Zygo Corporation
Laurel Brook Road
Middlefield, CT 06455
USA

Mx™ Scripting Guide
OMP-0601

This manual covers scripting information specifically related to creating Mx™ scripts.

For information on the Python Programming language, go to:

https://docs.python.org/3/

AMETEK®
ULTRA PRECISION TECHNOLOGIES

Revision Tracking

Revision	Date	Description
A	August 2015	Released document
B	January 2017	Mx™ 7.0 release

ZYGO SOFTWARE LICENSE AGREEMENT

PLEASE READ THIS SOFTWARE LICENSE AGREEMENT CAREFULLY BEFORE DOWNLOADING OR USING THE SOFTWARE.

BY CLICKING ON THE "ACCEPT" BUTTON, OPENING THE PACKAGE, DOWNLOADING THE PRODUCT, OR USING THE EQUIPMENT THAT CONTAINS THIS PRODUCT, YOU ARE CONSENTING TO BE BOUND BY THIS AGREEMENT. IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, CLICK THE "DO NOT ACCEPT" BUTTON AND THE INSTALLATION PROCESS WILL NOT CONTINUE, RETURN THE PRODUCT TO THE PLACE OF PURCHASE FOR A FULL REFUND, OR DO NOT DOWNLOAD THE PRODUCT.

Single User License Grant: Zygo Corporation ("Zygo") and its suppliers grant to Customer ("Customer") a nonexclusive and nontransferable license to use the Zygo software ("Software") solely on a single computer owned or leased by Customer or otherwise embedded in equipment provided by Zygo. Customer may not network this software or use in on more than one computer or computer terminal at the same time.

Customer may make one (1) archival copy of the Software provided Customer affixes to such copy all copyright, confidentiality, and proprietary notices that appear on the original.

EXCEPT AS EXPRESSLY AUTHORIZED ABOVE, CUSTOMER SHALL NOT: COPY, IN WHOLE OR IN PART, SOFTWARE OR DOCUMENTATION; MODIFY THE SOFTWARE; REVERSE COMPILE OR REVERSE ASSEMBLE ALL OR ANY PORTION OF THE SOFTWARE; OR RENT, LEASE, DISTRIBUTE, SELL, OR CREATE DERIVATIVE WORKS OF THE SOFTWARE.

Customer agrees that aspects of the licensed materials, including the specific design and structure of individual programs, constitute trade secrets and/or copyrighted material of Zygo. Customer agrees not to disclose, provide, or otherwise make available such trade secrets or copyrighted material in any form to any third party without the prior written consent of Zygo. Customer agrees to implement reasonable security measures to protect such trade secrets and copyrighted material. Title to Software and documentation shall remain solely with Zygo.

LIMITED WARRANTY. Zygo warrants that for a period of ninety (90) days from the date of shipment from Zygo: (i) the media on which the Software is furnished will be free of defects in materials and workmanship under normal use; and (ii) the Software substantially conforms to its published specifications. Except for the foregoing, the Software is provided AS IS. This limited warranty extends only to Customer as the original licensee. Customer's exclusive remedy and the entire liability of Zygo and its suppliers under this limited warranty will be, at Zygo or its service center's option, repair, replacement, or refund of the Software if reported (or, upon request, returned) to the party supplying the Software to Customer. In no event does Zygo warrant that the Software is error free or that Customer will be able to operate the Software without problems or interruptions.

This warranty does not apply if the software (a) has been altered, except by Zygo, (b) has not been installed, operated, repaired, or maintained in accordance with instructions supplied by Zygo, (c) has been subjected to abnormal physical or electrical stress, misuse, negligence, or accident, or (d) is used in hazardous activities.

DISCLAIMER. EXCEPT AS SPECIFIED IN THIS WARRANTY, ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS, AND WARRANTIES INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT OR ARISING FROM A COURSE OF DEALING, USAGE, OR TRADE PRACTICE, ARE HEREBY EXCLUDED TO THE EXTENT ALLOWED BY APPLICABLE LAW.

IN NO EVENT WILL ZYGO OR ITS SUPPLIERS BE LIABLE FOR ANY LOST REVENUE, PROFIT, OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL, OR PUNITIVE DAMAGES HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE EVEN IF ZYGO OR ITS SUPPLIERS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. In no event shall Zygo's or its suppliers' liability to Customer, whether in contract, tort (including negligence), or otherwise, exceed the price paid by Customer. The foregoing limitations shall apply even if the above-stated warranty fails of its essential purpose. SOME STATES DO NOT ALLOW LIMITATION OR EXCLUSION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES.

The above warranty DOES NOT apply to any beta software, any software made available for testing or demonstration purposes, any temporary software modules or any software for which Zygo does not receive a license fee. All such software products are provided AS IS without any warranty whatsoever.

This License is effective until terminated. Customer may terminate this License at any time by destroying all copies of Software including any documentation. This License will terminate immediately without notice from Zygo if Customer fails to comply with any provision of this License. Upon termination, Customer must destroy all copies of Software.

Software, including technical data, is subject to U.S. export control laws, including the U.S. Export Administration Act and its associated regulations, and may be subject to export or import regulations in other countries. Customer agrees to comply strictly with all such regulations and acknowledges that it has the responsibility to obtain licenses to export, re-export, or import Software.

This License shall be governed by and construed in accordance with the laws of the State of Connecticut, United States of America, as if performed wholly within the state and without giving effect to the principles of conflict of law. If any portion hereof is found to be void or unenforceable, the remaining provisions of this License shall remain in full force and effect. This License constitutes the entire License between the parties with respect to the use of the Software.

Restricted Rights - ZYGO's software is provided to non-DOD agencies with RESTRICTED RIGHTS and its supporting documentation is provided with LIMITED RIGHTS. Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph "C" of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19. In the event the sale is to a DOD agency, the government's rights in software, supporting documentation, and technical data are governed by the restrictions in the Technical Data Commercial Items clause at DFARS 252.227-7015 and DFARS 227.7202. Manufacturer is Zygo Corporation, Laurel Brook Rd., Middlefield, CT 06455. 201408

Table of Contents

Important—Before You Begin

This version of Python includes official Python organization libraries and modules. In addition, there are modules developed specifically for interaction with ZYGO's Mx™ software.

There is a wide variety of third party modules and libraries that have been created and made available by Python users. **DO NOT use these third party modules or libraries when creating scripts to be used with Mx software.** To ensure stable operation of the Mx software, use only the libraries and modules provided with this installation.

Overview

Zygo Corporation provides a Python 3 package, zygo, with all installations of Mx 6.3.0.0 and later. This document describes the functionality contained within this package. See the sample scripts provided in the Appendix for examples on how to get started with scripting.

Requirements

- Mx™ software versions 6.3.0.0 and above are installed with Python 3.4.x. Any other version of Python, that is one that was on the computer prior to the Mx installation/upgrade, must be uninstalled before continuing.
- Mx software 6.3.0.0 or above is installed.
- Basic understanding of scripting/programming and some mid-level experience writing scripts in Python.
- Python 3.4.x scripts cannot be used interchangeably with previous versions of Python, e.g. 2.7.
- Review of the Python Style Guide. Available at the following location: https://www.python.org/dev/peps/pep-0008/
- Strongly recommended is a review of the official Python documentation. Available at the following location: https://docs.python.org/3/

Python 3

Python 3.4.3 is installed as part of the Mx software installation process. It is important that the location and availability of the Python executable and standard libraries not change. Previous versions of Python which are not in the 3.4 branch (e.g., 2.7, 3.3) will not interfere with Mx scripting if they exist on the system. However, Mx does require its own installation of Python 3.4.3 for scripting to function.

Since Mx provides a standard installation of CPython, including the default installation's standard libraries, most of the built-in functionality of Python is available, including:

- Math functions
- Flow control
- Build-in data structures
- Functions and Classes

The interactive Python console/REPL (including IDLE) will be installed, but cannot be used for Mx™ scripting.

Mx™ 7.0 Scripting Enhancements

A number of improvements to Mx™ scripting have been made available in this release, including:

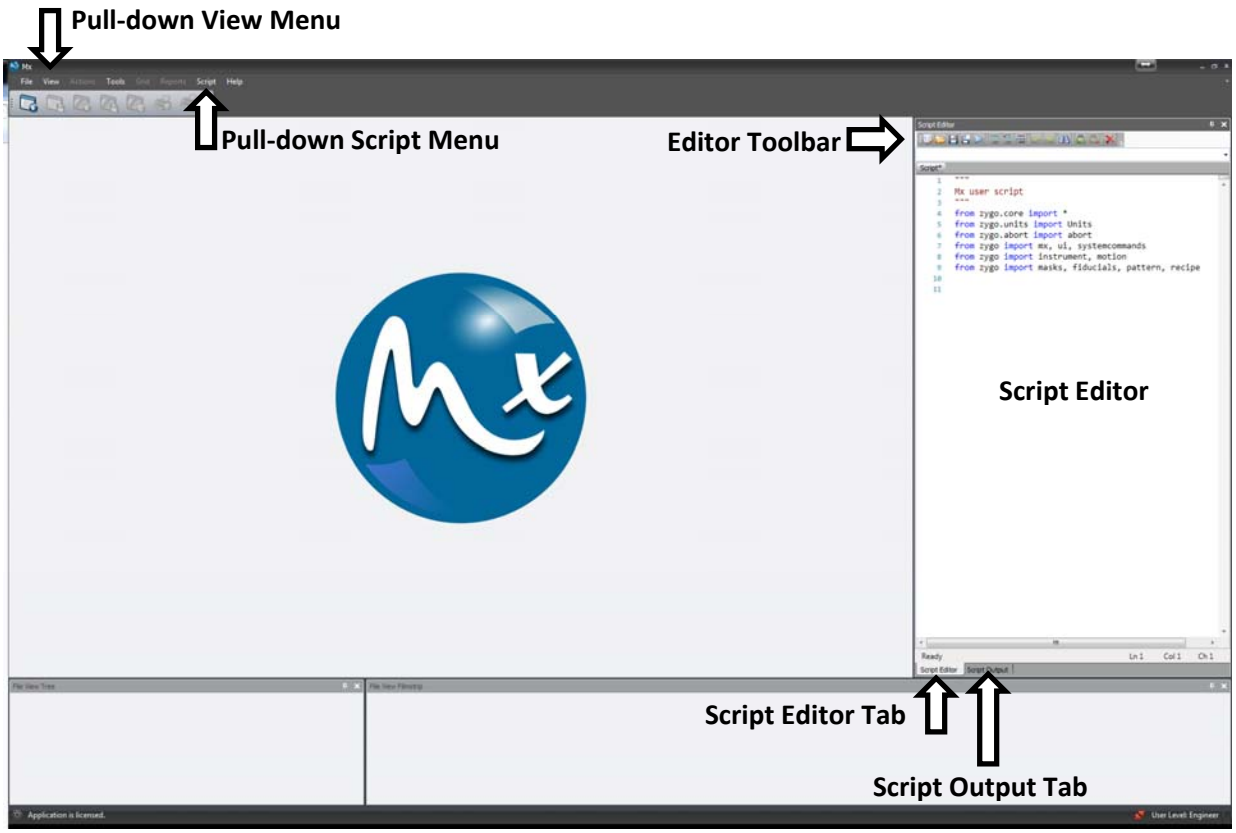
- In a script, the `__name__` attribute of a class, function, method, descriptor, or generator instance will contain its appropriate value. For top-level scripts, this will be `'__main__'`, as expected.
- The built-in `input()` method is available for keyboard input in the Mx™ Script Output window.
- A top-level script can be executed in "debug mode", enabling interactive console-based script debugging in the Script Output window. See "Debugging Techniques" in *Appendix A* of this manual for details.

Debugging Scripts

This guide also includes information for debugging scripts in Mx software. Refer to *Appendix D*.

Getting Started with the Script Editor

The Script Editor and Script Output windows can be accessed through the pull-down View or Script menus. Note that the Script menu is only available when no Mx™ application is open. Once opened, the scripting windows will be docked on the upper right-hand side of the Mx window. The layout of these windows can be adjusted. The image below shows the Script Editor and Script Output windows as a tabbed group.



Editing Scripts

Python scripts can be edited directly inside Mx from within the built-in Script Editor. The editor provides basic syntax highlighting, code formatting, and auto-completion. The editor can create new scripts, and open and edit scripts created both from within and outside Mx. Scripts can be run directly from within the editor.

With Python 3, whitespace must be consistent, and spaces and tabs cannot be mixed. The Mx script editor will convert each tab in the source file to four (4) spaces to conform to Python's Style Guide. Source should always use UTF-8 encoding. This is enforced by the built-in editor, but if an external editor is used, it may be necessary to force that editor to switch to UTF-8 mode by including the following declaration as the first line of each python source (.py) file:

```
# -*- coding: utf-8 -*-
```

Running Scripts

As mentioned in the previous section, scripts can be run directly from within the Mx script editor by clicking the Run icon in the editor's toolbar. Scripts run from the editor must exist on disk prior to execution, and there can be no pending changes. The editor will prompt to save if the script has not yet been saved. *Pending changes to an existing file will be automatically saved when the Run button is pressed.*

Mx also provides a Scripting toolbar which includes the ability to add customized Run Script buttons to the Mx UI (note: customized buttons require an application to be loaded in Mx). In the Scripting toolbar, the "Add Script Button" button will add a new "Run Script" button to the Scripting toolbar. Clicking the dropdown arrow associated with a Run Script button will display a popup menu which contains configuration options for the button, including associating the button with a given script. Configuring the button in this way applies to all copies of a particular button (i.e., if multiple instances of the button are added to the UI through toolbar customization, each instance shares the same settings). Customization of an individual instance of the button can be done through the button's context menu while in toolbar customization mode. Left-clicking a Run Script button will run the button's associated script.

