MATLAB® Compiler SDK™
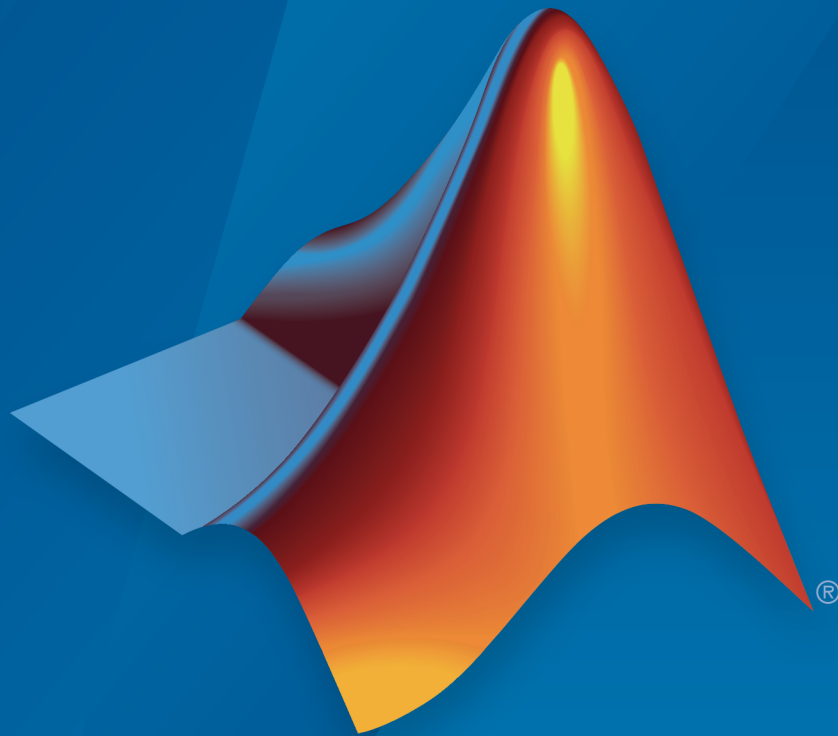
Python® User's Guide

MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

Technical support: www.mathworks.com/support/contact_us

Phone: 508-647-7000

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

MATLAB® Compiler SDK™ Python® User's Guide

© COPYRIGHT 2012–2016 by The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2015	Online only	New for Version 6.1 (Release 2015b)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)

Contents

3

Data Handling

Functions — Alphabetical List

4

Python Package Installation

System Requirements for Integration with MATLAB Compiler SDK Python Packages

MATLAB® Compiler SDK™ Python® package integration supports:

- Python 2.7
- Python 3.3
- Python 3.4

For more information on downloading and installing Python, see https://www.python.org/.

Note: The installed version of Python must match the bitness of the machine on which the application runs. For example, if running on a 64–bit Windows machine, you must install the 64–bit version of Python.

Install a MATLAB Compiler SDK Python Package

In this section...
"With the Generated Installer" on page 1-3
"Without the Generated Installer" on page 1-4

With the Generated Installer

The Library Compiler app generates an installer that installs the MATLAB Runtime and the files required to install the generated Python package. The app places the installer in the `for_redistribution` folder.

1 Copy the installer from the `for_redistribution` folder to the desired location.
2 Run the installer.
3 Note where the installer writes the Python package files.
4 When the installer finishes, open a command terminal in the folder containing the Python package files.
5 Run the Python set up script.

```
python setup.py install
```
6 Set the required environment variables.

