = 0x00020

PARENTDC = 0x00080

SAVEBITS = 0x00800

VREDRAW = 0x00001

class msl.loadlib.activex.**ExtendedWindowStyle**(*value*)

Bases: IntFlag

Extended window style flags. See extended-window-styles for more details.

DLGMODALFRAME = 0x0000001

NOPARENTNOTIFY = 0x0000004

TOPMOST = 0x0000008

ACCEPTFILES = 0x0000010

TRANSPARENT = 0x0000020

MDICHILD = 0x0000040

TOOLWINDOW = 0x0000080

WINDOWEDGE = 0x0000100

CLIENTEDGE = 0x0000200

CONTEXTHELP = 0x0000400

RIGHT = 0x0001000

RTLREADING = 0x0002000

LEFTSCROLLBAR = 0x0004000

CONTROLPARENT = 0x0010000

STATICEDGE = 0x0020000

APPWINDOW = 0x0040000

LAYERED = 0x0080000

NOINHERITLAYOUT = 0x0100000

NOREDIRECTIONBITMAP = 0x0200000

LAYOUTRTL = 0x0400000

COMPOSITED = 0x2000000

NOACTIVATE = 0x8000000

class msl.loadlib.activex.**MenuFlag**(*value*)

Bases: IntFlag

Menu item flags. See append-menu for more details.

BITMAP = 0x004

CHECKED = 0x008

DISABLED = 0x002

GRAYED = 0x001

MENUBARBREAK = 0x020

MENUBREAK = 0x040

OWNERDRAW = 0x100

POPUP = 0x010

SEPARATOR = 0x800

class msl.loadlib.activex.**MessageBoxOption**(*value*)

Bases: IntFlag

Message box options. See message-box for more details.

ABORTRETRYIGNORE = 0x000002

HELP = 0x004000

OKCANCEL = 0x000001

YESNO = 0x000004

ICONINFORMATION = 0x000040

ICONQUESTION = 0x000020

ICONSTOP = 0x000010

DEFBUTTON2 = 0x000100

DEFBUTTON3 = 0x000200

SYSTEMMODAL = 0x001000

TASKMODAL = 0x002000

DEFAULT_DESKTOP_ONLY = 0x020000

RIGHT = 0x080000

RTLREADING = 0x100000

SETFOREGROUND = 0x010000

TOPMOST = 0x040000

SERVICE_NOTIFICATION = 0x200000

class msl.loadlib.activex.**PositionFlag**(*value*)

Bases: IntFlag

Window position flags. See set-window-pos for more details.

ASYNCWINDOWPOS = 0x4000

DEFERERASE = 0x2000

DRAWFRAME = 0x0020

HIDEWINDOW = 0x0080

NOACTIVATE = 0x0010

NOCOPYBITS = 0x0100

NOMOVE = 0x0002

NOOWNERZORDER = 0x0200

NOREDRAW = 0x0008

NOSENDCHANGING = 0x0400

NOSIZE = 0x0001

NOZORDER = 0x0004

SHOWWINDOW = 0x0040

class msl.loadlib.activex.**ShowWindow**(*value*)

Bases: IntEnum

Show window commands. See show-window for more details.

HIDE = 0

SHOWNORMAL = 1

SHOWMINIMIZED = 2

SHOWMAXIMIZED = 3

SHOWNOACTIVATE = 4

SHOW = 5

MINIMIZE = 6

SHOWMINNOACTIVE = 7

SHOWNA = 8

RESTORE = 9

SHOWDEFAULT = 10

FORCEMINIMIZE = 11

class msl.loadlib.activex.**WindowStyle**(*value*)

Bases: IntFlag

Window style flags. See window-styles for more details.

POPUP = 0x80000000

CHILD = 0x40000000

MINIMIZE = 0x20000000

VISIBLE = 0x10000000

DISABLED = 0x08000000

CLIPSIBLINGS = 0x04000000

CLIPCHILDREN = 0x02000000

MAXIMIZE = 0x01000000

BORDER = 0x00800000

DLGFRAME = 0x00400000

VSCROLL = 0x00200000

HSCROLL = 0x00100000

SYSMENU = 0x00080000

THICKFRAME = 0x00040000

GROUP = 0x00020000

TABSTOP = 0x00010000

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.

class msl.loadlib.client64.**Client64**(*module32*, *, *add_dll_directory=None*, *append_environ_path=None*, *append_sys_path=None*, *host='127.0.0.1'*, *port=None*, *protocol=5*, *rpc_timeout=None*, *server32_dir=None*, *timeout=10*, ***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 class that is a wrapper around a 32-bit library. *Client64* 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* argument.

Changed in version 0.8: Added the *protocol* argument and the default *quiet* value became None.

Changed in version 0.10: Added the *server32_dir* argument.

Changed in version 1.0: Removed the deprecated *quiet* argument. The *host* value may now be *None*. Added the *add_dll_directory* argument.

Parameters

- **module32** (PathLike) – The name of, or the path to, a Python module that will be imported by the 32-bit server. The module must contain a class that inherits from *Server32*.
- **add_dll_directory** (PathLike | *Iterable*[PathLike] | *None*) – Add path(s) to the 32-bit server's DLL search path. See os.add_dll_directory() for more details. Available on Windows only.
- **append_environ_path** (PathLike | *Iterable*[PathLike] | *None*) – Append path(s) to the 32-bit server's os.environ['PATH'] variable. This may be useful if the library that is being loaded requires additional libraries that must be available on PATH.
- **append_sys_path** (PathLike | *Iterable*[PathLike] | *None*) – 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.

- **host** (*str* | *None*) – The hostname (IP address) of the 32-bit server. If None then the connection to the server is mocked. See *Mocking the connection to the server* for more details.
- **port** (*int* | *None*) – The port to open on the 32-bit server. If None, an available port will be used.
- **protocol** (*int*) – The pickle protocol to use.
- **rpc_timeout** (*float* | *None*) – 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, 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*).
- **server32_dir** (PathLike | *None*) – The directory where the frozen 32-bit server is located.

- **timeout** (*float*) – The maximum number of seconds to wait to establish a connection with the 32-bit server.
- **kwargs** (*Any*) – All additional keyword arguments are passed to the *Server32* subclass. The data type of each value is not preserved. It will be of type `str` at the constructor of the *Server32* subclass.

Raises

- `OSError` – If the 32-bit server cannot be found.
- ***ConnectionTimeoutError*** – If the connection to the 32-bit server cannot be established.

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.

---

property host: `str | None`

The host address of the 32-bit server.

property port: `int`

The port number of the 32-bit server.

property connection: `HTTPConnection | None`

The connection to the 32-bit server.

property lib32_path: `str`

The path to the 32-bit shared-library file.

request32(*name*, **args*, ***kwargs*)

Send a request to the 32-bit server.

Parameters

- **name** (*str*) – The name of a method, property or attribute of the *Server32* subclass.
- **args** (*Any*) – The arguments that the method in the *Server32* subclass requires.
- **kwargs** (*Any*) – The keyword arguments that the method in the *Server32* subclass requires.

Returns

Whatever is returned by calling *name*.

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.

Return type
Any

shutdown_server32(*kill_timeout=10*)

Shutdown the 32-bit server.

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

Changed in version 0.6: Added the *kill_timeout* argument.

