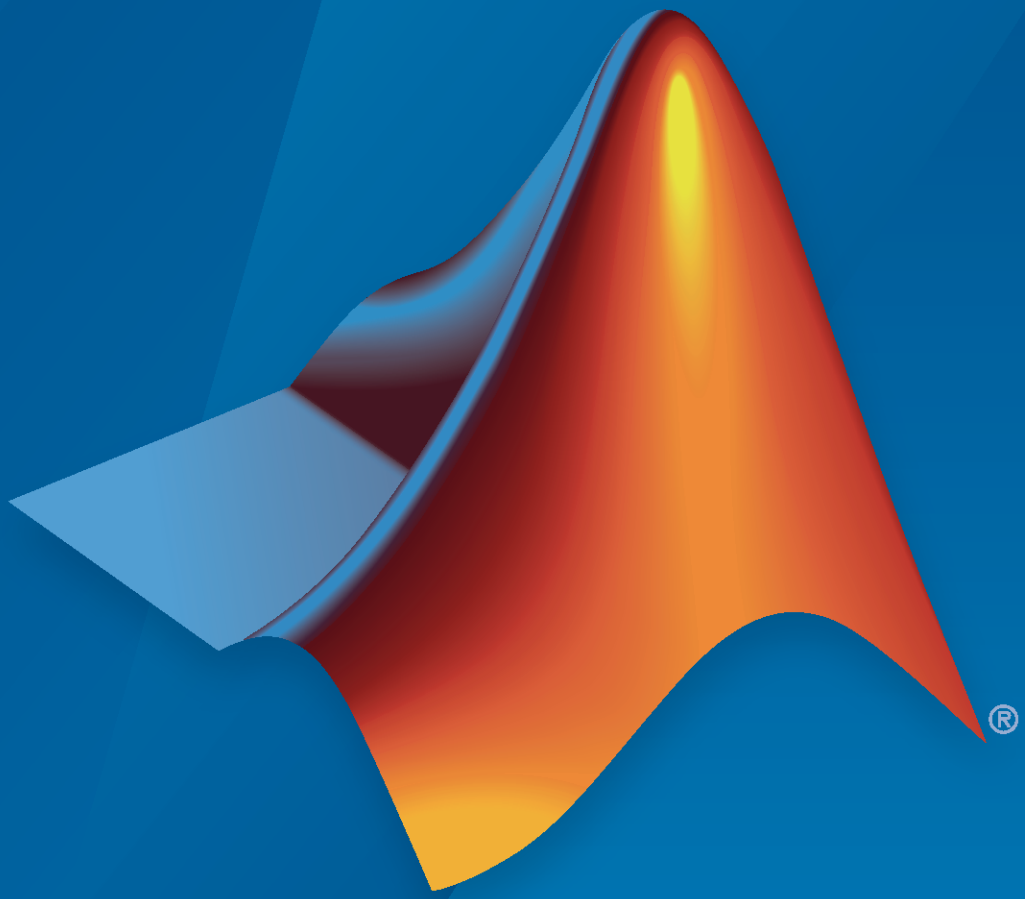
MATLAB® Compiler SDK™

Python® User's Guide



MATLAB®

R2022b

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

MATLAB® Compiler SDK™ Python® User's Guide

Revision History

September 2015	Online only	New for Version 6.1 (Release 2015b)
March 2016	Online only	Revised for Version 6.2 (Release 2016a)
September 2016	Online only	Revised for Version 6.3 (Release R2016b)
March 2017	Online only	Revised for Version 6.3.1 (Release R2017a)
September 2017	Online only	Revised for Version 6.4 (Release R2017b)
March 2018	Online only	Revised for Version 6.5 (Release R2018a)
September 2018	Online only	Revised for Version 6.6 (Release R2018b)
March 2019	Online only	Revised for Version 6.6.1 (Release R2019a)
September 2019	Online only	Revised for Version 6.7 (Release R2019b)
March 2020	Online only	Revised for Version 6.8 (Release R2020a)
September 2020	Online only	Revised for Version 6.9 (Release R2020b)
March 2021	Online only	Revised for Version 6.10 (Release R2021a)
September 2021	Online only	Revised for Version 6.11 (Release R2021b)
March 2022	Online only	Revised for Version 7.0 (Release R2022a)
September 2022	Online only	Revised for Version 7.1 (Release R2022b)

Contents

1 Python Integration

2 Data Handling

3 Functions

Python Integration

Integrate a Python Package

To integrate a MATLAB Compiler SDK Python Package:

1. In consultation with the MATLAB programmer, collect the MATLAB function signatures that comprise the services in the application.
2. Install and import the compiled Python package.

   See “Install and Import MATLAB Compiler SDK Python Packages” on page 1-3.
3. Write the Python code to initialize MATLAB Runtime and load the MATLAB code.

   See “Initialize MATLAB Runtime” on page 1-5.
4. Create the required MATLAB data for function inputs and outputs.

   See “matlab Python Module” on page 2-4.
5. Invoke the MATLAB functions.

