Musical MIDI Accompaniment

Plugins

Bob van der Poel
Wynndel, BC, Canada

bob@mellowood.ca

September 26, 2021

Table Of Contents

Chapter 1

Easy Guide

This is a DRAFT document.

This document is provided as a reference to advanced MMA users writing plugins. It does not pretend to be definitive or complete. It is a supplement to the main reference manual.

As mentioned in the MMA Reference Guide, a plugin must have its own directory. The name of this directory must be the same as the name you with to assign to the plugin (without the "@" prefix). So, the plugin @HELLO will have a directory `hello`.

In this directory you must have the following files:

1. `__init__.py` — this must be an empty file. Note that the name "init" has two leading and trailing underscores,
2. `plugin.py` — this contains your plugin code.

In the Reference Manual in the Plugins section, we discuss the fact that characters in filenames are, mostly, not case sensitive. However, we recommend that you use lower case characters for all file and directory names.

When the PLUGIN command finds the file `plugin.py` it is loaded into the running MMA program. The plugin loader will attempt to register three functions or methods from `plugin.py`:

1. `run(param-list)` — this is the code executed when your new command is encountered in a MMA script.
2. `trackRun(TrackName, param-list)` — this is the code executed when your new command is encountered as part of a track command.
3. `dataRun(param-list)` — this is code which will be inserted into a data line when called.
4. `printUsage()` — this code is used when you request a usage message from the command line using the "-I" option.

Chapter 2 Entry Points

Following are simple examples for each entry point recognized by MMA.

2.1 run(param-list):

The RUN() function is executed when a MMA script encounters the plugin's keyword in a non-track context. Following is a simple bit of Python code we used in the `hello` example shipped with MMA.

```
def run(l):
    print("Hello. This is the run() function in the MMA plugin.")
    print("We are at line %s in the MMA file %s." % (gbl.lineno, gbl.infile))

    if l:
        print("Args passed are:")
        for i in l:
            print(i)
```

In the first line we pass the parameter "l". This is set by the main MMA parser and contains a list of any parameters passed to the @HELLO function.

We leave the rest of the program as an exercise.

2.2 trackRun():

The next entry point is the TRACKRUN() function. This is executed when the parser finds the plugin's keyword in a track context. For example:

```
Plectrum-Main @Hello My command params
```

The difference between the simple and track versions is that trackRun() is also passed the "name" of the track. For example, the above example will have the first parameter set to the string "PLECTRUM-MAIN". Using the following code you can set a variable (in this case "self") to point to the Plectrum-Main class.

```
self = gbl.tnames[name]
```

For this to work, you will need to import MMA's global module into the namespace. We suggest you do that at the top of your plugin.py module:

```
        import MMA.gbl as gbl
```

Using the "self" variable you now have access to all of the variables associated with the track. You'll have to dig though the code a bit, but a few examples (most of the settings are in the form of a list with a value for every sequence point in your song):

```
self.volume  -- track specific volume
self.articulate -- track specifc articulations
self.sticky -- a True/False setting
```

For more variables we suggest you examine the pat.py module.

Yes, you can change these values from your plugin. Is that a good idea? Probably not! For an alternate method to change settings read the "returning values" chapter, below.

2.3 dataRun(param-list):

The DATARUN() function is executed when a MMA script encounters the plugin's keyword in a data line context. If you examine the very simple code for addgm/plugin.py you'll see a one line python function defined in the example file `plugins/addgm` shipped with MMA.

```
def dataRun(ln):
     return ['Gm'] + ln
```

All it does is to add "Gm" to the start of data line. However, there is nothing saying you can't do much, much more with this. The function is passed the entire data line (as a Python list). You could parse it and change chord names or types, etc.

- ♫ If no leading comment line number is present in your data line *and* the run() function **is** defined, the run() function (from the plugin) will be called.
- ♫ If there is not a leading comment line number and run() **has not** been defined, a comment line number based on MMA's internal tables is inserted into the start of the line and the dataRun() function (from the plugin) is called.
- ♫ In most cases you'll find it easiest to create a plugin which has a trackRun() and/or a run() function **or** a dataRun() function. Combining them will lead to your confusion!