Changed in version 0.8: Returns the (stdout, stderr) streams from the 32-bit server.

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

Returns
The (stdout, stderr) streams from the 32-bit server. Limit the total number of characters that are written to either stdout or stderr on the 32-bit server to be < 4096. This will avoid potential blocking when reading the stdout and stderr PIPE buffers.

Return type
tuple[*BinaryIO*, *BinaryIO*]

Note: This method gets called automatically when the reference count to the *Client64* object reaches zero (see `__del__()`).

---

msl.loadlib.exceptions module

Exception classes.

exception msl.loadlib.exceptions.**ConnectionTimeoutError**(*message*)

Bases: `OSError`

Raised when the connection to the 32-bit server cannot be established.

Parameters
message (*str*) – The error message.

Return type
None

exception msl.loadlib.exceptions.**Server32Error**(*value*, *, *name=''*, *traceback=''*)

Bases: `HTTPException`

Raised when an exception occurs on the 32-bit server.

Added in version 0.5.

Parameters

- **value** (*str*) – The error message.
- **name** (*str*) – The name of the exception type.

- **traceback** (*str*) – The exception traceback.

Return type
None

property name: str

The name of the exception type.

property traceback: str

The exception traceback.

property value: str

The error message.

exception msl.loadlib.exceptions.**ResponseTimeoutError**

Bases: OSError

Raised when a timeout occurs while waiting for a response from the 32-bit server.

Added in version 0.6.

msl.loadlib.freeze_server32 module

Create a 32-bit server for inter-process communication.

There is also a *command-line utility* to create a server.

msl.loadlib.freeze_server32.**main**(*, *spec=None*, *dest=None*, *imports=None*, *data=None*, *skip_32bit_check=False*)

Create a frozen server.

This function should be run using a 32-bit Python interpreter with PyInstaller installed.

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

Changed in version 0.10: Added the *dest* argument.

Changed in version 1.0: Removed the *requires_pythonnet* and *requires_comtypes* arguments. Added the *imports*, *data* and *skip_32bit_check* arguments.

Parameters

- **spec** (*str | None*) – The path to a spec file to use to create the frozen server.
- **dest** (*str | None*) – The destination directory to save the server to. Default is the current directory.
- **imports** (*str | Iterable[str] | None*) – The names of additional modules and packages that must be importable on the server.
- **data** (*str | Iterable[str] | None*) – The path(s) to additional data files, or directories containing data files, to be added to the frozen server. Each value should be in the form *source:dest_dir*, where *:dest_dir* is optional. *source* is the path to a file (or a directory of files) to add. *dest_dir* is an optional destination directory, relative to the top-level directory of the frozen server, to add the file(s) to. If *dest_dir* is not specified, the file(s) will be added to the top-level directory of the server.

- **skip_32bit_check** (*bool*) – In the rare situation that you want to create a frozen 64-bit server, you can set this value to `True` which skips the requirement that a 32-bit version of Python must be used to create the server. Before you create a 64-bit server, decide if *Mocking the connection to the server* is a better solution for your application.

Return type
None

> **Attention:** If a value for *spec* is specified, then *imports* nor *data* may be specified.

msl.loadlib.load_library module

Load a shared library.

`msl.loadlib.load_library.`**LibType**

The library type.

alias of `Literal`['cdll', 'windll', 'oledll', 'net', 'clr', 'java', 'com', 'activex']

class `msl.loadlib.load_library.`**PathLike**

A path-like object.

alias of TypeVar('PathLike', str, bytes, ~os.PathLike)

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.

Changed in version 0.4: Added support for Java archives.

Changed in version 0.5: Added support for COM libraries.

Changed in version 0.9: Added support for ActiveX libraries.

Parameters

- **path** (`PathLike`) – 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, then
  2. use `ctypes.util.find_library()`, then
  3. search `sys.path`, then
  4. search `os.environ['PATH']`.

  If loading a COM library, *path* may either be the

  – ProgID (e.g., `InternetExplorer.Application`), or the

  – CLSID (e.g., `{2F7860A2-1473-4D75-827D-6C4E27600CAC}`).

- **libtype** (*LibType | None*) – The library type.

  The following values are currently supported:

  - *cdll*: a library that uses the __cdecl calling convention (default)
  - *windll* or *oledll*: a library that uses the __stdcall calling convention
  - *net*: a Microsoft .NET library
  - *clr*: alias for *net* (Common Language Runtime)
  - *java*: a Java archive (`.jar` or `.class` files)
  - *com*: a COM library
  - *activex*: an ActiveX library

  Tip: Since the `.jar` or `.class` extension uniquely defines a Java library, *libtype* will automatically be set to *java* if *path* ends with `.jar` or `.class`.

- **kwargs** (*Any*) – All additional keyword arguments are passed to the object that loads the library. If *libtype* is
  - *cdll* → `CDLL`
  - *windll* → `WinDLL`
  - *oledll* → `OleDLL`
  - *net* or *clr* → all keyword arguments are ignored
  - *java* → `JavaGateway`
  - *com* → comtypes.CreateObject
  - *activex* → *`Application.load`*

Raises

- `OSError` – If the shared library cannot be loaded.
- `ValueError` – If the value of *libtype* is not supported.

property app: *`Application`* **|** `None`

Returns a reference to the ActiveX main application window.

When an ActiveX library is loaded, the window is not shown (to show it call *`show()`*) and the message loop is not running (to run it call *`run()`*).

Added in version 1.0.

cleanup()

Clean up references to the library.

Added in version 0.10.

Return type
None

property assembly: Any

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

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

---

property gateway

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

property lib: Any

Returns the reference to the library object.

For example, if *libtype* is

- *cdll* → CDLL
- *windll* → WinDLL
- *oledll* → OleDLL
- *net* or *clr* → *DotNet*
- *java* → JVMView
- *com* or *activex* → POINTER

property path: str

The path to the shared library file.

class msl.loadlib.load_library.**DotNet**(*items*, *path*)

Bases: object

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

Do not instantiate this class directly.

Parameters

- **items** (*dict*) – The items to use to update the internal __dict__ attribute.
- **path** (*str*) – The path to the .NET library file.

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.

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

Bases: 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. The module may import most of the standard python modules.

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 (see *LoadLibrary*)
- **libtype** (*LibType*) – The library type (see *LoadLibrary*).

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 (or hostname) to use for the server.
- **port** (*int*) – The port to open for the server.
- **kwargs** (*Any*) – All keyword arguments are passed to *LoadLibrary*.

property assembly

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

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

---

property lib

Returns the reference to the library object.

For example, if *libtype* is

- *cdll* → `CDLL`
- *windll* → `WinDLL`
- *oledll* → `OleDLL`
- *net* or *clr* → *DotNet*
- *com* or *activex* → `POINTER`

property path: str

The path to the shared library file.

static version()

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

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.

Return type

str

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.

Return type
None

static remove_site_packages_64bit()

Remove the site-packages directory from the 64-bit process.

By default, the site-packages directory of the 64-bit process is included in `sys.path` of the 32-bit process. Having the 64-bit site-packages directory available can sometimes cause issues. For example, comtypes imports numpy so if numpy is installed in the 64-bit process then comtypes will import the 64-bit version of numpy in the 32-bit process. Depending on the version of Python and/or numpy this can cause the 32-bit server to crash.