Mx offers several other ways to run a script. A new Auto Sequence operation, "Run Script", can be selected which allows the user to specify a script to be executed each sequence iteration. A "Run Script" recipe step enables a script to be run as part of a recipe. A "Run Script" processing sequence step adds the ability to insert a script as part of a sequence. Scripts can be run as Pattern or Stitch operations by selecting the "Run Script" operation in the appropriate tool panel.

Note that while it is possible to edit scripts externally, scripts which communicate with Mx must be executed from within Mx using one of the methods described above.

zygo Package

The `zygo` package contains several Python modules. Most of the modules communicate directly with Mx, whereas some only provide convenience data models, methods, etc., that are useful in writing Mx scripts. The `zygo` package is automatically visible from within the Mx script editor. All that is necessary to use a given module's functionality within the current script is to import the required module. Creating a new script from within the Mx script editor automatically adds calls to import all the modules in the `zygo` package. However, one may choose to remove unused imports or change the method used to import for convenience.

By convention, methods which begin with a single or double underscore (e.g., `mx._method2()`, `mx.__method2()`) are intended for use by the internal implementation, and should not be called from a script directly. These methods are subject to change without notice.

Importing Modules

Modules in this package can be imported into Mx scripts in the full variety of ways that any Python 3 package can be imported. Some examples are listed below:

- `from zygo import core`
  - Allows all public functionality within `core` module to be accessible via the `core` namespace, e.g., `core.SomeModuleAttribute`
- `import zygo.core`
  - Allows all public functionality within `core` module to be accessible from the `zygo.core` namespace, e.g., `zygo.core.SomeModuleAttribute`
- `import zygo.core as c`
  - Allows all public functionality within `core` module to be accessible from the `c` namespace, e.g., `c.SomeModuleAttribute`
  - `c` is an *alias* for `zygo.core`
  - This is the recommended method, especially for long module names
- `from zygo.core import *`
  - Allows all public functionality within `core` module to be accessible from the current module (current script) namespace, e.g., `SomeModuleAttribute`
  - Note that this method is generally discouraged for Python due to potential name conflicts. It also significantly limits the effectiveness of the Mx script editor's IntelliSense by requiring additional code to get a reference to the current module as well as resulting in a much longer list of available methods/attributes of the current module namespace.

core Module

The core module provides utility functionality shared by other Mx scripting modules. Much of its functionality does not communicate directly with Mx, but provides basic classes and methods that are useful in Mx scripts.

ZygoError Class

The ZygoError class is a Python Exception that originates from Mx. Not all calls to methods within the zygo package that error out will raise a ZygoError exception. For example, some calls can result in other Python exceptions such as TypeError or ValueError. However, all errors that Mx sends back to the Python script will be of type ZygoError. It has no additional attributes outside of Python's Exception.

```
class ZygoError(Exception)
    Exception generated from Mx or from the connection to Mx
```

Point2D Class

An (x, y) coordinate for representing some Mx object's location (e.g. mask, fiducial) is encapsulated by a Point2D object.

```
class Point2D(object)
    Represent (x,y) coordinate

    __init__(x, y)
        Initialize 2-dimensional point

        Parameters: • x: X-coordinate
                    • y: Y-coordinate
```

ZygoTask Class

The ZygoTask class encapsulates information pertaining to an asynchronous Mx operation. For some potentially long-running Mx calls, the script user can choose to call the method with a wait flag set to False, e.g.:

```
motion.move_x(2.5, units.MilliMeters, wait=False)
```

It is then up to the user to store and use as needed the ZygoTask result returned from the call. This ZygoTask object will then allow the user to check whether the task is done, ask to wait with a timeout, or ask for the result (if any) from the finished operation. It is not expected for the user to ever need to create a ZygoTask object directly. However, the user can utilize one returned from a non-blocking call if desired.

```
class ZygoTask(object)
    Represent information pertaining to an asynchronous Mx operation

    __init__(task_id, done_func, wait_func)
        Initialize task
```

```
        Parameters: • task_id: Unique task identifier
                    • done_func: Function to call for 'done' property - Takes task_id
                    • wait_func: Function to call for 'wait' method Takes task_id,
                                 timeout

    wait(timeout=None)
        Wait for operation to complete

        Parameters: • timeout: Maximum time to wait; None for infinite

    result(timeout=None)
        Return result from operation after waiting to complete

        Parameters: • timeout: Maximum time to wait; None for infinite

    done
        True if operation done; False otherwise
```

units Module

The units module contains an enumeration of all Mx-supported units of measurement.

```
class Units(enum.Enum)
    Enumeration of Mx-supported units

    NoUnits

    # Linear
    Angstroms
    CentiMeters
    Feet
    Inches
    Meters
    MicroInches
    MicroMeters
    MilliMeters
    NanoInches
    NanoMeters

    # etc.
```

Scripting includes support for more than 400 units. For a complete listing of the units defined in the units.Units enum, refer to the script editor's auto-complete feature.

mx Module

The mx module provides basic high-level Mx functionality. It only has module-level functions.

Application Methods

The mx module provides a few methods for handling applications. It can check to see if an application is open, get the path of the current open application, open an application given a valid file path, close the current application, and save the current application to a given filename. Note that any unsaved changes made to an application will be discarded when closing an app or opening a new app via a script.

```
is_application_open()
    Get value indicating if an Mx application is open

    Returns:    True if an application is open, False otherwise

get_application_path()
    Retrieves the full path of the current application

    Returns:    The full path of the current application if open; null otherwise

open_application(filename)
    Open requested Mx application

    Parameters: • filename: File name to load from

close_application()
    Close current Mx application

save_application_as(filename)
    Save current open Mx application as specified filename

    Parameters: • filename: File name to save to
```

Settings Methods

The mx module provides methods for handling settings. It can load settings given a valid filename and save the current settings to a given filename.

```
load_settings(filename)
    Loads Mx settings

    Parameters: • filename: File name to load from

save_settings(filename)
    Save Mx settings

    Parameters: • filename: File name to save to
```

Data Alignment Scaling Mode

The DataAlignmentScalingMode enumeration defines the available data alignment scaling modes used when averaging data.

```
class DataAlignmentScalingMode(IntEnum)
```

```
    Data alignment scaling modes

    isomorphic
    anamorphic
```

Fiducial Alignment Type

The `FiducialAlignmentType` enumeration defines the available fiducial alignment types used when subtracting data.

```
class FiducialAlignmentType(IntEnum)
    Alignment types used for fiducial alignment

    fixed
    variable
```

Data Methods

The `mx` module provides several methods for dealing with data. It can analyze the data. It can load and save data from/to a given filename. Likewise, it can do the same for signal data. It can also reset the current data.

```
analyze()
    Analyze current data

auto_save_data(update_sequence)
    Save the current data using the values in the AutoSequence AutoSaveData controls

    Parameters: • update_sequence: If true, increments any filename sequence values
    Returns:    The name of the file saved; None if not saved

load_data(filename)
    Load Mx data from specified filename

    Parameters: • filename: File name to load from

save_data(filename)
    Save Mx data to specified filename

    Parameters: • filename: File name to save to

load_signal_data(filename)
    Load Mx signal data from specified filename

    Parameters: • filename: File name to load from

save_signal_data(filename)
    Save Mx signal data to specified filename

    Parameters: • filename: File name to save to

load_and_average_data(file_pathnames, min_valid_pct, use_fiducial_alignment=False,
                      scaling_mode=DataAlignmentScalingMode.isomorphic)
    Load and average the specified data files

    Parameters: • file_pathnames: List of data filenames to load from
                • min_valid_pct: The minimum valid percent at a pixel location in the
                                 data matrix when averaging
```

```
                • use_fiducial_alignment: Whether to use fiducial alignment to align
                                          the data when averaging
                • scaling_mode: The DataAlignmentScalingMode type of scaling to use to
                                align the data; only used when using fiducial
                                alignment

subtract_data(filename, ignore_lateral_res=True, use_input_size=False,
              use_system_size=False, use_fiducial_alignment=False,
              alignment_type=FiducialAlignmentType.fixed, alignment_tolerance=1.0)
    Subtracts the given file from the current data

    Parameters: • filename: The fully qualified path of the file to subtract
                • ignore_lateral_res: Whether to ignore lateral resolution
                • use_input_size: Whether to use the input matrix data size
                • use_system_size: Whether to use the system reference data size
                • use_fiducial_alignment: Whether to use fiducial alignment
                • alignment_type: The alignment type; only respected when using
                                  fiducial alignment
                • alignment_tolerance: The alignment tolerance in pixels; only
                                       respected when using fiducial alignment

reset_data()
    Resets current data
```

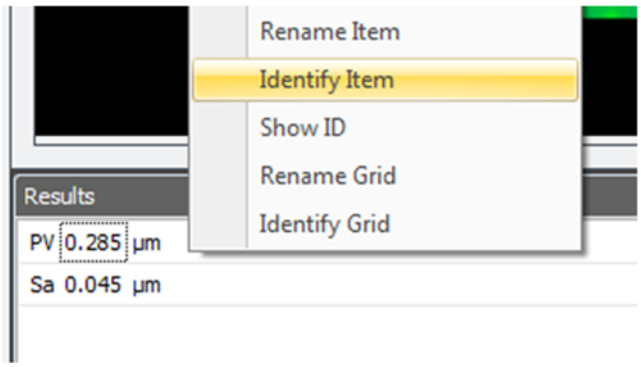
Results, Attributes, and Controls Methods

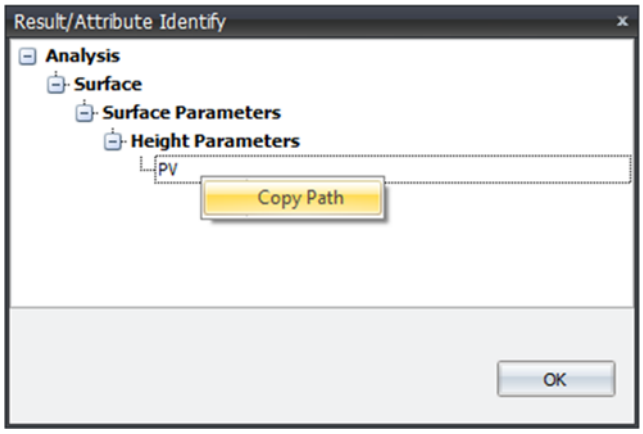
The `mx` module provides several methods for dealing with results, attributes, and controls. It can get and set numbers with specified units, where the units are of the `units.Units` type. It can get and set string and Boolean values. Selection controls can be set by string, specifying the value as the (case-sensitive) desired control string.

The path to a particular result, control, or attribute to get/set the value of is represented as a tuple of strings. For example:

- `('Analysis', 'Surface', 'Surface Parameters', 'Height Parameters', 'PV')`
- `('Analysis', 'Surface', 'Areal ISO Parameters', 'Height Parameters', 'Sa')`
- `('Analysis', 'Surface', 'RMS')`

This path can be identified by selecting the Identify option in the context menu of the specified item in a grid. There is an option to copy the path to the clipboard by right-clicking the item in the Identify dialog and selecting the Copy option.





Custom results must be created by the Mx Custom Results tool prior to being accessed in a script. The procedure is as follows:

- Select the “Tools > Custom Results” option from the Mx main menu.
- Leave the “Results and Attributes” section empty.
- Click the “New” button in the “Custom Results” section.
- Enter a unique Name, a Return Type, and a Unit Category for your custom result.
- Leave the expression editor empty and click “OK”.
- The custom result can now be accessed by path rooted at `('Analysis', 'Custom')`.

```
get_attribute_number(path, unit)
    Get an attribute value as a number at requested path with requested unit

    Parameters: • path: Path to attribute
                • unit: Units as a Units type or string for return value
    Returns:    Value at path in units

get_attribute_string(path)
    Get an attribute value as a string at requested path

    Parameters: • path: Path to attribute
    Returns:    Value at path

get_control_number(path, unit)
    Get a control value as a number at requested path with requested unit

    Parameters: • path: Path to control
                • unit: Units as a Units type or string for return value
    Returns:    Value at path in units

get_control_string(path)
    Get a control value as a string at requested path

    Parameters: • path: Path to control
    Returns:    Value at path

get_control_bool(path)
    Get a control value as a boolean at requested path

    Parameters: • path: Path to control
    Returns:    Value at path
```

```
get_result_number(path, unit)
    Get a result value as a number at requested path with requested unit

    Parameters: • path: Path to result
                • unit: Units as a Units type or string for return value
    Returns:    Value at path in units

get_result_string(path)
    Get a result value as a string at requested path

    Parameters: • path: Path to result
    Returns:    Value at path

get_result_bool(path)
    Get a custom result value as a boolean at requested path

    Parameters: • path: Path to result
    Returns:    Value at path

set_control_number(path, value, unit)
    Set a control value as a number at requested path with requested unit

    Parameters: • path: Path to control
                • value: Value to set at path
                • unit: Units as a Units type or string for value

set_control_string(path, value)
    Set a control value as a string at requested path

    Parameters: • path: Path to control
                • value: Value to set at path

set_control_bool(path, value)
    Set a control value as a boolean at requested path

    Parameters: • path: Path to control
                • value: Value to set at path

set_result_number(path, value, unit)
    Set a custom result value as a number at requested path with requested unit

    Parameters: • path: Path to custom result
                • value: Value to set at path
                • unit: Units as a Units type or string for value

set_result_string(path, value)
    Set a custom result value as a string at requested path

    Parameters: • path: Path to custom result
                • value: Value to set at path

set_result_bool(path, value)
    Set a custom result value as a boolean at requested path

    Parameters: • path: Path to custom result
                • value: Value to set at path
```

Other Results Methods

The `mx` module provides several other methods for dealing with results. It can clear and store current process statistics, clear a custom result, get tolerance state, and initiate Log Reports.

```
clear_process_stats()
    Clear process stats

store_process_stats()
    Store process stats

clear_custom_result(path)
    Clears the custom result

    Parameters: • path: Path to custom result

get_tolerance_pass_fail()
    Get the tolerance pass/fail state

is_tolerance_enabled()
    Gets whether or not the tolerance tool is enabled

log_reports()
    Causes Log Reports to be run if configured
```

