Unofficial DaVinci Resolve Scripting Documentation

Table of Contents

Version: 1.01
DaVinci Resolve version: 16.2.1

Official Resolve documentation updated as of 23 January 2020
Unofficial documentation updated as of 01 May 2020

About This Document

This document is a formatted copy of the official BlackmagicDesign DaVinci Resolve scripting documentation.

Keep in mind that this document might contain errors and might not be up to date with the current Resolve version.
If in doubt, always consult the official Resolve documentation provided by BlackmagicDesign.

In this package, you will find a brief introduction to the Scripting API for DaVinci Resolve Studio. Apart from this README.txt file, this package contains folders containing the basic import modules for scripting access (DaVinciResolve.py) and some representative examples.

From v16.2.0 onwards, the nodeIndex parameters accepted by SetLUT() and SetCDL() are 1-based instead of 0-based, i.e. 1 ⇐ nodeIndex ⇐ total number of nodes.

Overview

As with Blackmagic Design Fusion scripts, user scripts written in Lua and Python programming languages are supported. By default, scripts can be invoked from the Console window in the Fusion page, or via command line. This permission can be changed in Resolve Preferences, to be only from Console, or to be invoked from the local network. Please be aware of the security implications when allowing scripting access from outside of the Resolve application.

Using a script

DaVinci Resolve needs to be running for a script to be invoked.

For a Resolve script to be executed from an external folder, the script needs to know of the API location. You may need to set the these environment variables to allow for your Python installation to pick up the appropriate dependencies as shown below:

Mac OS X:

```
RESOLVE_SCRIPT_API="/Library/Application Support/Blackmagic Design/DaVinci
Resolve/Developer/Scripting/"
RESOLVE_SCRIPT_LIB="/Applications/DaVinci Resolve/DaVinci
Resolve.app/Contents/Libraries/Fusion/fusionscript.so"
PYTHONPATH="$PYTHONPATH:$RESOLVE_SCRIPT_API/Modules/"
```

Windows:

```
RESOLVE_SCRIPT_API="%PROGRAMDATA%\Blackmagic Design\DaVinci
Resolve\Support\Developer\Scripting\"
RESOLVE_SCRIPT_LIB="C:\Program Files\Blackmagic Design\DaVinci
Resolve\fusionscript.dll"
PYTHONPATH="%PYTHONPATH%;%RESOLVE_SCRIPT_API%\Modules\"
```

Linux:

```
RESOLVE_SCRIPT_API="/opt/resolve/Developer/Scripting/"
RESOLVE_SCRIPT_LIB="/opt/resolve/libs/Fusion/fusionscript.so"
PYTHONPATH="$PYTHONPATH:$RESOLVE_SCRIPT_API/Modules/"
(Note: For standard ISO Linux installations, the path above may need to be modified to
refer to /home/resolve instead of /opt/resolve)
```

As with Fusion scripts, Resolve scripts can also be invoked via the menu and the Console.

On startup, DaVinci Resolve scans the Utility Scripts directory and enumerates the scripts found in the Script application menu. Placing your script in this folder and invoking it from this menu is the easiest way to use scripts. The Utility Scripts folder is located in:

```
Mac OS X:   /Library/Application Support/Blackmagic Design/DaVinci
Resolve/Fusion/Scripts/Comp/
Windows:    %APPDATA%\Blackmagic Design\DaVinci Resolve\Fusion\Scripts\Comp\
Linux:      /opt/resolve/Fusion/Scripts/Comp/   (or /home/resolve/Fusion/Scripts/Comp/
depending on installation)
```

The interactive Console window allows for an easy way to execute simple scripting commands, to query or modify properties, and to test scripts. The console accepts commands in Python 2.7, Python 3.6 and Lua and evaluates and executes them immediately. For more information on how to use the Console, please refer to the DaVinci Resolve User Manual.

This example Python script creates a simple project:

```
#!/usr/bin/env python
import DaVinciResolveScript as dvr_script
resolve = dvr_script.scriptapp("Resolve")
fusion = resolve.Fusion()
projectManager = resolve.GetProjectManager()
projectManager.CreateProject("Hello World")
```

The resolve object is the fundamental starting point for scripting via Resolve. As a native object, it can be inspected for further scriptable properties - using table iteration and "getmetatable" in Lua and dir, help etc in Python (among other methods). A notable scriptable object above is fusion - it allows access to all existing Fusion scripting functionality.