Added in version 0.9.

Example:

```
import sys
from msl.loadlib import Server32

class FileSystem(Server32):

    def __init__(self, host, port, **kwargs):

        # remove the site-packages directory that was passed from 64-
↪bit Python
        # before calling the super() function to load the COM library
        path = Server32.remove_site_packages_64bit()

        super().__init__('Scripting.FileSystemObject', 'com', host,␣
↪port)

        # optional: add the site-packages directory back into sys.
↪path
        sys.path.append(path)
```

Returns
The path of the site-packages directory that was removed. Can be an empty string if the directory was not found in `sys.path`.

Return type
str

static is_interpreter()

Check if code is running on the 32-bit server.

If the same module is executed by both *Client64* and *Server32* then there may be only parts of the code that should only be executed by the correct bitness of the Python interpreter.

Added in version 0.9.

Example:

```
import sys
from msl.loadlib import Server32

if Server32.is_interpreter():
    # this only gets executed on the 32-bit server
    assert sys.maxsize < 2**32
```

Returns
Whether the code is running on the 32-bit server.

Return type
bool

static examples_dir()

Returns the directory where the example libraries are located.

Added in version 0.9.

Return type
str

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 or closing file handles before it shuts down.

Added in version 0.6.

Return type
None

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.`**is_pythonnet_installed**()

Checks if Python for .NET is installed.

Returns
Whether Python for .NET is installed.

Return type
bool

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.`**is_py4j_installed**()

Checks if Py4J is installed.

Added in version 0.4.

Returns
Whether Py4J is installed.

Return type
bool

`msl.loadlib.utils.`**is_comtypes_installed**()

Checks if comtypes is installed.

Added in version 0.5.

Returns
Whether comtypes is installed.

Return type
bool

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

Check if the **useLegacyV2RuntimeActivationPolicy** property is enabled.

By default, Python for .NET 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 version="1.0" encoding="utf-8" ?>
<configuration>
    <startup useLegacyV2RuntimeActivationPolicy="true">
        <supportedRuntime version="v4.0" />
        <supportedRuntime version="v2.0.50727" />
```

(continues on next page)

(continued from previous page)

```
    </startup>
</configuration>
```

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

Returns

A status flag and a message describing the outcome.

The flag will be one of the following values:

- -1: if there was a problem
- 0: if the .NET property was already enabled, or
- 1: if the property was created successfully.

Return type
tuple[int, str]

msl.loadlib.utils.**is_port_in_use**(*port*)

Checks whether the TCP port is in use.

Changed in version 0.10.0: Only check TCP ports (instead of both TCP and UDP ports). Uses the `ss` command instead of `netstat` on Linux.

Changed in version 0.7.0: Renamed from *port_in_use* and added support for macOS.

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

Returns
Whether the TCP port is in use.

Return type
bool

msl.loadlib.utils.**get_available_port**()

Returns a port number that is available.

Return type
int

msl.loadlib.utils.**wait_for_server**(*host*, *port*, *timeout*)

Wait for the 32-bit server to start.

Parameters

- **host** (*str*) – The hostname or IP address of the server.
- **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.

Return type
None

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.

Added in version 0.5.

Parameters
additional_keys (*str*) – 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
The keys are the Class ID's and each value is a `dict` of the information that was requested.

Return type
dict[str, dict[str, str | None]]

Example:

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

msl.loadlib.utils.**generate_com_wrapper**(*lib*, *out_dir=None*)

Generate a Python wrapper module around a COM library.

For more information see Accessing type libraries.

Added in version 0.9.

Parameters

- **lib** (*Any*) – The COM library to create a wrapper of.

  Can be any of the following

  - a *LoadLibrary* object
  - the *ProgID* or *CLSID* of a registered COM library as a `str`
  - a COM pointer instance (`POINTER()`)
  - an ITypeLib COM pointer instance (`POINTER()`)
  - a path to a library file (.tlb, .exe or .dll) as a `str`
  - a `tuple` or `list` specifying the GUID of a library, a major and a minor version number, plus optionally an LCID number, e.g.,

    ```
    ['{EAB22AC0-30C1-11CF-A7EB-0000C05BAE0B}', 1, 1]
    ```

  - an object with `_reg_libid_` and `_reg_version_` attributes
- **out_dir** (*str | None*) – The output directory to save the wrapper to. If not specified, the module is saved to the `../site-packages/comtypes/gen` directory.

Returns
The wrapper module that was generated.

Return type
ModuleType

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*, ***kwargs*)

Bases: *Server32*

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 (or hostname) to use for the server.
- **port** (*int*) – The port to open for the server.
- **kwargs** (*str*) – Optional keyword arguments. The keys and values are of type str.

add(*a*, *b*)

Add two integers.

The corresponding C++ code is

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

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

Parameters

- **a** (*int*) – First integer.
- **b** (*int*) – Second integer.

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

Return type
int

subtract(*a*, *b*)

Subtract two floating-point numbers *('float' refers to the C++ data type)*.

The corresponding C++ code is

```
float subtract(float a, float b) {
    return a - b;
}
```

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

Parameters

- **a** (*float*) – First floating-point number.
- **b** (*float*) – Second floating-point number.

Returns
The difference between *a* and *b*.

Return type
float

add_or_subtract(*a*, *b*, *do_addition*)

Add or subtract two double-precision numbers *('double' refers to the C++ data type)*.

The corresponding C++ code is

```
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*) – First double-precision number.
- **b** (*float*) – Second double-precision number.
- **do_addition** (*bool*) – Whether to add or subtract the numbers.

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

Return type
float

scalar_multiply(*a*, *xin*)

Multiply each element in an array by a number.

The corresponding C++ code is

```
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*) – Scalar value.
- **xin** (*Sequence[float]*) – Array to modify.

Returns
A new array with each element in *xin* multiplied by *a*.

Return type
list[float]

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

```
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
The string reversed.

Return type
str

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

```
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
The string reversed.

Return type
str

distance_4_points(*four_points*)

Calculates the total distance connecting 4 *Point*'s.

The corresponding C++ code is

```
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
four_points (FourPoints) – The points to use to calculate the total distance.

Returns
The total distance connecting the 4 points.

Return type
float

circumference(*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.

```
double distance_n_points(NPoints p) {
    if (p.n < 2) {
```

(continues on next page)

(continued from previous page)

```
        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;
}
```

Parameters

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

Returns

The estimated circumference of the circle.

Return type

float

class msl.examples.loadlib.cpp32.**Point**

Bases: Structure

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

This object can be pickle'd.

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

x

Structure/Union member

y

Structure/Union member

class msl.examples.loadlib.cpp32.**FourPoints**(*point1*, *point2*, *point3*, *point4*)

Bases: Structure

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

This object can be pickle'd.

```
struct FourPoints {
    Point points[4];
};
```

Parameters

- **point1** (Point) – The first point.
- **point2** (Point) – The second point.

- **point3** (Point) – The third point.
- **point4** (Point) – The fourth point.

points

Structure/Union member

class msl.examples.loadlib.cpp32.**NPoints**

Bases: 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.

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

n

Structure/Union member

points

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.

class msl.examples.loadlib.cpp64.**Cpp64**

Bases: *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*) – First integer.
- **b** (*int*) – Second integer.