   See “Invoke a Packaged MATLAB Function” on page 1-7 or “Invoke a Compiled MATLAB Function Asynchronously” on page 1-10.
6. Terminate each package using its `terminate()` function. If you do not call this function explicitly, it is called automatically when the program exits.

Install and Import MATLAB Compiler SDK Python Packages

After you generate a Python package using MATLAB Compiler SDK, you must install MATLAB Runtime and the package on the target machine. Import the package in your Python application to call the compiled MATLAB functions.

Install Python Package

If you create a package using the Library Compiler, the app generates an installer in the `for_redistribution` folder that installs MATLAB Runtime and the files required to install the generated Python package. If you create a package using `compiler.build.pythonPackage` or `mcc`, you can generate an installer using the `compiler.package.installer` function.

You can also distribute the Python package files manually. You must install MATLAB Runtime separately.

Using Generated Installer

1. Distribute the installer to the desired location.
2. Run the installer.
3. Note the location where the installer writes the Python package files.
4. When the installer finishes, open a system command prompt window and navigate to 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. Add the MATLAB Runtime folders to the library path environment variable. For details, see “Set MATLAB Runtime Path for Deployment”.

Using Package Files

1. Distribute the following files to integrate a Python package in an application:
   - Function signatures of the deployed MATLAB functions
   - `_init_.py` — initialization script for the Python package
   - `setup.py` — generated Python installer
2. Open a system command prompt window and navigate to 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. Install MATLAB Runtime that matches the version of MATLAB used to compile the Python package. For details, see “Install and Configure MATLAB Runtime”.
5. Add the MATLAB Runtime folders to the library path environment variable. For details, see “Set MATLAB Runtime Path for Deployment”.

Import Python Package

During compilation, you specify the package namespace, which is a period-separated list, such as `companyname.groupname.packageName`.

- If you use the `compiler.build.pythonPackage` function, you specify the namespace using the `PackageName` option. The name of the generated package is set to the last entry of the period-separated list.

  If not specified, `PackageName` defaults to the name of the first MATLAB file listed in the `FunctionFiles` argument.
- 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.

  If not specified, the namespace defaults to the name of the first MATLAB file listed in the command.

Import the package in your Python application using the `import` statement followed by the full namespace.

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 namespace `com.mathworks.addmatrix`, you import it as follows:

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

See Also

`compiler.build.pythonPackage` | `mcc`

Related Examples

- “Generate a Python Package and Build a Python Application”
- “Invoke a Packaged MATLAB Function” on page 1-7
- “Install and Configure MATLAB Runtime”
- “Initialize MATLAB Runtime” on page 1-5

External Websites

- `docs.python.org/3/installing/index.html`

Initialize MATLAB Runtime

When integrating compiled MATLAB functions into a Python application, your code must initialize MATLAB Runtime and any compiled packages in the application.

1. Call the `initialize_runtime()` function, which allows you to provide a list of startup options to MATLAB Runtime. This loads and starts MATLAB Runtime.
2. Use the `initialize()` function of each compiled package in the application to retrieve a handle that can be used to call the MATLAB functions within the package.

Provide MATLAB Runtime Startup Options

Note On macOS, 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 MATLAB Runtime's performance.
- `-nodisplay` — on Linux®, run 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 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 MATLAB Runtime session. Any subsequent calls are ignored.

Start MATLAB Runtime with Compiled MATLAB Functions

To invoke a compiled MATLAB function, load it into 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 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, 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

- “Install and Configure MATLAB Runtime”

Invoke a Packaged MATLAB Function

In this section...

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 MATLAB Function with 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 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 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]
```

Invoke a MATLAB Function in the Background

To invoke a MATLAB function `sumwrap` from the package `sumwrap` asynchronously, use this code:

```
>>> import sumwrap
>>> import matlab
>>> sw = sumwrap.initialize()
>>> a = matlab.double([[1, 2],[3, 4]])
>>> future = sw.sumwrap(a, 1, background=True)
>>> future.result()
matlab.double([[4.0,6.0]])
```

See Also

Related Examples

- “Initialize MATLAB Runtime” on page 1-5
- “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 MATLAB Runtime” on page 1-5
- “Generate a Python Package and Build a Python Application”

Create Python Application with Multiple MATLAB Functions

Supported platforms: Windows®, Linux, Mac

This example shows how to create a Python application that uses multiple MATLAB functions to compute data from a rectangle.

Prerequisites

Verify that you have a version of Python installed that is compatible with MATLAB Compiler SDK. For details, see MATLAB Supported Interfaces to Other Languages.

This example uses the following files in *`matlabroot`*`\extern\examples\compilersdk\python\rectangle\`:

MATLAB Functions	`getPointCoordinates.m`
	`getRectangleArea.m`
	`getRectangleCorners.m`
	`getRectangleHeight.m`
	`getRectangleWidth.m`
	`makePoint.m`
	`makeRectangle.m`
	`Point.m`
	`Rectangle.m`
	`rectangleDemo.m`

Python Application Code	`rectangleDriver.py`

Create and Run Python Application

1 At the MATLAB command prompt, copy the `rectangle` folder that ships with MATLAB to your work folder.