Running DaVinci Resolve in headless mode

DaVinci Resolve can be launched in a headless mode without the user interface using the -nogui command line option. When DaVinci Resolve is launched using this option, the user interface is disabled. However, the various scripting APIs will continue to work as expected.

Basic Resolve API

Some commonly used API functions are described below (*). As with the resolve object, each object is inspectable for properties and functions.

Resolve

Return to Basic Resolve API

Method	Return Type	Comment
`Fusion()`	Fusion	Returns the Fusion object. Starting point for Fusion scripts.
`GetMediaStorage()`	MediaStorage	Returns media storage object to query and act on media locations.
`GetProjectManager()`	ProjectManager	Returns project manager object for currently open database.
`OpenPage(pageName)`	None	Switches to indicated page in DaVinci Resolve. Input can be one of ("media", "cut", "edit", "fusion", "color", "fairlight", "deliver").

ProjectManager

Return to Basic Resolve API

Method	Return Type	Comment
`CreateProject(projectName)`	Project	Creates and returns a project if projectName (text) is unique, and None if it is not.
`DeleteProject(projectName)`	Bool	Delete project in the current folder if not currently loaded

Method	Return Type	Comment
`LoadProject(projectName)`	Project	Loads and returns the project with name = projectName (text) if there is a match found, and None if there is no matching Project.
`GetCurrentProject()`	Project	Returns the currently loaded Resolve project.
`SaveProject()`	Bool	Saves the currently loaded project with its own name. Returns True if successful.
`CloseProject(project)`	Bool	Closes the specified project without saving.
`CreateFolder(folderName)`	Bool	Creates a folder if folderName (text) is unique.
`GetProjectListInCurrentFolder()`	[project names...]	Returns a list of project names in current folder.
`GetFolderListInCurrentFolder()`	[folder names...]	Returns a list of folder names in current folder.
`GotoRootFolder()`	Bool	Opens root folder in database.
`GotoParentFolder()`	Bool	Opens parent folder of current folder in database if current folder has parent.
`OpenFolder(folderName)`	Bool	Opens folder under given name.
`ImportProject(filePath)`	Bool	Imports a project under given file path. Returns true in case of success.
`ExportProject(projectName, filePath)`	Bool	Exports a project based on given name into provided file path. Returns true in case of success.
`RestoreProject(filePath)`	Bool	Restores a project under given backup file path. Returns true in case of success.

Project

Return to Basic Resolve API

Method	Return Type	Comment
`GetMediaPool()`	MediaPool	Returns the Media Pool object.
`GetTimelineCount()`	int	Returns the number of timelines currently present in the project.
`GetTimelineByIndex(idx)`	Timeline	Returns timeline at the given index, 1 ⇐ idx ⇐ project.GetTimelineCount()
`GetCurrentTimeline()`	Timeline	Returns the currently loaded timeline.

Method	Return Type	Comment
`SetCurrentTimeline(timeline)`	Bool	Sets given timeline as current timeline for the project. Returns True if successful.
`GetName()`	string	Returns project name.
`SetName(projectName)`	Bool	Sets project name if given projectname (text) is unique.
`GetPresetList()`	[presets...]	Returns a list of presets and their information.
`SetPreset(presetName)`	Bool	Sets preset by given presetName (string) into project.
`GetRenderJobList()`	[render jobs...]	Returns a list of render jobs and their information.
`GetRenderPresetList()`	[presets...]	Returns a list of render presets and their information.
`StartRendering(index1, index2, ...)`	Bool	Starts rendering for given render jobs based on their indices. If no parameter is given rendering would start for all render jobs.
`StartRendering([idxs...])`	Bool	Starts rendering for given render jobs based on their indices. If no parameter is given rendering would start for all render jobs.
`StopRendering()`	None	Stops rendering for all render jobs.
`IsRenderingInProgress()`	Bool	Returns true is rendering is in progress.
`AddRenderJob()`	Bool	Adds render job to render queue.
`DeleteRenderJobByIndex(idx)`	Bool	Deletes render job based on given job index (int).
`DeleteAllRenderJobs()`	Bool	Deletes all render jobs.
`LoadRenderPreset(presetName)`	Bool	Sets a preset as current preset for rendering if presetName (text) exists.
`SaveAsNewRenderPreset(presetName)`	Bool	Creates a new render preset by given name if presetName(text) is unique.
`SetRenderSettings({settings})`	Bool	Sets given settings for rendering. Settings is a dict, with support for the keys: "SelectAllFrames", "MarkIn", "MarkOut", "TargetDir", "CustomName".
`GetRenderJobStatus(idx)`	{status info}	Returns a dict with job status and completion percentage of the job by given job index (int).
`GetSetting(settingName)`	string	Returns value of project setting (indicated by settingName, string). Check the section below for more information.