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

Return type
int

`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*) – First floating-point number.
- **b** (*float*) – Second floating-point number.

Returns
The difference between *a* and *b*.

Return type
float

`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*) – First double-precision number.
- **b** (*float*) – Second double-precision number.
- **do_addition** (*bool*) – Whether to add or subtract the numbers.

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

Return type
float

`scalar_multiply`(*a*, *xin*)

Multiply each element in an array by a number.

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

Parameters

- **a** (*float*) – Scalar value.
- **xin** (*Sequence[float]*) – Array to modify.

Returns
A new array with each element in *xin* multiplied by *a*.

Return type
list[float]

`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
The string reversed.

Return type
str

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
The string reversed.

Return type
str

distance_4_points(*points*)

Calculates the total distance connecting 4 *Point*'s.

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

Parameters
points (FourPoints) – The points to use to calculate the total distance. 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
The total distance connecting the 4 points.

Return type
float

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
The estimated circumference of the circle.

Return type
float

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.

class msl.examples.loadlib.dotnet32.**DotNet32**(*host*, *port*, ***kwargs*)

Bases: *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 (or hostname) to use for the server.
- **port** (*int*) – The port to open for the server.
- **kwargs** (*str*) – Optional keyword arguments. The keys and values are of type str.

get_class_names()

Returns the class names in the library.

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

Returns
The names of the classes that are available in *dotnet_lib32.dll*.

Return type
list[str]

add_integers(*a*, *b*)

Add two integers.

The corresponding C# code is

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

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

Parameters

- **a** (*int*) – First integer.
- **b** (*int*) – Second integer.

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

Return type
int

divide_floats(*a*, *b*)

Divide two C# floating-point numbers.

The corresponding C# code is

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

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

Parameters

- **a** (*float*) – First floating-point number.
- **b** (*float*) – Second floating-point number.

Returns
The quotient of *a* / *b*.

Return type
float

multiply_doubles(*a*, *b*)

Multiply two C# double-precision numbers.

The corresponding C# code is

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

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

Parameters

- **a** (*float*) – First double-precision number.
- **b** (*float*) – Second double-precision number.

Returns
The product of *a* * *b*.

Return type
float

add_or_subtract(*a*, *b*, *do_addition*)

Add or subtract two C# double-precision numbers.

The corresponding C# code is

```
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*) – First double-precision number.
- **b** (*float*) – Second double-precision number.
- **do_addition** (*bool*) – Whether to add or subtract the numbers.

Returns

a+b if *do_addition* is `True` else *a-b*.

Return type

float

scalar_multiply(*a*, *xin*)

Multiply each element in an array by a number.

The corresponding C# code is

```
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*) – Scalar value.
- **xin** (*Sequence[float]*) – Array to modify.

Returns

A new array with each element in *xin* multiplied by *a*.

Return type

list[float]

multiply_matrices(*a1*, *a2*)

Multiply two matrices.

The corresponding C# code is

```
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.

Parameters

- **a1** (*Sequence[Sequence[float]]*) – First matrix.
- **a2** (*Sequence[Sequence[float]]*) – Second matrix.

Returns

The result of *a1* * *a2*.

Return type

list[list[float]]

reverse_string(*original*)

Reverse a string.

The corresponding C# code is

```
public string reverse_string(string original)
{
    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
The string reversed.

Return type
str

add_multiple(*a*, *b*, *c*, *d*, *e*)

Add multiple integers. *Calls a static method in a static class.*

The corresponding C# code is

```
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*) – First integer.
- **b** (*int*) – Second integer.
- **c** (*int*) – Third integer.
- **d** (*int*) – Fourth integer.
- **e** (*int*) – Fifth integer.

Returns
The sum of the input arguments.

Return type
int

concatenate(*a*, *b*, *c*, *d*, *e*)

Concatenate strings. *Calls a static method in a static class.*

The corresponding C# code is

```
public static string concatenate(string a, string b, string c, bool␣
↪d, string e)
{
    string res = a + b + c;
```

(continues on next page)

(continued from previous page)

```
    if (d)
    {
        res += e;
    }
    return res;
}
```

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

Parameters

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

Returns

The strings concatenated together.

Return type

str

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: *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.

get_class_names()

Returns the class names in the library.

See the corresponding 32-bit *get_class_names()* method.

Returns

The names of the classes that are available in *dotnet_lib32.dll*.

Return type

list[str]

add_integers(*a*, *b*)

Add two integers.

See the corresponding 32-bit *add_integers()* method.

Parameters

- **a** (*int*) – First integer.
- **b** (*int*) – Second integer.

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

Return type
int

divide_floats(*a*, *b*)

Divide two C# floating-point numbers.

See the corresponding 32-bit *divide_floats()* method.

Parameters

- **a** (*float*) – First floating-point number.
- **b** (*float*) – Second floating-point number.

Returns
The quotient of *a* / *b*.

Return type
float

multiply_doubles(*a*, *b*)

Multiply two C# double-precision numbers.

See the corresponding 32-bit *multiply_doubles()* method.

Parameters

- **a** (*float*) – First double-precision number.
- **b** (*float*) – Second double-precision number.

Returns
The product of *a* * *b*.

Return type
float

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*) – First double-precision number.
- **b** (*float*) – Second double-precision number.
- **do_addition** (*bool*) – Whether to add or subtract the numbers.

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

Return type
float

`scalar_multiply`(*a*, *xin*)

Multiply each element in an array by a number.

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

Parameters

- **a** (*float*) – Scalar value.
- **xin** (*Sequence[float]*) – Array to modify.

Returns
A new array with each element in *xin* multiplied by *a*.

Return type
list[float]

`multiply_matrices`(*a1*, *a2*)

Multiply two matrices.

See the corresponding 32-bit *multiply_matrices()* method.

Parameters

- **a1** (*Sequence[Sequence[float]]*) – First matrix.
- **a2** (*Sequence[Sequence[float]]*) – Second matrix.

Returns
The result of *a1 * a2*.

Return type
list[list[float]]

`reverse_string`(*original*)

Reverse a string.

See the corresponding 32-bit *reverse_string()* method.

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

Returns
The string reversed.

Return type
str

`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*) – First integer.

- **b** (*int*) – Second integer.
- **c** (*int*) – Third integer.
- **d** (*int*) – Fourth integer.
- **e** (*int*) – Fifth integer.

Returns
The sum of the input arguments.

Return type
int

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*) – First string.
- **b** (*str*) – Second string.
- **c** (*str*) – Third string.
- **d** (*bool*) – Whether to include *e* in the concatenation.
- **e** (*str*) – Fourth string.

Returns
The strings concatenated together.

Return type
str

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*, ***kwargs*)

Bases: *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 (or hostname) to use for the server.
- **port** (*int*) – The port to open for the server.

- **kwargs** (*str*) – Optional keyword arguments. The keys and values are of type str.

static received_data(**args*, ***kwargs*)

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

Parameters

- **args** (*Any*) – The arguments.
- **kwargs** (*Any*) – The keyword arguments.

Returns

The *args* and *kwargs* that were received.

Return type

tuple[tuple[*Any*, ...], dict[*Any*, *Any*]]

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**

Bases: *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.

send_data(**args*, ***kwargs*)

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

Parameters

- **args** (*Any*) – The arguments that the *received_data()* method requires.
- **kwargs** (*Any*) – The keyword arguments that the *received_data()* method requires.

Returns

The *args* and *kwargs* that were returned from *received_data()*.

Return type

tuple[tuple[*Any*, ...], dict[*Any*, *Any*]]

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*, ***kwargs*)

Bases: *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 (or hostname) to use for the server.
- **port** (*int*) – The port to open for the server.
- **kwargs** (*str*) – Optional keyword arguments. The keys and values are of type str.

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

```
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*) – First 8-bit signed integer.
- **b** (*int*) – Second 8-bit signed integer.

