ConsoleTools Lua Framework

version 1.0

provided by notenlektorat.de

User Manual (macOS)

Table of Contents

Overview 3
Availability 3
CAVEAT UTILITOR 3

Installation and General Setup 4
Unpacking 4
Script-assisted installation 4

Using ConsoleTools in Dorico 8
Tutorial: keycommands and basic console use 8
Console use functions 11

Tool Setup File 19
Provide DoApp emulation, for testing in external Lua environment (optional) 19
Setup function declaration 19
Example function (for tutorial setup only) 20
Extend auxiliary library (optional) 20
Set up tools as available 20
Bind custom variable names (optional) 25
Set up profiles (optional) 25
List available tools and profiles on initialisation (optional) 27
Set up initial slot allocations (recommended) 27
Dorico console emulation (in external Lua environment only) 27

Appendices 28
Appendix A: Auxiliary library 28
Appendix B: Multi-language documentation encoding 29
Appendix C: Example Lua scripts 30
Appendix D: Embedding documentation in custom tool script files 32

Disclaimers 34

ConsoleTools Lua Framework

version 1.0 | 05/23

provided by notenlektorat.de

Overview

The notenlektorat ConsoleTools Lua framework provides Dorico users with the means to organise arbitrary Lua scripts as "tools" and access them by way of keycommand-controlled "tool slots".

In the context of the framework, the term tool can mean the execution of:

- an arbitrary Lua file (e.g., a recorded macro)
- a function from a table that was returned by a Lua file (i.e., an external tool library)
- a Dorico command string
- a local Lua function declared during setup, or a global Lua function

It is possible to pass arguments to tools. Default arguments can be specified for any tool during setup, and tools can be called with explicit arguments via the console, overriding any such default. Tools can also be bound to variables in the Lua environment, which can be called from the console.

A tool slot may hold multiple tools, with functions for cycling and toggling between them provided for use via keycommands. Assignment of tools to slots can be changed via console, or by application of custom profiles; cycling and toggling between profiles is likewise possible with keycommands.

The framework also provides functionality to store and recall basic documentation for tools.

Availability

The ConsoleTools framework is available as a full version (for purchase) and a free limited version. The versions differ as follows:

	Full Version	Limited Version
Number of tools that can be set up:	unlimited	no more than 10
Number of tool slots that can be set up:	unlimited	no more than 2
Number of variables that can be bound to tools:	unlimited	no more than 3
Default arguments can be set when allocating tools:	yes	no more then 20 times per session
Immediate tool execution of a tool cycled/toggled to:	yes	for no more than 3 tools at once
Tool source file reloading, to reflect *ad hoc* code edits:	yes	no
Included auxiliary library can be upgraded:	yes	no, only extendable

CAVEAT UTILITOR

- Setting up notenlektorat ConsoleTools is not particularly complicated. It is not an entirely straight-forward process, either, as it involves manual editing of Dorico's JSON data for keycommands. If in doubt, make backups of any relevant files before editing.
- The framework does **not** contain a noteworthy amount of Lua scripts for use with Dorico, merely some arbitrary examples to demonstrate how to set up and use a tool collection.
- No programming skills are required for using ConsoleTools. However, a basic understanding of the Lua language will be beneficial when using some of the advanced features.
- The framework makes use of Lua's `loadfile` function (in "text chunks only" mode), which by its nature comes with potential security issues. Users are advised to only run trusted scripts with the framework, just as they would when using Lua scripts directly in Dorico.

Please also acknowledge the disclaimers at the end of this document.

Installation and General Setup

Unpacking

For the purpose of these instructions, let it be assumed that the ConsoleTools zip file is unpacked to the directory /Users/Username. This creates a subdirectory /ConsoleTools, containing seven files:

assistedInstall.lua	a script facilitating installation and initial setup
ConsoleTools – User Manual (macOS).pdf	this user manual (for macOS)
ConsoleTools – User Manual (Windows).pdf	this user manual (for Windows)
consoletools.lua	the framework itself
initialiseConsoletools.lua	loads the framework into Dorico's Lua environment
tutorialToolSetup.lua	an example setup, demonstrating various use cases
userToolSetup.lua	a template for setting up a custom tool collection

… as well as two further subdirectories:

/example tool script files	containing some examples of Dorico Lua scripts
/keycommand files	initially empty

Script-assisted installation

- **Changing paths in assistedInstall.lua**

Open the file assistedInstall.lua in a text editor. At the start of the file are six variable declarations:

```
local pathInitialiseConsoletoolsFile = [[...initialiseConsoletools.lua]]
local pathConsoletoolsFile           = [[...consoletools.lua]]
local pathUserToolSetupFile          = [[...tutorialToolSetup.lua]]
local pathToolsDirectory             = [[...example tool script files]]
local pathDefaultToolLibraryFile     = [[xmpl_SK_OffsetAndVelocityAdjustmentsFromTempo.lua]]
local pathKeycommandDirectory        = [[...keycommand files]]
```

Replace the "…" portions to complete the appropriate paths, e.g.:

```
local pathInitialiseConsoletoolsFile = [[/Users/Username/ConsoleTools/initialiseConsoletools.lua]]
local pathConsoletoolsFile           = [[/Users/Username/ConsoleTools/consoletools.lua]]
local pathUserToolSetupFile          = [[/Users/Username/ConsoleTools/tutorialToolSetup.lua]]
local pathToolsDirectory             = [[/Users/Username/ConsoleTools/example tool script files]]
local pathDefaultToolLibraryFile     = [[xmpl_SK_OffsetAndVelocityAdjustmentsFromTempo.lua]]
local pathKeycommandDirectory        = [[/Users/Username/ConsoleTools/keycommand files]]
```

Make sure that no incorrect whitespace characters are included in between the double square brackets.

Providing the first four paths is mandatory; the remaining two can be left empty (like this: `[[]]`).

The fifth path (`pathDefaultToolLibraryFile`) specifies a Lua file that will be considered the default when retrieving tools from an external library. If only a file name is provided, it will be assumed that the file is located in the tools directory specified by the third path (`pathToolsDirectory`). If the default library file is stored in a different location, the whole path must be provided instead.

The sixth path (`pathKeycommandDirectory`) can also be left empty, in the unlikely case that you do not want to generate the files necessary for keycommand access.

If you want to move any of the relevant files to a different location, you can rerun assistedInstall.lua at a later time to change the paths accordingly. You can also make such changes manually in the file initialiseConsoletools.lua. However, when changing the location of initialiseConsoletools.lua itself, the script-assisted installation is recommended, as this will handle the necessary subsequent change of that path in all keycommand files automatically.

▪ **Running assistedInstall.lua in Dorico**

After saving the changes to assistedInstall.lua, start up Dorico. It is sufficient to have only the Steinberg Hub open. From there, load the file assistedInstall.lua via **Script › Run Script…**, which should bring up Dorico's Lua Script Console.

If the script finds a problem with any of the provided paths, an error report will be written to the console's output, with a description of the problem, if possible. In such a case, make the according changes in assistedInstall.lua and run it again. NB: do **not** use **Script › Run Last Script** for this, always use **Script › Run Script…**; if in doubt, completely restart Dorico before re-running.

If there are no problems with the provided paths, a status report to that effect will be written to the console, followed by prompts to enter one of three functions into the standard input (i.e., the field making up the lower half of the Script Console).

NB: the Lua language is case sensitive, so not typing the function names with lower and upper case letters exactly as shown below will result in an error.

Unless no directory for keycommand files was provided, you will likely want to proceed by entering:

```
writeAll()
```

… or alternatively:

```
writeAll(#)
```

… with `#` being the number of tool slots you want to set up (the default is two slots).

This will copy the necessary paths into the initialisation file and write a number of new files into the keycommand files directory.

It is also possible to only write the keycommand files, e.g. for creating additional slots at a later time:

```
writeKeycommandTriggerFiles(#)
```

As before, `#` stands for the number of slots wanted. Note that this always creates files for the given number of slots, instead of adding to any already existing slots. Therefore, to "upgrade" from two to three slots, enter `writeKeycommandTriggerFiles(3)`, and not `writeKeycommandTriggerFiles(1)`.

For the case that you do not want to create any keycommand files, a function for only updating the initialisation file is available:

```
insertPathsIntoInitFile()
```

If no keycommand file directory was provided, this will be the only function available.

A summary report will be written to the console after processing of a function. After successfully calling the appropriate function, close down Dorico completely (i.e., including the Steinberg Hub).

▪ **Editing Dorico keycommands JSON data**

To make the ConsoleTools framework's functionality accessible via regular Dorico keycommands, it is necessary to edit the relevant Dorico JSON file containing a user's custom keycommand overrides.

The JSON file (with # in the path examples below standing for the Dorico version) is located at:

/Users/[username]/LibraryApplication Support/Steinberg/Dorico #

It is possible that the directory contains multiple JSON files starting with "keycommands_", followed by a two-letter language abbreviation. If it is unclear which of several files is the relevant one, you should look at their contents in order to find out which one matches your current custom overrides.

NB: make sure that you really choose the directory concerned with user data, as the application level directory will contain a JSON file with the exact same name, editing which is not recommended.

Open the relevant JSON file in a text editor. If you have never worked with such a file, it can be helpful at this point to spend a moment on studying its structure and to get a general understanding of how commands are matched to specific keys or key sequences.

In short, a Dorico command string (of the kind that is captured when recording a macro script) is paired with one or more strings declaring the keycommand; if an already existing keycommand associated with the command string is to be vacated, this has to be stated as well.

For example, the following entry overrides the default keycommand for summoning the Jump Bar:

```
{
        "UI.ShowJumpBar" : [ "Ctrl+Shift+J", "DELETE:J" ]
},
```

Each keycommand trigger that you want to use needs its own JSON entry. When assistedInstall.lua has been used to write the trigger files, a template (forInsertionIntoDoricoJSON.template) will have been created in the provided keycommand files directory. It will contain a JSON entry for each created trigger file, with the appropriate command string already filled in, but missing a valid keycommand declaration. You can use this template by replacing the "…" portions with appropriate keycommands, and then copying-and-pasting it into the actual Dorico JSON file.

It is recommended to insert all ConsoleTools keycommands into the "global" context, to make sure that the framework can be accessed regardless of Dorico's specific system state at a given time:

```
{
        "common" : {
                "contexts" : [
                        {
                                "context" : "kGlobal",
                                "shortcuts" : [
                                        …               ← insert JSON entries within this block
                                ]
                        },
                        …
                ]
        },
        …
}
```

When deciding on suitable keycommands for your particular needs, keep in mind that it is possible not only to declare single keys (or single keys in combination with modifier keys) as keycommands, but also sequences of keystrokes, as well as more than one keycommand per command string.

