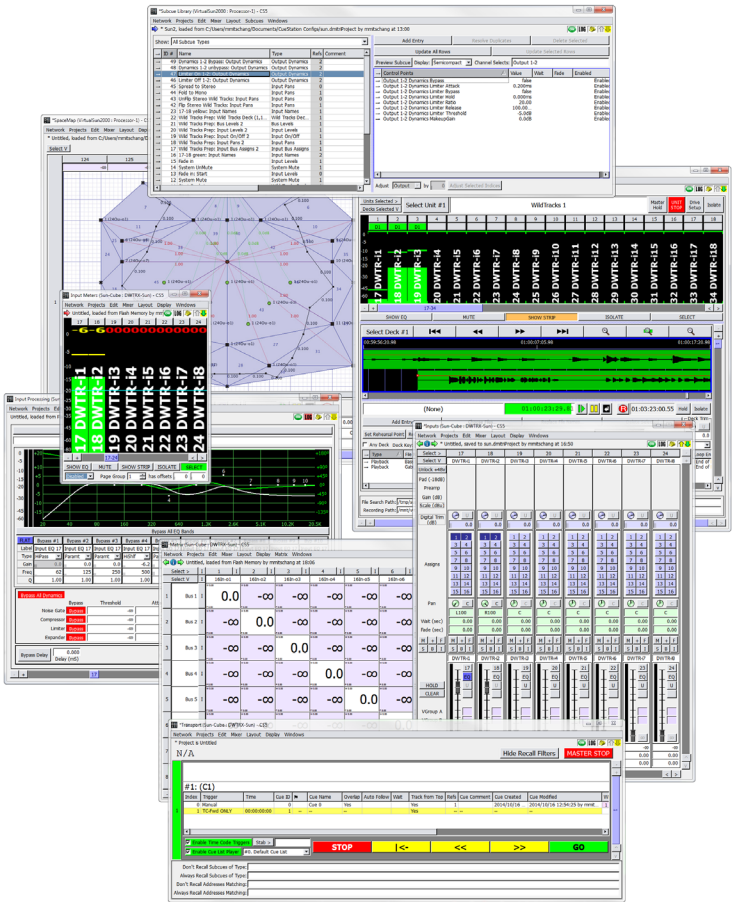
CueStation Server-Side Python

thinking sound Meyer Sound



python™

Keep these important operating instructions.
Check www.meyersound.com for updates.

CueStation Server-Side Python User Guide, PN 05.176.131.01 A

CONTENTS

CHAPTER 1: INTRODUCTION

HOW TO USE THIS MANUAL

As you read this user guide, you will encounter the following icons for notes, tips, and cautions:

NOTE: A note identifies an important or useful piece of information relating to the topic under discussion.

TIP: A tip offers a helpful tip relevant to the topic at hand.

CAUTION: A caution gives notice that an action may have serious consequences and could cause harm to equipment or personnel, or could cause delays or other problems.

Information and specifications are subject to change. Updates and supplementary information are available at www.meyersound.com.

Meyer Sound Technical Support is available at:

- **Tel:** +1 510 486.1166
- **Tel:** +1 510 486.0657 (after hours support)
- **Web:** www.meyersound.com/support
- **Email:** techsupport@meyersound.com

OVERVIEW

This document describes how Python® scripting is used for automation purposes with D-Mitri and CueStation 5.

Note that this document assumes at least a basic level of understanding of the Python scripting language. If you are unfamiliar with Python, it's recommended that you spend some time learning basic Python before continuing with this document.

There are plenty of good Python tutorials and documentation available online and offline, including these:

- Learning Python
- Byte of Python
- Begin Python

PYTHON AND CUESTATION 5

D-Mitri and CueStation 5 use Python for various internal purposes (such as populating the GUIs in the Command Subcue editor, and upgrading old projects). In addition, they can include Python scripts as part of a .dmitriProject file and run them from the CueStation GUI. This is useful, because there are sometimes specialized, application-specific automation requirements that don't fit easily into D-Mitri's regular Cue List automation structure. In these instances you can write a short Python script that implements the custom logic. The script can then be integrated into the project and executed when necessary. This greatly expands the set of possible behaviors that a D-Mitri system can be programmed to perform, and avoids the need to introduce external controllers or PCs to implement custom high-level logic.

CHAPTER 2: RUNNING A SERVER-SIDE PYTHON SCRIPT

RUNNING PYTHON SCRIPTS FROM THE COMMAND LINE

Python is integrated into D-Mitri and CueStation 5 by embedding a standard Python 2.7.5 parser and runtime into the executables. This embedded Python parser is the same as the stand-alone Python parser that you can download from www.python.org. You can run the embedded Python yourself from the command line, and interact with it just like the regular Python parser. The following is an example output from a Terminal window.

```
    jeremy-friesners-mac-pro-3:~ jaf$ cd /Applications/CueStation\ 5.5.2
    jeremy-friesners-mac-pro-3:CueStation 5.5.2 jaf$ ./VirtualD-Mitri.app/Contents/MacOS/
VirtualD-Mitri python
    Python 2.7.5 (r26:7891, Apr 21 2011, 03:06:04)
    >>> print "hello!"
    hello!
```

The same Python parser is embedded in to the dmitrid daemon that runs on all D-Mitri Modules (other than I/O) modules. If you SSH into a D-Mitri module, the parser can be run as follows:

```
    msli-DCP-10330685:~# /opt/msli/bin/dmitrid python
    Python 2.7.5 (r26:66714, Jun 16 2011, 09:41:41)
    >>> print "I'm on real hardware!"
    I'm on real hardware!
```

Running the Python interpreter interactively like this is a quick way to test python syntax, or to follow along with a Python tutorial, but for development you will want to run a Python script file that you've created, not type Python code directly into the interpreter. This works just like it does with the standard Python interpreter: you can just specify the name of the script file you want to run, and it will run it. For example, a file named hello.py is in the home directory, and it contains this script:

```
    print "Hello, World!"
```

That script can be run in VirtualD-Mitri's interpreter, for example:

```
    jeremy-friesners-mac-pro-3:CueStation 5.5.2 Beta jaf$ ./VirtualD-Mitri.app/Contents/
MacOS/VirtualD-Mitri python ~/hello.py
    Hello, World!
```

Running Python scripts from the command line is very efficient for developing and trouble-shooting the Python script, because you can simply keep the script file open in your favorite text editor, and whenever you make a change to the script you can simply choose “Save” in the text editor, then re-run the command in the Terminal window (using the up arrow key so you don't have to type it again, of course) to see if it works.

RUNNING PYTHON SCRIPTS FROM THE CUESTATION GUI

Running Python scripts from the command line is not very practical during a show, so once you have your Python script working, the next step is integrating that Python script into your CueStation 5 project file.

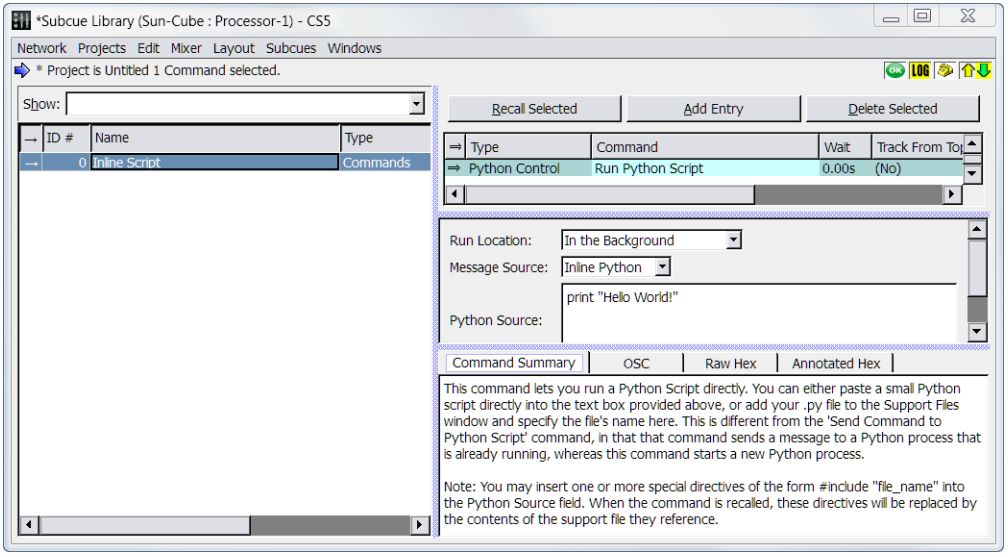
There are multiple ways to run server-side Python scripts in CueStation. The following methods integrate Python scripts into your CueStation 5 project file and allow for multiple ways of running scripts.