```
copyfile(fullfile(matlabroot,"extern","examples","compilersdk","python","rectangle"),"rectang
```

Navigate to the new `rectangle` folder in your work folder.

2 Examine the MATLAB function `rectangleDemo.m`.

The function creates two `Point` objects, then creates a `Rectangle` object using the points as corners. It then calculates and displays data about the rectangle.

```
function rectangleDemo()
    % RECTANGLEDEMO Construct a rectangle and print information about it

    pointZeroZero = makePoint(0, 0);
    pointThreeFour = makePoint(3, 4);
    rectA = makeRectangle(pointZeroZero, pointThreeFour);
    corners = getRectangleCorners(rectA);
    showPointCoordinates(corners.upperLeft, 'Upper left-hand corner');
    showPointCoordinates(corners.lowerLeft, 'Lower left-hand corner');
    showPointCoordinates(corners.upperRight, 'Upper right-hand corner');
    showPointCoordinates(corners.lowerRight, 'Lower right-hand corner');
    fprintf('Area: %.1f\n', area(rectA));
    fprintf('Height: %.1f\n', height(rectA));
    fprintf('Width: %.1f\n', width(rectA));
end

% This is an auxiliary function. It cannot be called outside rectangleDemo().
```

```
function showPointCoordinates(pt, desc)
    coordinates = getPointCoordinates(pt);
    fprintf('%s: (%.1f, %.1f)\n', desc, coordinates.X, coordinates.Y);
end
```

3. Build a Python package named `rectangleLib` using all of the MATLAB function files in the `rectangle` folder.

   a. Get the list of files with a `.m` extension in the current directory.

   ```
   functionfiles = dir('*.m')
   ```

   b. Save the filenames in a cell array.

   ```
   functionfiles = {functionfiles.name};
   ```

   c. Compile the Python package using `compiler.build.pythonPackage`.

   ```
   buildResults = compiler.build.pythonPackage(functionfiles,'PackageName','rectangleLib');
   ```

   For more details, see “Generate a Python Package and Build a Python Application”.

4. Write source code for a Python application that accesses the MATLAB functions.

   The sample application for this example is `rectangleDriver.py` in the `rectangle` folder.

   rectangleDriver.py

```
# Copyright 2019 The MathWorks, Inc.
import rectangleLib

def initializeLibrary():
    """ Initialize MATLAB rectangleLibrary library and retrieve a handle to it """
    handle = rectangleLib.initialize()
    return handle

class PyPoint:
    """Python class that uses a MATLAB class compiled with MATLAB Compiler SDK"""
    libHandle = 0

    @staticmethod
    def initializeHandle(handle):
        PyPoint.libHandle = handle

    # Initialize the point either from an existing Point (not PyPoint) object,
    # or from an X and Y pair.
    def __init__(self, pt=None, X=None, Y=None):
        if pt:
            if X or Y:
                raise ValueError('If pt is specified, neither X nor Y can be specified')
            else:
                self.data = pt
        else:
            if not X or not Y:
                raise ValueError('If pt is not specified, X and Y must be specified')
            else:
                self.data = PyPoint.libHandle.makePoint(X, Y)

    def getCoordinates(self):
        coordinates = PyPoint.libHandle.getPointCoordinates(self.data)
        return (coordinates['X'], coordinates['Y'])

class PyRectangle:
    """Python class that uses a MATLAB class compiled with MATLAB Compiler SDK"""

    # Class member that must be initialized before any PyRectangle object is instantiated
    libHandle = 0

    @staticmethod
    def initializeHandle(handle):
        PyRectangle.libHandle = handle

    # Initialize the point either from an existing Rectangle (not PyRectangle) object,
    # or from the coordinates of any two diagonally opposite corners.
    def __init__(self, rect=None, cornerAX=None, cornerAY=None, cornerBX=None, cornerBY=None):
        if rect:
            if cornerAX or cornerAY or cornerBX or cornerBY:
                raise ValueError(
                    'If rect is specified, no other keyword argument can be specified')
```

```
            else:
                self.data = rect
        else:
            if not cornerAX or not cornerAY or not cornerBX or not cornerBY:
                raise ValueError(
                    'If rect is not specified, the other keyword arguments must be specified')
            else:
                self.data = PyRectangle.libHandle.makeRectangle(
                    PyRectangle.libHandle.makePoint(cornerAX, cornerAY),
                    PyRectangle.libHandle.makePoint(cornerBX, cornerBY))

    def area(self):
        return PyRectangle.libHandle.getRectangleArea(self.data)

    def height(self):
        return PyRectangle.libHandle.getRectangleHeight(self.data)

    def width(self):
        return PyRectangle.libHandle.getRectangleWidth(self.data)

    def getCorners(self):
        return PyRectangle.libHandle.getRectangleCorners(self.data)

if __name__ == "__main__":

    libHandle = initializeLibrary()