There are six trigger files provided for each tool slot. For slot 1, these would be:

forKeycommand_toolslot1a_callCurrentTool.lua
forKeycommand_toolslot1b_cycleForward.lua
forKeycommand_toolslot1c_cycleBackward.lua
forKeycommand_toolslot1d_togglePrevious.lua
forKeycommand_toolslot1y_listCurrentAllocation.lua
forKeycommand_toolslot1z_helpForCurrentTool.lua

The first four trigger files provide the control functions for a slot: calling the current tool with default arguments (a), as well as changing the current tool by cycling through a (user-defined) collection of tools allocated to the slot (b,c and d). Assigning keycommands to the remaining two files is optional; they provide a status report for the slot (y) and documentation for its current tool, if available (z).

In addition to those for slot control, seven more files will have been created. Three of them provide means to switch between any user-defined slot profiles:

forKeycommand_profilesA_cycleForward.lua
forKeycommand_profilesB_cycleBackward.lua
forKeycommand_profilesC_togglePrevious.lua

… while the remaining four files offer a way to list quickly access status reports for the various components of the whole framework (assigning keycommand to them may be omitted):

forKeycommand_listA_listAllAvailableTools.lua
forKeycommand_listB_listAllAvailableProfiles.lua
forKeycommand_listC_listCurrentAllocationForAllSlots.lua
forKeycommand_listD_listAllCurrentCustomVariableNames.lua

The following example – which is also assumed for the tutorial further below – demonstrates a possible keycommand setup for two tool slots:

forKeycommand_toolslot1a_callCurrentTool	K
forKeycommand_toolslot1b_cycleForward	Option+K
forKeycommand_toolslot1c_cycleBackward	Ctrl+K
forKeycommand_toolslot1d_togglePrevious	Option+Shift+K
forKeycommand_toolslot1y_listCurrentAllocation	Ctrl+F1, Ctrl+K
forKeycommand_toolslot1z_helpForCurrentTool	Ctrl+F2, Ctrl+K
forKeycommand_toolslot2a_callCurrentTool	L
forKeycommand_toolslot2b_cycleForward	Option+L
forKeycommand_toolslot2c_cycleBackward	Ctrl+L
forKeycommand_toolslot2d_togglePrevious	Option+Shift+L
forKeycommand_toolslot2y_listCurrentAllocation	Ctrl+F1, Ctrl+L
forKeycommand_toolslot2z_helpForCurrentTool	Ctrl+F2, Ctrl+L
forKeycommand_profilesA_cycleForward	Option+,
forKeycommand_profilesB_cycleBackward	Option+.
forKeycommand_profilesC_togglePrevious	Option+/
forKeycommand_listA_listAllAvailableTools	Ctrl+F1, Ctrl+F1
forKeycommand_listB_listAllAvailableProfiles	Ctrl+F1, Ctrl+F2
forKeycommand_listC_listCurrentAllocationForAllSlots	Ctrl+F1, Ctrl+F3
forKeycommand_listD_listAllCurrentCustomVariableNames	Ctrl+F1, Ctrl+F4

- **Initialising the framework**

With custom keycommands properly defined, the ConsoleTools framework will be initialised for a Dorico session the first time that one of the relevant keycommands is invoked. Upon initialisation, the framework is loaded into Dorico's Lua environment, the tool collection is built up according to the tool setup file (as defined in initialiseConsoletools.lua), and the Script Console is brought up.

If you use the default tool setup file (tutorialToolSetup.lua), successful initialisation will result in printing to the console: 1.) a message on how to display help for the framework's primary functions 2.) a list of variables that have been bound to slots or tools, 3.) a list of all available tools, 4.) a list of all available profiles, and 5.) any messages caused by the function linked to the initial keycommand.

You should eventually replace the tutorial setup file with userToolSetup.lua, and adapt the latter to your needs. The tool setup file is processed only once, at initialisation. Any changes made to the tool setup will only take effect once Dorico is restarted.

Using ConsoleTools in Dorico

If, as is likely, you are considering to use the ConsoleTools framework as a way of accessing Dorico's Lua functionality in, first of all, an easy-to-use fashion, the amount of documentation below may be somewhat daunting. Don't be discouraged, though – ConsoleTools has been designed to make simple things simple. At the same time, the framework aims to be „super-chargeable" when extending it with the full power of the Lua language, should you feel so inclined.

You will hopefully find that, for the vast majority of your interactions with the framework, relying on the keycommands alone – as set up in the keycommands JSON file (see above) – is sufficient once you have set up a tool collection that suits your needs. Setting up a collection is detailed in the section Tool Setup File, further below.

The initial tool setup (tutorialToolSetup.lua) is intended as a practical introduction, allowing you to familiarise yourself with the basic principles of how to use the framework. The tutorial is designed within the constraints of the free-to-download version of ConsoleTools.

Tutorial: keycommands and basic console use

The following section is a walk-through of the basic commands for interacting with the framework. It assumes the tutorial tool setup, as well as the specific keycommands shown on the previous page. Do **not** try to run the tutorial file or any of the included files directly (via the **Script** menu) – with proper installation, the framework will initialise automatically upon first use of one of its keycommands.

For your very first steps with ConsoleTools after installing it, open a Dorico project with some notation (or simply enter some notes into a new project). Note: while all edits within the tutorial are trivial, it is recommended to make a backup if you use an existing file for trying things out.

In Write Mode, open the properties panel (at the bottom of the window) and select a note. Then use the keycommand K for tool slot 1.

The first thing that you will notice is that Dorico's Script Console opens up and displays a number of messages (if this does not happen, you probably already ran a script within the current Dorico session; in that case, bring up the console manually). You can safely ignore the console output for now. Nonetheless, it is recommended to leave the Script Console open when working with ConsoleTools, and to arrange it somewhere on the screen where it is visible while working in the Dorico project window.

NB: The first time the Script Console is brought up, it will cause the main application's focus to switch, and you must return focus to the project window manually before any further keycommands will register again. While there is currently no way to prevent this from happening, it also will occur only once during a Dorico session.

Next, check the properties panel – you should find that the color for the selection has been changed to red. Initially, slot 1 holds several tools for setting the Color property, and you have just applied one of them. There is a keycommand available for listing a slot's current tools; Ctrl+F1, Ctrl+K (that is: while keeping Ctrl pressed down, press F1 and K) will print this to the Script Console's output field:

```
Slot 1 currently contains:
  >>              ! to red
                  ! to blue
                  ! to transparent
```

The currently active tool is marked with >>. An exclamation mark preceding a tool's name denotes that this tool will be executed immediately upon becoming the slot's current tool.

Try out the keykommands for moving between the three tools of slot 1: Option+K and Ctrl+K will cycle forward and backward, and Option+Shift+K will switch to the previously used tool.

Use the keycommand L for slot 2. It holds a single tool, for deactivating the color property.

You can set up tool profiles in ConsoleTools, i.e.: presets of tool-to-slot allocations, tailor-made for certain tasks. The tutorial setup contains three such tool profiles. Use the keycommand Option+, to cycle to the next one. Since no profile is active at first, this will result in the first profile being selected for application, with this message printed to the Script Console:

```
Profile 'harmonics' (#1) to be applied | (2 tool slots)
```

With a note selected, use K again. Slot 1 now holds a different tool, which sets the Harmonics Type property for the note to "Artificial". The tool can do more, though. Press K twice in quick succession, and the Partial property is set to 4. Doing it three, four, five times sets the property to 3, 5 and 2, respectively, placing the most common artificial harmonics conveniently at your fingertips. Tool slot 2 (keycommand L) also holds a new tool now, for turning the Harmonics Type property off again.

Use the keycommand Option+, once more, to cycle to the next tool profile:

```
Profile 'offsets / velocity' (#2) to be applied | (2 tool slots)
```

Using the keycommands for the two tool slots, you will notice that nothing happens. That is, nothing except the following message being printed to the Script Console:

```
! no tempo set
```

This profile binds together a number of different tools for adjusting playback-related properties. It also goes already slightly beyond the mere sending of static command strings to Dorico, of the kind that you may be accustomed with if you have experimented with Dorico's macro recording capability. Instead, the property values are calculated in relation to a tempo, which must be provided first. To do this, switch to the Script Console and, in the lower field, enter:

```
bpm(80)
```

Go back to the project window and, with a note selected, use K once more. This time, the two properties Playback start offset and Velocity in the Notes and Rests panel have been turned on and set to 640 and 10, respectively. Return to the Script Console and enter another tempo value:

```
bpm(100)
```

Back in the project window, use K again; the Playback start offset property has now been set to 800, a different value than before.

To find out more about a tool, you can display its documentation, if any was provided. This can be done via the console (documented further below), but there is also a keycommand available for each slot to display documentation for its current tool. Ctrl+F2, Ctrl+K prints the following:

```
->Help for current tool of slot 1
  Description: apply portamento
     - sets the 'Notes and Rests'>'Playback start offset' property of a selected note, calculated as …
     - sets the 'Notes and Rests'>'Velocity' property of a selected note to 10
```

For a reminder of which tools are currently applied, you can use the keycommand Ctrl+F1, Ctrl+F3, which at this point will print:

```
* Contents of all current slots:
Slot 1 currently contains:
>>                apply portamento
Slot 2 currently contains:
>>                apply legato | slow
                  apply legato | fast
```

Using the keycommand L does now also result in properties being changed. If you cycle to the second slot's next tool (Option+L), you will notice that there is a message printed to the console:

```
Current tool for slot 2: apply legato | fast
```

… but there has been no actual change to the playback properties. That is because, unlike with the color tools earlier, the two tools in the second slot are not set to be executed immediately (thus, in the list above, they are not marked with an exclamation mark). For such a tool, you must use the slot's main keycommand after having changed the tool: pressing L again will now apply a different set of values for the Playback start offset and Velocity properties. Whether tools are executed immediately or not will depend very much on context and vary between different profiles.

Tool profiles are convenient, but there is a more flexible way of applying tools to slots, if needed. Let's assume you want to apply both the "slow legato" and the "fast legato" playback adjustments from above to several passages in a score, and you also want to mark up each case by color. For such a case, you can use tools *ad hoc*, by typing something like this into the console input field:

```
set(1,true,"idLS","idG")
set(2,true,"idLF","idB")
```

(You can simply copy-and-paste these two lines. They represent one of several ways to allocate tools via the console, which are documented in depth further below.)

The keycommand Ctrl+F1, Ctrl+F3 can once again be used to see how the slot allocation has changed:

```
* Contents of all current slots:
Slot 1 currently contains:
>>            !  apply legato | slow
              !  to green
Slot 2 currently contains:
>>            !  apply legato | fast
              !  to blue
```

With this configuration, you can simply select a passage and then cycle twice through the relevant slot – reducing to two actions what would normally take a lot of clicking and typing. (If you try this particular example out in practice, make sure to filter for notes first, if needed, just as when doing these operations manually. Eventually, of course, you could also set up your own tool for filtering notes, from a Dorico command string, and incorporate it in the cycle.)