Running Server-Side Scripts Inline

To add very short scripts (fewer than a few dozen lines):

1. Copy the text from your Python Script file.
2. In CueStation’s Subcue Library window, choose Subcues > New Subcue > Commands to create a new Command subcue.
3. Click Add Entry to create a new subcue entry. Set the Type to Python Control, then set the Command to Run Python Script.
4. Set the Message Source field to Inline Python.

5. Paste the text from your Python Script file into the “Python Source” text box. Whenever you recall that subcue, your python script will be executed. You can even modify/edit the script directly inside the subcue.

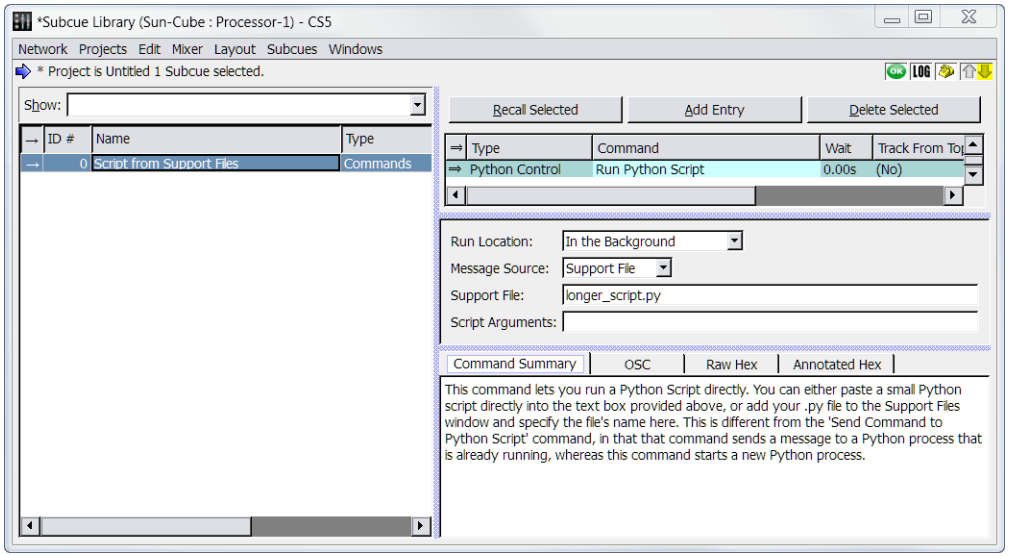


Running Server-Side Scripts with a Support File

To run longer scripts, or scripts that you want to recall from various subcues without having to make multiple copies of the script:

1. Drag the script file into CueStation’s Support Files window.
2. In CueStation’s Subcue Library window, choose Subcues > New Subcue > Commands to create a new Command subcue.
3. Click Add Entry to create a new subcue entry. Set the Type to Python Control, then set the Command to Run Python Script.

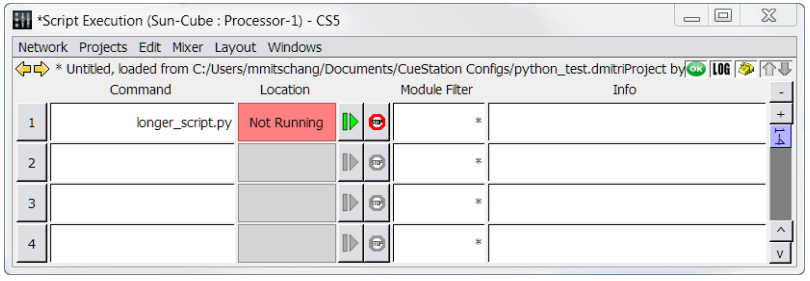
4. Set the Message Source field to Support File. Enter the filename of the script that you added to Support Files.



Running Server-Side Scripts from the Script Execution Window

To run a server-side Python script using the Script Execution window:

1. Drag the script file into CueStation's Support Files window.
2. In CueStation's Script Execution window, type the name of your support file into the Command column and press Enter. The .py extension is optional.



3. Click the Play button to run the script, and the Stop button to stop a script that is currently running.

Once you have one or more scripts running in the Script Execution window, it is possible to capture those settings into a Cue by either using the Capture window to capture a subcue of type Script Execution, or Control-drag from the controls in Script Execution window into the Cue Library or Subcue Library window.

INCLUDED PYTHON SCRIPTS

CueStation 5 and D-Mitri firmware ship with a number of server-side Python scripts included. These scripts can be used as-is, and they also serve as a set of examples showing how to implement various useful tasks as Python scripts under D-Mitri. A quick way to get started is to make a copy of a script file, modify the copy to suit your needs, then import the copy into the Support Files window.

For a detailed list of the included scripts, see Appendix A, “Included Server-Side Python Scripts.”

CHAPTER 3: WRITING PYTHON SCRIPTS

WRITING SCRIPTS FOR D-MITRI

When neither D-Mitri's built-in functionality nor any of the included example Python scripts are sufficient to meet your needs, the next step is to write your own Python script. Since the Python script execution mechanism used by D-Mitri is a vanilla Python interpreter process with your script's name as an argument, any valid Python script can be run in D-Mitri.

There are specific uses for a D-Mitri python script, such as connecting to dmixerd and or dcued, subscribing to control points, setting control point values, printing messages to the log, and so on. While it would be possible to write routines yourself that do all of these things, it isn't necessary, since D-Mitri (and VirtualD-Mitri) ship with these routines already written and included, so that all your script needs to do is call them and let them take care of the details.

This Python support code is included in the various sub-directories of the templates folder that start with an underscore ("_"). You are encouraged to examine the files in these directories; most of them contain extensive commenting describing what they do and how to use them. To locate the templates folder, see the directories listed in Appendix A, "Included Server-Side Python Scripts."

The directories are as follows:

Directory	Description
_python	Contains the standard (non-D-Mitri-specific) support files that ship with the Python interpreter.
_muscle	Contains files that support the MUSCLE networking protocol (used to connect to dcued and dmixerd).
_meyer	Contains files defining various D-Mitri-specific data structures and classes.
_osc	Contains code for constructing OpenSoundControl (OSC) messages.
_support	Contains some miscellaneous support code.
_scripts	Contains various example/utility Python scripts (as described in the previous section).

There are various ways to write a Python script for D-Mitri, but the BasicClient class is the easiest and quickest way to get the job done. BasicClient serves as a simple gateway to the standard D-Mitri script functionality. In particular, it will connect to dcued and dmixerd for you, and send and receive MUSCLE messages to and from those daemons on your behalf. By doing that, it gives your script most of the options available to any CueStation window. For example, your script can set control point values, subscribe to control point values (so that your script will be notified when they change) or print messages to the Log.

Let's take a look at a simple BasicClient-based Python script from the _scripts folder, fader_to_trim.py. When this script runs, it subscribes to the first 128 Input Level control points. Whenever the script is notified that an Input Level control has changed, the script responds by setting that input's Trim control point to the same value.

Before looking at the script below, you might want to take a moment to run the script in the Script Execution window, to see its effects in the Inputs window as you drag Input faders up and down.