Data Matrix Methods

The `mx` module provides several methods for dealing with the position and dimensions of an Mx data matrix (map/plot). The data matrix is specified by supplying the `ui.Control` object of a particular overlay. The data matrix corresponds to the currently visible plot in this overlay. The units are a `units.Units` value.

```
get_data_center_x(control, unit)
    Retrieves the x-coordinate of the geometric center of the control's plot

    Parameters: • control: A Control instance
                • unit: Units as a Units type or string for the value

get_data_center_y(control, unit)
    Retrieves the y-coordinate of the geometric center of the control's plot

    Parameters: • control: A Control instance
                • unit: Units as a Units type or string for the value

get_data_origin_x(control, unit)
    Retrieves the x-coordinate of the geometric origin of the control's plot

    Parameters: • control: A Control instance
                • unit: Units as a Units type or string for the value

get_data_origin_y(control, unit)
    Retrieves the y-coordinate of the geometric origin of the control's plot

    Parameters: • control: A Control instance
                • unit: Units as a Units type or string for the value

get_data_size_x(control, unit)
    Retrieves the x-dimension of the control's plot
```

```
    Parameters: • control: A Control instance
                • unit: Units as a Units type or string for the value


get_data_size_y(control, unit)
    Retrieves the y-dimension of the control's plot

    Parameters: • control: A Control instance
                • unit: Units as a Units type or string for the value
```

Annotations Grid Methods

The mx module provides several methods for dealing with Annotation grids in Mx. Annotations can be created, modified, deleted, and retrieved.

```
create_annotation(label, value)
    Create a new Annotation in Mx

    Parameters: • label: The label/name of the new annotation
                • value: The new string value

delete_annotation(path)
    Delete an annotation in Mx

    Parameters: • path: The label/name of the new annotation

set_annotation(path, value)
    Modify an existing Mx Annotation

    Parameters: • path: The path of the annotation
                • value: The new string value

get_annotation(path)
    Gets the string value associated with a given Mx annotation

    Parameters: • path: The path to the annotation
```

Logging Methods

The mx module provides several methods for writing to the system log/Event Viewer.

```
log_info(message)
    Logs a message with the Info level.

    Parameters: • message: The message to write to the system log

log_warn(message)
    Logs a message with the Warn level.

    Parameters: • message: The message to write to the system log

log_error(message)
    Logs a message with the Error level.

    Parameters: • message: The message to write to the system log

log_fatal(message)
```

```
    Logs a message with the Fatal level.

    Parameters: • message: The message to write to the system log
```

Miscellaneous Methods

Several other methods are provided by the mx module to perform various other tasks.

```
clear_script_console()
    Clears the Mx scripting console/output window

get_mx_version()
    Gets the Mx version number as a string
```

instrument Module

The instrument module provides functionality for requesting operations that require a tool. It has two IntEnum classes to define Align-View and Ring-Spot modes. The module defines a class pertaining to asynchronous acquisition operations. The rest of its contents are strictly module-level functions.

Align/View Mode

The AlignViewMode enumeration defines the available Align/View modes.

```
class AlignViewMode(IntEnum)
    Alignment mode of host instrument

    none  # Unknown or invalid
    align
    view
```

The instrument module has two module-level methods to get and set the align/view mode.

```
get_align_view_mode()
    Get align/view mode on host instrument

    Returns:    AlignViewMode

set_align_view_mode(mode)
    Set align/view mode on host instrument

    Parameters: • mode: AlignViewMode to set to
```

Ring/Spot Mode

The RingSpotMode enumeration defines the available Ring/Spot modes.

```
class RingSpotMode(IntEnum)
    Ring spot controller mode of host instrument

    none  # Unknown or invalid
    ring
    spot
```

The instrument module has two module-level methods to get and set the ring/spot mode.

```
get_ring_spot_mode():
    Get ring spot mode on host instrument

    Returns:    RingSpotMode

set_ring_spot_mode(mode):
    Set ring spot mode on host instrument

    Parameters: • mode: RingSpotMode to set to
```

AcquisitionTask Class

The `instrument` module defines the `AcquisitionTask` class which encapsulates information pertaining to acquisition operations. An `AcquisitionTask` object contains three `ZygoTask` objects, one for each of the three events: frame grab complete, acquisition complete, and measure complete. Note that `measure_task` is not valid for the `acquire()` operation.

```
class AcquisitionTask(object)
    Represent information pertaining to an asynchronous acquisition operation

    frame_grab_task
        Gets the task that checks for frame grab complete

    acquire_task
        Gets the task that checks for acquire complete

    measure_task
        Gets the task that checks for measure complete
```

Acquisition Methods

The `instrument` module has a few methods for acquiring data on the host instrument. It can acquire data with or without analysis. Acquisition can be performed synchronously or asynchronously by setting the `wait` parameter appropriately. These methods return an `AcquisitionTask` object.

```
acquire(wait=True)
    Acquire data on host instrument

    Parameters: • wait: True to wait for acquisition to complete
    Returns:    AcquisitionTask object for asynchronous acquire

measure(wait=True)
    Measure data on host instrument

    Parameters: • wait: True to wait for measurement to complete
    Returns:    AcquisitionTask object for asynchronous measure
```

Optimization Methods

The `instrument` module has several methods for optimizing the host instrument. It can auto optimize the tilt, focus, light-level, etc. It can also perform auto lateral calibration and auto center, provided the operation's prerequisites are met.

```
auto_focus()
    Perform auto focus on host instrument

auto_tilt()
    Perform auto tilt on host instrument

auto_focus_tilt()
    Perform auto focus and then auto tilt on host instrument

auto_light_level()
    Perform auto light level on host instrument

auto_lat_cal(value, unit)
```

```
    Perform auto lateral calibration

    Parameters: • value: Numeric value of the size of the calibration artifact
                • unit: Units corresponding to the value parameter

auto_center()
    Performs an auto center acquisition if available
```

Turret Methods

The `instrument` module has a couple of methods for dealing with the turret. It can get the current position and request a move to a specified position.

```
get_turret()
    Return current turret position on host instrument

    Returns:    Current turret possition as integer

move_turret(position)
    Move turret to specified position on host instrument

    Parameters: • position: Target turret position as integer
```

Zoom Methods

The `instrument` module has several methods for handling the zoom. It can get and set the current zoom, get the minimum and maximum zooms, lock and unlock the zoom.

```
get_zoom()
    Get current zoom value on host instrument

    Returns:    The current zoom value

set_zoom(zoom)
    Set zoom to specified value on host instrument

    Parameters: • zoom: Target zoom value

get_min_zoom()
    Get minimum zoom value allowable on host instrument

    Returns:    The minimum allowable zoom value

get_max_zoom()
    Get maximum zoom value allowable on host instrument

    Returns:    The maximum allowable zoom value

lock_zoom()
    Lock zoom on host instrument

unlock_zoom()
    Unlock zoom on host instrument
```

Light Level Methods

The `instrument` module can get and set the current light level on the host instrument.

```
get_light_level()
    Return current light level on host instrument

    Returns:    The current light level as a percentage

set_light_level(light_level)
    Set light to specified level on host instrument

    Parameters: • light_level: Target light level as a percentage
```

Wand Status Method

The `instrument` module can query, enable, and disable the wand on the host instrument.

```
is_wand_enabled()
    Gets whether or not the wand is enabled

    Returns:    True if the wand is enabled; False otherwise

set_wand_enabled(enabled)
    Enable/Disable wand on host instrument

    Parameters: • enabled: True to enable wand; False otherwise
```

Camera Information Methods

The `instrument` module can get certain details about the instrument's camera.

```
get_cam_res(unit)
    Gets the camera resolution of the active instrument, else the data resolution

    Parameters: • unit: Desired Units for the return value

get_cam_size_x(unit)
    Gets the X camera dimension resolution of the active instrument,
    else the X dimension of the surface data

    Parameters: • unit: Desired Units for the return value

get_cam_size_y(unit)
    Gets the Y camera dimension resolution of the active instrument,
    else the Y dimension of the surface data

    Parameters: • unit: Desired Units for the return value
```

Instrument Hardware Methods

The `instrument` module can retrieve instrument hardware information, and enable or disable the sleep mode of the host instrument. By default, sleep mode is disabled when a script is run.

```
get_system_serial_number()
    Retrieves the system serial number

    Returns:    The system serial number as a string; empty string if no serial number
```

```
                found

get_system_type()
    Returns a string representing the current system type

    Returns:    The string representation of the current system type

set_sleep_mode_enabled(enabled)
    Enable/Disable sleep mode on host instrument

    Parameters: • enabled: True to enable sleep mode; False otherwise
```

masks Module

The masks module provides functionality for loading/saving, retrieving information on, and manipulating masks. It contains two inner classes: Mask and Masks. These inner classes are described below.

Mask Class

The Mask class represents a single mask. It can be one of a variety of types, e.g. Acquisition, Surface. It can be manipulated by moving, rotating, and resizing. Its properties consist of its type, location, and size. A mask is not created directly from a script, but is instead retrieved from its Masks object container.

```
class Mask(object)
    Represent one Mx mask

    move_absolute(self, x, y)
        Move center of mask to specified absolute x, y position

        Parameters: • x: New mask center x-coordinate
                    • y: New mask center y-coordinate

    move_relative(self, x, y)
        Move center relative to current position by specified x, y amount

        Parameters: • x: X-offset to move center
                    • y: Y-offset to move center

    resize(self, height, width)
        Resize mask to specified height, width

        Parameters: • height: New mask height
                    • width: New mask width

    rotate(self, value, unit)
        Rotate mask counterclockwise by specified angle

        Parameters: • value: Numeric value to rotate mask by
                    • unit: Units corresponding to the value parameter

    center
        Return mask center X,Y coordinate

    height
        Return mask height

    width
        Return mask width

    type
        Return mask type
```

Masks Class

The Masks class represents the entire group of masks. When created, a Masks object is synchronized to the currently loaded group of masks in Mx and, thus, to the Mx Mask editor whether it is showing or not.

A Masks object can be created by calling its parameter-less initializer. A Masks object can be saved to and loaded from a file. The total number of masks or the number of masks of a given type can be retrieved. All of the masks can be cleared. A particular Mask object can be retrieved that is closest to a given location that is specified. It can also discriminate based on its type. A particular Mask object can be removed from the Masks object, resulting in the mask being removed from the Mask editor. This is accomplished by finding the Mask object of interest and calling the delete method of the Masks object. The Masks class is actually a container class. It can be iterated over to examine and/or manipulate all of the individual Mask objects. See the example below.

```
# Move, resize, rotate masks
masks = Masks()
for m in masks:
    m.move_relative(30.0, -10.0)
    m.resize(m.height + 30.0, m.width + 30.0)
    m.rotate(45, Units.Degrees)
    print('center = ({0.x:.2f}, {0.y:.2f})'.format(m.center), end=', ')
    print('height = {0.height:.2f}, width = {0.width:.2f}'.format(m))
```

```
class Masks(object)
    Represent a collection of Mx masks

    get_num_masks(self, mask_type=None)
        Return number of masks of specified type

        Parameters: • mask_type: Mask type (None for any type)
        Returns:    Number of masks of given type

    save(self, filename)
        Save masks to specified file

        Parameters: • filename: File name to save to

    load(self, filename)
        Load masks from specified file

        Parameters: • filename: File name to load from

    delete(self, mask)
        Remove specified mask from the collection of Mx masks

        Parameters: • mask: Mask object to remove

    clear(self, mask_type=None)
        Clear all masks of specified type

        Parameters: • mask_type: Mask type as string (None or '' for any type),
                                  e.g. Acquisition, Surface

    get_mask_closest_to(self, x, y, mask_type=None)
```

```
        Get mask of specified type closest to specified center coordinates

        Parameters: • x: X-coordinate
                    • y: Y-coordinate
                    • mask_type: Mask type as string (None or '' for any type),
                                 e.g. Acquisition, Surface
        Returns:    Mask
```

motion Module

The motion module provides functionality for communicating with the host instrument's stage. It provides an enumeration to represent stage axes. It provides methods for homing axes, retrieving current positions for axes, absolute moves, availability status, wait for move completions, and to set the pendant status.

AxisType Class

The AxisType IntEnum contains an enumeration of all Mx-supported standard stage axes. The rx, ry, and rz AxisTypes represent the pitch, roll, and theta stages respectively.

```
class AxisType(IntEnum)
    Axis name

    unknown
    x
    y
    z
    rx  # Pitch, rotation about x-axis
    ry  # Roll, rotation about y-axis
    rz  # Theta, rotation about z-axis
```

Home Axes

The motion module provides methods to home motorized axes. They take a single parameter, a wait flag that defaults to True. If its value is False, then the user should retrieve the return value which will be of type core.ZygoTask. From that return value, the user can check the home status, wait for a specified timeout, or wait until complete.

The two-parameter home method provides all the functionality of the methods described above, in a single method. The first parameter can be either a single value of type AxisType or an iterable (tuple, list, set) of AxisTypes. The wait flag is as previously described. This method is provided for backward-compatibility. In general, the single-parameter methods should be used.