The keycommand Ctrl+F1, Ctrl+F1 prints all available tools to the Script Console, so you do not have to rely on your memory when making *ad hoc* allocations. Each tool is listed with a numerical index, and, if provided, a unique string index, both of which can be used to match it to a slot.

There are ways to use tools directly through the console. One of them is calling tools that are bound to Lua variables. For example, selecting a note and then entering:

```
tr()
```

… into the console will execute the "to transparent" tool from the beginning of the tutorial, even if that tool is not allocated to any slot at all. Likewise, the `bpm` example from before was a variable-bound tool, not intended to be used through a tool slot ever.

As said earlier, ConsoleTools aims to make simple things simple. Nonetheless, users (as well as script developers) can do more complex things with it. The following section about Console use functions documents these capabilities in more detail. In the concluding Tool Setup File section you can learn how to build and manage your own personalised collection of tools and profiles, with many of the examples of this tutorial revisited for that purpose.

Console use functions

In addition to using the ConsoleTools framework by way of Dorico keycommands, it is also possible to access its functionality through the Dorico Script Console. On initialisation, six functions are declared as global variables to the Lua environment:

`set`	`slot`	`list`
`setProfile`	`bind`	`help`

If you prefer to use a different name for any of these functions, you can override them in line 17 of the file initialiseConsoletools.lua:

```
ConsoleTools_MainFunctionNameOverrides={set="",setProfile="",slot="",bind="",list="",help=""}
```

… by filling in a valid Lua identifier in between the corresponding pair of quotation marks.

Each function (and any of its variants, if applicable) is described in detail below. The symbol ► marks the statement of the general syntax of each function. Elements appearing in italics represent variable values. Optional arguments are enclosed in single square brackets.

▪ Console use of the `set` function

The `set` function is the most direct method to allocate tools to tool slots (the more convenient one being the application of profiles, see below). It comes with two different syntaxes: a simplified one intended for convenient *ad hoc* console use; and the full syntax, which considers some additional arguments. The full syntax is also somewhat more suited to be arranged in a way that allows for structured reading, which is why its use is recommended in the tool setup file.

A quick reference of several ways to use the `set` function is given by the examples at the end of this section. It will usually be sufficient to know the simplified syntax. Only venturing into more advanced uses of the framework may necessitate for you to familiarise yourself with the full syntax.

An important distinction between the two syntaxes is that the full syntax actually means the use of *two* (or even more) chained functions: `set(…)` does itself return a new function, and depending on which syntax the first set of arguments matches, the second function will either have to be called with further arguments (for the full syntax), or may be entirely ignored (for the simplified syntax). To emphasise this, the following examples have the brackets corresponding to function calls displayed in larger size.

Full syntax:

► `set`(*`slotId`*, [, *`bAddToCurrent`* [, *`bMakeCurrent`*]]) ([*`bCallWhenMadeCurrent`*,] *`toolRef1`* [, *`defArgs1`*] [, …])[…]

Simplified syntax:

► `set`(*`slotId`*, [*`bCallWhenMadeCurrent`*,] *`toolRef1`* [, *`defArgs1`*] [, …])

The *`slotId`* argument may be either a number, or a custom variable bound to a slot index (see use of the `bind` function, further below).

The full syntax can take two optional boolean arguments after *`slotId`*. The first, *`bAddToCurrent`*, may be passed as `true` if the tool (or tools) are to be added to what is already allocated to the slot; otherwise, the slot will be cleared first. (Note that the simplified syntax – as it does not consider *`bAddToCurrent`* – will always clear the respective slot.) If the second optional argument *`bMakeCurrent`* is passed as `true`, the added tool will also become the current tool for the slot; if several tools are added collectively, this will apply to the first one.

For the full syntax, the remaining arguments are passed to the secondary function, i.e., within the second pair of round brackets. For the simplified syntax they follow directly after the *`slotId`* argument.

The optional boolean argument *bCallWhenMadeCurrent* may be passed as `true`, which will set the added tool (or tools) to be executed immediately each time upon becoming the slot's current tool when cycling or toggling via keycommands. The purpose of this feature is foremost any use case where there is a pool of similar tasks which do not carry the risk of introducing non-trivial changes. For example, allocating to a slot a collection of tools for changing the color of a selection will allow, with this argument, to apply different colors by simply cycling through the slot's tools, instead of first changing the slot's current tool to the desired color tool, and then apply the current tool with an extra step. (This very scenario is demonstrated by the initial slot allocation of the tutorial setup.)

A tool or tool group to be added to the slot is specified by *toolRef*, which may, in fact, be more than one argument. Tools can be referenced either by sequences of integers, or with a unique string ID.

By example, some references of the tutorial setup, as shown with `list()` (see below), are these:

```
…
change color              4         idCOL
 change color to RGB       4,1      idRGB
  to red                    4,1,1   idR
  to green                  4,1,2   idG
  to blue                   4,1,3   idB
 change color to TR/OFF    4,2      idTO
  to transp.                4,2,1   idT
  to off                    4,2,2   idO
```

(For how to set up a tool collection, see the section Tool Setup File, further below.)

The `set` function matches the provided form of *toolRef* to a corresponding index sequence or string ID. For example:

- both the sequence `4,1,2` as well as the string `idG` refer to the tool "to green"
- both the sequence `4,1` as well as the string `idRGB` refer to the tool group comprising the three tools "to red", "to green" and "to blue"
- both the number `4` as well as the string ID `idCOL` refer to the tool group comprising the tool groups "change color to RGB" and "change color to TR/OFF"; i.e., five distinct tools in all

The rationale behind making tools referable by numerical indices is that entering numbers will often be quickest when re-allocating tools *ad hoc*. Keep in mind, however, that the numerical index of any specific tool may change when the tool setup file is edited. For that reason, defining string IDs provides you with a custom, unchanging and unique reference to any tool or tool group.

The optional final argument *defArgs* may be a Lua table (i.e., a list of arguments enclosed within curly brackets, like this: `{1,true,"kParenthesise"}`). If provided, the table's contents will be passed to the tool as arguments each time the respective instance of the tool function is called 1.) via keycommand, or 2.) without ephemeral override arguments via either the `slot` function (see below), or a custom variable bound to the slot (see `bind` function, below). Such "secondary" default arguments take precedence over a tool's "primary" default arguments, which may have been defined in the tool setup file. Note that if such a table is provided for a reference of a group of tools, it will apply for all tools of that group.

It is possible to pass several tool references to a single instance of the `set` function, which is especially useful with the simplified syntax, as it will always clear the slot first. To do this, add as many *toolRef* arguments as needed; each such reference may be followed by its own optional *defArgs* table. Adding two numerical indices after each other is done by inserting a `0` in between (this is unnecessary if the references are already seperated by a *defArgs* table).

With the full syntax, it is likewise possible to separate multiple tool references by function chaining; i.e.: the function that is returned by the primary call of `set` does return itself again when called. The arguments *`slotId`*, *`bAddToCurrent`* and *`bMakeCurrent`* of the initial call of `set` will apply for each reference passed to any of the subsequently returned functions, while the *`bCallWhenMadeCurrent`* argument will only apply to those references with which it was passed to an instance of the secondary function.

- **Examples for the set function**

Where applicable, examples are given in equivalent forms for simplified (left) and full syntax (right).

`set(1,4)` `set(1)(4)`

… clears slot 1 and then allocates to it the tool (or tool group) referenced by the index `4`.

`set(J, "idFOO")` `set(J)("idFOO")`

… clears the slot bound to the custom variable name J (see `bind` function, further below) and then allocates to it the tool (or tool group) referenced by the string ID `idFOO`.

`set(1,true,4,"idBAR")` `set(1)(true,4,"idBAR")`

… clears slot 1 and then allocates to it the tool (or tool group) referenced by the index `4` and the tool (or tool group) referenced by the string ID `idBAZ`, in that order. Each respective tool will be called immediately each time it becomes the current tool of slot 1.

`set(2,"idBAZ",4,1,3,0,4,1,1)` `set(2)("idBAZ",4,1,3,0,4,1,1)`

… clears slot 2 and then allocates to it the tool (or tool group) referenced by the string ID `idBAZ` and the tools (or tool groups) referenced by the indices `4,1,3` and `4,1,1`, in that order.

`[N/A]` `set(2)("idBAZ")(4,1,3)(4,1,1)`

… is equivalent to the preceding example.

`[N/A]` `set(2)("idBAZ")(true,4,1,3,0,4,1,1)`

… is equivalent to the preceding two examples, except that the added *`bCallWhenMadeCurrent`* argument applies to the two numerical references, but not to the string ID reference.

`[N/A]` `set(1,true)(3,2,1)`

… adds to the current tools of slot 1 the tool or (tool group) referenced by the index sequence `3,2,1`.

`[N/A]` `set(2,true,true)(3,2,2)`

… adds to the current tools of slot 2 the tool (or tool group) referenced by the index sequence `3,2,2`. If the reference is to a single tool, it is made the current tool of the slot; if the reference is to a tool group, the group's first tool is made the current tool of the slot.

`set(1,3,2,2,{7,true})` `set(1)(3,2,2,{7,true})`

… clears slot 1 and then allocates to it the tool (or tool group) referenced by the index sequence `3,2,2`. Calling the tool (or, in case of a tool group, any of the relevant tools) via the keycommand for slot 1, the arguments `7` and `true` will be passed to the tool function, taking precedence over any primary default arguments that may have been set up. Likewise, these arguments are passed when calling, without override arguments, custom variable bound to slot 1 (see `bind` function, further below); this also applies correspondingly to the following example.

`set(2,1,{"kMinim"},1,{"kCrotchet"},1,{"kQuaver"})` `set(2)(1,{"kMinim"})(1,{"kCrotchet"})(1,{"kQuaver"})`

… clears slot 2 and then allocates to it three instances of the the tool (or tool group) referenced by index `1`. Each instance, when called via the keycommand for slot 2, will be passed the argument from

the corresponding table (`"kMinim"` for the first instance, `"kCrotchet"` for the second, `"kQuaver"` for the last), taking precedence over any primary default arguments that may have been set up.

▪ **Console use of the `setProfile` function**

While tool profiles can probably be applied and switched most conveniently via keycommands, the `setProfile` function provides a way to apply a profile directly. (For how to set up tool profiles, see the section Tool Setup File, further below.)

► `setProfile(integerOrString)`

The function receives a single argument *`integerOrString`*. If the argument is an integer, the profile with that index will be applied (profiles are indexed in the order in which they were added during tool set-up). If the argument is a string, it can be either a profile's unique string ID, or the profile's description string (in full, or abridged).

