MATLAB® Compiler SDK™

Python® User's Guide
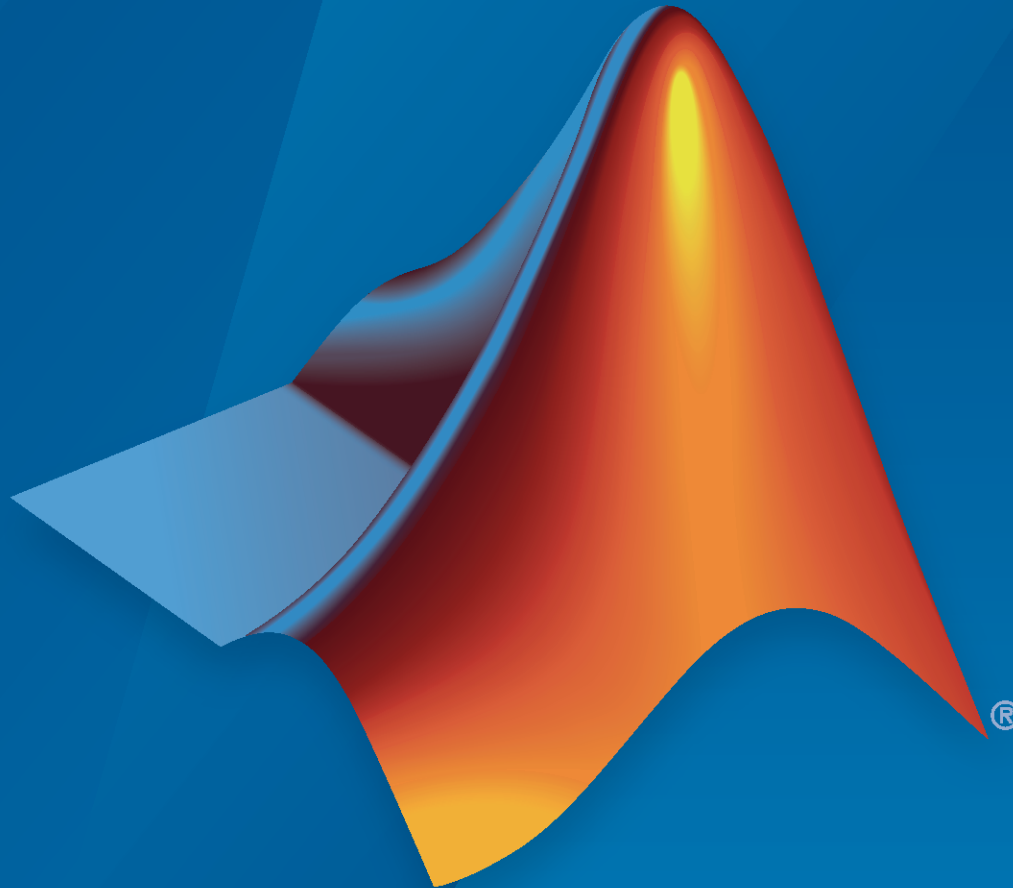
MATLAB®

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.
1 Apple Hill Drive
Natick, MA 01760-2098

Revision History

Contents

Functions

4

Python Package Installation

Install a MATLAB Compiler SDK Python Package

In this section...
"With the Generated Installer" on page 1-2
"Without the Generated Installer" on page 1-2

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 setup script. To install to a location other than the default, consult "Installing Python Modules" in the official Python documentation.

   ```
   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
   ```

   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 A command must be entered as a single line without white space between path components. Commands are wrapped here for readability.

Without the Generated Installer

If you already have the MATLAB Runtime installed, you can 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 setup script. To install to a location other than the default, consult "Installing Python Modules" in the official Python documentation.

```
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
```

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 A command must be entered as a single line without white space between path components. Commands are wrapped here for readability.

See Also

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 1-2.
2   In consultation with the MATLAB programmer, collect 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 3-3.
6   Invoke the MATLAB functions.

   See "Invoke a Compiled MATLAB Function" on page 2-6 or "Invoke a Compiled MATLAB Function Asynchronously" on page 2-8.
7   Terminate each package using its `terminate()` function. If you do not call this function explicitly, it is called automatically when the program exits.