The module also provides a method to check if an axis is homed. It takes a single AxisType and returns whether that axis has been homed.

```
home_x(wait=True)
    Home the x-axis

    Parameters: • wait: True to wait for requested axes to home

home_y(wait=True)
    Home the y-axis

    Parameters: • wait: True to wait for requested axes to home

home_z(wait=True)
    Home the z-axis

    Parameters: • wait: True to wait for requested axes to home
```

```
home_xy(wait=True)
    Home the x- and y-axes

    Parameters: • wait: True to wait for requested axes to home

home_xyz(wait=True)
    Home the x-, y-, and z-axes

    Parameters: • wait: True to wait for requested axes to home

home_r(wait=True)
    Home the roll-axis

    Parameters: • wait: True to wait for requested axes to home

home_p(wait=True)
    Home the pitch-axis

    Parameters: • wait: True to wait for requested axes to home

home_rp(wait=True)
    Home the roll- and pitch-axes

    Parameters: • wait: True to wait for requested axes to home

home_t(wait=True)
    Home the theta-axis

    Parameters: • wait: True to wait for requested axes to home

home_all(wait=True)
    Home all active axes

    Parameters: • wait: True to wait for requested axes to home

home(axes, wait=True)
    Home requested axes

    Parameters: • axes: AxisType or list of AxisTypes for which to request home
                • wait: True to wait for requested axes to home

is_homed(axis)
    Return whether specified axis is homed

    Parameters: • axis: AxisType for which to request home status
    Returns:    True is the specified axis is homed, False otherwise
```

Move Axes

The `motion` module provides methods to request a set of axes to move. It takes three or more input parameters. The first parameter(s) corresponds to the position(s) of the requested axis or axes. The next parameter is a Units type which applies to all specified position values. The last parameter is a `wait` flag that defaults to `True`. If its value is `False`, then the user should retrieve the return value which will be of type `core.ZygoTask`. From that return value, the user can check the move status, wait for a specified timeout, or wait until complete. The pitch and roll motion methods have an additional parameter specifying whether to perform a parcentric move.

```
move_x(x_pos, unit, wait=True)
    Move x-axis to requested position

    Parameters: • x_pos: Position to move x-axis to
                • unit: The unit used for the stage position parameter
                • wait: True to wait for requested axis to complete move
    Returns:    Task object with wait operation for asynchronous move

move_y(y_pos, unit, wait=True)
    Move y-axis to requested position

    Parameters: • y_pos: Position to move y-axis to
                • unit: The unit used for the stage position parameter
                • wait: True to wait for requested axis to complete move
    Returns:    Task object with wait operation for asynchronous move

move_z(z_pos, unit, wait=True)
    Move z-axis to requested position

    Parameters: • z_pos: Position to move z-axis to
                • unit: The unit used for the stage position parameter
                • wait: True to wait for requested axis to complete move
    Returns:    Task object with wait operation for asynchronous move

move_xy(x_pos, y_pos, unit, wait=True)
    Move xy-axes to requested position

    Parameters: • x_pos: Position to move x-axis to
                • y_pos: Position to move y-axis to
                • unit: The unit used for the stage position parameters
                • wait: True to wait for requested axes to complete move
    Returns:    Task object with wait operation for asynchronous move

move_xyz(x_pos, y_pos, z_pos, unit, wait=True)
    Move xyz-axes to requested position

    Parameters: • x_pos: Position to move x-axis to
                • y_pos: Position to move y-axis to
                • z_pos: Position to move z-axis to
                • unit: The unit used for the stage position parameters
                • wait: True to wait for requested axes to complete move
    Returns:    Task object with wait operation for asynchronous move

move_p(p_pos, unit, wait=True, parcentric=False)
    Move pitch-axis to requested position
```

```
    Parameters: • p_pos: Position to move pitch-axis to
                • unit: The unit used for the stage position parameter
                • wait: True to wait for requested axis to complete move
                • parcentric: True to perform a parcentric move
    Returns:    Task object with wait operation for asynchronous move

move_r(r_pos, unit, wait=True, parcentric=False)
    Move roll-axis to requested position

    Parameters: • r_pos: Position to move roll-axis to
                • unit: The unit used for the stage position parameter
                • wait: True to wait for requested axis to complete move
                • parcentric: True to perform a parcentric move
    Returns:    Task object with wait operation for asynchronous move

move_rp(r_pos, p_pos, unit, wait=True, parcentric=False)
    Move roll and pitch axes to requested position

    Parameters: • r_pos: Position to move roll-axis to
                • p_pos: Position to move pitch-axis to
                • unit: The unit used for the stage position parameters
                • wait: True to wait for requested axes to complete move
                • parcentric: True to perform a parcentric move
    Returns:    Task object with wait operation for asynchronous move

move_t(t_pos, unit, wait=True)
    Move theta-axis to requested position

    Parameters: • t_pos: Position to move theta-axis to
                • unit: The unit used for the stage position parameter
                • wait: True to wait for requested axis to complete move
    Returns:    Task object with wait operation for asynchronous move
```

Retrieve Current Position

The `motion` module provides methods to get the current positions of a set of axes. Their input parameter is the desired Units. The return value is the position in the requested Units.

```
get_x_pos(unit)
    Retrieve position of x axis in requested unit

    Parameters: • unit: Desired unit for return value
    Returns:    Axis position in requested unit

get_y_pos(unit)
    Retrieve position of y axis in requested unit

    Parameters: • unit: Desired unit for return value
    Returns:    Axis position in requested unit

get_z_pos(unit)
    Retrieve position of z axis in requested unit

    Parameters: • unit: Desired unit for return value
    Returns:    Axis position in requested unit

get_p_pos(unit)
    Retrieve position of pitch axis in requested unit
```

```
    Parameters: • unit: Desired unit for return value
    Returns:    Axis position in requested unit

get_r_pos(unit)
    Retrieve position of roll axis in requested unit

    Parameters: • unit: Desired unit for return value
    Returns:    Axis position in requested unit

get_t_pos(unit)
    Retrieve position of theta axis in requested unit

    Parameters: • unit: Desired unit for return value
    Returns:    Axis position in requested unit
```

Wait On Axes

The `motion` module provides a method to wait for a set of axes to stop moving. It takes two input parameters. The first parameter can be either a single value of type `AxisType` or an terable (tuple, list, set) of `AxisTypes`. The second parameter is a timeout value specified in milliseconds. If the requested axes finish moving before the timeout has elapsed then the method completes and returns `None`. However, if the timeout elapses before one or more specified axes has completed its move then a `core.ZygoError` exception is raised.

```
wait(axes, timeout=None)
    Wait for all requested axes to finish moves

    Parameters: • axes: AxisType list of AxisTypes for which to request wait
                • timeout: Maximum time to wait for axes to finish moving in
                           milliseconds; None for infinite
```

Axis Availability

The `motion` module provides a method for determining if an axis is available on the host instrument. It takes one parameter that is of type `AxisType`. It returns `True` if the axis is available on the host instrument. It will return `False` if it is not.

```
is_active(axis)
    Return whether specified axis is available or not

    Parameters: • axis: AxisType for which to request availability status
    Returns:    True if the specified axis is available, False otherwise
```

Pendant Status

The `motion` module provides a method for enabling or disabling the pendant on the host instrument. It takes one parameter that is of type `bool` or convertible to `bool`.

```
set_pendant_enabled(enabled)
    Enable or disable pendant

    Parameters: • enabled: True to enable pendant, False otherwise
```

Z-Stop Status

The `motion` module provides a method for checking the status of the z-stop on the host instrument. It returns `True` is the z-stop is set, `False` otherwise.

```
is_zstop_set()
    Return whether z-stop is set

    Returns:    True is z-stop is set, False otherwise
```

fiducials Module

The fiducials module provides functionality for loading/saving, retrieving information on, and manipulating fiducials in Mx.

Fiducial Class

The Fiducial class represents a single fiducial. It belongs to one and only one working set. It can be manipulated by moving, rotating, and resizing. Its properties consist of its location and size. A fiducial is not created directly from a script, but is instead retrieved from its Fiducials object container.

```
class Fiducial(object)
    Represent one Mx fiducial

    move_absolute(self, x, y)
        Move center of fiducial to specified absolute x, y position

        Parameters: • x: New fiducial center x-coordinate
                    • y: New fiducial center y-coordinate

    move_relative(self, x, y)
        Move center relative to current position by specified x, y amount

        Parameters: • x: X-offset to move center
                    • y: Y-offset to move center

    resize(self, height, width)
        Resize fiducial to specified height, width

        Parameters: • height: New fiducial height
                    • width: New fiducial width

    rotate(self, value, unit)
        Rotate fiducial counterclockwise by specified angle value and unit

        Parameters: • value: Numeric value of the angle to rotate fiducial by
                    • unit: The corresponding Units of the angle of rotation

    center
        Return fiducial center X,Y coordinate

    height
        Return fiducial height

    width
        Return fiducial width
```

Fiducials Class

The Fiducials class represents the entire group of all working sets of all fiducials. When created, a Fiducials object is synchronized to the currently-loaded group of fiducials in Mx and, thus, to the Mx Fiducial editor, whether it is showing or not.

A Fiducials object can be created by calling its parameter-less initializer. A Fiducials object can be saved to and loaded from a file. All of the fiducials in a given working set can be cleared. A particular

working set can be deleted resulting in all of its fiducials being deleted. A new empty working set can be created. A `Fiducials` object can be used to get the number of working sets currently available as well as the number of fiducials in a given working set or in all working sets. A particular `Fiducial` object can be retrieved that is closest to a given location that is specified. It can also discriminate based on its working set container. A particular `Fiducial` object can be removed from the `Fiducials` object, resulting in the fiducial being removed from the Fiducial editor. This is accomplished by finding the `Fiducial` object of interest and calling the `delete` method of the `Fiducials` object. The `Fiducials` class is actually a container class. It can be iterated over to examine and/or manipulate all of the individual `Fiducial` objects. An example follows.

```
fs = Fiducials()
print('Number of working sets: {0}'.format(fs.get_num_sets()))
print('Total number fiducials in all sets: {0}'.format(fs.get_num_fiducials_in_set()))
for w, f in fs:
    f.move_relative(30.0, -10.0)
    f.resize(f.height + 30.0, f.width + 30.0)
    f.rotate(45, Units.Degrees)
    print('working set = {0}'.format(w), end=' --> ')
    print('center = ({0.x:.2f}, {0.y:.2f})'.format(f.center), end=', ')
    print('height = {0.height:.2f}, width = {0.width:.2f}'.format(f))
```

```
class Fiducials(object)
    Represent a collection of Mx fiducial

    get_num_sets(self)
        Return number of working sets

        Returns:    Number of working sets

    get_num_fiducials(self, working_set=None)
        Return number of fiducials in given working set

        Parameters: • working_set: Working set for fiducial count (None for all)
        Returns:    Number of fiducials

    save(self, filename)
        Save fiducials to specified file

        Parameters: • filename: File name to save to

    load(self, filename)
        Load fiducial from specified file

        Parameters: • filename: File name to load from

    delete(self, fiducial)
        Remove specified fiducial from collection of Mx fiducials

        Parameters: • fiducial: Fiducial object to remove

    clear_set(self, working_set)
        Clear all fiducials of specified working set

        Parameters: • working_set: Fiducial working set as integer

    delete_set(self, working_set)
        Delete specified working set and all its fiducials
```

```
        Parameters: • working_set: Fiducial working set as integer

    add_set(self)
        Create a new empty working set

    get_fiducial_closest_to(self, x, y, working_set=None)
        Get working set and fiducial of fiducial closest to
        specified center coordinates

        Parameters: • x: X-coordinate
                    • y: Y-coordinate
                    • working_set: Working set as integer (None for any)
        Returns:    tuple of (working set, Fiducial)
```

pattern Module

The pattern module provides functionality for interfacing with Mx patterns. It contains only module-level functions.

Save/Load Pattern

Methods are provided that will load a pattern or a stitch into Mx from a given filename. A method is also provided to save the current pattern in Mx out to a given filename.

```
save(filename)
    Save current pattern to specified file

    Parameters: • filename: File name to save to

load(filename)
    Load pattern from specified file

    Parameters: • filename: File name to load from

load_stitch(filename)
    Load stitch from specified file

    Parameters: • filename: File name to load from
```

Run Pattern

There is a method to run the current pattern, pre-align the current pattern, and align the current pattern.

```
run()
    Run current pattern

prealign()
    Start pre-alignment on current pattern

align()
    Start alignment on current pattern
```

`recipe` Module

The `recipe` module provides functionality for interfacing with Mx recipes. It contains only module-level functions. It can save and load a recipe from a given filename, and run the current recipe. Its descriptions are below.

Save/Load Recipe

A method is provided that will load a recipe into Mx from a given filename. A method is also provided to save the current recipe in Mx out to a given filename.

```
save(filename)
    Save current recipe to specified file

    Parameters: • filename: File name to save to

load(filename)
    Load recipe from specified file

    Parameters: • filename: File name to load from
```

Run Recipe

There is a method to run the current recipe.

```
run()
    Run current recipe
```

systemcommands Module

The systemcommands module provides functionality for getting information on the Mx host system. It also provides methods for getting and setting of directories for different types of Mx files.

Host Information

The systemcommands module provides methods for retrieving the computer and operating system names.

```
get_os_name()
    Gets the operating system name of Mx host computer

    Returns:    The operating system name as a string

get_computer_name()
    Gets the computer name of the Mx host computer

    Returns:    The computer name as a string
```

FileTypes Class

The systemcommands module contains an enumeration, FileTypes, of all Mx-defined file type categories. It is used for any of the get and set directory methods which require a file_type parameter.

```
class FileTypes(Enum)
    Enumeration of Mx-supported file types

    Script
    Setting
    Application
    Data
    Recipe
    Mask
    Fiducial
    Proc_Stats
    Logging
    Slice

    # etc.
```

See Appendix A for the complete listing of Mx-supported file types.

Directories Information

The systemcommands module provides methods for getting/setting open/save directories for different file types, the Mx binary directory, and the current working directory. The file type input parameter is a FileTypes type described above.