▪ **Examples for the `setProfile` function**

The following examples assume the profiles from the tutorial tool setup, which would be listed as:

```
* Available Profiles
harmonics               1         H
offsets / velocity      2         OV
colors                  3         C
```

`setProfile(2)`

… will apply the second profile, "offsets / velocity".

`setProfile("C")`

… will apply the third profile, "colors".

`setProfile("harmonics")`

… will apply the first profile, "harmonics"; likewise will:

`setProfile("har")`

▪ **Console use of the `slot` function**

The `slot` function provides a way to pass override arguments to a slot's current tool. NB: an alternative method for this is binding a custom variable name to a slot (see `bind` function, below).

► `slot(slotId [, ephemeralOverrideArgument1 [, ephemeralOverrideArgument2 [, …]]])`

The *`slotId`* argument will usually be a number (the slot index). It may also be a custom variable bound to the respective slot; however, if such a variable has been set up, it is more effective to call it directly.

Any additional arguments will be passed to the tool instead of its primary or secondary default arguments. This override is ephemeral; the tool will revert to using its respective default arguments when called regularly afterwards. The primary default arguments are those that have been set up for a specific tool in the tool setup file. The secondary default arguments may have been provided when allocating a tool to the slot via the `set` function (see above); they override a tools primary default arguments.

Passing no additional arguments after the *`slotId`* argument will always result in the tool being passed its *default* arguments, instead of none. In other words, using a slot via the `slot` function without arguments is only equivalent to using it via its keycommand if the slot's current tool has been set up without *any* default arguments.

▪ **Example for the** `slot` **function**

When using the tutorial tool setup file, slot 1 will initially hold three tools which set the color property for a Dorico selection. Each tool is an instance of the same Lua function `fSetColor` (defined earlier in the file), but set up with different primary default arguments for red, blue or no color, respectively (see lines 72, 74 and 78 in tutorialToolSetup.lua). Thus, entering:

```
slot(1,"W")
```

… will result in `fSetColor` being called, but with the string `"W"` instead of the default argument, resulting in a command for setting the color to white.

▪ **Console use of the** `bind` **function**

The `bind` function makes it possible to flexibly allocate tools to Lua variables, making them accessible as regular Lua function calls via the Script Console. This may be done for a variety of reasons, e.g.: as an alternative to keycommands for rarely used tools, or in order to pass override arguments to a tool, or as an *ad hoc* way to arrange tools to better match a certain workflow. It can be used within the tool setup file to bind variable names upon initialisation.

► `bind(vn1, toBind1 [[, defaultArguments1] [, strDescription1 [, doc1]]], [vn2, toBind2 [, …]] [, …])`

The first argument `vn1` is the chosen variable name and must be a string of a valid Lua identifier (i.e., any string of letters, digits, and underscores, not beginning with a digit).

To bind a tool to the variable name, the second argument `toBind` must be:

- an integer, to bind the slot with that index to the variable name (an alternative method for calling a slot's current tool like a function is the `slot` function, see above)
- a Lua table containing either integers or a single string, to bind a tool from the tool collection to the variable name (the integers being the tool's index sequence as it appears when listing all available tools, or the string being that tool's unique ID string, if defined)
- a Lua function, to bind that function directly to the variable name

If a Lua table is provided as the optional third argument `defaultArguments`, the contents of that table will be passed to the bound tool whenever the variable is called without any override arguments. As with the `set` function, these arguments are "secondary" defaults, which will override any "primary" defaults that a tool from the tool collection may have been set up with.

Two further optional arguments can be passed, to provide 1.) a string as the description for the tool (`strDescription`), as it is to display when using `list.v()`, and 2.) a documentation string (`doc`), which will be used by the `help` function. You can provide `strDescription` without `doc`, but not the other way round.

When binding variable names to tool slots, any arguments passed as `strDescription` or `doc` will be ignored. When binding variable names to specific tools from the tool collection, that tool's description and documentation string (if provided during setup) will apply if `strDescription` or `doc` are omitted.

It is possible to bind more than one variable name with a single call of `bind`, by passing additional lists of arguments. As with the primary list, the variable name `vn` and the `toBind` argument are mandatory, while the remaining arguments are optional.

The whole list of arguments may be enclosed in a single Lua table (i.e., curly brackets). Doing so may be preferable when using the bind function within the tool setup file, as this reduces the risk of introducing syntax errors when making changes by outcommenting (a trailing comma is harmless within a table, but not for a pure list of arguments passed to a function).

To clear the binding of one or more custom variable names, pass `false` as the `toBind` argument:

```
bind(vn1, false [, vn2, false] [, …])
```

To clear a single variable name, the `false` may be omitted:

```
bind(vn)
```

… is equivalent to:

```
bind(vn,false).
```

▪ **Examples for the `bind` function**

NB: The tutorial setup (tutorialToolSetup.lua) does contain the following examples, albeit as a collective call to `bind`; it also provides an actual implementation of `fSetColor`, an example function that sets the color property for a Dorico selection.

```
bind("J",1,"K",2)
```

… will bind the global variables `J` and `K` to slots 1 and 2, respectively. Therefore, entering `J()` will call the current tool of slot 1, with that tool's default arguments (if defined); entering `J(7,true)` will also call the current tool of slot 1, but with the explicit arguments `7` and `true`.

```
bind("cl",fSetColor,{"P"},"set color (default: purple)")
```

… will bind the global variable `cl` to the Lua function `fSetColor`, with a single "secondary" default argument `"P"`, and a description string. Entering `cl()` executes `fSetColor`, passing the default argument `"P"`; this results in a Dorico command for setting the color to purple. However, passing a valid argument for `fSetColor`, like `cl("V")` or `cl("ffee82ee")` executes the function with that argument instead (both these arguments result in violet coloring). Note that this particular example, when entered into the Script Console, will only work because `fSetColor` is not locally scoped to the tool setup file.

```
bind("tr",{4,2,1})
```

… will bind the global variable `tr` to the tool stored to the collection by the index sequence `4,2,1`. With the tutorial setup, that particular tool is also an instance of the function `fSetColor`, but already with an appropriate "primary" default argument. Thus, entering `tr()` will default to setting the color property to transparent, but `tr` can still be passed valid arguments for `fSetColor`, just as was the case with `cl`. The strings for description and documentation default to those from the collection tool.

The remaining two examples are foremost suited for use in the tool setup file.

```
bind("tr",{"idT"})
```

… is – within the context of the tutorial setup – equivalent to the previous example. Referencing a tool from the tool collection by string ID is recommended when using `bind` within the tool setup file, since any subsequent changes to the tool collection may result in numerical indices referencing different tools than before.

```
bind("bpm",fromLuaLibrary("setTempo"))
```

… will bind the global variable `bpm` to the function returned by `fromLuaLibrary` (see below), and store for it the the provided description and documentation strings. In the context of the tutorial setup, this is an example for a tool that does only exist as a bound variable (as an auxiliary function to the portamento and legato tools in that setup).

▪ **Console use of the `list` function**

The `list` function provides a way to print overviews of the current state of the framework to the Script Console's output. It can be used in the following ways:

► `list()`

Prints a list of all available tools and a list of all available profiles (see `list.t()` and `list.p()` below).

► list.t()

Prints a list of the available tool collection, as defined in the tool setup file. Each tool or tool group is listed with 1.) its descriptive string, 2.) the numerical index, or indices, with which it can be referred to, and 3.) the unique ID string with which it can be referred to (if one was defined).

If tools are organised in tool groups, this is shown by indentation.

The following example shows an (abridged) list matching the collection of the tutorial setup:

```
* Available Tools
harmonics | cycle          1          idHC
harmonics | off           2          idHO
apply                     3
 apply portamento          3,1       idP
 apply legato              3,2       idLG
  apply legato | slow       3,2,1    idLS
  apply legato | fast       3,2,2    idLF
change color              4          idCOL
 change color to RGB       4,1       idRGB
  to red                    4,1,1    idR
  to green                  4,1,2    idG
  to blue                   4,1,3    idB
 change color to TR/OFF    4,2       idTO
  to transparent            4,2,1    idT
  to off                    4,2,2    idO
```

When using the limited version of ConsoleTools, any tools of the collection that have been disabled due to the number of allowed tools having been reached will have their names in square brackets.

► list.p()

Prints a list of available profiles, as defined in the tool setup file. Each profile is listed with 1.) its description, 2.) the numerical index with which it can be referred to (corresponding to the order in which profiles have been added to the setup), and 3.) the unique ID string with which it can be referred to.

► list.s([#, …])

Prints a list showing the tools allocated to each currently used tool slot. One or more numbers can be passed as arguments, which will narrow the list to the corresponding slots only.

If several tools are allocated to a slot, they are listed in the order that they can be cycled through; a slot's current tool is marked with a double arrow at the start of its line. Any tool set up to be executed immediately upon being made the slot's current tool will have its description be preceded by an exclamation mark. When using the limited version of ConsoleTools, any tools for which immediate execution has been disabled due to the number of allowed tools having been reached will have the exlamation mark enclosed in square brackets.

The following example shows a list matching the allocation of the the tutorial setup's "colors" profile:

```
* Contents of all current slots:
Slot 1 currently contains:
  >>             ! to red
                 ! to blue
                 ! to transparent
Slot 2 currently contains:
  >>               to off
```

► `list.v()`
Prints a list of all user-defined bound variable names. Names bound to slots are listed first, followed by those bound to specific tools from the tool collection, and, lastly, by those bound to other functions.

The following example shows a list matching the initially bound variables of the tutorial setup:

```
* All current bound variable names:
These custom variable names have been bound to SLOTS:
 J              -> Slot 1
 K              -> Slot 2
These custom variable names have been bound to TOOLS:
 tr             -> to transparent
These custom variable names have been bound to FUNCTIONS:
 bpm            -> set tempo
 cl             -> set color (default: purple)
```

The tool and function lists will include the corresponding descriptive strings, if available.

► `list.a()`
Prints all available lists, in this order: tools, profiles, slots, bound variables.

▪ **Console use of the `help` function**
The `help` function provides access to the documentation strings that tools may have been provided with during setup. It is designed to accept a variety of input forms, and to match a reference to an existing documentation string even if the reference is indirect (i.e.: for a tool for which no explicit documentation string was provided, but which can obtain it from another tool that it is derived from). For how to link documentation strings to tools, see the section Tool Setup File, further below.

► `help([`*`reference1`* `[, `*`reference2`* `[, …]]])`
When called without any reference, `help` will print a basic overview for the six console use functions.

When arguments are passed to `help`, the function will try to match each argument to a tool, and will print documentation for that tool, if found.

A *`reference`* argument may be:

- a string, as a reference by string ID, or a Lua table containing integers, as a reference by index or index sequence; to print documentation for the corresponding tool (or tools)
  e.g: `help("idP")` or `help({2})` or `help({3,2,1})`
- a number, as a reference to a slot; to print documentation for that slot's current tool
  e.g: `help(1)`
- a Lua variable; to print documentation for a tool bound to that variable
  e.g.: `help(bpm)`

Tool Setup File

The tool setup file is where you build and manage your personalised tool collection. Just as with the functions for console use, you should find that simple things are simple to set up; large parts of the following documentation will only be needed for using the framework's more advanced features.