Returns

The sum of *a* and *b*.

Return type

int

sum_16bit(*a*, *b*)

Add two 16-bit signed integers

Python only has one int data type to represent integer values. The *sum_16bit()* method converts the data types of *a* and *b* to be ctypes.c_int16.

The corresponding FORTRAN code is

```
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
```

See the corresponding 64-bit *sum_16bit()* method.

Parameters

- **a** (*int*) – First 16-bit signed integer.
- **b** (*int*) – Second 16-bit signed integer.

Returns

The sum of *a* and *b*.

Return type

int

sum_32bit(*a*, *b*)

Add two 32-bit signed integers.

Python only has one `int` data type to represent integer values. The *sum_32bit()* method converts the data types of *a* and *b* to be `ctypes.c_int32`.

The corresponding FORTRAN code is

```
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
```

See the corresponding 64-bit *sum_32bit()* method.

Parameters

- **a** (*int*) – First 32-bit signed integer.
- **b** (*int*) – Second 32-bit signed integer.

Returns

The sum of *a* and *b*.

Return type

int

sum_64bit(*a*, *b*)

Add two 64-bit signed integers.

Python only has one `int` data type to represent integer values. The *sum_64bit()* method converts the data types of *a* and *b* to be `ctypes.c_int64`.

The corresponding FORTRAN code is

```
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
```

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

Parameters

- **a** (*int*) – First 64-bit signed integer.
- **b** (*int*) – Second 64-bit signed integer.

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

Return type
int

multiply_float32(*a*, *b*)

Multiply two FORTRAN floating-point numbers.

The corresponding FORTRAN code is

```
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
```

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

Parameters

- **a** (*float*) – First floating-point number.
- **b** (*float*) – Second floating-point number.

Returns
The product of *a* and *b*.

Return type
float

multiply_float64(*a*, *b*)

Multiply two FORTRAN double-precision numbers.

The corresponding FORTRAN code is

```
function multiply_float64(a, b) result(value)
    !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'multiply_float64' :: multiply_
↪float64
    implicit none
```

(continues on next page)

(continued from previous page)

```
    real(8) :: a, b, value
    value = a * b
end function multiply_float64
```

See the corresponding 64-bit *multiply_float64()* method.

Parameters

- **a** (*float*) – First double-precision number.
- **b** (*float*) – Second double-precision number.

Returns

The product of *a* and *b*.

Return type

float

is_positive(*a*)

Returns whether the value of the input argument is > 0.

The corresponding FORTRAN code is

```
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*) – Double-precision number.

Returns

Whether the value of *a* is > 0.

Return type

bool

add_or_subtract(*a*, *b*, *do_addition*)

Add or subtract two integers.

The corresponding FORTRAN code is

```
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
```

(continues on next page)

(continued from previous page)

```
    else
        value = a - b
    endif
end function add_or_subtract
```

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

Parameters

- **a** (*int*) – First integer.
- **b** (*int*) – Second integer.
- **do_addition** (*bool*) – Whether to add or subtract the numbers.

Returns

a+b if *do_addition* is `True` else *a-b*.

Return type

int

factorial(*n*)

Compute the n'th factorial.

The corresponding FORTRAN code is

```
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

The factorial of *n*.

Return type

float

standard_deviation(*data*)

Compute the standard deviation.

The corresponding FORTRAN code is

```
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 (*Sequence[float]*) – The values to compute the standard deviation of.

Returns
The standard deviation of *data*.

Return type
float

besselJ0(*x*)

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

The corresponding FORTRAN code is

```
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
The value of BESSEL_J0(x).

Return type
float

reverse_string(*original*)

Reverse a string.

The corresponding FORTRAN code is

```
subroutine reverse_string(original, n, reversed)
    !DEC$ ATTRIBUTES DLLEXPORT, ALIAS:'reverse_string' :: reverse_
↪string
    !DEC$ ATTRIBUTES REFERENCE :: original, reversed
```

(continues on next page)

(continued from previous page)

```
    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
The string reversed.

Return type
str

add_1d_arrays(*a1*, *a2*)

Perform an element-wise addition of two 1D double-precision arrays.

The corresponding FORTRAN code is

```
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
```

See the corresponding 64-bit *add_1d_arrays()* method.

Parameters

- **a1** (*Sequence[float]*) – First array.
- **a2** (*Sequence[float]*) – Second array.

Returns
The element-wise addition of *a1* + *a2*.

Return type
list[float]

matrix_multiply(*a1*, *a2*)

Multiply two matrices.

The corresponding FORTRAN code is