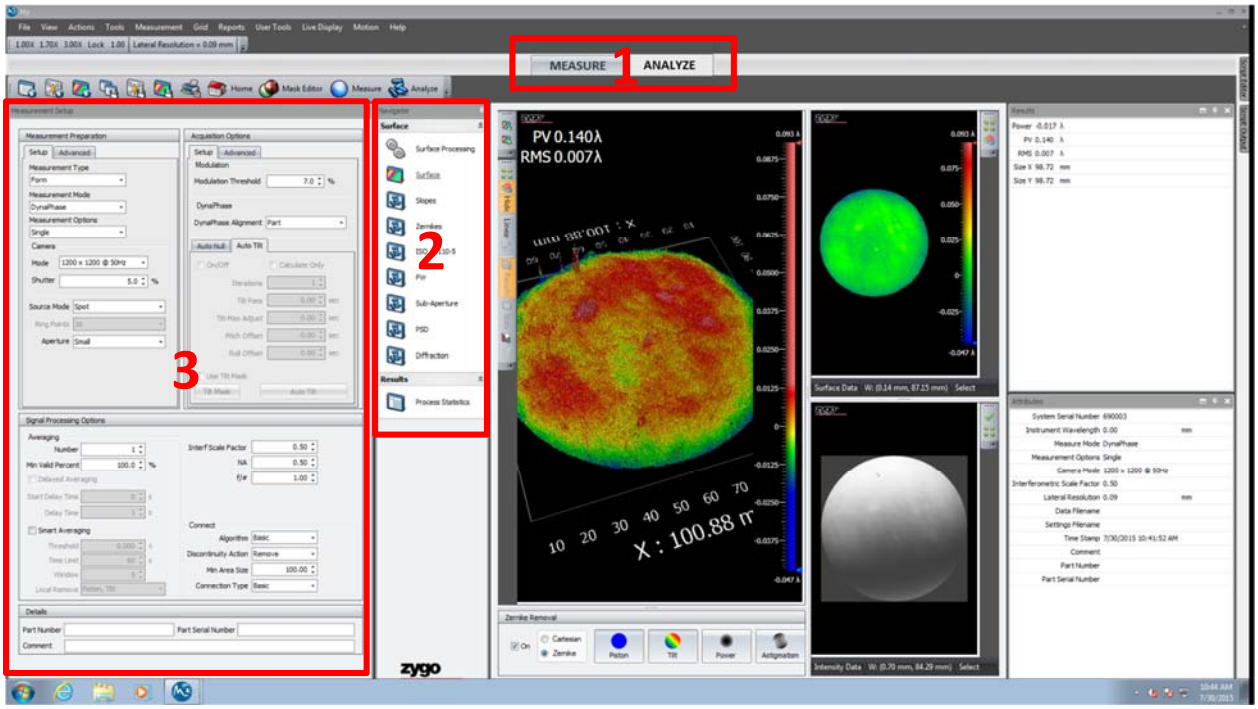
```
get_bin_dir()
    Gets the directory path containing Mx binaries

    Returns:    The absolute path to the Mx bin directory as a string
```

```
get_open_dir(file_type)
    Gets the directory path for given file type that Mx uses for opening a file

    Parameters: • file_type: The file type of interest, as a FileTypes type
    Returns:    The absolute path to the open directory for the requested file type

get_save_dir(file_type)
    Gets the directory path for given file type that Mx uses for saving a file

    Parameters: • file_type: The file type of interest, as a FileTypes type
    Returns:    The absolute path to the save directory for the requested file type

get_working_dir()
    Gets the current working directory of the Mx process

    Returns:    The absolute path to the Mx working directory as a string

set_open_dir(file_type, path)
    Set the directory path for the given file type that Mx uses for opening a file

    Parameters: • file_type: The file type of interest, as a FileTypes type
                • path: The absolute path to the open directory for the requested file
                        type as a string

set_save_dir(file_type, path)
    Set the directory path for the given file type that Mx uses for saving a file

    Parameters: • file_type: The file type of interest, as a FileTypes type
                • path: The absolute path to the save directory for the requested file
                        type as a string

list_files_in_dir(directory, extensions, recursive=False)
    Gets a list of all files in the given directory that match the given list of
    extensions

    Parameters: • directory: The directory to search
                • extensions: A list of extensions to match, as strings
                • recursive: True to search directory and all subdirectories; False
                             Otherwise
    Returns:    A list of all file path strings

list_files_in_open_dir(file_type)
    Returns a list of all the files of the given FileType in the type's primary open
    directory

    Parameters: • file_type: The file type of interest, as a FileTypes type
    Returns:    A list of file path strings of the requested type
```

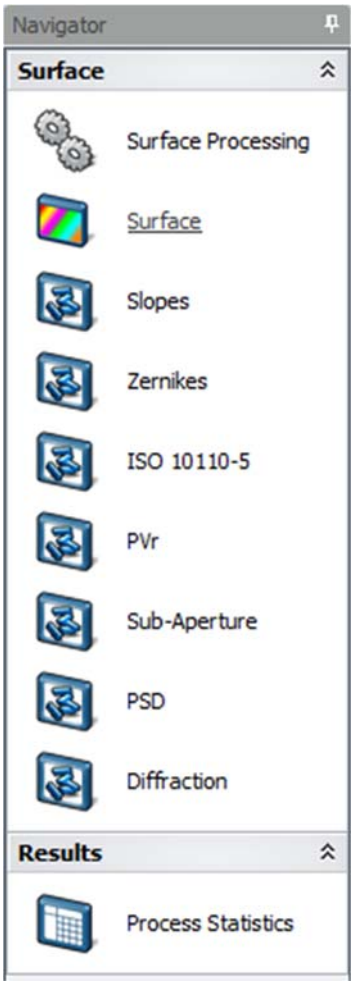
`ui` Module

The `ui` module provides functionality for communicating with the Mx GUI. It allows the script to display a modal dialog for showing text for message / warning / error, OK/Cancel confirmation, for retrieving input, and other uses. It allows the script to get access to tabs, groups, containers, dock panels, controls, etc. Some controls allow saving data and/or images to a file. Some controls have children controls. Some containers and dock panels allow the window to be minimized or normalized.

Mx GUI Components



The image above outlines the (1) **Tab**, (2) **Navigator**, and (3) **DockPanel** UI components. Every Mx app has a set of Tabs, referenced by their name as a string, e.g., "Measure", "Analyze". In this example, the active Tab is "Analyze". Each Tab contains a single Navigator and zero or more DockPanels. Following is a close-up view of the Navigator above:



The Navigator belonging to a particular Tab provides a simple way of identifying other Mx GUI components. **Groups**, the next level below Tabs in the Mx UI hierarchy, are displayed in the Navigator as the heading with the light-grey background. In this example, "Surface" and "Results" are the Groups found in the Analyze tab.

Within a Group, another level down the hierarchy, are **Containers** and **ContainerWindows**. "Surface", "Slopes", and "Zernikes" are examples of Containers in the Surface group, and "Surface Processing" is a ContainerWindow in the same group. The sole Container in the Results group is "Process Statistics". The difference between a Container and a ContainerWindow is that a Container is shown in the main area of Mx, while a ContainerWindow is displayed as its own window.

It is important to understand that Groups, Containers, and ContainerWindows do not *belong to* the Navigator – the Navigator is simply a list of the Tab's Groups, and a collection of links to each of the Containers and ContainerWindows within those Groups.

A **Control** is a UI component which is found within a Container, ContainerWindow, or another Control. In the context of the Mx GUI, a Control is one of several types of items which contain data that can be

acted on in specific ways. These items include: plots, plot details, plot histograms, process statistics, control charts, slice charts, and slice statistics.

The defining characteristic of a Control is that the Control's data can be saved/exported. Generally, the save_data() and save_image() API methods are equivalent to functionality available in the Control's context menu in the Mx GUI, e.g., Save Processed Data, Export Data, Save Bitmap. Some Controls may be used as parameters for other functions. For example, plot Controls are used as parameters in the Data Matrix methods in the `mx` module (see the appropriate section above).

Mx UI Controls should not be confused with the controls (radio buttons, checkboxes, combo boxes, etc.) used to change Mx settings. The Measurement Setup DockPanel pictured above contains a number of these settings controls. An Mx settings control is an item, similar to a result or attribute, which can be added to a control grid. See the section on Results, Attributes, and Controls in the `mx` module for more details.

Modal Dialogs

`DialogMode` Class

A `DialogMode` enumeration defines the available dialog modes. A dialog mode represents the "alert" level of the dialog, e.g., message, warning, error. It also specifies which buttons are available in the dialog box, e.g., OK, Cancel, Yes, No. It is used for any modal dialog that the script requests for Mx to display.

```
class DialogMode(IntEnum)
    Alert mode for dialog

    message_ok
    error_ok
    warning_ok
    confirm_yes_no
    error_ok_cancel
    warning_yes_no
    message_ok_cancel
```

Modal Dialog Methods

There are several ways to show a modal dialog for displaying information or retrieving a single input. A dialog has a mode as described above. A dialog can be displayed that waits for the user to press a button or it can be displayed for a specified length of time before closing.

```
show_dialog(text, mode, seconds=None)
    Show dialog with specified text and alert mode for requested duration

    Parameters: • text: String to display
                • mode: Alert mode for dialog
                • seconds: Number of seconds to show or None (Default) to wait for
                           user acknowledgement
    Returns:    Boolean dialog result if applicable; None otherwise

show_input_dialog(text, default_value, mode, max_length)
    Display user input request dialog

    Parameters: • text: String to display
```

```
                • default_value: Default value for user input
                • mode: Alert mode for dialog
                • max_length: Maximum length for user input
    Returns:    User input as parsed type

show_dropdown_dialog(text, selection_values, mode)
    Display user input dropdown dialog

    Parameters: • text: String to display
                • selection_values: List of values to add to the dropdown control,
                                    in display order
                • mode: Alert mode for dialog
    Returns:    Zero-based index of the selected item, or -1 if cancel/no pressed

show_file_dialog(type, make_dir_primary=False, allow_multiselect=False)
    Display an Mx file open dialog

    Parameters: • type: The specified systemcommands.FileTypes type for the dialog
                • make_dir_primary: Whether or not to save the selected directory as
                                    the default for the type
                • allow_multiselect: Whether or not to allow multiple file selections
                                     in the dialog
    Returns:    List of selected files, None for canceled dialog
```

Plot Palette

Palette Class

A Palette enumeration defines the available plot palette selections.

```
class Palette(IntEnum)
    Available plot palette selections

    Spectrum
    RWB
    Grey
    CMYK
    IcyCool
    Neon
    RedHot
    Bands
    Gold
    Red
    Binary
```

PaletteScaleMode Class

A PaletteScaleMode enumeration defines the available plot palette scale modes.

```
class DialogMode(IntEnum)
    Available plot palette scale modes

    PV
    Auto
    ThreeSigma
    Fixed
```

Plot Palette Methods

The ui module contains methods to change a plot's palette and palette scale.

```
set_plot_palette(control, palette_name=Palette.Spectrum)
```

```
    Get requested Mx tab

    Parameters: • control: The plot Control object
                • palette_name: The palette to use in the plot as a Palette type

set_plot_palette_scale(control, scale_mode=PaletteScaleMode.Auto, peak=10.0,
                       valley=0.0, unit=Units.MicroMeters)
    Set a plot palette scaling mode

    Parameters: • control: The plot Control object
                • scale_mode: The scaling mode as a PaletteScaleMode type
                • peak: The Fixed mode peak value
                • valley: The Fixed mode valley value
                • units: The units for the peak and valle
```

Miscellaneous Module-Level Methods

Toolbar Click

A toolbar can be clicked with a `ui` module-level function via its path as a tuple. This path can be in one of two formats. One format is a pair of category and button name, e.g., `('file', 'exit')`. The category can be found in the "Commands" tab of the Mx™ "Customize Toolbar" window. The other format is a path, potentially multiple depths, going from the toolbar name to the button name, e.g., `('main menu', 'file', 'exit')`.

```
click_toolbar_item(path)
    Click on toolbar item

    Parameters: • path: Toolbar button path
```

Image Grid Methods

The `ui` module provides a method for setting the current Image grid image in Mx.

```
set_image_grid(control, image_path)
    Set an image to show in the Image Grid

    Parameters: • control: The image grid Control object
                • image_path: The full path and name of the image file
```

Mx Application UI Methods

```
get_default_plot_control_path()
    Gets the path of the default Mx plot control

    Returns:    The path to the currently-defined default plot in Mx

get_home_container()
    Gets the home container

    Returns:    The container configured as the startup container in the home tab, or
                the first available container in the home tab if no startup container
                is defined

get_home_tab()
    Gets the home tab

    Returns:    The tab configured as the startup tab, or the first available tab if
                no startup tab configured
```

Processing Sequence Methods

The ui module provides a method for setting the current Image grid image in Mx.

```
set_sequence_step_state(sequence_id, sequence_step_description, is_on)
    Set a sequence step on/off state

    Parameters: • sequence_id: The sequence id from using Show Id
                • sequence_step_description: The sequence step description from the
                                             Step Properties
                • is_on: The True (on) or False (off) state of the step
```

Tab Class

The Tab class encapsulates the behavior and state of an Mx tab, e.g. Measure, Analyze. Two ui module-level functions exist for getting Tab objects. One function gets one Tab object with a specified name, while the other gets a tuple of all of them.

```
get_tab(name)
    Get requested Mx tab

    Parameters: • name: Name of tab
    Returns:    Tab object

get_tabs()
    Get available Mx tabs

    Returns:    Tuple of Tab objects
```

A Tab object has a Navigator object, a tuple of Group objects, and a tuple of DockPanel objects. From a Tab object, the script can get a specific Group or DockPanel or get a tuple of all of them. It can also display itself as the current tab with the show method.

```
class Tab(object)
    Represent Mx tab

    show()
        Show this tab

    get_group(group_name)
        Get requested group in navigator

        Parameters: • group_name: Name of group
        Returns:    Group object

    get_dock_panel(panel_name)
        Get requested dock panel in tab

        Parameters: • panel_name: Name of dock panel
        Returns:    DockPanel object

    dock_panels
        Tuple of dock panels in tab

    groups
        Tuple of groupings in tab
```

Group Class

The Group class encapsulates the behavior and state of an Mx navigator group. A Group object has a tuple of children Container or ContainerWindow objects. A Group object can be used to find one specific child by name or to return all in a tuple.

```
class Group(object)
    Represent Mx navigator group of containers

    get_container(container_name)
        Get requested container in navigator group

        Parameters: • container_name: Name of child
        Returns:    Container or ContainerWindow object

    containers()
        Tuple of children in grouping
```