2.4 printUsage():

The final entry point is the PRINTUSAGE() command. It receives no parameters and is only called when MMA finds a **-I plugin-name** command line argument. This code is then interpreted and the program ends.

PRINTUSAGE() should print a simple usage message.

Chapter 3

Preset Values

When a plugin is initialized the dictionary **PlugInName** is created in the plugin module's namespace. It contains the following items:

```
   'name' - the name of the plugin. The case of the containing
directory is preserved

   'dir'  - the name of the directory containing the plugin. For
example:  /home/bob/MMA/plugins

   'path' - a complete path entry to the plugin. For example:
/home/bob/MMA/plugins/hello/plugin.py

   'cmd'  - the registered command name. For example, @HELLO
```

You can access any of these values in your module using the notation:

```
zz = plugInName['name']
```

Chapter 4

Returning Values

In order to be of any use, a plugin will need to make modifications to your song or library file. You can, if you want, make these inside your function. However, we don't suggest you do. The API can change in the future. Plus, many settings rely on other basic MMA settings which are easy to miss. Much more robust is the method of creating string(s) with native (or even other plugin) commands and returning the to the main script.

This is easy to do.

Just create a list of commands and push them onto the programs input stack. Examine the `hello/plugin.py` module for actual code, but as a summary you'll need to:

1. Create lines for each line of code. In our example we use the array "ret" and simply append each line to the array.
2. Process each of the lines into an acceptable format. This is easy, it's just a matter of converting the string to a list of words. In our example we do:

   ```
   ret = [l.split() for l in ret]
   ```

3. Finally, return the value to the input queue.

   ```
   gbl.inpath.push(ret, [gbl.lineno] * len(ret))
   ```

   here we set the line number of each returned line to be the same as the current line being processed. Note: we're assuming that your plugin has imported "gbl" as detailed above.

Chapter 5 Standard Interface

To make life easier for plugin authors, a module, `pluginUtils.py`, has been added to MMA.[1]

5.1 How to use pluginUtils module

When you access MMA internals, you have to care about MMA changes. For example, the names of the members could change, and you should check the compatibility of your plugin with every new version of MMA that comes out.

The functions defined in pluginUtils are guaranteed to not change and to be updated together with MMAȦlso, they help you with documentation and argument parsing.

To explain how to use pluginUtils, we have commented the source of the StrumPattern plugin.

```
# We import the plugin utilities
from MMA import pluginUtils as pu

# ###################################
# # Documentation and arguments     #
# ###################################

# We add plugin parameters and documentation here.
# The documentation will be printed either calling printUsage() or
# executing "python mma.py -I pluginname".
# I suggest to see the output for this plugin to understand how code
# of this section will be formatted when printed.

# Minimum MMA required version.
pu.setMinMMAVersion(15, 12)

# A short plugin description.
pu.setDescription("Sets a strum pattern for Plectrum tracks.")

# A short author line.
```