    PyPoint.initializeHandle(libHandle)
    PyRectangle.initializeHandle(libHandle)

    print('Initializing module and creating a rectangle with corners (-1, 2) and (5, 10)...')
    rectA = PyRectangle(cornerAX=-1, cornerAY=2, cornerBX=5, cornerBY=10)
    print('Area: {0:.1f}'.format(rectA.area()))
    print('Height: {0:.1f}'.format(rectA.height()))
    print('Width: {0:.1f}'.format(rectA.width()))

    print('Corners:')
    corners = rectA.getCorners()
    for label in ('upperLeft', 'lowerLeft', 'upperRight', 'lowerRight'):
        pypoint = PyPoint(pt=corners[label])
        print('    {0}: {1}'.format(label, pypoint.getCoordinates()))

    print('')
    print('Executing rectangleDemo...')
    # nargout represents the number of output arguments. Its default value
    # is 1. rectangleDemo returns no output, so we need to explicitly set
    # nargout to 0.
    libHandle.rectangleDemo(nargout=0)
```

The `rectangleDriver` application:

a Defines two classes, `PyPoint` and `PyRectangle`
b Creates a handle to the package using `rectangleLib.initialize`
c Passes the package handle to the `PyPoint` and `PyRectangle` classes
d Creates a `PyRectangle` object and prints its area, height, and width
e Saves each corner as a `PyPoint` object and prints its coordinates
f Calls the `rectangleDemo` MATLAB function to create and display data from a different rectangle

5 Open a system command prompt window and navigate to the `rectangleLibpythonPackage` folder that contains your generated package.

6 Install the package using the `python` command.

```
python setup.py install
```

For more details, see “Install and Import MATLAB Compiler SDK Python Packages” on page 1-3.

7 Navigate up one directory and run the `rectangleDriver.py` application.

```
cd ..
python rectangleDriver.py
```

```
Initializing module and creating a rectangle with corners (-1, 2) and (5, 10)...
Area: 48.0
```

```
Height: 8.0
Width: 6.0
Corners:
    upperLeft: (-1.0, 2.0)
    lowerLeft: (-1.0, 10.0)
    upperRight: (5.0, 2.0)
    lowerRight: (5.0, 10.0)

Executing rectangleDemo...
Upper left-hand corner: (0.0, 0.0)
Lower left-hand corner: (0.0, 4.0)
Upper right-hand corner: (3.0, 0.0)
Lower right-hand corner: (3.0, 4.0)
Area: 12.0
Height: 4.0
Width: 3.0
```

Note On macOS, you must use the `mwpython` script instead of `python`. For example, `mwpython rectangleDriver.py`.

The `mwpython` script is located in the *`matlabroot`*`/bin` folder, where *`matlabroot`* is the location of your MATLAB or MATLAB Runtime installation.

See Also

`mwpython` | `compiler.build.pythonPackage`

More About

- “Generate a Python Package and Build a Python Application”

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 Compiler SDK for Python allows you to run compiled Python packages without needing a licensed copy of MATLAB, whereas MATLAB Engine API for Python requires MATLAB.
- 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. To call a built-in MATLAB function using MATLAB Compiler SDK, you can create a wrapper function and include it in the package.

  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 MATLAB Runtime” on page 1-5
- “Get Started with MATLAB Engine API for Python”
- “Start and Stop MATLAB Engine for Python”
- “Call MATLAB Functions from Python”

Data Handling

Pass Data Between MATLAB and Python

Pass Data 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 Python Module” on page 2-4)
`double`, `single`	`float`
Complex (any numeric type)	`complex`
`int8`, `uint8`, `int16`, `uint16`, `int32`	`int`
`uint32`, `int64`, `uint64`	`int`
`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 (such as the `containers.Map` type)	`matlab.object` MATLAB returns a reference to a `matlab.object`, not the object itself. You cannot pass a `matlab.object` between MATLAB sessions.
MATLAB value object (such as the `categorical` type)	Opaque object. You can pass a value object to a MATLAB function, but you cannot create or modify it.
Other object (for example, Java object)	Not supported
Function handle	Not supported
Sparse array	Not supported
String array	Not supported
Structure array	Not supported

Pass Data from Python to MATLAB

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 Python Module” on page 2-4)	Numeric array
`float`	`double`
`complex`	Complex `double`
`int`	`int32`(Windows) `int64`(Linux and Mac)
`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

See Also

Related Examples

- “matlab Python Module” on page 2-4
- “Use MATLAB Arrays in Python” on page 2-10

matlab Python Module

In this section...

The `matlab` Python module provides array classes to represent arrays of MATLAB numeric types as Python variables so that MATLAB arrays can be passed between Python and MATLAB.

MATLAB Classes in the matlab Python Module

- You can use MATLAB numeric arrays in Python code by importing the `matlab` Python package and calling the necessary constructors. For example:

  ```
  import matlab
  a = matlab.double([[1, 2, 3],[4, 5, 6]])
  ```

  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 an array with an optional `initializer` input argument that contains numbers. The `initializer` argument must be a Python sequence type such as a `list`, `tuple`, or `range`. You can specify `initializer` to contain multiple sequences of numbers.
- You can initialize an array with an optional `vector` input argument that contains input of size 1-by-N. If you use `vector`, you cannot use `initializer`.
- You can create a multidimensional array using one of the following options:
  - Specify a nested sequence without specifying the size.
  - Specify a nested sequence and also specify a `size` input argument that matches the dimensions of the nested sequence.
  - Specify a one-dimensional sequence together with a multidimensional size. In this case, the sequence will be assumed to represent the elements in column-major order.
- You can create a MATLAB array of complex numbers by setting the optional `is_complex` keyword argument to `True`.
- You can use custom types for initializing MATLAB arrays in Python. The custom type should implement the Python buffer protocol. One example is `ndarray` in NumPy.

Class from matlab Python Package	Constructor Call in Python
matlab.double	matlab.double(initializer=None|vector=None, size=None, is_complex=False)
matlab.single	matlab.single(initializer=None|vector=None, size=None, is_complex=False)
matlab.int8	matlab.int8(initializer=None|vector=None, size=None, is_complex=False)
matlab.int16	matlab.int16(initializer=None|vector=None, size=None, is_complex=False)
matlab.int32	matlab.int32(initializer=None|vector=None, size=None, is_complex=False)
matlab.int64	matlab.int64(initializer=None|vector=None, size=None, is_complex=False)
matlab.uint8	matlab.uint8(initializer=None|vector=None, size=None, is_complex=False)
matlab.uint16	matlab.uint16(initializer=None|vector=None, size=None, is_complex=False)
matlab.uint32	matlab.uint32(initializer=None|vector=None, size=None, is_complex=False)
matlab.uint64	matlab.uint64(initializer=None|vector=None, size=None, is_complex=False)
matlab.logical	matlab.logical(initializer=None|vector=None, size=None)[a]

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

Properties and Methods of MATLAB Classes in the matlab Python Package

All MATLAB arrays created with `matlab` package constructors have the following properties and methods:

Properties

Property Name	Description	Examples

`size`	A tuple of integers representing the dimensions of an array	`>>> a = matlab.int16([[1, 2, 3],[4, 5,`
		`>>> a.size`
		`(2, 3)`

`itemsize`	An integer representing the size in bytes of an element of the array	`>>> a = matlab.int16()`
		`>>> a.itemsize`
		`2`
		`>>> b = matlab.int32()`
		`>>> b.itemsize`
		`4`

Methods

Method Name	Purpose	Examples

`clone()`	Return a new distinct object with contents identical to the contents of the original object	`>>> a = matlab.int16([[1, 2, 3],[4, 5,`
		`>>> b = a.clone()`
		`>>> print(b)`
		`[[1,2,3],[4,5,6]]`
		`>>> b[0][0] = 100`
		`>>> b matlab.int16([[100,2,3],[4,5,6]])`
		`>>> print(a )`
		`[[1,2,3],[4,5,6]]`

`real()`	Return the real parts of elements that are complex numbers, in column-major order, as a 1-by-N array	`>>> a = matlab.int16([[1 + 10j, 2 + 20j`
		`>>> print(a.real())`
		`[1,4,2,5,3,6]`

`imag()`	Return the imaginary parts of elements that are complex numbers, in column-major order, as a 1-by-N array	`>>> a = matlab.int16([[1 + 10j, 2 + 20j`
		`>>> print(a.imag())`
		`[10,0,20,0,30,0]`

`noncomplex()`	Return elements that are not complex numbers, in column-major order, as a 1-by-N array	`>>> a = matlab.int16([[1, 2, 3],[4, 5,`
		`>>> print(a.noncomplex())`
		`[1,4,2,5,3,6]`

• `reshape(`*`dim1`*`,`*`dim2`*`,...,`*`dimN`*`)`
• `reshape((`*`dim1`*`,`*`dim2`*`,...,`*`dimN`*`))`
• `reshape([`*`dim1`*`,`*`dim2`*`,...,`*`dimN`*`])`
	Reshape the array according to the dimensions and return the result	`>>> a = matlab.int16([[1, 2, 3],[4, 5,`
		`>>> print(a)`
		`[[1,2,3],[4,5,6]]`
		`>>> a.reshape(3, 2)`
		`>>> print(a)`
		`[[1,5],[4,3],[2,6]]`

`toarray()`	Return a standard Python `array.array` object constructed from the contents. Applicable for one-dimensional sequences only.	`>>> a = matlab.int16([[1, 2, 3],[4, 5,`
		`>>> a[0].toarray()`
		`array('h', [1, 2, 3])`
		`>>> b = matlab.int16([[1 + 10j, 2 + 20j`
		`>>> b.real().toarray()`
		`array('h', [1, 4, 2, 5, 3, 6])`

`tomemoryview()`	Return a standard Python `memoryview` object constructed from the contents	`>>> a = matlab.int16([[1, 2, 3],[4, 5,`
		`>>> b = a.tomemoryview()`
		`>>> b.tolist()`
		`[[1, 2, 3], [4, 5, 6]]`
		`>>> b.shape`
		`(2, 3)`

Create a MATLAB Array with N Elements

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)`.

Multidimensional MATLAB Arrays in Python

In Python, you can create multidimensional MATLAB arrays of any numeric type. Use a nested Python list 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]]
```

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. Elements are taken from the original array in column-major order.

```
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]]
```