To familiarise yourself with the fundamental use cases, it is sufficient to only study those sections below that are marked to the left with a vertical line.

The file to be used as tool setup file is provided within the initialisation file initialiseConsoletools.lua, via the variable `pathUserToolSetupFile`. When first installing the framework, this path will be for an example file, tutorialToolSetup.lua, which serves as a practical introduction. It contains several sections (marked by outcommented headings), each of which is explained in detail below. The tutorial file is best viewed with a monospaced font.

Once you are familiar with setting up a tool collection, you should replace the tutorial file with your own user setup file. To do so, you may use the provided template userToolSetup.lua, or simply rename and adapt the tutorial file; at that point, make the appropriate edit in initialiseConsoletools.lua.

Provide DoApp emulation, for testing in external Lua environment (optional)

When running a tool setup file outside of Dorico, filling in the full paths of the initialisation file and the tool setup file itself in this section (between the double square brackets) will provide a simple emulation of the DoApp variable otherwise provided by the Dorico application, to facilitate testing of the tool setup, or development of custom Lua components. For how to use the enabled emulation, see the final section Dorico console emulation, below.

NB: the paths for default tools directory and default tool library file are still retrieved from the initialisation file. If needed for the emulation, they can be changed within the tool setup file via two dedicated functions in the auxiliary library, `.ct.setDefaultDirectory` and `.ct.setDefaultToolLibraryFile`.

Setup function declaration

The ConsoleTools framework provides six dedicated functions for setting up and managing a tool collection. For defining tools and profiles, and adding components to the auxiliary library, three primary functions are available via `ConsoletoolsByNotenlektorat.getPrimarySetupFunctions()`:

```
local addTool,addProfile,addToAuxLibrary = ConsoletoolsByNotenlektorat.getPrimarySetupFunctions()
```

Furthermore, since structuring arguments for the `addTool` function can become unwieldy at times, three secondary functions are available via `ConsoletoolsByNotenlektorat.getSecondarySetupFunctions()`:

```
local fromLuaScript,fromLuaLibrary,group = ConsoletoolsByNotenlektorat.getSecondarySetupFunctions()
```

How to use each of these functions is detailed in the relevant following sections. The symbol ► marks the statement of the general syntax of each function. Elements appearing in italics represent variable values. Optional arguments are enclosed in single square brackets.

In addition to the primary and secondary setup function (which must be explicitly required, as shown), all six console use functions (see above) are already available as global variables within the scope of the tool setup file, and can be used accordingly. The tool setup file may also be a suitable place to declare any other global variables that you may want tools to have access to.

Note that the six setup functions are declared as local variables in the tutorial file. In the unlikely case that you want to use any of them via the Dorico Script Console, make them available as global variables within the code of the tool setup file.

Example function (for tutorial setup only)

The function `fSetColor` is used as an example in the context of the tutorial setup, to demonstrate certain use cases. Usually, tools will not be provided as functions written directly in the tool setup file.

Extend auxiliary library (optional)

The framework's included auxiliary library (see Appendix A) can be extended, either by upgrading (i.e., replacing the initial version with a later one published by notenlektorat) or by adding your own custom functions and sublibraries.

► `addToAuxLibrary(`*`filename`* `[, `*`pathDirectory`*`])`

The name of the file from which to load the extension or to upgrade must be provided as the first argument. The path of the file's directory may be provided as the second argument *pathDirectory*; if omitted, the file is assumed to be located in the default tools directory, as provided via the variable `pathToolsDirectory` in initialiseConsoletools.lua.

To add library additions from several files, call `addToAuxLibrary` repeatedly, with the according file information. Note, however, that loading an upgrade file provided by notenlektorat will replace the whole library; therefore, upgrade files should always be loaded before any other additions.

For information on how to provide your own custom additions to the library, see Appendix A.

NB: upgrading the auxiliary library is only possible with the full version of ConsoleTools. However, it is still possible to extend the library of the limited version with custom additions.

Set up tools as available

Tools are set up as part of the tool collection via the `addTool` function, which has the following syntax:

► `addTool(`*`stringOrTableAsDescription`*`, `*`commandStringOrTable`* `[, `*`strRef`* `[, `*`doc`*`]])`

Each call of `addTool` results in a new numerical index added to the tool collection.

The first argument *stringOrTableAsDescription* provides the description of the tool (or tool group) as it will appear when printing available tools with the `list` function.

Usually, *stringOrTableAsDescription* will be a string; for certain cases when arranging tools into groups it may be a string enclosed in a table. It may omitted if the second argument is provided with the `group` function (see below), but not otherwise.

The second argument *commandStringOrTable* may be a Dorico command string, or a Lua table, either containing the tool function and its default arguments (if any), or containing another Lua table, to add a group of tools. The respective use cases are described further below.

Cases involving Lua tables as second argument tend to be tedious; it is usually preferable to use one of the secondary setup functions instead, as those are designed to return the necessary arguments from a streamlined input. Note: for some of the following examples, cases with a Lua table as second argument have their curly brackets corresponding to that table in larger size, for emphasis.

The third argument *stringRef* is optional, but recommended. It may be a unique identifier string by which the tool (or tool group) can always be referenced – a tool's numerical index is the result of the order in which tools where added in the tool setup file, and may therefore change when the tool setup is re-arranged. If a particular *stringRef* has already been assigned to another tool, the reference string will not be assigned a second time, and a warning message will be printed to the console.

The fourth argument *doc* is optional. It may be a string, to be printed by the `help` function (see above) when requesting documentation for the tool.

Alternatively, *doc* may be a multi-language encoding Lua table (as described in Appendix B). Note that tools from external files may have documentation embedded (see Appendix D), which will be extracted automatically; an explicit *doc* argument will override any such embedded documentation.

▪ **Adding a tool from a Dorico command string**

You can make any single Dorico command string into a tool by providing that string as the second argument to the `addTool` function:

```
addTool("harmonics | off",  [[UI.InvokePropertyEnableSwitch?Type=kNoteHarmonicType&Value=false]], "idH0")
```

In the tutorial setup file, the line above adds a tool for turning off the Dorico property for harmonic type. The tool will be assigned the numerical index 2, since this is the second call to `addTool` in the (tutorial) setup file; it will also be assigned the unique string identifier `idH0`.

To provide the command string, Lua's long string format is recommended, i.e.: enclosing the string in double square brackets, as in the example above. Command strings captured by Dorico's own macro recording feature are already formatted in this way and can thus be copied straightforwardly. NB: the string is not checked to be a valid Dorico command string; if you provide an invalid string, the tool will have no effect, as invalid strings are ignored by the Dorico application.

▪ **Adding a tool from a function**

You can make any function into a tool by providing it within a Lua table as the second argument to the `addTool` function. In practice, tools will rarely be set up in this way, as tool functions will usually come from external files – for this, the secondary setup functions are more convenient. But even when using those, being familiar with the general syntax for adding functions is helpful:

► `addTool(`*description*`, {`*fTool* `[, `*defaultArg1* `[, …]]} [, `*strRef*`, [`*doc*`]])`

The tool function *fTool* must be the table's first element; if there are other elements contained in the table (at indices iterable by Lua's `ipairs` function), these will be stored as default arguments for the tool. For example, the following line would add a tool which, when called without override arguments, simply prints the numbers 3, 2 and 1 to the console:

```
addTool("Lua's print as a tool", {print, 3, 2, 1})
```

▪ **Tool groups**

It is possible to group multiple tools together, and even to nest groups. This allows for more flexibility for managing tools, and also provides a convenient way for assigning tools collectively to a tool slot.

Setting up tool groups directly does involve nested Lua tables as arguments for `addTool`. This is a tedious and error-prone method; is recommended to use the secondary setup function `group` instead. Unless you are interested in understanding the framework as a Lua developer, you may safely skip ahead.

To add a tool group to the collection, `addTool`'s second argument must be a Lua table containing exactly one element, another table, at index 1. The inner table may contain a description for the group at index 1, and more tables as further elements (iterable by Lua's `ipairs`), one for each tool or subgroup:

► `addTool([`*description*`,] {{[`*descrGroup*`,] `*tToolOrSubGroup1* `[, …]}} [, `*strRefGroup*`, [`*docGroup*`]])`

Note: with such a table, `addTool`'s argument *description* may be omitted; *descrGroup* will apply instead.

The inner table's first argument *descrGroup* may be a string, or a table containing a string. The distinction is only relevant when nesting groups, in which case it can be used in the same way as described below for the descriptions of a group's items.

For a tool to be added to the group, the respective table *tToolOrSubGroup* must be structured like this:

► addTool({{*descrGroup*, {*descrTool*, {*fTool* [, …]} [, *strRefTool*, [*docTool*]]} [, …]}} [, …])

The table's first element *descrTool* may be a string, which will be concatenated with the next-higher description (here, this would be *descrGroup*). Alternatively, *descrTool* may be a string contained within a table, which will result in that string becoming the tool's description without any concatenation.

The second element must be a table, containing as its first element the tool function *fTool*; any further elements (as iterable by Lua's ipairs function) will be stored as the tool's primary default arguments.

To nest groups, the respective table *tToolOrSubGroup* must simply repeat the structure for grouping elements, without the need for an explicit description element. Therefore, *tSubGroup* in this example:

► addTool({{*descrGroup*, {*descrSubGroup*, *tSubGroup* [, *strRefSubGroup*, [*docSubGroup*]]} [, …]}} [, …])

… could be something like {{*tToolAsItem1*, *tToolAsItem2*, *tSubSubGroupAsItem3*, *tToolAsItem4*}}.

The optional *strRef* and *doc* elements, for both adding tools and adding subgroups, are analogous to the corresponding arguments of the addTool function. Documentation may be provided simply as a string, or as a multi-language encoding (see Appendix B) . Documentation for a group will "cascade down", meaning that any group element without its own explicit documentation will default to the group's.

The following table:

```
local tGroupColorTools=
{ { "change color ",
    { "to RGB",
      { { {{"to red"},         {fSetColor,"R"},        "idR"},
          {{"to green"},       {fSetColor,"G"},        "idG"},
          {{"to blue"},        {fSetColor,"W"},        "idB"},
      } },
      "idRGB"
    },
    { "to TR/OFF",
      { { {{"to transparent"}, {fSetColor,"00000000"}, "idT"},
          {{"to off"},         {fSetColor,false},      "idO"},
      } },
      "idTO"
    },
} };
```

…, when passed to the addTool function:

```
addTool(tGroupColorTools,"idCOL","- first argument…")
```

…, is equivalent to how the coloring tools are set up in the tutorial setup file with the group function, as described in the following section.