Here is the fader_to_trim Python script (aka templates/_scripts/fader_to_trim.py):

```
    1    from dmitri_script import *
    2
    3    # A very simple script:  whenever an input fader is moved, we set its trim knob to
the same value!
    4    class FaderToTrimClient(BasicClient):
    5       def ConnectedToServer(self):
    6          self.Log("Python script fadertotrim.py has connected to the servers!  Try
moving the input faders!", MUSCLE_LOG_WARNING)
    7          self.AddMixerSubscription(BlockAddressSet(ControlPointAddress(ciInput,
ciLevel, 0), ControlPointAddress(ciInput, ciLevel, 127)));
    8
    9       # This method is called whenever a subscribed value is changed
    10      def ControlPointValueUpdated(self, address, value):
    11         if value.GetValueType() == VALUE_TYPE_FLOAT:
    12            self.SetControlPointValues(SingleAddressSet(ControlPointAddress(ciInput,
ciTrim, address[2])), SingleValueSet(value))
    13
    14   # Execution starts here. Just create a FaderToTrimClient object and run it!
    15   if __name__ == "__main__":
    16      FaderToTrimClient().RunEventLoop(
```

The first line tells the Python interpreter to first read and parse the contents of the dmitri_script file (templates/_meyer/dmitri_script.py). The dmitri_script file, in turn, will tell the Python interpreter to read in a number of other files that are commonly used by BasicClient and friends. Including all of the common files this way saves you from having to specify which files are relevant explicitly, so this is a good line to include at the top of any BasicClient-based script.

Line #3 is a comment, and has no effect on the execution of the script. It's there only to help a human reader understand the script's purpose.

Line #4 tells Python that we want to define a subclass of the BasicClient class. Our subclass will be named FaderToTrimClient, and it will contain all the functionality of a regular BasicClient, plus some additional functionality which we will define just below.

Line #5 defines a method in our FaderToTrimClient class, called ConnectedToServer. This is an example of a “hook” method (a method that you define) that the BasicClient will call at the expected time. In this instance, the BasicClient class is guaranteed to call a method named ConnectedToServer when the script has successfully connected to the server daemons. Our implementation of ConnectedToServer does two things: It writes a warning message to the Log (Line #6), and then it subscribes to the first 128 Input Level control points. By making this subscription, the script is requesting to be notified of (a) those control points' current values, and (b) those control points' updated values, any time those control points change in the future.

Line #10 defines a second “hook” method. In this case, we define ControlPointValueUpdated, which is a method that BasicClient will call whenever it receives a control-point-value update from dmixerd. Note that this method takes two additional arguments (in addition to Python's required “self” argument): (address), and (value). When called, (address) will be a ControlPointAddress object representing the address that changed, and (value) will be a ControlPointValue object representing the new value associated with that address.

Lines #11 and #12 implement the body of the ControlPointValueUpdated method. Line #11 is just a sanity check to make sure that the received value is actually a floating point value. Assuming the type was as expected, Line #12 is executed. Line #12 tells D-Mitri to set the Input Trim control point of the same Input channel to the same value, which is the behavior we wanted for this script.

The final lines (#15 and #16) are a bit of boilerplate that need to be added to the bottom of all BasicClient Python scripts. These final two lines are necessary to actually start the process of code execution happening, since the lines previous to these only defined a class with some code in it.

Line #15 checks to make sure that this file is actually the file that is intended to be run (and in particular that we're not just importing this file from a different file without intending to run any of the code in it right now). Given that it is true, Line #16 creates a FaderToTrimClient object, and starts its event loop running by calling its RunEventLoop() method. RunEventLoop() won't return until the Python script is ready to go away, but internally it will be setting up network connections, sending and receiving MUSCLE messages to the dmixerd and dcued daemons, and calling the various “hook” methods that are defined in basic_client.py when their associated events have occurred. All you need to do is define those hook methods in your subclass if you want them to do something particular in response to an event.

APPENDIX A: INCLUDED SERVER-SIDE PYTHON SCRIPTS

INCLUDED FILES

CueStation 5 and D-Mitri firmware ship with a number of server-side Python scripts included. These scripts can always be run using any of the techniques described in Chapter 2, “Running a Server-Side Python Script.” since they are in the Python interpreter's default path. These files are located in the following directories:

- Windows 32-bit operating systems:
  C:\Program Files\Meyer Sound\CueStation 5\templates_scripts
- Windows 64-bit operating systems:
  C:\Program Files (x86)\Meyer Sound\CueStation 5\templates_scripts
- Mac OSX operating systems:
  right click the VirtualD-Mitri icon in Finder and choose Show Package Contents, then navigate to *Contents/Resources/templates/_scripts*

These scripts can be used as-is, and they also serve as a set of examples showing how to implement various useful tasks as Python scripts under D-Mitri.

A list of the included Python scripts is as follows:

Script	Actions
adjust_cuelist_timecode.py	Adds a specified number of SMPTE frames to all the values in the “Time” column of the specified Cue List. Usage Example: `adjust_cuelist_timecode cuelist=0 delta=100`
adjust_level.py	Adds or subtracts a specified number of decibels to the specified fader control points. Usage Example: `adjust_level input 1-8 Level +5.4`

Script	Actions
batch_put_subcues.py	Adds or removes the specified subcues to/from the specified cues. Usage Example: `batch_put_subcues put=1,3-6 remove=9-10 cues=15-20,30 cuelist=1 cuelist=2:4-8` (puts subcues 1,3,4,5,6 into cues 15-20 and 30, plus all the cues in cuelist #1 and cuelist #2 indices 4-8, and remove subcues 9 and 10 from same)
capture_differences_by_type.py	Tells dcued to “Capture Differences” for all subcues of the specified type(s) in the specified cue. Usage Example: `capture_differences_by_type 16 Input Levels, Input Trims` (captures differences for subcues of type Input Levels and Input Trims in cue #16)

Script	Actions

conditional_recall.py	This script allows for some specified Python code to be executed. When executed, the specified Python code is able to access the values of various specified control points, which it can use to decide what it wants to do (recall a cue, print a message to the log, or anything else a Python script can do).
	
	Usage Example: This script is intended to run in the Script Execution window with no arguments. Once it is running, you can recall one or more "Python Control / Send Command To Python Script" commands to send instructions to the running conditional_recall script. One such conditional recall script is as follows:
	`input 1-2 Level`
	
	`input 3 Trim`
	
	`if (val("Input 1 Level") >= 0.0) and (val("Input 3 Trim") < -0.0):`
	
	`print "Condition met! Input 2 Level is: ", val("Input 2 Level")`
	
	`self.RecallSubcue(123)`
	Note that the instructions are composed of two parts: Lines listing the control points that the code is interested in the current values of, then a blank line, and then the Python code to execute once the values of the specified control points have been discovered. In this (contrived) example, the included Python code checks to see if "Input 1 Level" is greater than or equal to 0dB, AND if "Input 3 Trim" is less than 0dB. If both conditions are true, the code then prints a debug message to the log and recalls subcue #123.

copy_addrs.py	This script copies the values from one set of control points to another set of control points. Usage Example: `copy_addrs Input 1-10 Trim to Input 11-20 Trim` (Sets the values of Input 11-20 Trim equal to the values of Input 1-10 Trim, respectively)

Script	Actions
fader_to_trim.py	A trivial example script. When running, it watches the first 128 Input Level control points, and whenever one of those values changes, it sets the Input Trim control point of the same channel to the same value. This script's purpose is just to demonstrate how control point subscriptions work, not to be useful. Usage Example: `fader_to_trim`
hello.py	A trivial example script. When run, it prints a message to the Log, then counts to 100 in the “Info” cell of its row in the Script Execution window, then exits. Usage Example: `hello`
master_cps.py	This script implements “Master Control Point” logic for a given set of master and slave control points. Specifically, it subscribes to the specified set of “slave” control points, and watches their states. Whenever more than a specified number of “slave” control point values are set to true at the same time, the “master” control points get set to true also. Whenever fewer than the specified number of “slave” control points are set to true, the “master” control points get set to false. Usage Example: `master_cps Input 1-8 Solo / Input 9-16 Solo / 1` (Causes Inputs 1-8 Solo to be set to true if and only if at least one of the control points in the “Input 9-16 Solo” set has a value of true)
mini_page.py	This script watches a specified Page Group, and whenever the page group's value changes, this script recalls a specified cue or subcue. This can be useful if you want to invoke specified behaviors (custom CC2 mappings, for example) that are associated with various pages in a page group, in addition to the usual control-points-mapping behavior. Usage Example: `mini_page s1,s5,s10,c20,c30; setpagegroup 1` (Will recall subcue #1 when page Group 1 is at position 1, subcue #5 at position 2, and so on)