On Linux®:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:mcrroot/runtime/glnxa64:
      mcrroot/bin/glnxa64:mcrroot/sys/os/glnxa64:
      mcrroot/sys/opengl/lib/glnxa64
setenv XAPPLRESDIR mcrroot/v90/X11/app-defaults
```

On OS X:

```
setenv DYLD_LIBRARY_PATH ${DYLD_LIBRARY_PATH}:mcrroot/runtime/maci64:
      mcrroot/sys/os/maci64:mcrroot/bin/maci64
```

Note: If `LD_LIBRARY_PATH` is not defined on Linux, remove `${LD_LIBRARY_PATH}:` from the code to set the environment variables. Similarly, on OS X, remove `${DYLD_LIBRARY_PATH}:` if `DYLD_LIBRARY_PATH` is not defined.

Note: *mcrroot* is the full path to the MATLAB Runtime installation.

Note: Commands must be entered as a single line.

Without the Generated Installer

If you already have the MATLAB Runtime installed, you can just install the Python package.

1 Copy the contents of the `for_redistribution_files_only` folder to the desired location.
2 Open a command terminal in the folder containing the Python package files.
3 Run the Python set up script.

   ```
   python setup.py install
   ```
4 Set the required environment variables.

   On Linux:

   ```
   setenv LD_LIBRARY_PATH $LD_LIBRARY_PATH:mcrroot/runtime/glnxa64:
         mcrroot/bin/glnxa64:mcrroot/sys/os/glnxa64:
         mcrroot/sys/opengl/lib/glnxa64
   setenv XAPPLRESDIR mcrroot/v90/X11/app-defaults
   ```

   On OS X:

   ```
   setenv DYLD_LIBRARY_PATH $DYLD_LIBRARY_PATH:mcrroot/runtime/maci64:
         mcrroot/sys/os/maci64:mcrroot/bin/maci64
   ```

Note: *mcrroot* is the full path to the MATLAB Runtime installation.

Note: Commands must be entered as a single line.

Related Examples

- "Import Compiled Python Packages" on page 2-3
- "Initialize the MATLAB Runtime" on page 2-4

2

Python Integration

Integrate a Python Package

To integrate a MATLAB Compiler SDK Python Package:

1 Install the compiled Python Package.

See "Install a MATLAB Compiler SDK Python Package" on page 1-3
2 In consultation with the MATLAB programmer, agree on the MATLAB function signatures that comprise the services in the application.
3 Import the compiled Python package.

See "Import Compiled Python Packages" on page 2-3.
4 Write the Python code to initialize the MATLAB Runtime, and load the MATLAB code.

See "Initialize the MATLAB Runtime" on page 2-4.
5 Create the required MATLAB data for function inputs and outputs.

See "MATLAB Arrays as Python Variables" on page 3-2.
6 Evaluate the MATLAB functions.

See "Evaluate a Compiled MATLAB Function" on page 2-6 or "Evaluate a Compiled MATLAB Function Asynchronously" on page 2-9.
7 Terminate the MATLAB Runtime using the `terminate()` function of the generated package.

Import Compiled Python Packages

The MATLAB Compiler SDK Python target generates the Python code into a package that must be imported to the Python runtime before you can use the compiled MATLAB functions. You specify the package name and the namespace when compiling the MATLAB functions.

- If you use the Library Compiler app, you specify the package name with the **Library Name** field and the namespace with the **Namespace** field.

  The **Library Name** field defaults to the name of the first MATLAB file listed in the app. You can leave the **Namespace** field empty.

- If you use the `mcc` function, you specify the package name and namespace as part of the `-W python:`*namespace.packageName* flag.

  Specifying the namespace is optional.

For example, if you compile your MATLAB functions and specify the package named `addmatrix`, you import it as follows:

```
import addmatrix
```

Related Examples

- "Create a Python Application with MATLAB Code"
- "Compile Python Packages with Library Compiler App"
- "Compile Python Packages from Command Line"

Initialize the MATLAB Runtime

When integrating compiled MATLAB functions into a Python application, your code must initialize the MATLAB Runtime:

1   Optionally, provide a list of startup options to the MATLAB Runtime using the `initialize_runtime()` function.
2   Start the MATLAB Runtime, and load a MATLAB Runtime instance using the `initialize()` function of each compiled package used in the application.

Provide MATLAB Runtime Startup Options

Note: On Mac OS X, you must pass the MATLAB Runtime options to the `mwpython` command when starting Python. Use `-mlstartup` followed by a comma-separated list of MATLAB Runtime options. MATLAB Runtime options passed to `initialize_runtime()` are ignored.

The MATLAB Runtime has two startup options that you can specify:

*   `-nojvm` — disable the Java® Virtual Machine, which is enabled by default. This can help improve the MATLAB Runtime performance.
*   `-nodisplay` — on Linux, run the MATLAB Runtime without display functionality.

You specify these options before you initialize the compiled MATLAB functions. You do so by calling the `initialize_runtime()` method of a generated Python package with the MATLAB Runtime options. The list of MATLAB Runtime options is passed as a list of strings. For example, to start the MATLAB Runtime for the package `addmatrix` with no display and no Java Virtual Machine:

```
import addmatrix

addmatrix.initialize_runtime(['-nojvm', '-nodisplay'])
```

If your application uses multiple Python packages, you call `initialize_runtime()` from only one package. The first call sets the run-time options for the MATLAB Runtime session. Any subsequent calls are ignored.

Start MATLAB Runtime with Compiled MATLAB Functions

To evaluate a compiled MATLAB function, load it into the MATLAB Runtime. Do this by calling the `initialize()` method of the generated Python package. The `initialize()` method returns an object that can be used to evaluate the compiled MATLAB functions in the package. For example, to start the MATLAB Runtime and load the MATLAB functions in the `addmatrix` package, use:

```
import addmatrix

myAdder = addmatrix.initialize()
```

Note: If the `initialize_runtime()` function is not called prior to the `initialize()` function, the MATLAB Runtime is started with no startup options.

More About

- "Integrate Python Packages on Mac OS X" on page 2-11

Evaluate a Compiled MATLAB Function

In this section...

Evaluate a compiled MATLAB function using the Python object returned from the `initialize()` function.

```
result1,...resultN = my_client.function_name(in_args, nargout=nargs,
                                             stdout=out_stream,
                                             stderr=err_stream)
```

- *my_client* — Name of object returned from `initialize()`
- *function_name* — Name of the function to invoke
- *in_args* — Comma-separated list of input arguments
- *nargs* — Number of expected results. The default value is `1`.
- *out_stream* — Python `StringIO` object receiving the console output. The default is to direct output to the console.
- *err_stream* — Python `StringIO` object receiving the error output. The default is to direct output to the console.

Each variable on the left side of the function call is populated with a single return value.

Note: If you provide less than *nargs* variables on the left side of the function call, the last listed variable will contain a list of the remaining results. For example

```
result1, result2 = myMagic.triple(5,nargout=3)
```

leaves `result1` containing a single value and `result2` containing a list with two values.

Invoke a MATLAB Function that Returns a Single Output

To invoke the MATLAB function `result = mutate(m1, m2, m3)` from the package `mutations`, you use this code:

```
import mutations
import matlab

myMutator = mutations.initialize()

m1 = matlab.double(...)
m2 = matlab.double(...)
m3 = matlab.double(...)

result = myMutator.mutate(m1,m2,m3)
```

Invoke a MATLAB Function that Returns Zero Outputs

To invoke the MATLAB function `mutate(m1,m2,m3)` from the package `mutations`, you use this code:

```
import mutations
import matlab

myMutator = mutations.initialize()

m1 = matlab.double(...)
m2 = matlab.double(...)
m3 = matlab.double(...)

myMutator.mutate(m1,m2,m3,nargout=0)
```

Receive Multiple Results as Individual Variables

To invoke the MATLAB function `c1,c2 = copy(o1,o2)` from the package `copier`, use this code:

```
>>> import copier
>>> import matlab
>>> myCopier = copier.initialize()
>>> c1,c2 = myCopier.copy("blue",10,nargout=2)
>>> print(c1)
"blue"
>>> print(c2)
10
```

Receive Multiple Results as a Single Object

To invoke the MATLAB function `copies = copy(o1,o2)` from the package `copier`, use this code:

```
>>> import copier
>>> import matlab
>>> myCopier = copier.initialize()
>>> copies = myCopier.copy("blue",10,nargout=2)
>>> print(copies)
["blue",10]
```

Related Examples

- "Initialize the MATLAB Runtime" on page 2-4
- "Create a Python Application with MATLAB Code"

Evaluate a Compiled MATLAB Function Asynchronously

Asynchronously evaluate a compiled MATLAB function that uses the Python object returned from the `initialize()` function by passing `async = True`.