Import Compiled Python Packages

The MATLAB Compiler SDK Python target generates the Python code into a package that must be imported into Python 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 name as `addmatrix` with no namespace, you import it as follows:

```
import addmatrix
```

If you compile your MATLAB functions using the `mcc` command with the option `-W python:com.mathworks.addmatrix`, you import it as follows:

```
import com.mathworks.addmatrix
```

See Also

Related Examples

- "Generate a Python Package and Build a Python Application"
- "Package Python Applications from Command Line"

Initialize the MATLAB Runtime

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

1 Call the `initialize_runtime()` function, which allows you to provide a list of startup options to the MATLAB Runtime.
2 Use the `initialize()` function of each compiled package in the application. The first time the function is called, a MATLAB Runtime is loaded and started.

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 invoke 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 invoke 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 before a call to `initialize()` function, the MATLAB Runtime is started with no startup options.

Note You cannot import `matlab.engine` after importing your component. For more information on `matlab.engine`, see "Start and Stop MATLAB Engine for Python".

See Also

More About

- "Integrate Python Packages on Mac OS X" on page 2-9

Invoke a Compiled MATLAB Function

In this section...
"Invoke a MATLAB Function with a Single Output" on page 2-6
"Invoke a MATLAB Function with Zero Outputs" on page 2-7
"Receive Multiple Results as Individual Variables" on page 2-7
"Receive Multiple Results as a Single Object" on page 2-7

Invoke 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 contains 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 with 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([1,2,3])
m2 = matlab.double([10,20,30])
m3 = matlab.double([100,200,300])

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

Invoke a MATLAB Function with 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([1,2,3])
m2 = matlab.double([10,20,30])
m3 = matlab.double([100,200,300])

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]
```

See Also

Related Examples
- "Initialize the MATLAB Runtime" on page 2-4
- "Generate a Python Package and Build a Python Application"

Invoke a Compiled MATLAB Function Asynchronously

Asynchronously invoke 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([1,2,3])
>>> m2 = matlab.double([10,20,30])
>>> m3 = matlab.double([100,200,300])
>>> 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.

See Also

Related Examples
- "Initialize the MATLAB Runtime" on page 2-4
- "Generate a Python Package and Build a Python Application"

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 "Generate a Python Package and Build a Python Application" you enter `mwpython getmagic.py`.

See Also

`mwpython`

Differences Between MATLAB Engine API for Python and MATLAB Compiler SDK

MATLAB Engine API for Python enables you to call MATLAB as a computational engine. The main differences between MATLAB Engine API for Python and MATLAB Compiler SDK for Python are as follows:

- MATLAB Engine API for Python starts a MATLAB session out-of-process, which executes MATLAB as a separate process. MATLAB Compiler SDK for Python starts MATLAB Runtime in-process.
- You can use MATLAB Engine API for Python to call built-in or user-written MATLAB functions. MATLAB Compiler SDK for Python can only call user-written MATLAB functions.

  For an example of calling user-written MATLAB code with MATLAB Engine API for Python, see "Call User Scripts and Functions from Python".
- MATLAB Engine API for Python allows you to work with a workspace, while MATLAB Compiler SDK for Python does not. Therefore you cannot call MATLAB classes (handles) with MATLAB Compiler SDK for Python.

  For more information on MATLAB Engine workspace, see "Use MATLAB Engine Workspace in Python".

See Also

Related Examples

- "Generate a Python Package and Build a Python Application"
- "Initialize the MATLAB Runtime" on page 2-4
- "Get Started with MATLAB Engine API for Python"
- "Start and Stop MATLAB Engine for Python"
- "Call MATLAB Functions from Python"

Data Handling

Pass Data to MATLAB from Python

When you pass data as input arguments to MATLAB functions from Python, MATLAB converts the data into equivalent MATLAB data types.

Python Input Argument Type	Resulting MATLAB Data Type (scalar unless otherwise noted)
`matlab` numeric array object (see "MATLAB Arrays as Python Variables" on page 3-3)	Numeric array
`float`	`double`
`complex`	Complex `double`
`int`	`int32`(Windows®) `int64`(Linux and Mac)
`long` [a]	`int64`
`float('nan')`	NaN
`float('inf')`	Inf
`bool`	`logical`
`str`	`char`
`bytearray`	`uint8` array
`bytes`	`uint8` array
`dict`	Structure if all keys are strings Not supported otherwise
`list`	Cell array
`set`	Cell array
`tuple`	Cell array
`memoryview`	Not supported
`range`	Cell array
None	Not supported
module.type	Not supported

a.   long is a data type of Python 2.7 only

See Also

Related Examples

- "MATLAB Arrays as Python Variables" on page 3-3
- "Use MATLAB Arrays in Python" on page 3-9
- "Handle Data Returned from MATLAB to Python" on page 3-8

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. Other MATLAB types are also supported, as listed in "Pass Data to MATLAB from Python". For information on installing the `matlab` Python package, see "Install a MATLAB Compiler SDK Python Package" on page 1-2.

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

`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

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

`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 on Windows, `matlab.int64` is converted to `int32` in MATLAB. Also, MATLAB cannot return an `int64` array to Python.
b. In Python 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 following table:

Attribute or Method	Purpose
`size`	Size of array returned as a tuple

Attribute or Method	Purpose
`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]]
```