```
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
```

Note: FORTRAN stores multidimensional 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 returned.

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

Parameters

- **a1** (*Sequence[Sequence[float]]*) – First matrix.
- **a2** (*Sequence[Sequence[float]]*) – Second matrix.

Returns

The product of *a1 * a2*.

Return type

list[list[float]]

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: *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.

sum_8bit(*a*, *b*)

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

See the corresponding 32-bit *sum_8bit()* method.

Parameters

- **a** (*int*) – First 8-bit signed integer.
- **b** (*int*) – Second 8-bit signed integer.

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

Return type
int

sum_16bit(*a*, *b*)

Send a request to add two 16-bit signed integers.

See the corresponding 32-bit *sum_16bit()* method.

Parameters

- **a** (*int*) – First 16-bit signed integer.
- **b** (*int*) – Second 16-bit signed integer.

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

Return type
int

sum_32bit(*a*, *b*)

Send a request to add two 32-bit signed integers.

See the corresponding 32-bit *sum_32bit()* method.

Parameters

- **a** (*int*) – First 32-bit signed integer.
- **b** (*int*) – Second 32-bit signed integer.

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

Return type
int

sum_64bit(*a*, *b*)

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

See the corresponding 32-bit *sum_64bit()* method.

Parameters

- **a** (*int*) – First 64-bit signed integer.
- **b** (*int*) – Second 64-bit signed integer.

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

Return type
int

multiply_float32(*a*, *b*)

Send a request to multiply two FORTRAN floating-point numbers.

See the corresponding 32-bit *multiply_float32()* method.

Parameters

- **a** (*float*) – First floating-point number.
- **b** (*float*) – Second floating-point number.

Returns

The product of *a* and *b*.

Return type

float

multiply_float64(*a*, *b*)

Send a request to multiply two FORTRAN double-precision numbers.

See the corresponding 32-bit *multiply_float64()* method.

Parameters

- **a** (*float*) – First double-precision number.
- **b** (*float*) – Second double-precision number.

Returns

The product of *a* and *b*.

Return type

float

is_positive(*a*)

Returns whether the value of the input argument is > 0.

See the corresponding 32-bit *is_positive()* method.

Parameters

a (*float*) – Double-precision number.

Returns

Whether the value of *a* is > 0.

Return type

bool

add_or_subtract(*a*, *b*, *do_addition*)

Add or subtract two integers.

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

Parameters

- **a** (*int*) – First integer.
- **b** (*int*) – Second integer.
- **do_addition** (*bool*) – Whether to add or subtract the numbers.

Returns

a+*b* if *do_addition* is `True` else *a*-*b*.

Return type
int

`factorial`(*n*)

Compute the n'th factorial.

See the corresponding 32-bit *factorial()* method.

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

Returns
The factorial of *n*.

Return type
float

`standard_deviation`(*data*)

Compute the standard deviation.

See the corresponding 32-bit *standard_deviation()* method.

Parameters
data (*Sequence[float]*) – The values to compute the standard deviation of.

Returns
The standard deviation of *data*.

Return type
float

`besselJ0`(*x*)

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

See the corresponding 32-bit *besselJ0()* method.

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

Returns
The value of `BESSEL_J0(x)`.

Return type
float

`reverse_string`(*original*)

Reverse a string.

See the corresponding 32-bit *reverse_string()* method.

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

Returns
The string reversed.

Return type
str

add_1d_arrays(*a1*, *a2*)

Perform an element-wise addition of two 1D double-precision arrays.

See the corresponding 32-bit *add_1d_arrays()* method.

Parameters

- **a1** (*Sequence[float]*) – First array.
- **a2** (*Sequence[float]*) – Second array.

Returns
The element-wise addition of *a1* + *a2*.

Return type
list[float]

matrix_multiply(*a1*, *a2*)

Multiply two matrices.

See the corresponding 32-bit *matrix_multiply()* method.

Parameters

- **a1** (*Sequence[Sequence[float]]*) – First matrix.
- **a2** (*Sequence[Sequence[float]]*) – Second matrix.

Returns
The product of *a1* * *a2*.

Return type
list[list[float]]

msl.examples.loadlib.kernel32 module

A wrapper around the 32-bit Windows kernel32.dll library.

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

Kernel32 is the 32-bit server and *Kernel64* is the 64-bit client for inter-process communication.

Note: The kernel32.dll library is a standard Windows library and therefore this example is only valid on a Windows computer.

---

class msl.examples.loadlib.kernel32.**Kernel32**(*host*, *port*, ***kwargs*)

Bases: *Server32*

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

Parameters

- **host** (*str*) – The IP address (or hostname) to use for the server.
- **port** (*int*) – The port to open for the server.

- **kwargs** (*str*) – Optional keyword arguments. The keys and values are of type str.

get_time()

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

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

Returns
The current date and time.

Return type
datetime

class msl.examples.loadlib.kernel32.**SystemTime**

Bases: Structure

Example of creating a ctypes.Structure.

See SYSTEMTIME for a description of the struct.

WORD

alias of 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.

---

class msl.examples.loadlib.kernel64.**Kernel64**

Bases: *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
The current date and time.

Return type
datetime

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.

class msl.examples.loadlib.labview32.**Labview32**(*host*, *port*, ***kwargs*)

Bases: *Server32*

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

Parameters

- **host** (*str*) – The IP address (or hostname) to use for the server.

- **port** (*int*) – The port to open for the server.
- **kwargs** (*str*) – Optional keyword arguments. The keys and values are of type str.

stdev(*x*, *weighting=0*)

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

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

Parameters

- **x** (*Sequence[float]*) – The data to calculate the mean, variance and standard deviation of.
- **weighting** (*int*) – Whether to calculate the sample (*weighting* = 0) or the population (*weighting* = 1) standard deviation and variance.

Returns

The mean, variance and standard deviation.

Return type

tuple[float, float, float]

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.

class msl.examples.loadlib.labview64.**Labview64**

Bases: *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.

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

- **weighting** (*int*) – Whether to calculate the sample (*weighting* = 0) or the population (*weighting* = 1) standard deviation and variance.

Returns
The mean, variance and standard deviation.

Return type
tuple[float, float, float]

1.5 Create a custom 32-bit server

If you want to create a custom 32-bit server, you will need

- a 32-bit version of Python (version 3.8 or later) installed
- PyInstaller installed in the 32-bit Python environment (ideally, you would use a **virtual environment** to install the necessary packages to create the server)

Some reasons why you may want to create a custom 32-bit server are that you want to

- run the server on a different version of Python,
- install a different version of comtypes or pythonnet (on Windows),
- install additional packages on the server (e.g., numpy, my_custom_package),
- embed your own data files in the frozen server

Using pip from a 32-bit Python interpreter, run

```
pip install msl-loadlib pyinstaller
```

You may want to install additional packages as well.

You have two options to create the 32-bit server

1) *Using the API*

2) *Using the CLI*

and you have two options to use the newly-created server

1. Copy the `server32-*` file to the `../site-packages/msl/loadlib` directory where you have MSL-LoadLib installed in your 64-bit version of Python to replace the existing server file.

2. Specify the directory where the `server32-*` file is located as the value of the `server32_dir` keyword argument in *Client64*.

1.5.1 Using the API

Create a script that calls the *freeze_server32.main()* function with the appropriate keyword arguments, for example,

```
from msl.loadlib import freeze_server32
freeze_server32.main(imports='numpy')
```

and run your script using a 32-bit Python interpreter.

1.5.2 Using the CLI

When MSL-LoadLib is installed, a console script is included (named *freeze32*) that may be executed from the command line to create a new frozen 32-bit server.

To see the help for *freeze32*, run

```
freeze32 --help
```

For example, if you want to include your own package and data files, you would run

```
freeze32 --imports my_package --data .\mydata\lib32.dll
```

1.6 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.

1.6.1 C++

Source code for the example C++ library.

cpp_lib.h

```
// cpp_lib.h
// Contains the declaration of exported functions.
//

#if defined(_MSC_VER)
    // Microsoft
    #define EXPORT __declspec(dllexport)
#elif defined(__GNUC__)
    // G++
    #define EXPORT __attribute__((visibility("default")))
#else
#   error "Unknown EXPORT semantics"
#endif

struct Point {
    double x;
    double y;
};

struct FourPoints {
    Point points[4];
};

struct NPoints {
```

(continues on next page)

(continued from previous page)

```
    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
    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);

}
```

cpp_lib.cpp

```
// cpp_lib.cpp
// Examples that show how to pass various data types between Python and a C++␣
↪library.
//
// Compiled using:
//   g++ cpp_lib.cpp -fPIC -shared -Bstatic -Wall -o cpp_lib64.so
//
#include <math.h>
#include "cpp_lib.h"

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

(continues on next page)

(continued from previous page)

```
float subtract(float a, float b) {
    return a - b;
}

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

void scalar_multiply(double a, double* xin, int n, double* xout) {
    for (int i = 0; i < n; i++) {
        xout[i] = a * xin[i];
    }
}

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) {
```

(continues on next page)

(continued from previous page)

```
        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;
}
```

1.6.2 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 -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
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
        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
```

(continues on next page)

(continued from previous page)

```
    !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
```

1.6.3 Microsoft .NET

Source code for the example C# library.

dotnet_lib.cs

```
// 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)
        {
```

(continues on next page)

(continued from previous page)