▪ **Adding tool groups with the group function**

The general syntax for the group function, shown here:

► group({*descrGroup*, *descr1*, *tItem1* [, *strRef1* [, *doc1*]] [, *descr2*, *tItem2* [, *strRef2* [, *doc2*]] [, …]]})

… is probably better understood by also studying the practical example on the next page.

The arguments for the `group` function can be enclosed in a single Lua table (i.e.: curly brackets). This is strictly optional, but makes changing or re-arranging a group structure in the setup file at a later time easier, with Lua's syntax being more permissive for table elements than for function arguments.

The initial argument *descrGroup* will usually be a string. However, when nesting groups, it may also be a table containing a string, as described below.

For each item of the group, up to four arguments can be provided: *descr*, *tItem*, *strRef* and *doc*.

The first two arguments, *descr* and *tItem*, are mandatory when adding a tool, but can also be replaced altogether with another call to `group`, which will create a nested group.

An item's *descr* argument may simply be a string, which will be concatenated with the next-higher description (here, this would be *descrGroup*). Alternatively, *descr* may be a string within a table; this will result in that string becoming the item's description without any concatenation.

In the example given below, the distinction can be seen in the way how, in the output of the `list` function, the very first description `"change color "` is prepended to the plain-string descriptions of both subgroups (`"to RGB"` and `"to TR/OFF"`), while the table-enclosed strings for each tool result in the tools being listed with their respective description, without any concatenation.

Note that the output of the `group` function allows for `addTool`'s first argument (which normally would give the description of the respective tool) to be omitted; *descrGroup* will apply instead. Likewise, when groups are nested, the argument *descr* can be left out for a subgroup (i.e.: another use of `group` in the place of *tItem*), with the description provided for the subgroup again applying instead.

To add a tool function to the group, *tItem* must be a table, containing as its first element the tool function. Any further elements will be stored as the tool's primary default arguments. This is analogous to the second argument of the `addTool` function (see above), except that the `group` function will not accept a Dorico command string as *tItem*.

The optional arguments *strRef* and *doc*, for both adding tools and adding subgroups, are analogous to the corresponding arguments of the `addTool` function (see above).

A group's reference ID *strRef*, if provided, will reference all of the group's items.

For groups, documentation will "cascade down": group items without explicit documentation will default to that of the group. Calling the `help` function (see above) on any of the tools defined in the example below displays the same documentation, provided once (indicated as `"- first argument…"`).

- **Example for use of the `group` function**

This section from the tutorial setup file is a single call to `addTool`, adding five different tools (all from the same tool function `fSetColor`, each with a different default argument), organised in nested groups:

```
addTool(group({"change color ",
               group({ "to RGB",
                     {"to red"},         {fSetColor,"R"},            "idR",
                     {"to green"},       {fSetColor,"G"},            "idG",
                     {"to blue"},        {fSetColor,"B"},            "idB",
                   }),                                               "idRGB",
               group({ "to TR/OFF",
                     {"to transparent"}, {fSetColor,"00000000"},     "idT",
                     {"to off"},         {fSetColor,false     },     "idO",
                   }),                                               "idTO",
            }),                                                      "idCOL", "- first argument…"
)
```

It adds the following arrangement of tools to the collection (as printed by the `list` function):

```
change color              4         idCOL
 change color to RGB       4,1      idRGB
  to red                    4,1,1    idR
  to green                  4,1,2    idG
  to blue                   4,1,3    idB
 change color to TR/OFF    4,2      idTO
  to transparent            4,2,1    idT
  to off                    4,2,2    idO
```

With this being the fourth call of `addTool` in the tutorial setup, the root index for all tools is 4, with index sequences being assigned according to the nesting structure. In addition to each tool having its own reference ID, the three groups themselves have IDs as well. Thus, entering `set(1,"idRGB")` will allocate three tools to slot 1, and `set(1,"idCOL")` will allocate five tools.

▪ **Adding a tool from an external Lua script with the `fromLuaScript` function**

It is easy enough in Lua to derive a tool function from an external script file, which you could then pass to `addTool`, as described above. It is more convenient, however, to use the secondary setup function `fromLuaScript` for this purpose, as it automates turning the file into a function:

► `addTool(`*`descr`*`, {fromLuaScript(`*`filename`*` [, `*`pathDirectory`*`]) [, `*`defaultArg1`*` [, …]]} [, …])`

The name of the script file must be provided as the first argument. The path of the file's directory may be provided as the second argument *pathDirectory*; if omitted, the file is assumed to be located in the default tools directory, as provided via the variable `pathToolsDirectory` in initialiseConsoletools.lua.

`fromLuaScript` will check whether the file exists and, if so, if it contains valid Lua code. In case of an error, a message will be printed to the console. Any embedded documentation contained in the file (see Appendix D) is automatically extracted with the tool function. To not override the embedded description (if provided), `false` may be passed as the *descr* argument for the enclosing call to `addTool`.

▪ **Example for use of the `fromLuaScript` function**

The following line from the tutorial setup file:

```
addTool("harmonics | cycle",{fromLuaScript("xmpl_setArtificialHarmonicAndCycleThroughPartials.lua"),{4,3,5,2}},"idHC")
```

… adds a tool from the external file xmpl_setArtificialHarmonicAndCycleThroughPartials.lua, and stores the table `{4,3,5,2}` as the tool's single default argument (in the context of that example script, the table denotes that triggering the tool repeatedly will cycle through those values for setting the Dorico property for the partial of an harmonic; see also Appendix C). The tool will be assigned the numerical index 1, since this is the first call to `addTool` in the setup file; it will also be assigned the unique string identifier `idHC`.

▪ **Adding a tool from an external Lua Library with the `fromLuaLibrary` function**

A tool library is a way of bundling tool functions, for example as a suite for related tasks, or simply as a personal collection of useful custom tool functions. Any Lua file that returns a table when executed is considered a tool library by the `fromLuaLibrary` function.

► `addTool(`*`descr`*`, {fromLuaLibrary(`*`index`*` [, `*`filename`*` [, `*`pathDirectory`*`]]) [, `*`defaultArg1`*` [, …]]} [, …])`

The first argument *index* is the table index (usually a string or number) under which the tool function is stored in the library. The second and third argument can be omitted if a default tool library file path was provided via the variable `pathDefaultToolLibraryFile` in initialiseConsoletools.lua (see Installation and General Setup, above), in which case the tool will be extracted from the default library file.

To extract from another library file, provide its file name as the second argument. The file's directory may be provided as the third argument *pathDirectory*; otherwise, the file is assumed to be located in the default tools directory, as provided via the variable `pathToolsDirectory` in initialiseConsoletools.lua.

Just as the previous function `fromLuaScript`, `fromLuaLibrary` will check whether the indicated file exists and, if so, if it contains valid Lua code; it also checks whether the file did return a table and, if so, if a function is stored at the provided index. In case of an error, a message will be printed to the console. Any relevant embedded documentation (see Appendix D) is automatically extracted and transferred with the tool function. To not override the embedded description (if provided), `false` may be passed as the *descr* argument for the enclosing call to `addTool`.

▪ **Examples for use of the `fromLuaLibrary` function**

The following section from the tutorial setup file:

```
addTool(group({"apply ",
               "portamento ",            {fromLuaLibrary("portamento")    }, "idP",
               group({ "legato ",
                       "| slow",         {fromLuaLibrary("legato"),330, 40}, "idLS",
                       "| fast",         {fromLuaLibrary("legato"),100,120}, "idLF",
               }),                                                          "idLG",
              })
)
```

… adds three tools from the default library file, all of which are concerned with changing playback-related properties of selected notes (see Appendix C for a basic explanation of the tutorial's default library file, xmpl_SK_OffsetAndVelocityAdjustmentsFromTempo.lua). The legato function is turned into two different tools, with appropriate default arguments for the respective use cases (legato within slow or fast tempo). The portamento function is turned into a tool directly.

`fromLuaLibrary` (as well as `fromLuaScript`, for that matter) can be used in any context where a tool function is expected. In the tutorial setup, it is also used to require a tool function to be bound to a Lua variable:

```
bind("bpm",fromLuaLibrary("setTempo"))
```

Bind custom variable names (optional)

It may be useful to bind some tools to Lua variables upon initialisation. This is done by simply placing the appropriate calls to the console use function `bind`, as one would type them into the Script Console. The example in the tutorial setup file is a collective call to bind tools to five variable names; each case is explained in more detail in the documentation for the `bind` function (see above). Note the notation with curly brackets enclosing all arguments, as is recommended for the tool setup file.

Set up profiles (optional)

The ConsoleTools framework allows to define tool profiles, i.e., custom presets of tool-to-slot allocations. Profiles are set up with the `addProfile` function, which has the following general syntax:

► `addProfile`(*strDescr*, *profileId* [, *fToCallBeforeApplication*, […]])(*slotId*)(*toolref1* [, …]) […]

Setting up profiles does involve function chaining, similar to the full syntax of the console use function `set`. In fact, the first function returned (receiving the *slotId* argument) behaves largely like `set`, in that it returns yet another function, which then receives the references for the tools to be allocated to the specified slot (as well as the same optional arguments that apply for the corresponding function returned by `set`). In contrast to `set`, however, that latter function does not return itself again, but instead a new instance of the previously returned function, expecting a new *slotId* argument, and so on.

The arguments for the chained function calls are documented in detail below. The relevant code from the tutorial file, included afterwards, should make these rather abstract descriptions more clear.

► addProfile(*strDescr, profileId* [, *fToCallBeforeApplication*, [, …]])(*slotId*)(*toolref1* [, …]) […]

For the initial call of addProfile, the first argument *strDescr* provides the description of the profile as it will appear when printing available tools with the list function; it must be a string. The second argument *profileId* must be a unique ID string; it is recommended to keep this string succinct, so that it is convenient to use with the console use function setProfile.

The optional third argument *fToCallBeforeApplication* may be a function. If provided, this function will be called each time the profile is applied (prior to the actual allocation of tools to slots), with all further arguments following *fToCallBeforeApplication* passed as well to that function. (Given the advanced nature of the use case, there is no corresponding example included in the tutorial file, on the assumption that developers having a need for this will probably be familiar with the concept.)

When called with appropriate arguments, addProfile returns another function:

► addProfile(*strDescr, profileId* [, …])(*slotId*)(*toolref1* [, …]) […]

Its only argument is *slotId*, which must be a number (the slot index) or a custom variable bound to a slot index (if such a variable has been defined in the previous block, see above).