Use Custom Types to Initialize MATLAB Arrays

You can use custom types such as the `ndarray` in NumPy for initializing MATLAB arrays in Python. The custom type should implement the Python buffer protocol.

```
import matlab
import numpy

nf = numpy.array([[1.1, 2,2, 3.3], [4.4, 5.5, 6.6]])
md = matlab.double(nf)
ni32 = numpy.array([[1, 2, 3], [4, 5, 6]], dtype='int32')
mi32 = matlab.int32(ni32)
```

See Also

Related Examples

- “Use MATLAB Arrays in Python” on page 2-10
- “Pass Data to MATLAB from Python”

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 Python Module” on page 2-4
- “Pass Data to MATLAB from Python”

Functions

compiler.build.pythonPackage

Create Python package for deployment outside MATLAB

Syntax

```
compiler.build.pythonPackage(FunctionFiles)
compiler.build.pythonPackage(FunctionFiles,Name,Value)
compiler.build.pythonPackage(opts)
results = compiler.build.pythonPackage( ___ )
```

Description

`compiler.build.pythonPackage(FunctionFiles)` creates a Python package using the MATLAB functions specified by `FunctionFiles`.

`compiler.build.pythonPackage(FunctionFiles,Name,Value)` creates a Python package with additional options specified using one or more name-value arguments. Options include the package name, output directory, and additional files to include.

`compiler.build.pythonPackage(opts)` creates a Python package with options specified using a `compiler.build.PythonPackageOptions` object `opts`. You cannot specify any other options using name-value arguments.

`results = compiler.build.pythonPackage( ___ )` returns build information as a `compiler.build.Results` object using any of the input argument combinations in previous syntaxes. The build information consists of the build type, paths to the compiled files, and build options.

Examples

Create Python Package Using File

Create a Python package using a function file that generates a magic square.

In MATLAB, locate the MATLAB function that you want to deploy as a Python package. For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Build a Python package using the `compiler.build.pythonPackage` command.

```
compiler.build.pythonPackage(appFile);
```

The build function creates the following files within a folder named `magicsquarepythonPackage` in your current working directory:

- `GettingStarted.html`
- `includedSupportPackages.txt`
- `example`

- `mccExcludedFiles.log`
- `readme.txt`
- `requiredMCRProducts.txt`
- `setup.py`
- `unresolvedSymbols.txt`

Customize Python Package

Create a Python package and customize it using name-value arguments.

For this example, use the files `flames.m` and `flames.mat` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','flames.m');
MATFile = fullfile(matlabroot,'extern','examples','compiler','flames.mat');
```

Build a Python package using the `compiler.build.pythonPackage` command. Use name-value arguments to specify the package name, add a MAT-file, and enable verbose output.

```
compiler.build.pythonPackage(appFile,'PackageName','FlamesApp',...
    'AdditionalFiles',MATFile,...
    'Verbose','on');
```

Create Multiple Python Packages Using Options Object

Create multiple Python packages using a `compiler.build.PythonPackageOptions` object.

For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
```

Create a `PythonPackageOptions` object using `appFile`. Use name-value arguments to specify a common output directory, disable automatic detection of data files, and enable verbose output.

```
opts = compiler.build.PythonPackageOptions(appFile,...
    'OutputDir','D:\Documents\MATLAB\work\PythonPackageBatch',...
    'AutoDetectDataFiles','off',...
    'Verbose','on')
```

```
opts =

  PythonPackageOptions with properties:

           FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler\magicsquare
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: off
          SupportPackages: {'autodetect'}
                  Verbose: on
                OutputDir: 'D:\Documents\MATLAB\work\PythonPackageBatch'
```

Build the Python package using the `PythonPackageOptions` object.

```
compiler.build.pythonPackage(opts);
```

To compile using the function file `hello.m` with the same options, use dot notation to modify the `FunctionFiles` argument of the existing `PythonPackageOptions` object before running the build function again.

```
opts.FunctionFiles = fullfile(matlabroot,'extern','examples','compiler','hello.m');
compiler.build.pythonPackage(opts);
```

By modifying the `FunctionFiles` argument and recompiling, you can compile multiple components using the same options object.

Get Build Information from Python Package

Create a Python package and save information about the build type, generated files, included support packages, and build options to a `compiler.build.Results` object.

Compile using the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
results = compiler.build.pythonPackage('magicsquare.m');
```

```
results =

  Results with properties:

            BuildType: 'pythonPackage'
                Files: {3×1 cell}
IncludedSupportPackages: {}
              Options: [1×1 compiler.build.PythonPackageOptions]
```

The `Files` property contains the paths to the following:

- `example` folder
- `setup.py`
- `GettingStarted.html`

Input Arguments

`FunctionFiles` — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `["myfunc1.m","myfunc2.m"]`

Data Types: `char` | `string` | `cell`

`opts` — Python package build options

`compiler.build.PythonPackageOptions` object

Python package build options, specified as a `compiler.build.PythonPackageOptions` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'Verbose','on'`

`AdditionalFiles` — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the Python package, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: `'AdditionalFiles',["myvars.mat","data.txt"]`

Data Types: `char` | `string` | `cell`

`AutoDetectDataFiles` — Flag to automatically include data files

`'on'` (default) | on/off logical value

Flag to automatically include data files, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then data files that you provide as inputs to certain functions (such as `load` and `fopen`) are automatically included in the package.
- If you set this property to `'off'`, then you must add data files to the package using the `AdditionalFiles` option.

Example: `'AutoDetectDataFiles','off'`

Data Types: `logical`

`PackageName` — Name of Python package

character vector | string scalar

Name of the Python package, specified as a character vector or a string scalar. Specify `'PackageName'` as a namespace, which is a period-separated list, such as `companyname.groupname.component`. The name of the generated package is set to the last entry of the period-separated list. The name must begin with a letter and contain only alphabetic characters and periods.

If not specified, `PackageName` defaults to the name of the first MATLAB file listed in the `FunctionFiles` argument.

Example: `'PackageName','mathworks.pythonpackage.mymagic'`

Data Types: `char` | `string`

`SampleGenerationFiles` — MATLAB sample files

character vector | string scalar | cell array of character vectors | string array

MATLAB sample files used to generate sample Python files for functions included with the package, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `'SampleGenerationFiles',["sample1.m","sample2.m"]`

Data Types: `char` | `string` | `cell`

`OutputDir` — Path to output directory
character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the package name appended with `pythonPackage`.

Example: `'OutputDir','D:\Documents\MATLAB\work\mymagicpythonPackage'`

Data Types: `char` | `string`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

Data Types: `char` | `string` | `cell`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control build verbosity, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to `'off'`, then the command window does not display progress information.

Example: `'Verbose','on'`

Data Types: `logical`

Output Arguments

`results` — Build results
`compiler.build.Results` object

Build results, returned as a `compiler.build.Results` object. The `Results` object contains:

- Build type, which is `'pythonPackage'`
- Paths to the following items:
  - `example` folder
  - `setup.py`
  - `GettingStarted.html`
- A list of included support packages
- Build options, specified as a `PythonPackageOptions` object

Version History

Introduced in R2021a

See Also

`compiler.build.PythonPackageOptions`

compiler.build.PythonPackageOptions

Options for building Python packages

Syntax

```
opts = compiler.build.PythonPackageOptions(FunctionFiles)
opts = compiler.build.PythonPackageOptions(FunctionFiles,Name,Value)
```

Description

`opts = compiler.build.PythonPackageOptions(FunctionFiles)` creates a `PythonPackageOptions` object using MATLAB functions specified by `FunctionFiles`. Use the `PythonPackageOptions` object as an input to the `compiler.build.pythonPackage` function.

`opts = compiler.build.PythonPackageOptions(FunctionFiles,Name,Value)` creates a `PythonPackageOptions` object with options specified using one or more name-value arguments. Options include the package name, output directory, and additional files to include.

Examples

Create Python Package Options Object Using File

Create a `PythonPackageOptions` object using file input.

For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`.

```
appFile = fullfile(matlabroot,'extern','examples','compiler','magicsquare.m');
opts = compiler.build.PythonPackageOptions(appFile)
```

```
opts =

  PythonPackageOptions with properties:

           FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler'}
             PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: on
          SupportPackages: {'autodetect'}
                  Verbose: off
                OutputDir: '.\magicsquarepythonPackage'
```

You can modify the property values of an existing `PythonPackageOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =

  PythonPackageOptions with properties:
```

```
            FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler'}
              PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: on
          SupportPackages: {'autodetect'}
                  Verbose: on
                OutputDir: '.\magicsquarepythonPackage'
```

Use the `PythonPackageOptions` object as an input to the `compiler.build.pythonPackage` function to build a Python package.

```
buildResults = compiler.build.pythonPackage(opts);
```

Customize Python Package Options Object

Create a `PythonPackageOptions` object and customize it using name-value arguments.

For this example, use the file `magicsquare.m` located in *`matlabroot`*`\extern\examples\compiler`. Use name-value arguments to specify the output directory and disable automatic detection of data files.

```
opts = compiler.build.PythonPackageOptions('magicsquare.m',...
    'OutputDir','D:\Documents\MATLAB\work\MagicPythonPackage',...
    'AutoDetectDataFiles','off')
```

```
opts =

  PythonPackageOptions with properties:

            FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler'}
              PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: off
          SupportPackages: {'autodetect'}
                  Verbose: off
                OutputDir: 'D:\Documents\MATLAB\work\MagicPythonPackage'
```

You can modify the property values of an existing `PythonPackageOptions` object using dot notation. For example, enable verbose output.

```
opts.Verbose = 'on'
```

```
opts =

  PythonPackageOptions with properties:

            FunctionFiles: {'C:\Program Files\MATLAB\R2022b\extern\examples\compiler'}
              PackageName: 'example.magicsquare'
    SampleGenerationFiles: {}
          AdditionalFiles: {}
      AutoDetectDataFiles: off
          SupportPackages: {'autodetect'}
                  Verbose: on
                OutputDir: 'D:\Documents\MATLAB\work\MagicPythonPackage'
```

Use the PythonPackageOptions object as an input to the compiler.build.pythonPackage function to build a Python package.

```
buildResults = compiler.build.pythonPackage(opts);
```

Input Arguments

FunctionFiles — Files implementing MATLAB functions

character vector | string scalar | cell array of character vectors | string array

Files implementing MATLAB functions, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. File paths can be relative to the current working directory or absolute. Files must have a .m extension.