```
future = my_client.function_name(in_args, nargout=nargs,
                                 stdout=out_stream,
                                 stderr=err_stream,
                                 async=True)
```

- *my_client* — Name of object returned from `initialize()`
- *function_name* — Name of the function to invoke
- *in_args* — Comma-separated list of input arguments
- *nargs* — Number of results expected from the server
- *out_stream* — Python `StringIO` object receiving the console output
- *err_stream* — Python `StringIO` object receiving the error output

When the `async` keyword is set to `True`, the MATLAB function is placed into a processing queue and a Python `Future` object is returned. You use the `Future` object to retrieve the results when the MATLAB function is finished processing.

To invoke the MATLAB function `c1,c2= copy(o1,o2)` from the package `copier` asynchronously, use the following code:

```
>>> import mutations
>>> import matlab
>>> myMutator = mutations.initialize()
>>> m1 = matlab.double(...)
>>> m2 = matlab.double(...)
>>> m3 = matlab.double(...)
>>> resultFuture = myMutator.mutate(m1,m2,m3, async=True)
>>> while !resultFuture.done():
...     time.sleep(1)
...
>>> result = resultFuture.result()
```

Tip You can cancel asynchronous requests using the `cancel()` method of the `Future` object.

Related Examples

Integrate Python Packages on Mac OS X

To use MATLAB Compiler SDK Python packages on Mac OS X, use the `mwpython` script. The `mwpython` script is located in the *mcrroot*`\bin` folder. *mcrroot* is the location of your MATLAB Runtime installation.

For example, to run the example in "Create a Python Application with MATLAB Code" you enter `mwpython getmagic.py`.

See Also
`mwpython`

Data Handling

MATLAB Arrays as Python Variables

In this section...

The `matlab` Python package provides array classes to represent arrays of MATLAB numeric types as Python variables.

Create MATLAB Arrays in Python

You can create MATLAB numeric arrays in a Python session by calling constructors from the `matlab` Python package (for example, `matlab.double`, `matlab.int32`). The name of the constructor indicates the MATLAB numeric type. You can pass MATLAB arrays as input arguments to MATLAB functions called from Python. When a MATLAB function returns a numeric array as an output argument, the array is returned to Python.

You can initialize the array with an optional `initializer` input argument that contains numbers. The `initializer` argument must be a Python sequence type such as a list or a tuple. The optional `size` input argument sets the size of the initialized array. To create multidimensional arrays, specify `initializer` to contain multiple sequences of numbers, or specify `size` to be multidimensional. You can create a MATLAB array of complex numbers by setting the optional `is_complex` keyword argument to `True`. The `mlarray` module provides the MATLAB array constructors listed in the table.

Class from `matlab` Package	Constructor Call in Python	MATLAB Numeric Type

`matlab.double`	`matlab.double(`
	`initializer=None,`
	`size=None,`
	`is_complex=False)`
		Double precision

`matlab.single`	`matlab.single(`
	`initializer=None,`
	`size=None,`
	`is_complex=False)`
		Single precision

Class from `matlab` Package	Constructor Call in Python	MATLAB Numeric Type
`matlab.int8`	`matlab.int8( initializer=None, size=None, is_complex=False)`	8-bit signed integer
`matlab.int16`	`matlab.int16( initializer=None, size=None, is_complex=False)`	16-bit signed integer
`matlab.int32`	`matlab.int32( initializer=None, size=None, is_complex=False)`	32-bit signed integer
`matlab.int64`[a]	`matlab.int64( initializer=None, size=None, is_complex=False)`	64-bit signed integer
`matlab.uint8`	`matlab.uint8( initializer=None, size=None, is_complex=False)`	8-bit unsigned integer
`matlab.uint16`	`matlab.uint16( initializer=None, size=None, is_complex=False)`	16-bit unsigned integer
`matlab.uint32`	`matlab.uint32( initializer=None, size=None, is_complex=False)`	32-bit unsigned integer
`matlab.uint64`[b]	`matlab.uint64( initializer=None, size=None, is_complex=False)`	64-bit unsigned integer
`matlab.logical`	`matlab.logical( initializer=None, size=None)`[c]	Logical

a.   In Python 2.7 on Windows, `matlab.int64` is converted to `int32` in MATLAB. Also, MATLAB cannot return an `int64` array to Python.

b.   In Python 2.7 on Windows, `matlab.uint64` is converted to `uint32` in MATLAB. Also, MATLAB cannot return a `uint64` array to Python.

    c.   Logicals cannot be made into an array of complex numbers.

When you create an array with N elements, the size is 1-by-N because it is a MATLAB array.