Script	Actions
print_config.py	Trivial example script. This script prints the Active Config to the log, and also demonstrates iterating over all the Input indices in the Active Config, by printing each of them to the log in sequence. Usage Example: `print_config`
print_events.py	Trivial example script. This script subscribes to Events matching the specified wildcard pattern, and whenever an Event notification is received, it prints the event string to the log. Usage Example: `print_events *`
python_client.py	This script demonstrates how a Python script can connect to the dcued and dmixerd daemons without using the BasicClient class. It's meant to be run from a Terminal window, since it accepts commands from stdin, and prints results to stdout. BasicClient is almost always a better way to implement communication between the Python script and the daemons, however, since it hides the networking implementation details and is therefore much simpler to work with.
random_playlist.py	This script walks a specified directory tree and compiles a list of files in that tree. It then shuffles that list into a random order, and uploads it as a Wild Tracks track to the specified Wild Tracks deck. Note that this script generally needs to run on the Wild Tracks module, since that is where the directory tree will be. Usage Example: `random_playlist unit=1 deck=1 dir=/mnt/wtdrives/left/wtrxaudio voices=1,2`
recall_random_cue_list_entry.py	This script is meant to run in the Script Execution window. When it runs, it subscribes to the Cue List contained in the specified Cue List Player, and when it receives a User Command “doit” (for example via a “Python Control / Send Command To Python Script” command) it recalls one of the Cue List entries in that Cue List, at random. `Usage Example: recall_random_cue_list_entry 1` (will set up the script to be ready to recall random entries from the Cue List in Cue List Player #1)

Script	Actions
send_chat_text.py	Trivial example script, to generate how a Python script can send “chat” text to the CueStation Chat window. This script sends a single Chat message to the Chat window, then exits. Usage Example: `send_chat_text Sssssssoooooo... you thought you could hide from me, Potter?`
test_basic_client.py	Trivial example script, to serve as a demonstration of how to subclass a BasicClient to implement custom control point logic. When run, this script subscribes to various Input Fader control points, and whenever a particular Input Fader is moved, it responds by moving other Input Faders in an inverse fashion. It also causes the Input Names to change over time, in a counting-to-100 fashion. Usage Example: `test_basic_client`
time_of_day.py	This script implements time-of-day cue-recalls for D-Mitri. To use it, you start it running with arguments specifying which cues/subcues are to be recalled at which times of day. Usage Example: `timeofday c25@4:32PM s33@16:09:24 c2@3:15AM [...]`

APPENDIX B: BasicClient CLASS DETAILS

This section gives some more information about the methods in BasicClient class. This section is meant as a summary only; for complete documentation of these methods, refer to the templates/_meyer/basic_client.py file and read the method-header-comments written at the top of each method's definition.

The BasicClient class is meant to be subclassed by your script, so that your script can provide its own code to be executed in response to `method` calls by the BasicClient class event loop. With that in mind, we can divide the methods in the BasicClient class into two categories:

- Utility Methods - called by your code in order to get BasicClient to do something. See “Utility Methods” on page 24.
- Hook Methods - called by the BasicClient event loop at the expected times. You have the option of implementing in your subclass, in order for your code to be executed at those times. See “Hook Methods” on page 36.

UTILITY METHODS

The following is a list of the “utility methods” available in the BasicClient class:

RunEventLoop(self, servername=None)

This method is typically the only method you'll ever call on your BasicClient-subclass object from outside the object itself. This method runs the event loop that handles all of the network and event processing for your script, and won't return until the script is ready to quit (for more information, see “Writing Scripts for D-Mitri” on page 13).

Usage Example:

```
    # typically this will be at the bottom of your script
    if __name__ == “__main__”:
       MyBasicClientSubclass().RunEventLoop()   # Creates a MyBasicClientSubclassObject, and
runs the event loop
```

GetArgumentsAsString(self)

A convenience method. This returns the arguments that this script was launched with, as a single string. For example, entering `myscript a b c` into the Script Execution window and pressing the Play button results in this method returning `a b c`.

Usage example:

```
    print “This script was run with the following arguments:  ”, self.GetArgumentsAsString()
```

GetArgumentsAsList(self)

A convenience method. This returns the arguments that this script was launched with, as a Python List. For example, entering `myscript a b c` into the Script Execution window and pressing the Play button results in this method returning `['a', 'b', 'c']`.

Usage example:

```
    print “This script was run with the following arguments:  ”, self.GetArgumentsAsList()
```

GetArgumentsAsDictionary(self)

A convenience method. This returns the arguments that this script was launched with, as a Python Dictionary. For example, entering `myscript a b=25 c=hello` into the Script Execution window and pressing the Play button results in this method returning `{'a': '', 'b' :'25', 'c' :'hello}`.

Usage Example:

```
    print "This script was run with the following arguments:  ",
self.GetArgumentsAsDictionary()
```

GetArgumentAsString(self, argName, defVal=None)

A convenience method. This returns the value associated with a named argument, converted to a string. For example, if a script is running with the arguments `foo=bar a=1 b=0 c=5.3`, the method `GetArgumentAsString("foo")` returns `bar`.

GetArgumentAsInt(self, argName, defVal=None, base=0)

A convenience method. This returns the value associated with a named argument, converted to an integer. For example, if a script is running with the arguments `foo=bar a=1 b=0 c=5.3`, the method `GetArgumentAsInt("a")` returns `1`.

GetArgumentAsBool(self, argName, defVal=None)

A convenience method. This returns the value associated with a named argument, converted to a boolean value. For example, if a script is running with the arguments `foo=bar a=1 b=0 c=5.3`, the method `GetArgumentAsBool("b")` returns `False`.

GetArgumentAsFloat(self, argName, defVal=None)

A convenience method. This returns the value associated with a named argument, converted to a floating point number. For example, if a script is running with the arguments `foo=bar a=1 b=0 c=5.3`, the method `GetArgumentAsFloat("b")` returns `5.3`.

GetExecutionIndex(self)

Returns the “execution index” of this script, shown in the row of the Script Execution window that this script is executing in. If this script is executed outside the Script Execution window, this method returns `-1`.

Usage Example:

```
    if (self.GetExecutionIndex() >= 0):
      print “This script is running in row #%i in the Script Execution window.” %
(self.GetExecutionIndex()+1)
    else:
      print “This script is running in the background.”
```

InvalidatePulseTime(self)

Calling this method tells BasicClient that you've changed your mind about when it should next call the `Pulse()` hook method. Soon after you call this method, BasicClient calls your `GetPulseTime()` hook method again, and your `GetPulseTime()` code can then return the new time at which `Pulse()` should be called next.

Usage Example:

```
    self._nextPulseTime = time.time()  # assume GetPulseTime() has been defined to return
self._nextPulseTime
    self.InvalidatePulseTime()
    # self.Pulse() is now scheduled to be called on the next iteration of RunEventLoop()'s
event loop
```

SetControlPointValues(self, addresses, values, flags=0)

Call this method to set a specified set of control points to a specified set of values. Note that the arguments take set-objects (SingleAddressSet, or ListAddressSet, or TableAddressSet, and for values a SingleValueSet, or ListValueSet, or etc), so that you can set multiple control points at once if you want to do so.

Usage Examples:

```
    self.SetControlPointValues(SingleAddressSet(ControlPointAddress(ciInput, ciLevel, 0)),
SingleValueSet(ControlPointAddress(VALUE_TYPE_FLOAT, -90.0)))
    self.SetControlPointValues(ParseAddressSet(“Input 1-8 Mute”),
SingleValueSet(ControlPointAddress(VALUE_TYPE_BOOL, True)))
```

AddMixerSubscription(self, addresses)

Call this method to subscribe to a specified set of ControlPointAddresses. Soon after you do this, your `ControlPointValueUpdated()` method will be called to let you know the current values of the addresses you specified; and it will be called again whenever any of the specified addresses' values change.