```
            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)
    {
```

(continues on next page)

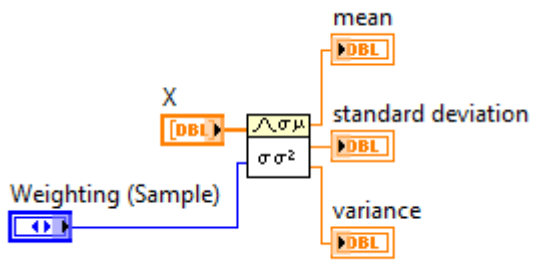
(continued from previous page)

```
        char[] charArray = original.ToCharArray();
        Array.Reverse(charArray);
        return new string(charArray);
    }

}

// A static class
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;

    }

}
```

1.6.4 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

/*!
 * 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);

#ifdef __cplusplus
} // extern "C"
#endif

#pragma pack(pop)
```

1.6.5 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. */
```

(continues on next page)

(continued from previous page)

```
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) {
```

(continues on next page)

(continued from previous page)

```
    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

```
package nz.msl.examples;

public class MathUtils {

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

  /** 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;

```

(continues on next page)

(continued from previous page)

```
/** 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;

/*
 *
 * Define the constructors.
 *
 *
 */

/** 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 {@code rows} x {@code cols} Matrix filled with zeros. */
public Matrix(int rows, int cols) {
  m = new double[rows][cols];
}

/** Create a {@code rows} x {@code 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 {@code rows} x {@code cols} Matrix that is filled with
 * uniformly-distributed random values that are within the range
 * {@code min} to {@code max}.
 */
```

(continues on next page)

(continued from previous page)

```
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 {@code 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()));
  } 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;
            }
```

(continues on next page)

(continued from previous page)

```
        }
      return c;
    }
  }

  /**
   * Solves {@code b = Ax} for {@code 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();

      // 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);
```

(continues on next page)

(continued from previous page)

```
      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();
}

/*
 *
 * The public methods.
 *
 *
 */

/** 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]));
    }
```

(continues on next page)

(continued from previous page)

```
    sb.append("\n");
  }
  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 {@code row} and {@code col}. */
public double getValue(int row, int col) {
  return m[row][col];
}

/** Sets the value at {@code row} and {@code col} to be {@code 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++)
```

(continues on next page)

(continued from previous page)

```
      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;
  }
}

/** 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",
```

(continues on next page)

(continued from previous page)

```
        m.length, m[0].length));
    }
  }


  /*
   *
   * Private methods.
   *
   *
   */

  /**
   * Create the Lower, L, and Upper, U, triangular matrices, such that M=LU.
   * Does not use pivoting.
   */
  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 https://people.inf.ethz.ch/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);
```

(continues on next page)

(continued from previous page)

```
      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;
    } 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();
```

(continues on next page)

(continued from previous page)

```
    }

    // 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);
  }

}
```

1.7 FAQ

Answers to frequently asked questions.

1.7.1 Access the console output of the 32-bit server

You have access to the console output of the 32-bit server once it has shut down. For example, suppose your 64-bit client class is called *MyClient*, you would do something like:

```
c = MyClient()
c.do_something()
c.do_something_else()
stdout, stderr = c.shutdown_server32()
print(stdout.read())
print(stderr.read())
```

If you want to be able to poll the console output in real time (while the server is running) you have two options:

1. Have the 32-bit server write to a file and the 64-bit client read from the file.
2. Implement a *polling* method on the 32-bit server for the 64-bit client to send requests to. The following is a runnable example:

```
import os

from msl.loadlib import Client64
from msl.loadlib import Server32

class Polling32(Server32):

    def __init__(self, host, port):
        # Loading a "dummy" 32-bit library for this example
        path = os.path.join(Server32.examples_dir(), 'cpp_lib32')
        super().__init__(path, 'cdll', host, port)

        # Create a list to store 'print' messages in
        self._stdout = []

        # Use your own print method instead of calling the builtin print()␣
↪function
        self.print('Polling32 has been initiated')

    def say_hello(self, name):
        """Return a greeting."""
        self.print(f'say_hello was called with argument {name!r}')
        return f'Hello, {name}!'

    def poll(self):
        """Get the latest message."""
        try:
            return self._stdout[-1]
        except IndexError:
            return ''

    def flush(self):
        """Get all messages and clear the cache."""
        messages = '\n'.join(self._stdout)
        self._stdout.clear()
        return messages

    def print(self, message):
        """Append a message that you would have wanted to print to the␣
↪console."""
        self._stdout.append(message)

class Polling64(Client64):

    def __init__(self):
        super().__init__(__file__)

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

(continues on next page)

(continued from previous page)

```
        return send

# Only execute this section of code on the 64-bit client (not on the 32-bit␣
↪server).
# Typically, you may write the Server32 class and the Client64 class in␣
↪separate files.
if not Server32.is_interpreter():
    p = Polling64()
    print('poll ->', p.poll())
    print('say_hello ->', p.say_hello('world'))
    print('poll ->', p.poll())
    print('flush ->', repr(p.flush()))
    print('poll ->', repr(p.poll()))
    p.shutdown_server32()
```

Running the above script will output:

```
poll -> Polling32 has been initiated
say_hello -> Hello, world!
poll -> say_hello was called with argument 'world'
flush -> "Polling32 has been initiated\nsay_hello was called with argument
↪'world'"
poll -> ''
```

1.7.2 Freezing the MSL-LoadLib package

If you want to use PyInstaller or cx-Freeze to bundle MSL-LoadLib in a frozen application, the 32-bit server must be added as a data file.

For example, using PyInstaller on Windows you would include an `--add-data` option

```
pyinstaller --add-data "..\site-packages\msl\loadlib\server32-windows.exe:."
```

where you must replace the leading `..` prefix with the parent directories to the file (i.e., specify the absolute path to the file). On Linux, replace `server32-windows.exe:.` with `server32-linux:.`

If the server is loading a .NET library that was compiled with .NET < 4.0, you must also add the `server32-windows.exe.config` data file. Otherwise, you do not need to add this config file.

cx-Freeze appears to automatically bundle the 32-bit server (tested with cx-Freeze version 6.14.5) so there may not be anything you need to do. If the *server32* executable is not bundled, you can specify the absolute path to the *server32* executable as the `include_files` option for the `build_exe` command.

You may also wish to *Create a custom 32-bit server* and add your custom server to your application.

1.7.3 Mocking the connection to the server

You may mock the connection to the server by passing in `host=None` when you instantiate *Client64*. Also, the *Server32* may need to decide which library to load.

When the connection is mocked, both *Client64* and *Server32* instances will run in the same Python interpreter, therefore the server must load a library that is the same bitness as the Python interpreter that the client is running in. The `pickle` module is not used when the connection is mocked, so there is no overhead of using a file as a middle step to process requests and responses (which has a side effect that a mocked connection can return objects in a server's response that are not pickleable).

One reason that you may want to mock the connection is that you wrote a lot of code that had to load a 32-bit library but now a 64-bit version of the library is available. You may also need to support the 32-bit and 64-bit libraries at the same time. Instead of making a relatively large change to your code, or managing different code bases, you can simply specify a keyword argument when instantiating your client class to decide whether to use the 32-bit library or the 64-bit library and the client class behaves exactly the same.

Here is an example on how a client (running within 64-bit Python) can have a server load a 32-bit library or a 64-bit library.