```
import matlab
A = matlab.int8([1,2,3,4,5])
print(A.size)

(1, 5)
```

The initializer is a Python list containing five numbers. The MATLAB array size is 1-by-5, indicated by the tuple (1,5).

MATLAB Array Attributes and Methods in Python

All MATLAB arrays created with `matlab` package constructors have the attributes and methods listed in the table below:

Attribute or Method	Purpose
size	Size of array returned as a tuple
reshape(size)	Reshape the array as specified by the sequence size

Multidimensional MATLAB Arrays in Python

In Python, you can create multidimensional MATLAB arrays of any numeric type. Use two Python lists of floats to create a 2-by-5 MATLAB array of doubles.

```
import matlab
A = matlab.double([[1,2,3,4,5], [6,7,8,9,10]])
print(A)

[[1.0,2.0,3.0,4.0,5.0],[6.0,7.0,8.0,9.0,10.0]]
```

The `size` attribute of A shows it is a 2-by-5 array.

```
print(A.size)

(2, 5)
```

Index Into MATLAB Arrays in Python

You can index into MATLAB arrays just as you can index into Python lists and tuples.

```
import matlab
A = matlab.int8([1,2,3,4,5])
print(A[0])
```

```
[1,2,3,4,5]
```

The size of the MATLAB array is `(1,5)`; therefore, `A[0]` is `[1,2,3,4,5]`. Index into the array to get 3.

```
print(A[0][2])
```

```
3
```

Python indexing is zero-based. When you access elements of MATLAB arrays in a Python session, use zero-based indexing.

This example shows how to index into a multidimensional MATLAB array.

```
A = matlab.double([[1,2,3,4,5], [6,7,8,9,10]])
print(A[1][2])
```

```
8.0
```

Slice MATLAB Arrays in Python

You can slice MATLAB arrays just as you can slice Python lists and tuples.

```
import matlab
A = matlab.int8([1,2,3,4,5])
print(A[0][1:4])
```

```
[2,3,4]
```

You can assign data to a slice. This example shows an assignment from a Python list to the array.

```
A = matlab.double([[1,2,3,4],[5,6,7,8]])
A[0] = [10,20,30,40]
print(A)
```

```
[[10.0,20.0,30.0,40.0],[5.0,6.0,7.0,8.0]]
```

You can assign data from another MATLAB array, or from any Python iterable that contains numbers.

You can specify slices for assignment, as shown in this example.

```
A = matlab.int8([1,2,3,4,5,6,7,8])
A[0][2:4] = [30,40]
A[0][6:8] = [70,80]
print(A)

[[1,2,30,40,5,6,70,80]]
```

Note: Slicing MATLAB arrays behaves differently from slicing a Python list. Slicing a MATLAB array returns a view instead of a shallow copy.

Given a MATLAB array and a Python list with the same values, assigning a slice results in different results.

```
>>>mlarray = matlab.int32([[1,2],[3,4],[5,6]])
>>>py_list = [[1,2],[3,4],[5,6]]
>>>mlarray[0] = mlarray[0][::-1]
>>>py_list[0] = py_list[0][::-1]
>>>mlarray[0]
matlab.int32([[2,2],[3,4],[5,6]])
>>>py_list
[[2,1],[3,4],[5,6]]
```

Reshaping MATLAB Arrays in Python

You can reshape a MATLAB array in Python with the `reshape` method. The input argument, `size`, must be a sequence that does not change the number of elements in the array. Use `reshape` to change a 1-by-9 MATLAB array to 3-by-3.