Usage Examples:

```
    self.AddMixerSubscription(ListAddressSet([ControlPointAddress(ciInput, ciLevel, 0),
ControlPointAddress(ciOutput, ciTrim, 5)]))
    self.AddMixerSubscription(ParseAddressSet("Bus 1-4 Output 1-4 Level"))
```

RemoveMixerSubscription(self, addresses)

Call this method if you want to cancel the subscription on one or more ControlPointAddresses that you had previously subscribed to. After calling this, you will no longer receive `ControlPointValueUpdated()` calls when those control points' values change. Note that the set of control points specified here do not need to exactly match the set previously passed to `AddMixerSubscription()`; control points can be subscribed to and unsubscribed from on an individual basis.

Usage Examples:

```
    self.RemoveMixerSubscription(ListAddressSet([ControlPointAddress(ciInput, ciLevel, 0),
ControlPointAddress(ciOutput, ciTrim, 5)]))
    self.RemoveMixerSubscription(ParseAddressSet("Bus 1-4 Output 1-4 Level"))
```

RemoveAllMixerSubscriptions(self)

Call this method to cancel ALL control point subscriptions this script had previously set up. After calling this, you won't get any more `ControlPointValueUpdated()` calls until you resubscribe.

Usage Example:

```
    self.RemoveAllMixerSubscriptions()
```

RequestControlPointValues(self, requestKey, addresses, flags = 0)

This method lets you request the current values of a set of control points, without setting up a subscription or getting notifications when they change later on. Soon after you call this method, the `ControlPointValuesRequestReturned()` hook method will be called, with arguments containing your requestKey, your set of addresses, and those addresses' associated values.

Usage Example:

```
    self.RequestControlPointValues("my-key-1", BlockAddressSet(ControlPointAddress(ciInput,
ciLevel, 0), ControlPointAddress(ciInput, ciLevel, 15)))
    self.RequestControlPointValues("borscht", ParseAddressSet("Bus 1-4 Output 1-4 Level"))
```

AddAutomationSubscription(self, path, quiet=0)

Call this method to subscribe to a specified set of nodes in dcued's Project Database. Soon after you call this method, your `DatabaseNodeUpdated()` method (and possibly also your `DatabaseNodeIndex*()` methods) inform you of the current contents of Project Database nodes that match your specified path. You'll also receive more calls to those methods later on, if the contents of those nodes changes.

Usage Examples:

```
    self.AddAutomationSubscription("default/cues/*")
    self.AddAutomationSubscription("default/subcues/*/*")
    self.AddAutomationSubscription("default/maps/0/trisets/*")
    self.AddAutomationSubscription("log/*")
```

RemoveAutomationSubscription(self, path)

Call this method to cancel Project Database subscriptions that you previously set up via `AddAutomationSubscription()`. Note that in the case, the path you specify should match a previous path string that you passed to `AddAutomationSubscription()`, since Project Database subscriptions are maintained per-path, not per-node.

Usage Example:

```
    self.RemoveAutomationSubscription("log/*")
```

RemoveAllAutomationSubscriptions(self)

Call this method to cancel ALL Project Database subscriptions that this script had previously set up. After calling this, you won't get any more `DatabaseNodeUpdated()` callbacks (or etc) until you resubscribe.

Usage Example:

```
self.RemoveAllAutomationSubscriptions()
```

RequestAutomationNodeValues(self, path)

This method is very similar to `AddAutomationSubscription()`, except that instead of setting up a persistent, “live” query, it executes a one-shot instantaneous query instead. Database nodes matching (path) will be downloaded exactly once, and `DatabaseNodeUpdated()` will be called once for each matching node.

RequestAutomationNodeSubtrees(self, queryPaths, batchID, maxDepth=-1)

This method allows for bulk-downloading of one or more entire subtrees of the automation database. This is a one-shot instantaneous query. It will result in `AutomationNodeSubtreesReturned()` being called exactly once, after the results have been received. `(queryPaths)` may be a single node-path string, similar to those passed in to `RequestAutomationNodeValues()`, or it may be a list or tuple containing several such strings. `(batchID)` is an arbitrary string that you may pass in; it will be passed back to `AutomationNodeSubtreesReturned()` verbatim, to help that code identify which call to `RequestAutomationNodeSubtrees()` caused that particular call to `AutomationNodeSubtreesReturned()`. If specified, `(maxdepth)` lets you limit the maximum depth of the downloaded node-subtrees to a specified depth. By default it is -1, meaning that the entire subtrees will be downloaded, no matter how deep they are.

AddEventSubscription(self, pattern)

Call this to request a notification whenever events are received whose event-name-strings match the specified pattern.

Usage Examples:

```
self.AddEventSubscription(“midi * ch 1 pg change bank 0 pg 2”)
self.AddEventSubscription(“midi * ch * note on note * vel *”)
self.AddEventSubscription(“*”)  # tell me about ALL the events!
```

RemoveEventSubscription(self, pattern)

Call this to cancel any previous event-subscriptions that match the specified pattern.

Usage Example:

```
self.RemoveEventSubscription("midi * ch * note on note * vel *")
```

RemoveAllEventSubscriptions(self)

Call this to cancel ALL previous event-subscriptions.

Usage Example:

```
self.RemoveAllEventSubscriptions()
```

Log(self, logtext, level=qnet_log_entry.MUSCLE_LOG_INFO)

Call this to print a line of text to the Log.

Usage Example:

```
self.Log("It's the end of the world as we know it", MUSCLE_LOG_CRITICALERROR)
```

SendPythonUserCommand(self, slotIndex, commandString)

Call this to send a string to a currently running Python script. `(slotIndex)` can be the slot number where the Python script is running in the Script Execution window (0-31), or one of the following special values:

- -1 = send to all python scripts (whether shown in the Script Execution window or not)
- -2 = send to all "foreground" python scripts (all scripts shown in the Script Execution window)
- -3 = send to all "background" python scripts (all scripts not shown in the Script Execution window)

`(commandString)` will be interpreted as Python source code by default, or it may have a special meaning if the receiving Python script has overridden its `UserCommandReceived()` method to specially parse the strings it receives.

SendPythonUserCommandToGUI(self, clientName, windowName, commandString)

This version of `SendPythonUserCommand()` command is used to send commands to client-side Python scripts.

- `(clientName)` A string indicating which client computers to send the command to. May contain wildcards.
- `(windowName)` A string indicating which window names on the matching client computers to send the command to. May contain wildcards.
- `(commandString)` will be interpreted as Python source code by default, or it may have a special meaning if the receiving Python script has overridden its `UserCommandReceived()` method to specially parse the strings it receives.

RunPythonScript(self, slotIndex, scriptText, scriptArgs)

Sends a command to run the specified Python script in the specified slot in the Script Execution window.

- `(slotIndex)` is the slot to run the script in (0-31), or -1 if you want it to run in the background.
- `(scriptText)` is the python source code to execute. If you wish to execute a Python file instead, you can do so by including one or more lines in the text. For example:

  ```
  #include "some_support_file.py"
  ```

  These tokens will be expanded into a copy of the contents of the specified file, if that file can be found in the Support Files window or Python path.
- `(scriptArgs)` A string containing any command line arguments you wish to pass to the script when it runs.

SendMessageToServer(self, msg, whichdaemon)

Call this to send a specific MUSCLE Message object to a particular server. Typically you don't need to call this directly, as the other methods will create and send the appropriate Messages for you; but if there is ever a function that the BasicClient API doesn't implement, this method allows you to send the necessary Messages yourself.

Usage Example:

```
    msg = Message(CUEPROJECT_COMMAND_GENERATEVLANREPORT)
    msg.PutInt32(CUEPROJECT_NAME_VENUEID, 0)
    self.SendMessageToServer(msg, "cued")   # ask cued to send back a VLAN report about Venue
#0 for us
```

EndEventLoop(self)

This method tells BasicClient's event loop that you'd like the event loop to quit now. Soon after you call this method, RunEventLoop() will return, at which point you can exit the Python script's process.