```
import sys

from msl.loadlib import Client64
from msl.loadlib import Server32

class MockableServer(Server32):

    def __init__(self, host, port, **kwargs):
        # Decide which library to load on the server.
        # You could check if the server is running in a 32-bit
        # or a 64-bit version of Python (like is shown here) or
        # check "if host is None:", which is `True` when the
        # connection is mocked.
        if sys.maxsize > 2 ** 32:
            path = 'path/to/64bit/c/library.so
        else:
            path = 'path/to/32bit/c/library.so
        super().__init__(path, 'cdll', host, port)

class MockableClient(Client64):

    def __init__(self, **kwargs):
        super().__init__(__file__, **kwargs)

if __name__ == '__main__':
    client_uses_32bit_library = MockableClient()
    client_uses_64bit_library = MockableClient(host=None)
```

1.8 License

1.9 Developers

- Joseph Borbely <joseph.borbely@measurement.govt.nz>

1.10 Release Notes

1.10.1 Version 1.0.0 (in development)

- Added
  - *Client64* now accepts `None` as the value of *host*, which will mock the connection to the server
  - support for Python 3.12
  - type annotations (**PEP 484** and **PEP 561** using inline types)
  - `freeze32` console script to create a new server
- Changed
  - convert to a **PEP 420** implicit namespace package
  - the *requires_pythonnet* and *requires_comtypes* arguments to *freeze_server32.main()* were removed and the *imports*, *data* and *skip_32bit_check* arguments were added

  - constants (e.g., *IS_WINDOWS*) were moved to a *constants.py* file, these constants are meant for internal use only

- Removed
  - support for Python 2.7, 3.5, 3.6 and 3.7
  - the deprecated *quiet* parameter

1.10.2 Version 0.10.0 (2023-06-16)

This release will be the last to support Python 2.7, 3.5, 3.6 and 3.7

- The 32-bit server is frozen with the following versions
  - `server32-windows.exe` – Python 3.11.4, pythonnet 3.0.1, comtypes 1.2.0
  - `server32-linux` – Python 3.11.4 (built with GLIBC 2.27)
- Added
  - can now specify the destination directory when freezing the 32-bit server
  - the *server32_dir* keyword argument to *Client64* (fixes issue #35)
  - Support for Python 3.10 and 3.11
  - *LoadLibrary* and *Client64* can be used as a context manager (The with statement)
  - *LoadLibrary.cleanup()* method
  - `~/.local/share/py4j` to the search paths when looking for the `py4j<version>.jar` file
- Changed
  - *is_port_in_use()* only checks TCP ports and it uses the `ss` command instead of `netstat` on linux
  - the example libraries for FORTRAN now depend on *libgfortran5* on linux
- Fixed
  - issue #31 - suppress console popups when using `pythonw.exe`
  - issue #24 - starting the 32-bit server could block forever by not honouring the timeout

1.10.3 Version 0.9.0 (2021-05-13)

- The 32-bit server is frozen with the following versions
  - `server32-windows.exe` – Python 3.7.10, pythonnet 2.5.2, comtypes 1.1.10
  - `server32-linux` – Python 3.7.10, pythonnet 2.4.0
- Added
  - support for loading an ActiveX library
  - the following static methods to *Server32*: *remove_site_packages_64bit()*, *is_interpreter()*, *examples_dir()*
  - the *generate_com_wrapper()* function

- Changed
  - the `sys.coinit_flags` attribute is now set to `COINIT_MULTITHREADED` (only if this attribute was not already defined prior to importing *msl.loadlib*)
- Fixed
  - `Client64.__del__` could have written a warning to stderr indicating that no `self._conn` attribute existed
  - `sys:1: ResourceWarning: unclosed file <_io.BufferedReader name=...>` warnings could be written to stderr when a *Client64* object is destroyed
  - issue #23 - the `useLegacyV2RuntimeActivationPolicy` property was no longer created

1.10.4 Version 0.8.0 (2021-02-20)

- Added
  - support for Python 3.9
  - the *protocol* keyword argument to *Client64*
  - the ability to request non-callable attributes from the 32-bit server class (e.g., methods that use the *@property* decorator and class/instance variables)
- Changed
  - `server32-windows.exe` uses Python 3.7.10, pythonnet 2.5.2 and comtypes 1.1.8
  - `server32-linux` uses Python 3.7.10 and pythonnet 2.4.0 (there are problems with later versions of pythonnet on 32-bit linux, see issue #1210 for more details)
  - call `clr.AddReference` before `clr.System.Reflection.Assembly.LoadFile` when loading a .NET library
  - use PIPE's for *stdout* and *stderr* for the 32-bit server subprocess and for the py4j *GatewayServer*
  - *shutdown_server32()* now returns the *(stdout, stderr)* streams from the 32-bit server subprocess
  - the *quiet* keyword argument for *Client64* is deprecated
- Fixed
  - issue #21 - an `UnsupportedOperation: fileno` exception was raised when running within the Spyder IDE
- Removed
  - `cygwin` from the *IS_WINDOWS* check

1.10.5 Version 0.7.0 (2020-03-17)

- Added
  - support for Python 3.8
  - compiled the C++ and FORTRAN examples for 64-bit macOS
- Changed
  - the frozen server32 executable uses Python 3.7.7 (Windows and Linux), pythonnet 2.4.0 (Windows and Linux) and comtypes 1.1.7 (Windows)
  - use `__package__` as the logger name
  - renamed `port_in_use()` to *is_port_in_use()* and added support for checking the status of a port in macOS
  - changes to how a .NET library is loaded: include the System namespace by default, do not automatically create a class instance
- Removed
  - Support for Python 3.4

1.10.6 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 COM *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

1.10.7 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 *lib* will be *None*
  - the default timeout value when waiting for the 32-bit server to start is now 10 seconds
  - the *Client64* class now raises *Server32Error* if the 32-bit server raises an exception
  - the *Client64* class now inherits from *object* and the reference to *HTTPConnection* is now a property value
  - the *__repr__* methods no longer include the id as a hex number
- Fixed
  - set `sys.stdout = io.StringIO()` if *quiet=True* on the server

1.10.8 Version 0.4.1 (2018-08-24)

- Added
  - the `version_info` namedtuple now includes a *releaselevel*
  - Support for Python 3.7
- Fixed
  - Issue #11
  - `utils.wait_for_server()` raised *NameError: name 'TimeoutError' is not defined* for Python 2.7
  - `utils.port_in_use()` raised *UnicodeDecodeError* (PR #9)
  - `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
  - pythonnet is now an optional dependency on Windows and py4j is now optional for all OS

  - rename *Dummy* example to *Echo*

- Removed
  - Support for Python 3.3

1.10.9 Version 0.4.0 (2018-02-28)

- Added
  - Py4J wrapper for loading `.jar` and `.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 `socket.socket.bind` to select an available port instead of checking of calling `utils.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

1.10.10 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

1.10.11 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

1.10.12 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)

1.10.13 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

1.10.14 Version 0.2.2 (2017-03-03)

- refreeze server32 executables

1.10.15 Version 0.2.1 (2017-03-02)

- fix releaselevel bug

1.10.16 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

1.10.17 Version 0.1.0 (2017-02-15)

- Initial release

PYTHON MODULE INDEX

m

INDEX

C

D

N

O

P

R

S

T

V

W

X

Y