The next function in the chain receives the references for tools to be allocated to the specified slot:

► addProfile(*strDescr, profileId* [, …])(*slotId*)([*bCallWhenMadeCurrent*,] *toolRef1* [, *defArgs1*] [, …]) […]

In regard to arguments that can be passed to it, it behaves exactly as the corresponding function call of the console use function set (see above), with references as numerical indices or unique string IDs, and optional default arguments enclosed in a Lua table. For immediate tool execution when toggling or cycling the slot's tool, the optional preceding boolean *bCallWhenMadeCurrent* may be passed as true.

If continued, the function chain alternates between the two previous functions from this point on:

► addProfile(*strDescr, profileId* [, …]) (*slotId*)([*bool*,] *toolRef* [, …]) (*slotId*)([*bool*,] *toolRef* [, …]) …etc.

In the tutorial tool setup file, three distinct profiles are defined:

```
addProfile("offsets / velocity","OV")
    (1)(3,1)
    (2)(3,2)

addProfile("harmonics","H")
    (1)("idHC")
    (2)("idHO")

addProfile("colors","C")
    (J)(true, 4,1,1, 0, 4,1,3, 0, 4,2,1)
    (K)("idTO")
```

The first line of each example is the initial call, with the two strings for description and unique ID. The remaining two lines contain the tool allocations, with one line per slot. For the first example, tools are referenced by numerical index (note that the index sequence 3,2 does actually allocate a tool group); for the second example tools are referenced by their unique ID. The third example specifies the slots by their Lua variables; it also includes the use of the boolean to flag tools for being called immediately upon becoming the slot's current tool.

The arrangement with linebreaks and indentation is merely for readability. The first example may just as well be written as `addProfile("offsets / velocity","OV")(1)(3,1)(2)(3,2)`. On the other hand, a multi-tool allocation as with the third example might be split into several allocations, if preferred:

```
addProfile("colors","C")
    (J)(true, 4,1,1)
    (J)(true, 4,1,3)
    (J)(true, 4,2,1)
    (K)("idTO")
```

List available tools and profiles on initialisation (optional)

A call to the console use function `list` (see above); to be outcommented or removed if so preferred.

Set up initial slot allocations (recommended)

It is recommended to define a slot allocation to be applied upon initialisation. This is done by simply placing the appropriate calls to console use functions `set` or `setProfile` (see above), as one would type them into the Script Console.

The example in the tutorial setup file allocates tools for coloring to slots 1 and 2. Note that the arguments match those of the “colors” profile, defined earlier in the file. Thus, replacing the current two lines in this block with `setProfile("C")` would do the same.

Dorico console emulation (in external Lua environment only)

This block is where you can simulate the use of the framework outside of Dorico, when running the tool setup file in an editor capable of interpreting Lua. This is predominantly intended for development, e.g. when writing your own Lua scripts for use with ConsoleTools; it may also be useful for testing changes to a tool setup.

```
do --[[ Dorico console emulation (in external Lua environment only) ]]
  if DoApp and DoApp.isEmulation then print("\nDORICO CONSOLE USE EMULATION: ")
                ← place emulation statements within this block
  end
end
```

Code in this block will not be executed when the framework is run within Dorico, and otherwise only if emulation has been enabled at the start of the tool setup file (see above). With active emulation, any statement placed in the block will be executed as if it was typed into the Script Console.

There is no actual Dorico application variable `DoApp` present in an external environment, but the variable of the same name provided by the emulation acts as a substitute. Just as the actual variable, it contains a function `.DoApp()`, which returns a pseudo-object with the method `:doCommand()`. A string passed to that method will be printed (with the prefix `">> [DOR]: "`) to the standard output, allowing to check the command strings that would be sent to Dorico.

For example, placing a call of the current tool of slot 1 for the tutorial setup into the block:

```
slot(1)
```

… will result in the following printout:

```
>> [DOR]: UI.InvokePropertyChangeValue?Type=kEventColour&Value=string: "#ffff0000"
```

Appendices

The following appendices may be of interest for you if you want to develop Lua scripts for use in connection with ConsoleTools.

Appendix A: Auxiliary library

Included with the ConsoleTools Lua framework comes an auxiliary library, aimed at facilitating development of Lua scripts for Dorico.

▪ Contents

The initial version (0.1.0) of the auxiliary library contains:

```
.checkLibrary                              function
.generateDoricoPropertyChangeFunction      function
.generateStaticDoricoCommandFunction       function
.getVersionString                          function
.printIf                                   function
.provideGlobalLog                          function
.toDoricoFractionString                    function
.toInteger                                 function
.ct                                        sublibrary
.ct.extractStringByLanguageIfAvailable      function
.ct.setDefaultDirectory                     function
.ct.setDefaultToolLibraryFile               function
.ct.setPreferredLanguage                    function
.dor                                       sublibrary
.dor.get                                    sublibrary
.dor.get.appVersionAsString                  function
.forDoricoCommandString                    sublibrary
.forDoricoCommandString.formatAsBool        function
.forDoricoCommandString.formatAsInt         function
.forDoricoCommandString.formatAsList        function
.forDoricoCommandString.formatAsNull        function
.forDoricoCommandString.formatAsString      function
.forDoricoCommandString.formatValue         function
.ui                                        sublibrary
.ui.genericCycle                            function
.ui.thresholdTimeHasBeenUnderrun            function
```

▪ Requiring the auxiliary library

The library is provided as a property of the DoApp variable:

```
local CTL = DoApp.CTL
```

The `CTL` property is also present with the emulation of DoApp, if enabled (see above); this makes it possible to access the library outside of Dorico.

▪ Detailed documentation for the auxiliary library

Documentation for library items can be accessed by way of the `.help` property, e.g.:

```
print(CTL.ui.genericCycle.help)
```

Items of the initial library (or of future upgrades by notenlektorat) will always have a `.help` property; other extensions of the library may not. Accessing `.help` for any sublibrary will, however, at least lists its contents, even if no further documentation was provided.

▪ Extending the auxiliary library

The auxiliary library can be extended through the setup function `addToAuxLibrary` (during tool setup, see above). A lua file for extending the library, when processed by `addToAuxLibrary`, will receive as its arguments the DoApp variable, as well as the auxiliary library itself. The file must return a Lua table; this table will be traversed via Lua's `pair` function. For each *key*/*value* pair, the library will be extended

- with a function at *key* if *value* is a function
- with a function at *key* if *value* is a table with a function at index 1; the table may also contain documentation for the function at index `help`
- with a sublibrary at *key* if *value* is a table *not* containing a function at index 1, with all indices of the table traversed recursively; the table may also contain documentation for the sublibrary at index `help` – Note: if a sublibrary already exists at index *key*, the table's elements will be added to that sublibrary

Accordingly, the following example Lua file would 1.) add a function `calculateTempo`, with the provided string as documentation, 2.) add a sublibrary `generate`, with the functions `row12tone` and `durations`, and 3.) add a function `formatForConsoleOutput` to the existing sublibrary `ui`, with documentation provided as a multi-language encoding (see Appendix B):

```
local DoApp, CTL = table.unpack({...})
local tableToReturn = {
   calculateTempo = {
      function(...) … end, help = "The function calculateTempo receives the following arguments […]"
   },
   generate = {
      row12tone = function() … end, durations = function() … end,
   },
   ui = {
      formatForConsoleOutput = {
         function(toFormat, indent) … end,
         help = {
            EN = "The function receives two arguments, toFormat and indent; […] ",
            DE = "Die Funktion erhaelt zwei Argumente, zuFormatieren und einruecken; […] ",
            args = {EN = {"toFormat", "indent"}, DE = {"zuFormatieren", "einruecken"}}
         }
      },
   },
}
return tableToReturn
```

Appendix B: Multi-language documentation encoding

At several points, script developers can provide documentation for tools. The most straight-forward way for this is to provide a string. However, it is also possible to do it in a way that allows users to have the documentation displayed in the language of their choice, if available.

The following five language keys are currently reserved: `EN`, `DE`, `FR`, `IT` and `ES`.

Users can set their preferred language in initialiseConsoletools.lua, by replacing the language key in line 19. If documentation for a tool is provided for the respective key, that string will be used; otherwise, the string for English (if provided) or the default string will be used.

A multi-language encoding is provided as a table instead of a string, e.g.:

{*strDefaultString*, EN=*strEnglishDocumentation*, FR=*strFrenchDocumentation* [, …] }

Users can provide multi-language encodings in this form within the tool setup file. In practice, however, such encodings are probably only of interest for script developers.

When documenting tool functions that take arguments, a multi-language encoding table may include the optional index `args`, holding a table that behaves similarly to a regular multi-language encoding, with the difference that it does not store a single string for a language, but a table containing the names (as strings) of the arguments. For documentation with an `args` index, a representation of the function syntax, using the argument names, will be displayed at the start of the documentation printout. Likewise, when documenting extensions of the auxiliary library, the `help` property for a function may be provided as a multi-language encoding with an `args` index. For an example of the latter, see the file code at the end of Appendix A.

With the auxiliary library function `.ct.extractStringByLanguageIfAvailable`, multi-language encodings can be evaluated by developers (however, this function does not consider the `arg` index, if present).

Appendix C: Example Lua scripts

This appendix provides basic commentary for the example Lua scripts included with ConsoleTools, focusing in particular on how the framework's auxiliary library can facilitate script development.

▪ xmpl_setArtificialHarmonicAndCycleThroughPartials.lua

This is a stand-alone tool script which, for a selection of notes, sets the Harmonics Type property to "Artificial" when called first; when called repeatedly within a certain time interval, it cycles the Partial property through a (user-customisable) sequence of values. It also is an example for a script that plays to the strengths of the ConsoleTools framework: the workflow that it makes possible can not realistically be achieved by running scripts via Dorico's **Script** menu. The script can be adapted with little effort to control other Dorico properties in the same way.

The first section (lines 4 to 19) provides variables of general usefulness, and as such is also included in the tool script template described later. These three variables are declared:

- `args` is a table containing any varargs passed to the function (e.g., these would be whatever is passed as default arguments when the tool is added in the tool setup file).
- `toDorico` is a function that streamlines the somewhat unwieldy notation found in Lua files from Dorico's macro recorder – `toDorico(str)` is equivalent to `app:doCommand(str)`. It also provides a simple emulation (similar to the one available with the initialised framework): in an external environment it prints strings to the standard output that would be sent to `DoApp` in Dorico.
- `CTL` holds the auxiliary library included with ConsoleTools (see Appendix A); if the framework is not present in the Lua environment, accessing this table will result in a warning message printed to the standard output before the script fails.

Note that the function `toDorico` is not actually used in this script. Instead, a slightly more sophisticated way of sending commands will be employed via the auxiliary library.

The next section (lines 24 to 62) sets up some script-specific variables and utilities.

First, any relevant varargs are retrieved (line 24): `tExplicitPartials` can be a table with integers (as values for the Partials property), in the order to be cycled through; `o_thresholdIntervalInMilliseconds` can be used to change the time interval with which the library function `ui.thresholdTimeHasBeenUnderrun` registers related subsequent tool calls (see line 71). By way of example: in the tutorial setup file (see line 59 in tutorialToolSetup.lua), an instance of this tool is provided with the table `{4,3,5,2}` as explicit partials; since no second argument is given, the time interval used will be the default value (300ms) of the threshold time function.