Usage Example:

```
    self.EndEventLoop()
```

ExecuteTextCommand(self, cmdStr)

This method lets you tell D-Mitri to execute an arbitrary ASCII text command (as described in the document cs5_text_commands.txt)

Usage Examples:

```
    self.ExecuteTextCommand("recall cue 5")
    self.ExecuteTextCommand("update subcue 17; update cue 199; log hello")
    self.ExecuteTextCommand("set Input 3-5 Level = -5.0")
```

RecallCueEntry(self, cueListID, entryID, instant=0, cuelistplayer=0)

This method lets you tell D-Mitri to recall a specified Cue Entry in a specified Cue List. Note that the Cue Entry is specified by its Entry ID number, not its position in the Cue List.

Usage Example:

```
    self.RecallCueEntry(0, 0)
```

RecallCue(self, cueID, instant=0, cuelistplayer=0)

This method lets you tell D-Mitri to recall a specified Cue.

Usage Example:

```
self.RecallCue(55)
```

RecallSubcueEntry(self, cueID, entryID, instant=0, cuelistplayer=0)

This method lets you tell D-Mitri to recall a specified Subcue Entry within a specified Cue.

Usage Example:

```
self.RecallSubcueEntry(55, 12)
```

RecallSubcue(self, subcueID, instant=0, cuelistplayer=0)

This method lets you tell D-Mitri to recall a specified Subcue.

Usage Example:

```
self.RecallSubcue(192)
```

UpdateCue(self, cueID, flags=0)

This method lets you tell D-Mitri to update a specified Cue.

Usage Example:

```
self.UpdateCue(46)
```

UpdateSubcue(self, subcueID, flags=0)

This method lets you tell D-Mitri to update a specified Subcue.

Usage Example:

```
self.UpdateSubcue(123)
```

SendTriggerEvent(self, eventName)

This method lets you tell D-Mitri to post an Event with a specified event-name string.

Usage Example:

```
    self.SendTriggerEvent("midi 0 ch 0 note on note 3 vel 5")  # fake a MIDI note-on event
```

Go(self, cuelistplayers=None)

This method lets you tell D-Mitri "press the GO button" on a specified Cue List Player.

Usage Examples:

```
    self.Go()    # Go on Cue List Player #1
    self.Go(1)   # Go on Cue List Player #2
    self.Go(ParseIndexRangeFromString("1-4,6-8",True))  # Go on a number of Cue List Players
```

Stop(self, cuelistplayers=None)

This method lets you tell D-Mitri "press the STOP button" on a specified Cue List Player.

Usage Examples:

```
    self.Stop()    # Stop on Cue List Player #1
    self.Stop(1)   # Stop on Cue List Player #2
    self.Stop(ParseIndexRangeFromString("1-4,6-8",True))  # Stop on a number of Cue List
Players
```

MoveBy(self, howMany, cuelistplayers=None)

This method lets you tell D-Mitri to move the upcoming-cue-index (aka the yellow bar) on a specified Cue List Player down or up by `(howMany)` rows.

Usage Examples:

```
    self.MoveBy(-1)    # Skip-to-previous on Cue List Player #1
    self.MoveBy(1, 4)  # Skip-to-next on Cue List Player #5
    self.MoveBy(-10000, 2)  # Skip-to-top on Cue List Player #3
    self.MoveBy(10000, ParseIndexRangeFromString("1-4,6-8",True))  # Skip-to-end on a number
of Cue List Players
```

StartWatchingSocket(self, sock, eventTypes)

This method is for advanced users who want their Python script to interact with one or more additional network sockets, in addition to the ones that BasicClient uses internally to connect to dcued and dmixerd. `StartWatchingSocket()` will cause your `SocketReadReady()` and `SocketWriteReady()` hook methods to be called at the appropriate times for the specified socket.

StopWatchingSocket(self, sock, eventTypes)

This method is used to cancel `SocketReadReady()` and `SocketWriteReady()` hook method callbacks on the specified socket.

StopWatchingAllSockets(self, eventTypes)

This method is used to cancel `SocketReadReady()` and `SocketWriteReady()` hook method callbacks on all sockets.

HOOK METHODS

The following is a list of the “hook methods” available in the BasicClient class. Methods in this list are meant to be called by the RunEventLoop() method in the BasicClient class, and so you generally won't want to call these yourself. These methods can be defined in your subclass, so your script will operate as desired when these methods get called.

ConnectedToServer(self):

Called when the script's TCP connections dmixerd and dcued have both been successfully completed and are ready to be used. This is often a good place to do setup work such as making any initial subscriptions, or printing a “Script is running now” type of message to the Log.

Usage Example:

```
    def ConnectedToServer(self):
       self.Log("The Python script is now running and connected to the servers!")
```

ControlPointValuesUpdated(self, addrs, values):

Called when a set of address/value updates have been received from dmixerd. Note that the default implementation of this method will parse out the sets and call `ControlPointValueUpdated()` for each address/value pair in the sets, so it's generally easier to redefine that method instead of this one.

Usage Example:

```
    def ControlPointValuesUpdated(self, addrs, values):
       print "Got addrs = ", addrs
       print "Got values = ", values
       BasicClient.ControlPointValuesUpdated(addrs, values)  # so ControlPointValueUpdated()
will be called too
```

ControlPointValueUpdated(self, address, value):

Called by `ControlPointValuesUpdated()` when a particular address/value update has been received.

Usage Example:

```
    def ControlPointValueUpdated(self, addr, value):
      print "Got addr = ", addr
      print "Got value = ", value
```

ControlPointValuesRequestReturned(self, requestKey, addrs, values):

Called when the data requested by a previous `RequestControlPointValues()` call has been received. Note that this call will have the same `(requestKey)` string as the one you passed in to the `RequestControlPointValues()` call, so that it is possible for you to determine which `ControlPointValuesRequestReturned()` call corresponds to which earlier `RequestControl-PointValues()` call.

Usage Example:

```
    def ControlPointValuesRequestReturned(self, requestKey, addrs, values):
      print "I got the response to my previous RequestControlPointValues(%s, %s) call!" %
(requestKey, addrs)
      print "Returned values are: ", values
```

EventOccurred(self, eventName):

Called when an event was posted that matches one or more of your subscribed event-name-string patterns.

Usage Example:

```
    def EventOccurred(self, eventName):
      print "Event [%s] occurred!" % eventName
```

DatabaseNodeUpdated(self, path, valueMsg):

Called when one of the nodes (whose node-path matches the path-pattern passed to a previous call to `AddAutomationSubscription()`) is added, or changed, or removed.

Usage Example:

```
def DatabaseNodeUpdated(self, path, valueMsg):
   if (valueMsg != None):
      print "Database node at [%s] has a new Message value:" % path
      valueMsg.PrintToStream()
   else:
      print "Database node at [%s] was deleted." % path
```

DatabaseNodeIndexCleared(self, path):

These methods are called to let you know when a node-index of an ordered-children/indexed node (the Log or a Cue List, for example) has been cleared.

Usage Example:

```
def DatabaseNodeIndexCleared(self, path):
   print "Node index of node [%s] was cleared." % path
```

DatabaseNodeIndexEntryInserted(self, path, insertedAtIndex, nodeName):

These methods are called to let you know when a node-index of an ordered-children/indexed node (the Log or a Cue List, for example) has been modified.

Usage example:

```
def DatabaseNodeIndexEntryInserted(self, path, insertedAtIndex, nodeName):
   print "Inserted node [%s] at index %i of path %s." % (nodeName, insertedAtIndex, path)
```

DatabaseNodeIndexEntryRemoved(self, path, removedAtIndex, nodeName):

These methods are called to let you know when a node-index of an ordered-children/indexed node (the Log or a Cue List, for example) has been removed.

Usage example:

```
def DatabaseNodeIndexEntryRemoved(self, path, removedAtIndex, nodeName):
   print "Removed node [%s] from index %i of path %s." % (nodeName, removedAtIndex, path)
```

AutomationNodeSubtreesReturned(self, batchID, subtreesMessage):

- Called after `RequestAutomationNodeSubtrees()` was called, when the downloaded subtrees have become available.
- `(batchID)` is the same string that was passed in to `RequestAutomationNodeSubtrees()`.
- `(subtreesMessage)` contains all of the downloaded subtrees' data. Each entry in `(subtreesMessage)` represents a single downloaded subtree (the field name is the subtree's root path).