```
import matlab
A = matlab.int8([1,2,3,4,5,6,7,8,9])
A.reshape((3,3))
print(A)

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

Related Examples

-

Use MATLAB Arrays in Python

This example shows how to use MATLAB arrays in Python.

The `matlab` package provides new Python data types to create arrays that can be passed to MATLAB functions. The `matlab` package can create arrays of any MATLAB numeric or logical type from Python sequence types. Multidimensional MATLAB arrays are supported.

Create a MATLAB array in Python, and call a MATLAB function on it.

```
import matlab
from production_server import client
client_obj = client.MWHttpClient("http://localhost:9910")
x = matlab.double([1,4,9,16,25])
print(client_obj.myArchive.sqrt(x))

[[1.0,2.0,3.0,4.0,5.0]]
```

You can use `matlab.double` to create an array of doubles given a Python list that contains numbers. You can call a MATLAB function such as `sqrt` on `x`, and the return value is another `matlab.double` array.

Create a multidimensional array. The `magic` function returns a 2-D array to Python scope.

```
a = client_obj.myArchive.magic(6)
print(a)

[[35.0,1.0,6.0,26.0,19.0,24.0],[3.0,32.0,7.0,21.0,23.0,25.0],
 [31.0,9.0,2.0,22.0,27.0,20.0],[8.0,28.0,33.0,17.0,10.0,15.0],
 [30.0,5.0,34.0,12.0,14.0,16.0],[4.0,36.0,29.0,13.0,18.0,11.0]]
```

Call the `tril` function to get the lower triangular portion of `a`.

```
b = client_obj.myArchive.tril(a)
print(b)

[[35.0,0.0,0.0,0.0,0.0,0.0],[3.0,32.0,0.0,0.0,0.0,0.0],
 [31.0,9.0,2.0,0.0,0.0,0.0],[8.0,28.0,33.0,17.0,0.0,0.0],
```

```
[30.0,5.0,34.0,12.0,14.0,0.0],[4.0,36.0,29.0,13.0,18.0,11.0]]
```

More About

- "MATLAB Arrays as Python Variables" on page 3-2

Functions — Alphabetical List

mwpython

Start a Python session using a MATLAB Compiler SDK Python package on Mac OS X

Syntax

```
mwpython [-verbose] [py_args] [-mlstartup opt[,opt]]
python_scriptname
mwpython [-verbose] [py_args] [-mlstartup opt[,opt]] -c cmd
mwpython [-verbose] [py_args] [-mlstartup opt[,opt]] -m mod
```

Description

`mwpython [-verbose] [py_args] [-mlstartup opt[,opt]]`
`python_scriptname` starts a Python session that executes a Python script.

`mwpython [-verbose] [py_args] [-mlstartup opt[,opt]] -c cmd` starts
Python session that executes a Python command.

`mwpython [-verbose] [py_args] [-mlstartup opt[,opt]] -m mod` starts a
Python session that executes a Python module.

Input Arguments

py_args — Python arguments

Python arguments, specified as a comma-separated list.

opt[,opt] — MATLAB Runtime startup options
-nojvm | -nodisplay | -logfile

MATLAB Runtime startup options, specified as a comma-separated list.

- `-nojvm` — disable the Java Virtual Machine, which is enabled by default. This can help improve the MATLAB Runtime performance.
- `-nodisplay` — on Linux, run the MATLAB Runtime without display functionality.

`python_scriptname` — Python script to execute

Python script to execute, specified as a string with a `.py` extension.

`cmd` — Python command to execute

Python command to execute, specified as a string.

`mod` — Python module to execute

Python module to execute, specified as a string.

Examples

Execute Python script with in verbose mode

```
mwpython -verbose myfile.py
```

Execute Python module with arguments

```
mwpython -m mymod arg1 arg2
```

Introduced in R2015b