`Navigator` Class

The `Navigator` class encapsulates the behavior and state of an Mx navigator. It can pin and unpin itself to/from its current location.

```
class Navigator(object)
    Represent Mx navigator

    pin(do_pin)
        Pin/Unpin navigator

        Parameters: • do_pin: True to pin; False to unpin
```

`DockPanel` Class

The `DockPanel` class encapsulates the behavior and state of an Mx dock panel. It can pin and unpin itself to/from its current location.

```
class DockPanel(object)
    Represent Mx dock panel

    pin(do_pin)
        Pin/Unpin dock panel

        Parameters: • do_pin: True to pin; False to unpin
```

`Container` Class

The `Container` class encapsulates the behavior and state of an Mx container. It can show itself and return a tuple of all of its children `Control` objects.

```
class Container(object)
    Represent Mx container

    show()
        Show this container

    controls
        Tuple of children controls contained within

    plots
        Tuple of child plot controls contained within
```

`ContainerWindow` Class

The `ContainerWindow` class encapsulates the behavior and state of an Mx container window. It behaves very similar to both a `Container` object and a `Window` object. It can show itself, close itself, minimize (`to_back`), and normalize (`to_front`). It has a tuple of any of its accessible child `Control` objects.

```
class ContainerWindow(object)
    Represent Mx container window, e.g., Pattern Popup

    show()
        Show this control window
```

```
    close()
        Close window

    to_front()
        Bring window to front

    to_back()
        Send window to back

    controls
        Tuple of children controls contained within

    open
        True if window currently open; False otherwise

    plots
        Tuple of child plot controls contained within
```

Window Class

The `Window` class encapsulates the behavior and state of an Mx window. There are `ui` module-level functions for opening the desired window via its unique name, e.g. Mask. The function will return the corresponding `Window` object. See below.

```
show_mask_editor()
    Display mask editor

    Returns:    Window object

show_fiducial_editor()
    Display fiducial editor

    Returns:    Window object
```

A `Window` object behaves similarly to a `ContainerWindow` object. It can close itself, save data to file, save image to file, minimize (`to_back`), and normalize (`to_front`). It has a tuple of any of its accessible children `Control` objects. A `Window` object is not retrieved via any other `Window`, `Container`, `ContainerWindow`, or `Control` object.

```
class Window(object)
    Represent Mx non-container window, e.g. Mask, Fiducials

    close()
        Close window

    save_data(file_path)
        Save data to file. File extension used to determine file type.

        Parameters: • file_path: Target file path to save data to

    save_image(file_path)
        Save image to file. File extension used to determine file type.

        Parameters: • file_path: Target file path to save image to

    to_front()
        Bring window to front
```

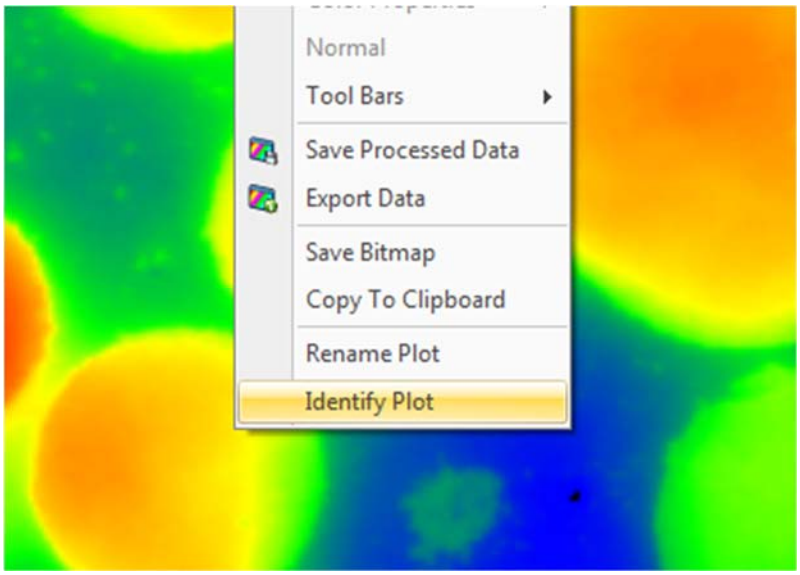
```
    to_back()
        Send window to back

    controls
        Tuple of children controls contained within

    open
        True if window currently open; False otherwise
```

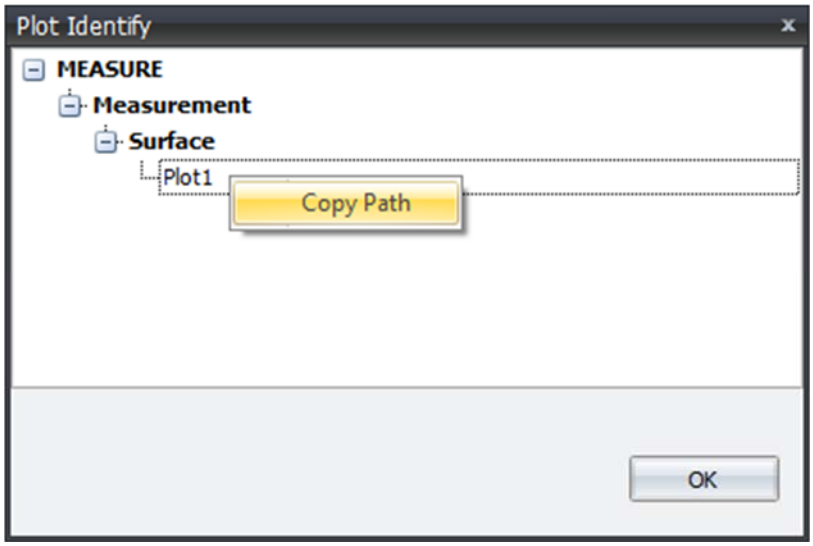
Control Class

The Control class encapsulates the behavior and state of an Mx UI control. Control objects that are accessible from any parent Control, Container, or ContainerWindow objects are also accessible via a ui module-level function via its path as a tuple.

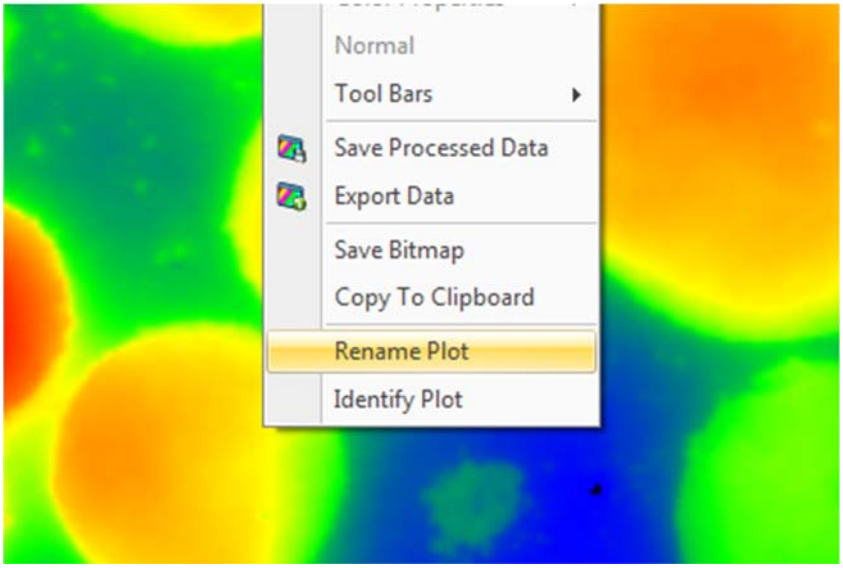
```
get_control(path)
    Get control located at path

    Parameters: • path: Path to control
    Returns:    Control object
```

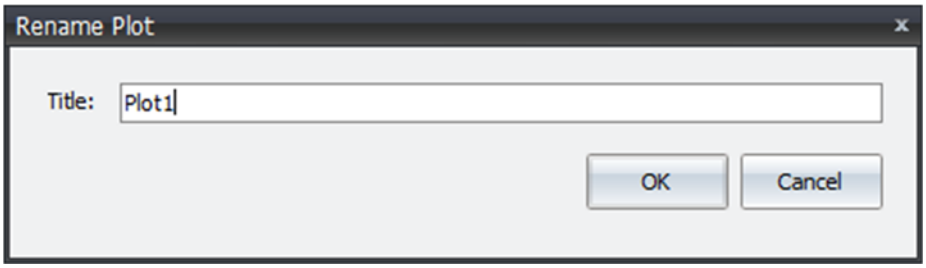
The path to a Control can be obtained via the appropriate Identify option in the control's context menu.





Note that controls must have unique paths in order to be retrieved. For certain controls, e.g., plots, process statistics, and grids, the control name can be changed via the Rename item in the control's context menu. Control names are saved with Mx settings. Be aware that, when loading settings, control names are not applied until the control is visible.





A `Control` object can save data to file, save an image to file, and overlay plot control toolbar buttons can be clicked. It has a tuple of any of its accessible children `Control` objects.

```
class Control(object)
    Represent Mx control

    save_data(file_path, optional_params=None)
        Save data to file. File extension used to determine file type.

        Parameters: • file_path: Target file path to save data to
                    • optional_params: The optional process stats, CodeV, or Sdf
                                       parameters

    save_image(file_path)
        Save image to file. File extension used to determine file type.

        Parameters: • file_path: Target file path to save image to

    click_toolbar_item(path)
        Click on toolbar item

        Parameters: • path: Toolbar button path

    controls
        Tuple of children controls contained within
```

Saving Control Data

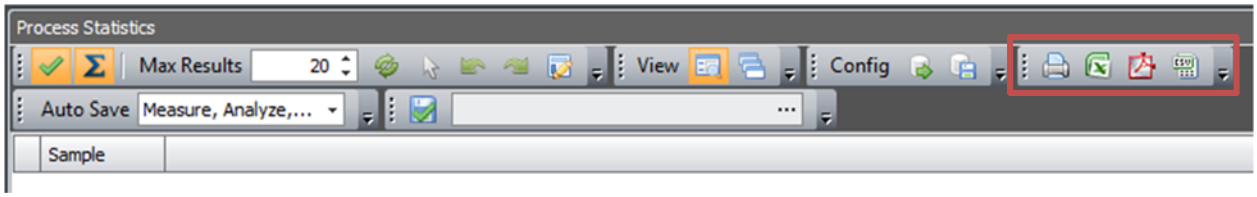
`save_data()` may require additional options, supplied as the `optional_params` parameter, based on the type of control/data being saved. There are three classes which are used to supply these options: `ProcStatsParams` for saving process statistics, `CodeVParams` for exporting Code V Data files, and `SdfParams` for exporting SDF files.

Saving Process Stats data requires `optional_params` be set to an object of type `ProcStatsParams`. The `ProcStatsParams` class contains two members, `simple_mode` and `standard_format`.

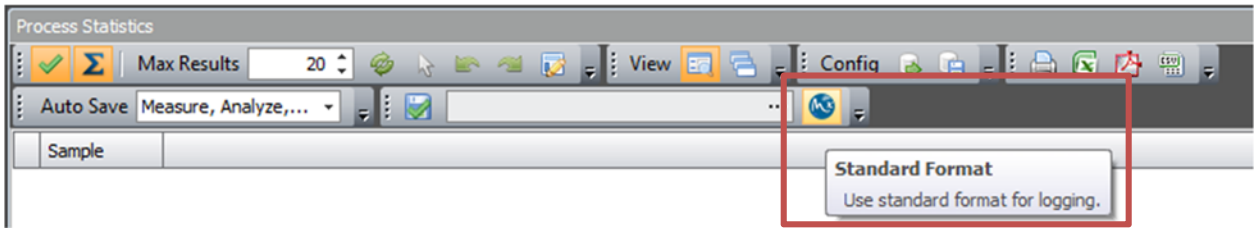
```
class ProcStatsParams(object)
        Process statistics save parameters

        card_view
            Determines the simple mode view format to save.

            If True, process statistics will be saved in the Card View format.
            It False, process statistics will be saved in the Table View format.
            This only applies when simple_mode is True.

        simple_mode
            Determines the process statistics save mode.

            If True, the data will be saved in the format corresponding to one
            of the export buttons in the Process Stats control.
            If False, the data will be saved in one of the autolog formats.

        standard_format
            Determines the process statistics save format,
```

```
            This value corresponds to the AutoLog>Standard Format button in
            the Process Stats control.
```

Setting `simple_mode` to `True` corresponds to the functionality of the highlighted buttons below:



The `standard_format` property corresponds to the Standard Format button of the AutoLog toolbar. By default, this button is not shown and is off. As such, typical behavior would be to set this property to `False`.