UnknownMessageReceivedFromServer(self, msg, whichdaemon):

This method is called when the `RunEventLoop()` method of the BasicClient class receives a Message that it doesn't recognize. If your script does recognize the Message, it can implement an appropriate response by redefining this method in your subclass.

Usage Example:

```
def UnknownMessageReceivedFromServer(self, msg, whichdaemon):
   print "Got Unknown Message from daemon [%s]:" % whichdaemon
   msg.PrintToStream()
```

UserCommandReceived(self, cmd):

This method is called when a text command is sent to your Python script from elsewhere in the system, such as the recall of a “Python Control / Send Command To Python Script” Command Subcue. Note that the default implementation of this method tries to execute (cmd) as Python source code, but if you want to handle the incoming text differently, you can redefine this method in your subclass to do whatever you want instead.

Usage Example:

```
def UserCommandReceived(self, cmd):
   print "User sent me a command:  [%s]" % cmd
```

GetPulseTime(self, prevTime):

This method is called by the event loop when it wants to know at what time it should next call the `Pulse()` method. This method can either return None (like the default implementation always does) to indicate that `Pulse()` doesn't ever need to be called, or it can return a floating point value in seconds (using the same conventions as used by the Python `time.time()` function) indicating when `Pulse()` should next be called. This is useful for time-driven behavior (for example updating an animated display via a control point).

Usage Example:

```
    def GetPulseTime(self, prevTime):
      return time.time()+1.0   # Tell the event loop to call Pulse() exactly 1 second from
now
```

Pulse(self, scheduledTime):

This method is called at the time previously indicated by the return value of a call to `GetPulseTime()`. `GetPulseTime()` will be called again after `Pulse()` returns, and an additional call to `GetPulseTime()` can also be forced at any time by calling `InvalidatePulseTime()`.

Usage Example:

```
    def Pulse(self, scheduledTime):
      if (now >= self._quittingTime):  # assume self._quittingTime was previously returned
by GetPulseTime()
        print "Hey, time to go!"
        self.EndEventLoop()
```

SocketReadReady(self, sock):

This method is called when a watched user socket receives `select()` as ready-for-read inside the event loop.

SocketWriteReady(self, sock):

This method is called when a watched user socket receives `select()` as ready-for-write inside the event loop.

APPENDIX C: OTHER RELEVANT CLASSES

INCLUDED FILES

The BasicClient class is your primary interface to D-Mitri-specific functionality, but there are other classes that you may need to use from time to time. The intent and basic functionality of these classes is summarized here; for full documentation, see the comments in the relevant .py files.

Classes are listed below, categorized by the file they are declared in. Files are sorted roughly by how likely it is you are to need classes in those files, with the most-common files listed first.

File	Description
_meyer/dmitri_script.py	This file is a simple “front end” file that just imports a number of other commonly used files. For organizational purposes, it is advisable to include a “from dmitri_script import *” line at the beginning of all of your D-Mitri Python scripts.
_support/cp_indices.py	This file defines all of the tokens with a “ci” prefix that are used to make up the non-numeric portions of a ControlPointAddress. For example, ControlPointAddress(ciInput, ciLevel, 2) is internally expressed as an oct-tuple (-32758, -32744, 2, 0, 0, 0, 0, 0), but specifying the magic numbers for Input and Level directly like that would make your script difficult to decipher, so it's better to use these constants instead.
_meyer/basic_client.py	`BasicClient` This class is discussed at length in previous sections of this document.
_meyer/control_point_address.py	`ControlPointAddress` An object of class ControlPointAddress represents a single D-Mitri control point address. In D-Mitri, a control point address is a oct-tuple (a set of 8 numbers). Generally the first few numbers in the oct-tuple will be special negative values that indicate categories (see the cp_indices.py section above for an example), followed by zero or more non-negative tokens representing numeric indices within that category, followed by enough zeros to pad out the length of the address to 8 indices. Note that each index is considered to be a 16-bit signed value.
_meyer/control_point_value.py	`ControlPointValue` A ControlPointValue object is a typed variant object that holds exactly one value. The value can be a String, float, boolean, int32, int64, Point, Message, Config, Address, ByteBuffer, or a VList (which is a list of sub-values). The ControlPointValue object contains both the value itself, and a VALUE_TYPE_* tag that indicates what the value's current type is.

File	Description

_meyer/control_point_address_set.py	This file defines various classes that are containers for sets of ControlPointAddress objects. It is these container classes that can be passed in to AddMixerSubscription(), SetControlPointValues(), and other methods of the BasicClient class.
	
	`SingleAddressSet`
	This is an address set that always contains exactly one ControlPointAddress.
	
	`ListAddressSet`
	This is an address set that contains a list of ControlPointAddresses.
	
	`TableAddressSet`
	This is an address set that contains a dictionary, where ControlPointAddresses are the keys in the dictionary.
	
	`MultiAddressSet`
	This is an address set that contains zero or more “child” control point address sets. When iterated over, this address set will act as the concatenation of all of its child address sets.
	
	`BlockAddressSet`
	This is an address set that contains a “block” of addresses. A block of addresses is defined by giving a range of indices on one or more axes; for example “Input 1-8 Level” is a one-dimensional block containing 8 (implicit) ControlPointAddresses, whereas “Bus 1-4 Output 5-8 Level” is a two-dimensional block containing 16 (implicit) ControlPointAddresses.
	
	`IndexedBlockAddressSet`
	This is a more elaborate version of a BlockAddressSet. It works similarly to a BlockAddressSet, except that the ranges of indices aren't necessarily contiguous. For example, an IndexedBlockAddressSet could specify “Input 1,3,5 Level” which is a one-dimensional set containing 3 (implicit) ControlPointAddresses, or “Bus 1-4,6-9,12 Output 5-8,20-30,40-50 Level” which is a two-dimensional set containing 234 (implicit) ControlPointAddresses.

File	Description

_meyer/control_point_value_set.py	This file defines various classes that are containers for sets of `ControlPointValue` objects. It is these container classes that can be passed in to `SetControlPointValues()`, and are received in hook functions like `ControlPointValuesRequestReturned()`.
	
	`SingleValueSet`
	This is a value set that always contains exactly one ControlPointValue.
	
	`ListValueSet`
	This is a value set that contains a list of ControlPointValues.
	
	`MultiValueSet`
	This is a value set that contains zero or more “child” control point value sets. When iterated over, this value set will act as the concatenation of all of its child value sets.

_meyer/control_point_utilities.py	This file contains various useful functions dealing with control point address sets and control point value sets. In particular, it has commands for converting human-readable strings (such as “Input 5-10 Level”) into their equivalent control point address set objects and back, as well as a function for parsing individual control point address string, and functions for adding sets to Messages, and retrieving them back out again.
_meyer/config.py	`Config` An object of the Config class represents the current configuration of a D-Mitri system. It contains methods for finding out system details such as the number of inputs in the D-Mitri system, what their index values are, and so on for outputs, auxes, buses, and etc. Objects of type Config are generally only seen if your script subscribes to the (ciSystem, ciConfig) control point address, in which case you will receive the value for that address as a Config object. You can then use that Config object to set up subscriptions (etc) that automatically match the geometry of the configuration that is currently in use.

_meyer/command_subcue_data.py	`CommandSubcueData`
	This class represents the contents of a “Command” subcue in the Subcue Library. You would typically only use this class if you were subscribing to subcues (via AddAutomationSubscription()) or creating your own Command subcues to add to the project.
	
	`CommandEntry`
	This class represents on Command inside a Command subcue. A command subcue contains a list of these.

File	Description
_meyer/control_point_address_info.py	`ControlPointAddressInfo` This class represents information about a type of Control Point.
_meyer/instantiate_db_object.py	This file implements a `DBObjectFromMessage()` function, that takes a MUSCLE Message and creates and returns the corresponding Python object (QNetCue, QNetSubcue, etc) that was contained in that Message.