Example: ["myfunc1.m","myfunc2.m"]

Data Types: char | string | cell

Name-Value Pair Arguments

Specify optional pairs of arguments as Name1=Value1,...,NameN=ValueN, where Name is the argument name and Value is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose Name *in quotes.*

Example: 'Verbose','on'

AdditionalFiles — Additional files

character vector | string scalar | cell array of character vectors | string array

Additional files and folders to include in the Python package, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute.

Example: 'AdditionalFiles',["myvars.mat","data.txt"]

Data Types: char | string | cell

AutoDetectDataFiles — Flag to automatically include data files

'on' (default) | on/off logical value

Flag to automatically include data files, specified as 'on' or 'off', or as numeric or logical 1 (true) or 0 (false). A value of 'on' is equivalent to true, and 'off' is equivalent to false. Thus, you can use the value of this property as a logical value. The value is stored as an on/off logical value of type matlab.lang.OnOffSwitchState.

- If you set this property to 'on', then data files that you provide as inputs to certain functions (such as load and fopen) are automatically included in the package.
- If you set this property to 'off', then you must add data files to the package using the AdditionalFiles option.

Example: 'AutoDetectDataFiles','off'

Data Types: logical

PackageName — Name of Python package

character vector | string scalar

Name of the Python package, specified as a character vector or a string scalar. Specify `'PackageName'` as a namespace, which is a period-separated list, such as `companyname.groupname.component`. The name of the generated package is set to the last entry of the period-separated list. The name must begin with a letter and contain only alphabetic characters and periods.

If not specified, `PackageName` defaults to the name of the first MATLAB file listed in the `FunctionFiles` argument.

Example: `'PackageName','mathworks.pythonpackage.mymagic'`

Data Types: `char` | `string`

`SampleGenerationFiles` — MATLAB sample files
character vector | string scalar | cell array of character vectors | string array

MATLAB sample files used to generate sample Python files for functions included with the package, specified as a character vector, a string scalar, a string array, or a cell array of character vectors. Paths can be relative to the current working directory or absolute. Files must have a `.m` extension.

Example: `'SampleGenerationFiles',["sample1.m","sample2.m"]`

Data Types: `char` | `string` | `cell`

`OutputDir` — Path to output directory
character vector | string scalar

Path to the output directory where the build files are saved, specified as a character vector or a string scalar. The path can be relative to the current working directory or absolute.

The default name of the build folder is the package name appended with `pythonPackage`.

Example: `'OutputDir','D:\Documents\MATLAB\work\mymagicpythonPackage'`

Data Types: `char` | `string`

`SupportPackages` — Support packages
`'autodetect'` (default) | `'none'` | string scalar | cell array of character vectors | string array

Support packages to include, specified as one of the following options:

- `'autodetect'` (default) — The dependency analysis process detects and includes the required support packages automatically.
- `'none'` — No support packages are included. Using this option can cause runtime errors.
- A string scalar, character vector, or cell array of character vectors — Only the specified support packages are included. To list installed support packages or those used by a specific file, see `compiler.codetools.deployableSupportPackages`.

Example: `'SupportPackages',{'Deep Learning Toolbox Converter for TensorFlow Models','Deep Learning Toolbox Model for Places365-GoogLeNet Network'}`

Data Types: `char` | `string` | `cell`

`Verbose` — Flag to control build verbosity
`'off'` (default) | on/off logical value

Flag to control build verbosity, specified as `'on'` or `'off'`, or as numeric or logical `1` (`true`) or `0` (`false`). A value of `'on'` is equivalent to `true`, and `'off'` is equivalent to `false`. Thus, you can use

the value of this property as a logical value. The value is stored as an on/off logical value of type `matlab.lang.OnOffSwitchState`.

- If you set this property to `'on'`, then the MATLAB command window displays progress information indicating compiler output during the build process.
- If you set this property to `'off'`, then the command window does not display progress information.

Example: `'Verbose','on'`

Data Types: `logical`

Output Arguments

`opts` — Python package build options
`PythonPackageOptions` object

Python package build options, returned as a `PythonPackageOptions` object.

Version History

Introduced in R2021a

See Also

`compiler.build.pythonPackage`

myDeployedModule.initialize

Python module to 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 Packaged MATLAB Function” on page 1-7

myDeployedModule.initialize_runtime

Python module to 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'])
myobj = myDeployedModule.initialize()
```

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

See Also

`myDeployedModule.terminate`

Topics

“Initialize MATLAB Runtime” on page 1-5

myDeployedModule.terminate

Python module to 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 Packaged MATLAB Function” on page 1-7

myDeployedModule.wait_for_figures_to_close

Python module to 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
```

Version History

Introduced in R2015b