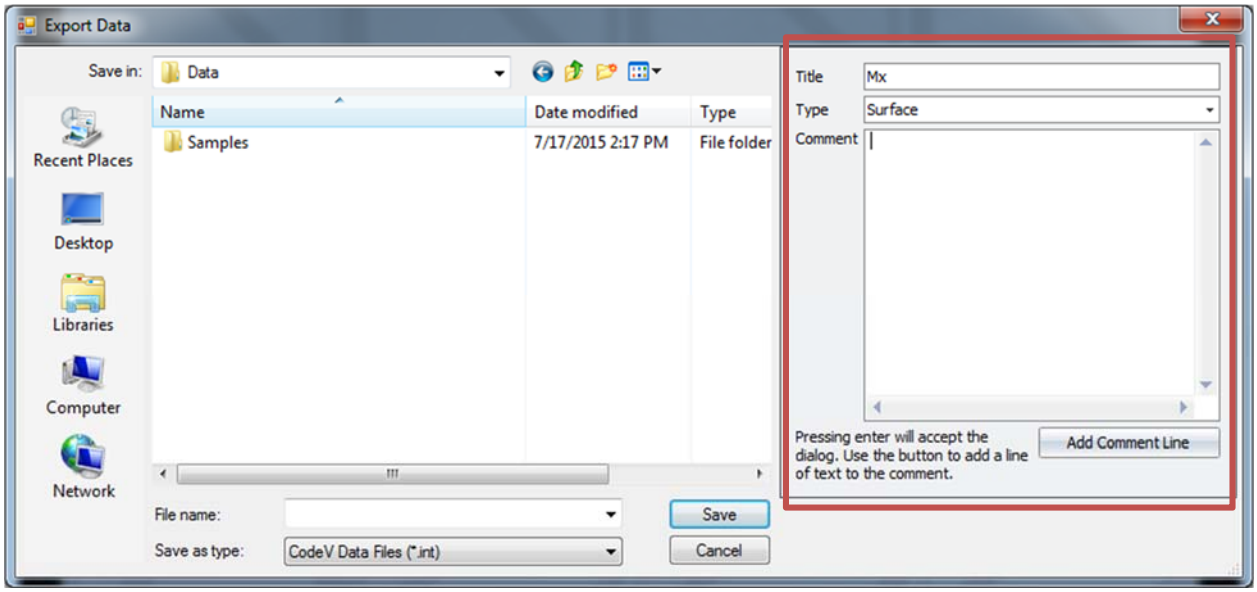


Exporting overlay data requires optional_params to be either an object of the `CodeVParams` or the `SdfParams` type, depending on the type of file being exported (.int and .sdf, respectively).

The `CodeVParams` class contains three properties, `title`, `type`, and `comment`:

```
class CodeVParams(object)
        CodeV export parameters

        title
            The CodeV Title export field

        type
            The CodeV Type export field

        comment
            The CodeV Comment export field
```

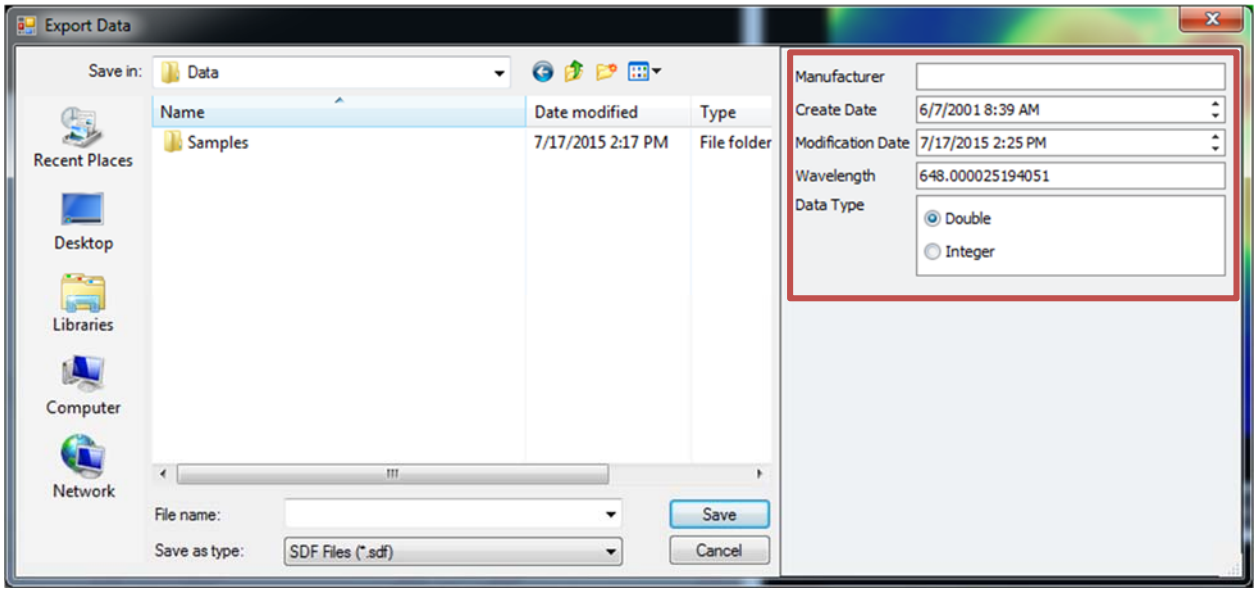
These properties correspond to the fields in the Export Data dialog as shown below. The `type` string parameter must be one of 'Wavefront', 'Surface', or 'Filter'.



The `SdfParams` contains five properties: `manufacturer`, `create_date`, `modification_date`, `wavelength`, and `data_type`:

```
class SdfParams(object)
        Sdf export parameters

        manufacturer
            The Sdf Manufacturer export field

        create_date
            The Sdf CreateDate export field

        modification_date
            The Sdf ModificationDate export field

        wavelength
            The Sdf Wavelength export field

        data_type
            The Sdf DataType export field
```

These properties correspond to the fields in the Export Data dialog as shown below. `create_date` and `modification_date` are expected to be `datetime` objects. `wavelength` is a double value. The `data_type` string parameter must be one of 'Integer' or 'Double'.


Export Data
Save in:
Data
Recent Places
Desktop
Libraries
Computer
Network
Name
Date modified
Type
Samples
7/17/2015 2:17 PM
File folder
File name:
Save as type:
SDF Files (*.sdf)
Save
Cancel
Manufacturer
Create Date
6/7/2001 8:39 AM
Modification Date
7/17/2015 2:25 PM
Wavelength
648.000025194051
Data Type
Double
Integer

Appendix A

Mx Directory Types

All
UI_Application
Script
Csv
Swli
AFC_Measurement
ShortTerm
DeltaPsi
Bin
Xml
Setting
Application
Data
Signal_Data
Programming
Application_Reference
VisionPro_Persistance
Image
Recipe
Result
Text
Mask
Fiducial
Zernike
Average
Proc_Stats
Logging
Slice
Region
Region_Stats
Cal_Data
Cal_Data_Archive

Appendix B

Example Programs

Disclaimer

THE SAMPLE CODE IS PROVIDED "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ZYGO OR ANY AFFILIATED COMPANY, ITS OR THEIR OFFICERS, EMPLOYEES OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) SUSTAINED BY YOU OR A THIRD PARTY, HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT ARISING IN ANY WAY OUT OF THE USE OF THIS SAMPLE CODE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

mx_basic.py

```
"""
Sample script for basic mx module functionality,
e.g. application loading, unloading, and status.
"""

import os.path
import time

from zygo import mx
import zygo.systemcommands as sc


# Get and print application status
print('Is application open? {0}'.format(mx.is_application_open()))
print('Sleeping ...')
time.sleep(5)

# Open application
app_file = r'C:\Users\zygo\Documents\Mx\Apps\Micro.appx'
print('Opening application: "{0}" ...'.format(app_file))
mx.open_application(app_file)
print('Sleeping ...')
time.sleep(5)

# Get and print application status
if mx.is_application_open():
    print('Application opened: "{0}"'.format(mx.get_application_path()))
else:
    print('Application is not open.')
print('Sleeping ...')
time.sleep(5)

# Close application, get and print application status
print('Closing application ...')
mx.close_application()
print('Is application open? {0}'.format(mx.is_application_open()))
```

mx_intermediate.py

```
"""
Sample script for intermediate mx module functionality,
e.g. application and data loading, retrieving results and attributes,
modifying control values, and saving application.
"""

import os.path
import time

from zygo.units import Units
from zygo import mx
import zygo.systemcommands as sc


# File below expected to exist in the current open directory
# for Application files
app_filename = 'Micro.appx'

# File below expected to exist in Samples subfolder of the
# current open directory for Data files.
data_filename = 'Plot3D.datx'

# Get and print application status
print('Is application open? {}'.format(mx.is_application_open()))

# Open application
open_app_dir = sc.get_open_dir(sc.FileTypes.Application)
app_file_path = os.path.join(open_app_dir, app_filename)
print('Opening application: "{}" ...'.format(app_file_path))
mx.open_application(app_file_path)

# Get and print application status
print('Is application open? {}'.format(mx.is_application_open()))

# Load a sample data file
open_data_dir = sc.get_open_dir(sc.FileTypes.Data)
data_file_path = os.path.join(open_data_dir, 'Samples', data_filename)
print('Loading data: "{}" ...'.format(data_file_path))
mx.load_data(data_file_path)

# Retrieve results and attributes

## Paths to the results and attributes of interest
pv_path = ("Analysis", "Surface", "Surface Parameters", "Height Parameters", "PV")
sa_path = ("Analysis", "Surface", "Areal ISO Parameters", "Height Parameters", "Sa")
lat_res_path = ("Instrument",
                "Measurement Setup",
                "Acquisition",
                "Lateral Resolution")
data_filename_path = ("System", "Load", "Data Filename")

## Create aliases for Units to save typing
um_unit = Units.MicroMeters
nm_unit = Units.NanoMeters

## Retrieve the result and attribute data from Mx
pv = mx.get_result_number(pv_path, um_unit)
sa = mx.get_result_number(sa_path, nm_unit)
lat_res = mx.get_attribute_number(lat_res_path, um_unit)
data_file = mx.get_attribute_string(data_filename_path)
```

```
## Display the information
print("PV = {} {}".format(pv, um_unit.name))
print("Sa = {} {}".format(sa, nm_unit.name))
print("Lateral Resolution = {} {}".format(lat_res, um_unit.name))
print("Data Filename = {}".format(data_file))

# Change the Measurement Type control value
meas_type_path = ("Instrument",
                  "Measurement Setup",
                  "Acquisition",
                  "Measurement Type")
print('Current Measurement Type: {}'.format(mx.get_control_string(meas_type_path)))
meas_type = 'Intensity SnapShot'
print('Changing Measurement Type to: {}'.format(meas_type))
mx.set_control_string(meas_type_path, meas_type)

# Save the application
save_app_dir = sc.get_save_dir(sc.FileTypes.Application)
save_app_filename = '{}_test.appx'.format(os.path.splitext(app_filename)[0])
save_app_path = os.path.join(save_app_dir, save_app_filename)
print('Saving application as: "{}" ...'.format(save_app_path))
mx.save_application_as(save_app_path)
print('Sleeping ...')
time.sleep(5)

# Close application, get and print application status
print('Closing application ...')
mx.close_application()
print('Is application open? {}'.format(mx.is_application_open()))
```

motion_basic.py

```
"""
Sample script for basic motion module functionality.
"""

import os.path

from zygo.units import Units
from zygo import instrument, mx, motion
import zygo.systemcommands as sc


# File below expected to exist in current open directory for Application files.
# See mx_basic.py for examples.
app_file = 'Micro.appx'

# Open application if not open
orig_app_open = mx.is_application_open()
if not orig_app_open:
    print('Opening application ...')
    app_dir = sc.get_open_dir(sc.FileTypes.Application)
    mx.open_application(os.path.join(app_dir, app_file))

# Create (x, y) coordinate pairs
targets = ((0.0, 0.0), (0.5, 0.0),
           (0.0, 0.5), (0.5, 0.5))

# Loop through each target coordinate pair
# CAUTION: This will move the XY stage to the absolute positions
# specified above, and will trigger an auto-focus at each position.
for target in targets:
    # Get current positions of x and y axes
```

```
    xy_unit = Units.MilliMeters
    x_pos = motion.get_x_pos(xy_unit)
    y_pos = motion.get_y_pos(xy_unit)
    print('XY position:')
    print('\tBefore = ({0} {2}, {1} {2})'.format(
        x_pos, y_pos, xy_unit.name))

    # Move stage to next position
    motion.move_xy(target[0], target[1], xy_unit)

    # Get new current positions of x and y axes.
    x_pos = motion.get_x_pos(xy_unit)
    y_pos = motion.get_y_pos(xy_unit)
    print('\tAfter = ({0} {2}, {1} {2})'.format(
        x_pos, y_pos, xy_unit.name))

    # Optimize focus and get before and after positions of the z-axis,
    # rounded to 3 decimal places.
    z_unit = Units.MicroMeters
    print('Z position:')
    print('\tBefore = {:.3f} {}'.format(
        motion.get_z_pos(z_unit), z_unit.name))
    instrument.auto_focus()
    print('\tAfter = {:.3f} {}'.format(
        motion.get_z_pos(z_unit), z_unit.name))

print("Done.")
```

Appendix C

Common File Tasks

Representing File Paths

In Python, file pathnames are represented using string values, e.g.:

```
sample_data = 'C:\\Users\\zygo\\Documents\\Mx\\Data\\Samples\\Plot3D.datx'
```

It is common to find Windows file paths expressed as Python raw string literals, e.g.:

```
sample_data = r'C:\Users\zygo\Documents\Mx\Data\Samples\Plot3D.datx'
```

Both are equivalent. The `r` prefix instructs Python to interpret the string with most escape sequence processing disabled. That is, in a raw string, a backslash is just a backslash. Sequences such as `'\n'` are treated literally rather than interpreted as, in this example, a newline. There cannot be any whitespace between the `r` and the beginning of the string. Also, raw string literals cannot contain a backslash as the last character. For example, the following is invalid syntax:

```
samples_folder = r'C:\Users\zygo\Documents\Mx\Data\Samples\'
```

Joining File Paths

Since file pathnames are string values, all of Python's built-in string manipulation facilities are available to operate on pathnames, e.g., concatenation, slicing, and formatting. In addition, Python includes the standard module `os.path` which provides common pathname manipulations. One of the most frequently used of these is the `os.path.join(`*`path, *paths`*`)` function:

```
base_dir = r'C:\Users\zygo\My Documents\Mx\Data'
sample_data = os.path.join(base_dir, 'Samples', 'Plot3D.datx')
```

This method intelligently concatenates each pathname component parameter, inserting backslashes, removing duplicate backslashes, handling empty parameters, etc. This is particularly useful for joining pathname components which may come from different sources, e.g., user input, Mx API methods, and hardcoded strings.

Using Special Paths

Mx provides several directory methods in the `systemcommands` module, such as `get_open_dir(`*`file_type`*`)` and `get_bin_dir()`, which assist the script writer by providing easy access to paths defined in the Mx Options dialog. Likewise, Python provides a number of methods for accessing the pathnames of certain special paths.