_meyer/mixer_parameter_subcue_data.py	This file defines classes describing Mixer Parameter subcues (all the subcues in the Subcue Library, except for Command subcue and Wild Tracks Deck subcues).
	
	`MixerParameterSubcueData`
	This class defines a single Mixer Parameter subcue.
	
	`MixerParameterAddressesEntry`
	This class defines one entry in a Mixer Parameter subcue (one row in the Mixer Parameter Editor GUI). Each MixerParameterSubcueData contains a list of these.
	
	`MixerParameterValueTarget`
	This class holds the value, wait time, and fade time for a particular MixerParameterAddressesEntry. Each MixerParameterAddressesEntry contains one of these.

_meyer/qnet_access_policy_header.py	`QNetAccessPolicyHeader` This class represents the header data of an Access Policy (the info that is seen in the table at the left hand side of the Access Policies window).
_meyer/qnet_access_policy_data.py	`QNetAccessPolicyData` This class represents the body data of an Access Policy (the info that is seen at the right-hand side of the Access Policies window when you click on an Access Policy).
_meyer/qnet_alias.py	`QNetAlias` This class represents a D-Mitri alias (as seen in the Aliases table in the Mixer Configuration window)
_meyer/qnet_channel_set.py	`QNetChannelSet` This class represents an Index Set (as seen in the Index Sets table in the Capture window). Note that the name of this class is an anochronism, `QNetIndexSet` would be a more accurate name.
_meyer/qnet_chat_entry.py	`QNetChatEntry` This class represents a single row of chat-text in the Chat window.

File	Description
_meyer/qnet_cue.py	`QNetCue` This class represents a single Cue (as seen in the table at the upper left of the Cue Library window). Note that this class does not contain the subcue entries or subcues that are logically "owned" by the cue; that data has to be retrieved separately (as `QNetSubcueEntry` and `QNetSubcueHeader` objects).
_meyer/qnet_cue_entry.py	`QNetCueEntry` This class represents a single Cue Entry (as seen in the table at the upper left of the Cue List window). Note that this class does not contain the cue that is referenced by the cue entry object; that data has to be retrieved separately (as a `QNetCue` object).
_meyer/qnet_cue_list.py	`QNetCueList` This class represents a single Cue List (each item in the combo box at the top of the Cue List window represents a `QNetCueList` object. Note that this class does not contain the cue entries that is referenced by the cue list object; that data has to be retrieved separately (as `QNetCueEntry` objects).
_meyer/qnet_db_object.py	`QNetDBObject` This class is a superclass that all of the other database-object classes derive from. This class contains the fields that all of the subclasses have in common, such as Name, Comment, Creation Date, Modification Date, Creator, Modifier, and so on. By keeping all of that data in a common superclass, we avoid code duplication.
_meyer/qnet_key_mapping.py	`QNetKeyMapping` This class represents a single Key Mapping (one row in the Key Mappings window).
_meyer/qnet_log_entry.py	`QNetLogEntry` This class represents a single Log entry (one row in the Log window).
_meyer/qnet_parameter_set.py	`QNetParameterSet` This class represents a single Control Point Set entry (one row in the Control Point Sets table in the Capture window). Note that the name of this class is an anachronism; QNetControlPointSet would be a more accurate name.
_meyer/qnet_protocol.py	This file contains various constants related to the D-Mitri system such as port numbers, command ID codes, etc.
_meyer/qnet_space_link.py	`QNetSpaceLink` This class represents a link between a speaker node and virtual or derived node, as seen in the SpaceMap window.

File	Description
_meyer/qnet_space_-map.py	QNetSpaceMap This class represents a single SpaceMap, as seen in the SpaceMap window. Note that this object does not contain the contents of the SpaceMap, only its basic information (such as its name). The contents would need to be downloaded separately as QNetSpaceNodes, QNetTrisets, QNetSpaceLinks, QNetSpacePics, etc.
_meyer/qnet_space_node.py	QNetSpaceNode This class represents a single speaker/virtual/derived/silent node, as seen in the SpaceMap window.
_meyer/qnet_space_pic.py	QNetSpacePic This class represents a single imported bitmap, as seen in the SpaceMap window.
_meyer/qnet_subcue_entry.py	QNetSubcueEntry This class represents a single subcue entry within a Cue (a row in the upper-right hand table of the Cue Library or Cue List windows). Note that this class does not contain the subcue itself.
_meyer/qnet_subcue_-header.py	QNetSubcueHeader This class represents the header information for a single subcue (a row in the left-hand table of the Subcue Library window). Note that this object does not contain the subcue's actual contents, which would need to be downloaded separately as a CommandSubcueData, WildTracksDeckSubcueData, or MixerParameterSubcueData object.
_meyer/qnet_subcue_-type_header.py	QNetSubcueTypeHeader This class represents the header of a Subcue Type (a row in the table at the upper left corner of the Subcue Types tab in the Capture window). Note that this object does not contain the contents of the Subcue Type, which would have to be downloaded separately as a QNetSubcueTypeData object.
_meyer/qnet_subcue_-type_data.py	QNetSubcueTypeData This class represents the contents of a Subcue Type (as in the upper right and lower portions of the Subcue Types tab in the Capture window, when a Subcue Type has been selected).
_meyer/qnet_support_file_-header.py	QNetSupportFileHeader This class represents the header data for a Support File (a row in the Support Files window). Note that this object does not contain the contents of the support file itself, which would have to be downloaded separately.

File	Description
_meyer/qnet_trajectory.py	`QNetTrajectory` This class represents a single Trajectory as seen in the SpaceMap window. Note that this class contains only the header data (such as the trajectory's name), not the trajectory nodes themselves, which would have to be downloaded separately as `QNetTrajectoryNodes`.
_meyer/qnet_trajectory_node.py	`QNetTrajectoryNode` This class represents a single node inside a Trajectory, as seen in the SpaceMap window.
_meyer/qnet_triset.py	`QNetTriset` This class represents a single triset inside a SpaceMap, as seen in the SpaceMap window.

_meyer/wild_tracks_deck_subcue_data.py	`WildTracksDeckSubcueData`
	This class represents a single Wild Tracks Deck subcue.
	
	`WildTracksDeckEntry`
	This class represents one entry in a Wild Tracks Deck subcue (one row as seen in the Wild Tracks Deck Editor). A `WildTracksDeckSubcueData` contains a set of `WildTracksDeckEntry` objects.
	
	`WildTracksGainEnvelope`
	This class represents a gain-envelope for a Wild Tracks Deck Entry. Each Wild Tracks Deck Entry may own zero or more `WildTracksGainEnvelope` objects.

_meyer/auto_select_state.py	`AutoSelectState` This class represents an object that remembers how to set Channel-Select control points when it is recalled. `MixerParameterSubcueData` and `WildtracksDeckSubcueData` both derive from this class.
_muscle/message.py	`Message` This class represents a single MUSCLE Message object. Messages are used to communicate data to and from the dmixerd and dcued server processes.
_muscle/message_transceiver_thread.py	`MessageTransceiverThread` This class implements an I/O thread that is used to send and receive Message objects to a remote location without slowing down the main scripting thread. The BasicClient class uses these objects behind the scenes on your behalf, so you won't typically need to interact with them directly.

File	Description
_muscle/storage_reflect_-constants.py	This file contains various MUSCLE command constants and field names. The BasicClient class uses these on your behalf, so you won't typically need to interact with them directly.
_muscle/zlib_utility_func-tions.py	This file contains functions for zlib-compressing and decompressing data. The other classes will call these functions on your behalf when necessary, so you won't typically need to interact with them directly.
_osc/OSC.py	This file contains functions for encoding OSC commands. Since BasicClient uses MUSCLE protocol and not OSC protocol, you won't typically need these functions for anything.
_support/client_con-stants.py	This file contains constants identifying each of the window types in CueStation 5.
_support/cs_constants.py	This file contains some constants used in conjunction with D-Mitri SpaceMap and Wild Tracks.

Meyer Sound Laboratories Inc.
2832 San Pablo Ave.
Berkeley, CA 94702

www.meyersound.com
+1 510 486.1166

CueStation Server-Side Python User Guide, PN 05.176.131.01 A