Then a global storage object is retrieved from the auxiliary library:

```
26 local log=CTL.provideGlobalLog("___xmpl_setArtificialHarmonicAndCycleThroughPartials")
```

The returned object, associated with the id passed to `.provideGlobalLog`, will be always the same during a Dorico session, and can be used to store and recall values over separate calls of the script (or even to exchange values between different scripts).

The auxiliary library is also used to generate two functions that will send the actual Dorico commands:

```
28 local dorSetHarmonicType=CTL.generateDoricoPropertyChangeFunction("kNoteHarmonicType")
29 local dorSetPartial=CTL.generateDoricoPropertyChangeFunction("kNoteHarmonic","int")
```

These functions handle the work of compiling a valid Dorico command string from a value, ensuring, among other things, that the value is properly formatted. (It should be noted that the generator function does know nothing at all about particular Dorico properties; it is up to developers to provide the arguments appropriate for the property in question.)

As a final preparation, two specific utilities are declared. `fillInExplicitPartialsIfNeeded` will provide the sequence of partials to be used, either from `tExplicitPartials` (the values of which it will validate), or, in its absence, from a default value table. `generateIdsFromPartials` turns any particular sequence of partials into id strings associated with that sequence; these ids make it possible to use multiple instances of the script (each with its own sequence) in parallel, i.e., by allocation to several tool slots.

The next section (lines 67 to 77) contains the main functionality of the script.

The two utilities are called to provide the sequence of partials and the associated ids. In line 69, the library function `.ui.genericCycle` is then used to turn the sequence into an object-like table with basic functionality for cycling through those values. The use of `log.init` ensures that the cycle object is the same for each respective sequence upon repeat execution of the script.

A side note: this script's sole use of `log.init` to initialise and also recall values is, technically, not good code, as the initial value is always sent to the `log` object, even though it is stored only once. The storage and retrieval of the cycling object is particularly bad form, because it includes the generation of a whole new cycling object every time. The best-practice way for that task would be to merely recall a value for the id first, and to use the `.init` method only if no value was returned:

```
local cycle=log(cycleId) or log.init(cycleId,CTL.ui.genericCycle(partialsToCycleThrough))
```

However, in the context of a Dorico Lua script, an inefficiency such as this usually will be of no practical consequence at all.

The library function `.ui.thresholdTimeHasBeenUnderrun` determines how much time has elapsed since the script has been last called for the respective sequence of partials. If that time exceeds the threshold interval, or if the script is called for the first time, `dorSetHarmonicType` is used to send a command string to enable the property for artifical harmonics, and the cycle object is reset to its initial state:

```
72 dorSetHarmonicType("kArtificial")
73 cycle.reset()
```

If, however, the script has been called again within the interval (the `else` branch), the current value in the sequence is retrieved while incrementing the cycle. The retrieved value is passed to `dorSetPartial`:

```
75 local strCommand,valAsString=dorSetPartial(cycle.up(true))
```

If a function from `.generateDoricoPropertyChangeFunction` did successfully compile a command string, the string is returned, and `nil` otherwise; the original argument, coerced to a string, is returned as a second value (whether a command was compiled or not), e.g. for use in an error message (as in line 76). Within this script – since any user-provided sequence of partials is validated earlier –, the only way that the `dorSetPartial` function could fail is if no explicit sequence of partials was provided and the default table within `fillInExplicitPartialsIfNeeded` was edited by mistake to contain a non-integer. Still, whenever a command string is generated from variable values, it is recommended to include a check for success.

The script concludes with two Lua long comments (lines 82 to 89), providing embedded documentation; for more details on this, see Appendix D.

- **xmpl_SK_OffsetAndVelocityAdjustmentsFromTempo.lua**

This is an example for a tool library script, i.e., it returns a table with functions. It has been adapted from original scripts by Sampo Kasurinen. The library provides a small number of tools that can facilitate the adjustment of offset and velocity properties of notes in relation to a specific tempo, as may be done to polish playback with certain sample libraries.

The first section (lines 6 to 25) is the same as in the previous example script, with one addition: it also declares the local function `embed`, which can be used to embed documentation with library items. See Appendix D for how to use this function.

The next section (lines 30 to 43) sets up some script-specific variables and utilities.

Just as described for the previous script, a global storage object is retrieved from the auxiliary library, as well as two functions for compiling the actual command strings:

```
30 local log=CTL.provideGlobalLog("___xmpl_SK_OffsetAndVelocityAdjustmentsFromTempo")

32 local dorSetVelocity=CTL.generateDoricoPropertyChangeFunction("kNoteVelocity","int")
33 local dorSetOffset=CTL.generateDoricoPropertyChangeFunction("kEventPlayStartOffset","int")
```

The `"int"` flags for formatting property values are especially relevant here: the values from the script's calculations will more often than not be floating point numbers, and using those directly in the command string would make the the string invalid. By specifying the format, the necessary conversion of the values is handled automatically.

The utilities `setVelocity` and `setOffset` add a simple validation mechanism to the property change functions, which prints a message to the console if, due to an invalid argument, no command string was compiled. Likewise, if the tempo value needed for the library's calculations has not yet been provided, the utility `tempoIsSet` prints a warning message, while also returning the check value.

The next section (lines 48 to 65) sets up the actual tool functions as local functions:

- `setTempo` is the function through which users can specify the tempo value that is the basis for the calculations below; the value is stored in the `log` object (under the key `"tempoInBPM"`)
- both `applyPortamento` as well as `applyLegato` will first confirm that a tempo value has been provided (this is done via the `tempoIsSet` utility, which, in turn, checks for the value through the `log` object); if so, `setOffset` and `setVelocity` are used to send the Dorico commands, with the offset value being calculated with a contextual formula.

Finally (lines 69 to 95), documentation is added for each tool function with the `embed` function, and the tools are arranged in a table. That table, the actual tool library, is then returned by the script.

▪ **xmpl_templateStandAloneScript.lua**
This is a simple template for a stand-alone tool script, with the general structure from the harmonics tool script described above.

A script based on this template, when executed via Dorico's **Script** menu, should generally work as intended even without the ConsoleTools framework. Likewise, running it in a an editor capable of interpreting Lua will allow to test its behaviour with the simple `DoApp` emulation set up in the "General declarations" section. Note, however, that using the auxiliary library (assigned to the variable `CTL` in line 15) will only work in Dorico with the framework initialised, and it will require running the script via a tool setup file with enabled Dorico emulation (see above) when used in an external environment.

The script's single command string (`[[Application.About]]`) brings up the About Dorico dialog, and is obviously meant to be replaced with useful code.

▪ **xmpl_templateToolLibraryScript.lua**
This is a simple template for a tool library, with the general structure from the offset-and-velocity library script above. The dummy functions, as well as the examples for embedded documentation, are included entirely for demonstrative purposes and are meant to be replaced with useful code.

Appendix D: Embedding documentation in custom tool script files

Tool scripts to be used with ConsoleTools can include documentation in a form that users can access through the console use function `help`. Two items may be provided as documentation: a description (as used, e.g., when displaying tools with `list`), and the documentation text itself; each item is optional, and may be provided as a string or a multi-language encoding (see Appendix B).

▪ **Embedding documentation for a stand-alone script**

For a script that is not a tool library script, documentation embedding is done via Lua long comments marked with two reserved string sequences, followed by a colon:

```
--[[CT_description: … ]]
--[[CT_documentation: … ]]
```

If such a long comment is found in the script, the string sequence following the colon is extracted; any whitespace characters on both ends of the sequence are trimmed automatically. The irregular escape sequence \] can be used to encode the character], to avoid two subsequent such characters ending the long comment too soon.

An extracted string starting with the { character and ending with the } character will be assumed to describe a multi-language encoding, in regular Lua notation.

For an example, see lines 82 to 89 of the file xmpl_setArtificialHarmonicAndCycleThroughPartials.lua.

▪ **Embedding documentation for tool library script functions**

For a tool library script (i.e., a Lua file that returns a table containing functions), documentation for each tool function may be added with a dedicated function provided by the framework:

```
local embed=ConsoletoolsByNotenlektorat.embedDocumentationInLuaLibrary
```

It has the the following syntax:

► `embed(`*`fTool`* `[, `*`documentation`* `[, `*`description`*`]])`

Note that the description is the last argument, with the rationale that the actual documentation is less likely to be omitted than the description (the latter, in practice, will often be replaced by users with their own custom strings when adding tools in the tool setup file).

For an example, see lines 69 to 93 of the file xmpl_SK_OffsetAndVelocityAdjustmentsFromTempo.lua.

Disclaimers

General
Dorico libraries and Lua scripts are provided by notenlektorat “as is” and “with all faults.” notenlektorat makes no representations or warranties of any kind concerning the safety, suitability, lack of viruses, or other harmful components of such Dorico libraries and/or Lua scripts. There are inherent dangers in the use of any software, and you are solely responsible for determining whether such Dorico libraries and/or Lua scripts are compatible with your equipment and other software installed on your equipment. You are also solely responsible for the protection of your equipment and backup of your data, and notenlektorat will not be liable for any damages you may suffer in connection with using, modifying, or distributing Dorico libraries and/or Lua scripts provided by notenlektorat.

Dorico library files (*.doricolib) and pre-loadable Dorico library files (*.preloadable.doricolib)
Library items imported through doricolib files by notenlektorat are generally designed in a way that any items provided through Dorico’s default “scorelibrary.xml” file are unaffected; any deliberate divergence from this practice will usually be explicitly stated in the main documentation above. If in doubt, users are nonetheless advised to double-check the details of a manual import via the comparison tools in the Library Manager before applying any changes.

If a doricolib file by notenlektorat is designed to interoperate with other (e.g. alternate) doricolib files, or with auxiliary Lua scripts, making changes to the “entityID” strings contained within the xml structure making up the doricolib file may prevent such interoperability.

Lua script files (*.lua)
While Lua scripting is available within Dorico, it is not officially supported. The underlying functionality may be changed or removed with each version of the Dorico scorewriting software, which may cause existing Lua scripts to cease operating as intended, or at all.

Due to current technical limitations, it is possible that running a Lua script may cause Dorico to end up in an undesirable state, resulting in generic error messages (which must be dismissed by the user) or, in rare cases, freezing or crashing of the Dorico application. The Dorico-specific commands used by notenlektorat’s Lua scripts will usually be of modest complexity, making errors with irreversable consequences unlikely; a list of conditions to be met when running a particular script may be provided in the main documentation above, meant to further minimise these risks. If in doubt, users are nonetheless advised to take appropriate measures to prevent data loss.