See Also

Related Examples

- "Use MATLAB Arrays in Python" on page 3-9
- "Pass Data to MATLAB from Python"

Handle Data Returned from MATLAB to Python

When MATLAB functions return output arguments, MATLAB converts the data into equivalent Python data types.

MATLAB Output Argument Type (scalar unless otherwise noted)	Resulting Python Data Type
Numeric array	`matlab` numeric array object (see "MATLAB Arrays as Python Variables" on page 3-3)
`double` `single`	`float`
Complex (any numeric type)	`complex`

`int8`
`uint8`
`int16`
`uint16`
`int32`
	`int`

`uint32`
`int64`
`uint64`
	`int`
	`long`

`NaN`	`float('nan')`
`Inf`	`float('inf')`
`logical`	`bool`
`char` array (1-by-N, N-by-1) `char` array (M-by-N)	`str` Not supported
structure	`dict`
Row or column cell array	`list`
M-by-N cell array	Not supported
MATLAB handle object (`table`, `containers.Map`, categorical array)	Not supported
Other object (e.g., Java)	Not supported
Function handle	Not supported
Sparse array	Not supported
String array	Not supported
Structure array	Not supported

See Also

Related Examples
- "MATLAB Arrays as Python Variables" on page 3-3
- "Use MATLAB Arrays in Python" on page 3-9
- "Pass Data to MATLAB from Python" on page 3-2

Use MATLAB Arrays in Python

To use MATLAB arrays in Python, you can either install the Python engine before running your packaged application, as described in "Install MATLAB Engine API for Python", or use `import mypackage` before `import matlab` in the following programs.

The MATLAB Engine API for Python provides a Python package named `matlab` that enables you to call MATLAB functions from Python. The `matlab` package provides constructors to create MATLAB arrays in Python. It can create arrays of any MATLAB numeric or logical type from Python sequence types. Multidimensional MATLAB arrays are supported. For a list of other supported array types, see "Pass Data to MATLAB from Python".

Examples

1   Create a MATLAB array in Python, and call a MATLAB function on it. Assuming that you have a package named `mypackage` and a method called `mysqrt` inside the package, you can use `matlab.double` to create an array of doubles given a Python list that contains numbers. You can call the MATLAB function `mysqrt` on `x`, and the return value is another `matlab.double` array as shown in the following program:

```
import matlab
import mypackage
pkg = mypackage.initialize()
x = matlab.double([1,4,9,16,25])
print(pkg.mysqrt(x))
```

The output is:

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

2   Create a multidimensional array. The `magic` function returns a 2-D array to Python scope. Assuming you have method called `mysqrt` inside `mypackage`, you can use the following code to call that method:

```
import matlab
import mypackage
pkg = mypackage.initialize()
x = matlab.double([1,4,9,16,25])
print(pkg.mymagic(6))
```

The output is:

```
[[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]]
```

See Also

More About

*   "MATLAB Arrays as Python Variables" on page 3-3
*   "Pass Data to MATLAB from Python"

Functions

myDeployedModule.initialize

Initialize package and return a handle

Syntax

```
myobj = myDeployedModule.initialize()
```

Description

`myobj = myDeployedModule.initialize()` initializes a package consisting of one or more deployed MATLAB functions. The return value is used as a handle on which any of the functions can be executed.