To get the path to the folder containing the currently-executing script, use the `os.getcdw()` function in the standard Python `os` module.

To get the currently logged-in user's home directory, the os.path module contains the `os.path.expanduser(path)` function, which will replace a leading tilde ('~') character with the appropriate home path. The following two examples are equivalent:

```
my_documents = os.path.expanduser(r'~\My Documents')
```

```
home_dir = os.path.expanduser('~')
my_documents = os.path.join(home_dir, "My Documents")
```

There is also a method, `os.path.expandvars(path)`, to expand Windows environment variables:

```
my_documents = os.path.expandvars(r'%userprofile%\My Documents')
```

```
home_dir = os.path.expandvars('%userprofile%')
my_documents = os.path.join(home_dir, "My Documents")
```

Walking a Directory

To walk a directory, use the `os.walk(top, topdown=True, onerror=None, followlinks=False)` function, provided by Python's standard `os` module (see the official Python documentation for complete details). For example, to load every ".datx" file in Mx's data samples folder:

```
from os import path, walk
from zygo import mx


for root, dirs, files in os.walk(r'C:\Users\zygo\Documents\Mx\Data\Samples'):
    for f in files:
        if f.endswith('.datx'):
            filepath = path.join(root, f)
            mx.load_data(filepath)
```

Appendix D

Debugging Techniques for Mx Scripts

Introduction

This appendix describes basic techniques for debugging scripts in Mx™, as well as an introduction to the Python Debugger facilities exposed in Mx™.

Requirements

- Mx™ version 6.3.0.15 or above must be installed with scripting enabled
- Familiarity with Python and Mx Scripting
- Experience debugging Python is recommended

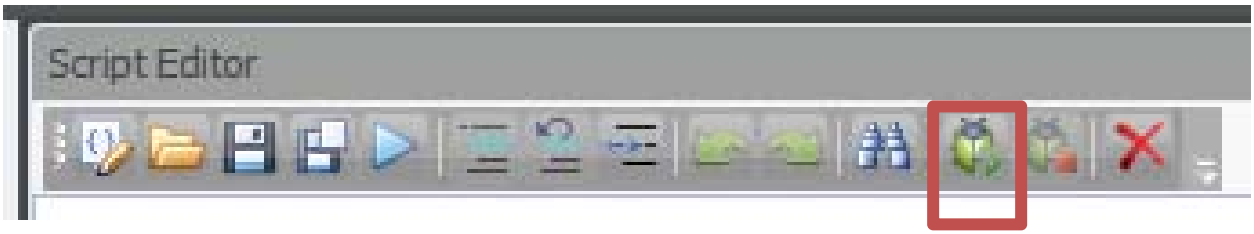
Starting the Debugger

There are two options to start the debugger: directly from the Mx™ Script Editor, or by embedding debugger commands in the target script. Regardless of the method, interaction with the debugger, once started, occurs in the Script Output window.

In this section, we will focus on the first method, initiating a debug session from the Script Editor.

Debugging from the Script Editor

Starting the debugger from the Script Editor will place Mx™ into script debug mode. The script to be debugged must be loaded in the Mx™ Script Editor and visible as the active tab. Click the "Start Debugging" button, highlighted below, to enter script debug mode and begin debugging the active script.



Embedding Debug Commands in a Script

Mx™ allows embedding certain pdb module functions in a script to start the debugger. The pdb module is part of the Python standard library. By using these methods, Mx™ will *not* enter the global script

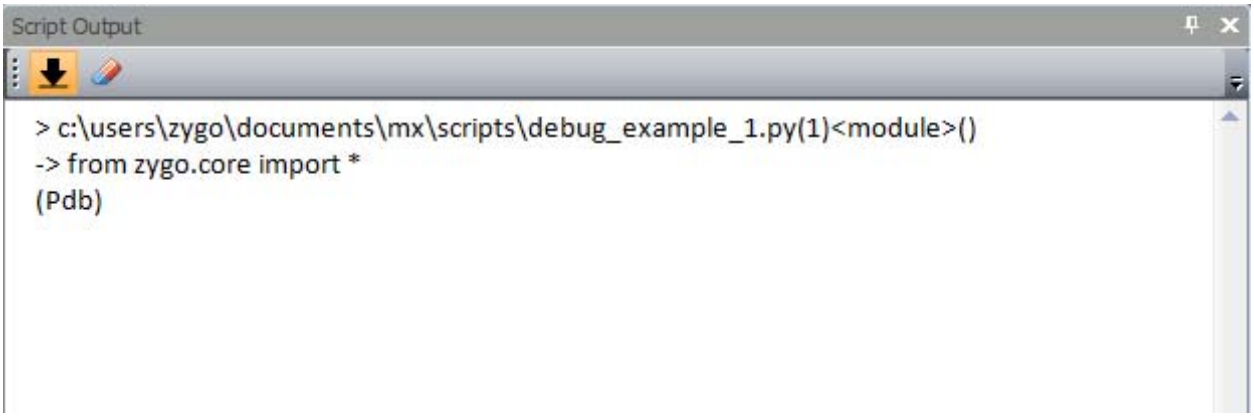
debug mode. The `pdb` module functions only affect the script(s) in which they exist. The same restrictions for debugging via the editor, described below, apply.

The allowed debugger methods are `pdb.set_trace()`, to enter the debugger at the calling stack frame, and `pdb.post_mortem(traceback=None)`, to enter post-mortem debugging of the given `traceback` object. See the official Python documentation for further information.

Scripts which make use of `pdb` module functions should be started via any of the normal methods of running a script in Mx™, and not by clicking the "Start Debugging" button in the Script Editor.

Debugging

Mx™ includes support for a subset of the Python 3.4 `pdb` module to facilitate script debugging. When debugging begins, execution of the script will pause before running the first line of executable code, and you will be greeted by the debugger prompt, **(Pdb)**, in the Script Output window:



The first line of the output, beginning with the greater-than sign (**>**), keeps track of the debugger's current location in three parts. The first part is the path of the current script:

c:\users\zygo\documents\mx\scripts\debug_example_1.py

Following the path is the line number, in parenthesis, of the next line to be executed when the debugger resumes execution. Here, it is the first line of the script. However, this may not always be the case if, for example, the script begins with comments or docstrings. Since the debugger will only stop on executable statements, the first debuggable line of code may be further down the script.

The last part of the first line is the name of the current function. **<module>()** is a special name indicating that the debugger is currently at the top-level of a script, outside of any functions.

The second line of the output, beginning with the arrow (**->**), is the next-statement pointer. This displays the line of code corresponding to the line number in parenthesis described above. This is the next line the debugger will process when execution resumes.

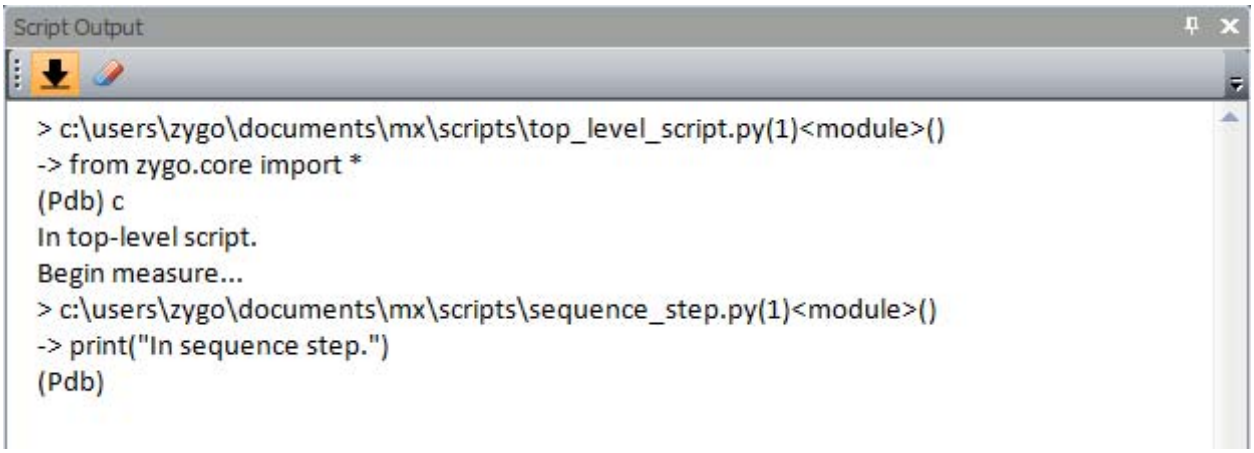
The third line of the output, **(Pdb)**, is the debugger's prompt, indicating that the debugger is paused waiting for user input.

Be aware that the next-line number and indicator (**->**) technically refer to the debugger's location in the current stack frame, which may not be the next line to execute. This distinction is made when navigating up or down frames in the stack trace.

Script Debug Mode

When in script debug mode, any scripts which are subsequently executed – directly or indirectly – will be run under the control of the debugger. When the first top-level script (i.e., the script that was active in the Script Editor when debugging began) exits, Mx™ will exit script debug mode. Since only one script can receive input in Mx™ at a time, the currently-executing script will have focus in the Script Output window.

In the following example, the script *top_level_script.py* is loaded and active in the script editor. This script will print a message and begin an instrument measurement. The measurement will trigger an analysis. A Surface Processing sequence Run Script step is added to Mx™ and configured to execute *sequence_step.py*. Mx is placed into debug mode, and the first script is executed, by clicking the "Start Debugging" button in the script editor.



Inspecting the output, it can be seen that the debugger paused on line 1 of *top_level_script.py*. The **c(ontinue)** command (described later) was issued by the user, which instructed to debugger to continue execution of the current script. The script printed its messages to the window and the `measure()` command was sent to Mx™. At this point, the script (and, therefore, the debugger) paused waiting for Mx™ to return control to the script process.

When the Run Script sequence step was encountered during the analysis phase of the measurement, the script, *sequence_step.py*, was executed under debug mode. The sixth line of the Script Output window shows that the debugger stopped on the first line of the corresponding script. At this point, the debugger has displayed its prompt and is waiting for input from the user. User input has been directed

to the second script. When this script terminates (not shown here), control will be directed back to the first script. Once the first script ends, debugging will stop and Mx™ will exit script debug mode.

Asynchronous Scripts

Special care must be taken when debugging scripts which contain asynchronous commands (e.g., measurements, motion). User input will always be directed to the process of the most-recently started active script. If a script is expecting input, and a second script begins execution, all input to the Script Output window will be directed to the second script. *This switch can occur while attempting to interact with the first script.* You will not be able to send standard input to the first script until the second script terminates execution. Output, however, is displayed as it is received *from all executing scripts.* Therefore, it is possible that output from different scripts may be interleaved.

Debugger Commands

As mentioned earlier, Mx™ includes support for a subset of the Python 3.4 `pdb` module to facilitate script debugging. Due to the nature of the way Python integrates with Mx™, there are certain cases in which behavior may diverge from what may be expected. Most notable are the following:

- The global symbol table dictionary returned by `globals()`, and the module-level local symbol table dictionary returned by `locals()`, will be populated with additional items both from Mx™ and from the debugger.
- The **run** and **restart** debugger commands are not available.

Following is a brief synopsis of the most common debugger commands available in Mx™, adapted from the Python documentation. Commands are case-sensitive. Many commands can be abbreviated. For example, the command **w(here)** means that either **w** or **where** can be used to enter the where command. Arguments to the commands, where they exist, follow the command and are separated from the command by whitespace. Optional arguments are enclosed in square brackets ([]); the brackets themselves are not typed. For more details on debugging Python, including additional commands and options, consult the official documentation, available at:

https://docs.python.org/3.4/library/pdb.html#debugger-commands

Command	Description
h(elp) [command]	With no argument, prints a list of debugger commands (note that some commands may not be available in Mx™). With a *command* as argument, displays help on that command.
w(here)	Prints a stack trace, with the most recent stack frame at the bottom.
d(own) [count]	Move the current stack frame down *count* levels in the stack trace (to a newer frame). Defaults to one if *count* is omitted.
u(p) [count]	Move the current stack frame up *count* levels in the stack trace (to an older frame). Defaults to one if *count* is omitted.
b(reak) [lineno | function]	Lists or sets breakpoints in the current file.

	With a *lineno* argument, sets a break at the specified line number. With a *function* argument, sets a break at the first executable statement within that function. With no arguments, lists all breakpoints.
cl(ear) [bpnumber [bpnumber …]]	With no argument, clears all breakpoints in the current script, asking for confirmation. With a space separated list of breakpoint numbers, clears those breakpoints.
s(tep)	Executes the current line. If the line contains a function call, enters that function and stops at the first executable statement, if possible. Otherwise, continues execution until the next line in the current function is reached or it returns.
n(ext)	Continues execution until the next line in the current function is reached or it returns. Unlike the step command, **next** will not step inside any called functions.
r(eturn)	Continue execution until the current function returns.
c(ont(inue))	Continues execution, stopping only if a breakpoint is encountered.
j(ump) lineno	Sets the next line that will be executed.
l(ist) [first[, last]]	Lists the source code for the current file. With no arguments, lists the 11 lines centered on the current line. Subsequent **list** commands will continue the previous listing. With . as the argument, lists the 11 lines around the current line. With two arguments, lists the given range or, if *last* is less than *first*, interprets the second argument as a count. The current line in the current frame is indicated by an arrow (->).
ll	Lists all source code for the current function or frame.
p expression	Evaluates the expression in the current context and prints its value.
pp expression	Similar to the **p** command, except the value is pretty-printed using the `pprint` module.
whatis expression	Prints the type of the expression.
! statement	Executes a single-line statement in the context of the stack frame. The exclamation point can be omitted unless the first word of the statement is a debugger command.
q(uit)	Quits the debugger. The script being executed is aborted.

N

O

P

R

S

T

U

W

Z

Zygo Corporation

Middlefield, Connecticut United States

Phone: 860-347-8506 or 800-994-6669

E-mail: inquire@zygo.com

Please visit our website for other locations.

www.zygo.com