[1]The module has been authored by Ignazio Di Napoli ¡neclepsio@gmail.com¿ and all kudos should go to him.

```
 pu.setAuthor("Written by Ignazio Di Napoli <neclepsio@gmail.com>.")

 # Since this is a track plugin, we set the track types for which it can
 # be used. You can specify more than one. To allow for all track types,
 # call setTrackType with no arguments.
 # For non-tracks plugin, don’t call setTrackType at all.
 # Whether the plugin is track or non-track is decided by the entry point,
 # but help will use this information.
 pu.setTrackType("Plectrum")

 # We add the arguments for the command, each with name, default value and a
 # small description. A default value of None requires specifying the argument.
 pu.addArgument("Pattern",     None,  "see after")
 pu.addArgument("Strum",       "5",   "strum value to use in sequence (see Plectrum
 # Some arguments are omitted -- you can see the full source code if you’re interest

 # We add a small doc. %NAME% is replaced by plugin name.
 pu.setPluginDoc("""
 The pattern is specified as a string of comma-separed values, that are equally spac

...... Use the command ‘‘mma -i strumpattern’’ to see the complete documentation!

 Each time it’s used, %NAME% creates a clone track of the provided one using the voi
 Its name is the name of the main track with an appended "-Muted", if you need to ch

 This plugin has been written by Ignazio Di Napoli <neclepsio@gmail.com>.
 Version 1.0.
 """)



 # ###################################
 # # Entry points                    #
 # ###################################

 # This prints help when MMA is called with -I switch.
 # Cannot import pluginUtils directly because it wouldn’t recognize which
 # plugin is executing it.
 def printUsage():
     pu.printUsage()

 # This is a track plugin, so we define trackRun(trackname, line).
 # For non-track plugins we use run(line).
 # When using this library, only one of the two can be used.
 def trackRun(trackname, line):
     # We check if track type is correct.
     pu.checkTrackType(trackname)
     # We parse the arguments. Errors are thrown if something is wrong,
     # printing the correct usage on screen. Default are used if needed.
     # parseCommandLine also takes an optional boolean argument to allow
     # using of arguments not declared with pu.addArgument, default is False.
     args = pu.parseCommandLine(line)

     # This is how we access arguments.
     pattern = args["Pattern"]
     strum   = args["Strum"]
     # [Omissis]

     # Here I omit plugin logic, this is not interesting for explaining
     # pluginUtils.
     # Let’s just pretend we have the result of the computation:
     all_normal = "{1.0 +5 90 80 80 80 80 80;}"
     all_muted = "z"

     # If you don’t send all the commands together the result is commands
     # are reversed since each is pushed as the very next command to be executed.
     # So we save all the commands (with addCommand) and send them at the end
     # (with sendCommands).

     pu.addCommand("{} Sequence {}".format(trackname, all_normal))
     pu.addCommand("{}-Muted SeqClear".format(trackname))
     pu.addCommand("{}-Muted Copy {}".format(trackname, trackname))
     pu.addCommand("{}-Muted Voice MutedGuitar".format(trackname))
     pu.addCommand("{}-Muted Sequence {}".format(trackname, all_muted))
     pu.sendCommands()
```

5.2 Function documentation

Following are the available functions in the pluginUtil.py module.

addArgument(name, default, doc) Adds an argument for the plugin, to be using in parseArguments and printUsage. If you do not want to provide a default, use None to trigger an error if the user not specifies the argument. When the plugin is used, the arguments have to be specified as in Call.

addCommand(cmd) Adds a MMA command string to the queue. The queue must be sent using sendCommands, so they are pushed together. If you don’t send all the commands together the result is commands are reversed since each one is pushed as the very next command to be executed.

checkTrackType(name) Checks if the track type is coherent with the ones specified with setTrackTypes. If not, throws an error.

error(string) Prints an error and halts execution.

getSysVar(name) Returns a system variable. For example, getSysVar(”Time”) is the same as *MMA* command $_Time.

getVar(name) Returns a user variable.

getName() Returns the name of plugin, according to the directory the user installed it.

parseCommandLine(line, allowUnkown=False) Parses the plugin command line and returns a dictionary with arguments and values according what has been defined with addArgument. If allowUnknown is True, it allows to use undeclared argument names. Else, throws an error. Throws an error if an argument with no default is not assigned. Hhe arguments have to be specified as in Call.

printUsage() Prints documentation using data from addArgument, setAuthor, setDescription, setSynopsis and setTrackType.

sendCommands() Sends the commands added with addCommand to the queue.

setAuthor(author) Sets the author.

setDescription(descr) Sets a short description of the plugin.

setSynopsis(descr) Sets a synopsis of the plugin.

setMinMMAVersion(major, minor) Sets the miminum *MMA* version required, throws an error if not met.

setPluginDoc(doc) Sets the documentation for the plugin. %NAME% is replaced by plugin name.

setTrackType(*types) Sets track types to which the plugin applies; if empty everythi ng is accepted.

warning(string) Prints a warning.

Chapter 6

Tutorial

A short tutorial on writing a simple plugin. Stay (yes!) tuned.