Examples

Create a Handle to a Deployed MATLAB Function

This example shows how to create a handle to a package named `myDeployedModule`. This handle is then used for calling a deployed MATLAB function called `makesqr`.

```
import myDeployedModule

myobj = myDeployedModule.initialize()

print(myobj.makesqr(3))

myobj.terminate()
```

Output Arguments

`myobj` — Output a handle to deployed MATLAB functions
Python object

Output a handle to deployed MATLAB functions, returned as a Python object used to execute deployed MATLAB functions.

See Also
`myDeployedModule.terminate`

Topics
"Invoke a Compiled MATLAB Function" on page 2-6

myDeployedModule.initialize_runtime

Initialize MATLAB Runtime with a list of startup options

Syntax

```
myobj = myDeployedModule.initialize_runtime()
```

Description

`myobj = myDeployedModule.initialize_runtime()` initializes the MATLAB Runtime with a list of startup options that affects all packages opened within the script.

If it is not called explicitly, it is executed automatically, with an empty list of options, by the first call to `initialize()`. Do not call `initialize_runtime()` after calling `initialize()`. There is no corresponding `terminate_runtime()` call. The MATLAB Runtime terminates automatically when the script or session ends.

Input Arguments

`in_args` — Startup options to MATLAB Runtime
comma separated list of options

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

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

Output Arguments

`myobj` — Output a handle to deployed MATLAB functions
Python object

Output a handle to deployed MATLAB functions, returned as a Python object used to execute deployed MATLAB functions.

Examples

Specify MATLAB Runtime Options

This example shows how to specify MATLAB Runtime options when creating a handle to a package named myDeployedModule.

```
import myDeployedModule

myobj = myDeployedModule.initialize_runtime(['-nojvm', '-nodisplay'])

print(myobj.makesqr(3))
```

```
myobj.terminate()
```

See Also
`myDeployedModule.terminate`

Topics
"Initialize the MATLAB Runtime" on page 2-4

myDeployedModule.terminate

Close a package

Syntax

```
myDeployedModule.terminate()
```

Description

myDeployedModule.terminate() closes a package consisting of one or more deployed MATLAB functions. myDeployedModule.terminate() can be called on a package handle, after which no functions can be called on the handle.

If you exit from a script or session, myDeployedModule.terminate() is called automatically. Hence, calling it explicitly is optional, but a good idea because it frees resources at that point. Alternatively, you can use quit() or exit().

Examples

Close a Handle to a Deployed MATLAB Function

This example shows how to create a handle to a package named myDeployedModule, and close the handle after calling a deployed MATLAB function.

```
import myDeployedModule

myobj = myDeployedModule.initialize()

print(myobj.makesqr(3))

myobj.terminate()
```

See Also
myDeployedModule.initialize | myDeployedModule.initialize_runtime

Topics
"Invoke a Compiled MATLAB Function" on page 2-6

myDeployedModule.wait_for_figures_to_close

Wait for all graphical figures to close before continuing

Syntax

```
myDeployedModule.wait_for_figures_to_close()
```

Description

`myDeployedModule.wait_for_figures_to_close()` enables the deployed application to process graphics events. The purpose of `myDeployedModule.wait_for_figures_to_close()` is to block execution of a calling program as long as figures created in deployed MATLAB code are displayed.

This function can only be called after `initialize()` has been called and before `terminate()` has been called. If this function is not called, any figure windows initially displayed by the application briefly appear, and then the application exits.

Examples

Keep a Figure in MATLAB Function Open

This example shows how to keep a MATLAB plot open after it is invoked using the `showplot` function in a package named `myDeployedModule`.

```
import myDeployedModule

myobj = myDeployedModule.initialize()

myobj.showplot()

myobj.wait_for_figures_to_close()

myobj.terminate()
```

See Also
`myDeployedModule.terminate`

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 character array with a .py extension.

cmd — Python command to execute

Python command to execute, specified as a character array.

mod — Python module to execute

Python module to execute, specified as a character array.

Note If you want to use a specific version of Python, set the PYTHONHOME environment variable on your machine to point to the location of your desired Python installation prior to invoking mwpython.

Examples

Execute a Python Script in Verbose Mode

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

Execute a Python Module with Arguments

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

Introduced in R2015b