Method	Return Type	Comment
`SetSetting(settingName, settingValue)`	Bool	Sets a project setting (indicated by settingName, string) to the value (settingValue, string). Check the section below for more information.
`GetRenderFormats()`	{render formats..}	Returns a dict (format → file extension) of available render formats.
`GetRenderCodecs(renderFormat)`	{render codecs...}	Returns a dict (codec description → codec name) of available codecs for given render format (string).
`GetCurrentRenderFormatAndCodec()`	{format, codec}	Returns a dict with currently selected format 'format' and render codec 'codec'.
`SetCurrentRenderFormatAndCodec(format, codec)`	Bool	Sets given render format (string) and render codec (string) as options for rendering.

MediaStorage

Return to Basic Resolve API

Method	Return Type	Comment
`GetMountedVolumeList()`	[paths...]	Returns a list of folder paths corresponding to mounted volumes displayed in Resolve's Media Storage.
`GetSubFolderList(folderPath)`	[paths...]	Returns a list of folder paths in the given absolute folder path.
`GetFileList(folderPath)`	[paths...]	Returns a list of media and file listings in the given absolute folder path. Note that media listings may be logically consolidated entries.
`RevealInStorage(path)`	None	Expands and displays a given file/folder path in Resolve's Media Storage.
`AddItemListToMediaPool(item1, item2, ···)`	[clips...]	Adds specified file/folder paths from Media Storage into current Media Pool folder. Input is one or more file/folder paths. Returns a list of the MediaPoolItems created.
`AddItemListToMediaPool([items ···])`	[clips...]	Adds specified file/folder paths from Media Storage into current Media Pool folder. Input is an array of file/folder paths. Returns a list of the MediaPoolItems created.

MediaPool

Return to Basic Resolve API

Method	Return Type	Comment
`GetRootFolder()`	Folder	Returns the root Folder of Media Pool
`AddSubFolder(folder, name)`	Folder	Adds a new subfolder under specified Folder object with the given name.
`CreateEmptyTimeline(name)`	Timeline	Adds a new timeline with given name.
`AppendToTimeline(clip1, clip2, ···)`	Bool	Appends specified MediaPoolItem objects in the current timeline. Returns True if successful.
`AppendToTimeline([clips])`	Bool	Appends specified MediaPoolItem objects in the current timeline. Returns True if successful.
`AppendToTimeline([{clipInfo}, ···])`	Bool	Appends list of clipInfos specified as a dict of "mediaPoolItem", "startFrame" (int), "endFrame" (int).
`CreateTimelineFromClips(name, clip1, clip2,···)`	Timeline	Creates a new timeline with specified name, and appends the specified MediaPoolItem objects.
`CreateTimelineFromClips(name, [clips])`	Timeline	Creates a new timeline with specified name, and appends the specified MediaPoolItem objects.
`CreateTimelineFromClips(name, [{clipInfo}])`	Timeline	Creates a new timeline with specified name, appending the list of clipInfos specified as a dict of "mediaPoolItem", "startFrame" (int), "endFrame" (int).
`ImportTimelineFromFile(filePath)`	Timeline	Creates timeline based on parameters within given file.
`GetCurrentFolder()`	Folder	Returns currently selected Folder.
`SetCurrentFolder(Folder)`	Bool	Sets current folder by given Folder.
`DeleteClips([clips])`	Bool	Deletes the specified clips in the media pool
`DeleteFolders([subfolders])`	Bool	Deletes the specified subfolders in the media pool
`MoveClips([clips], targetFolder)`	Bool	Moves specified clips to target folder.
`MoveFolders([folders], targetFolder)`	Bool	Moves specified folders to target folder.

Folder

Return to Basic Resolve API

Method	Return Type	Comment
`GetClipList()`	[clips...]	Returns a list of clips (items) within the folder.
`GetName()`	string	Returns user-defined name of the folder.
`GetSubFolderList()`	[folders...]	Returns a list of subfolders in the folder.

MediaPoolItem

Return to Basic Resolve API

Method	Return Type	Comment
`GetMetadata(metadataType)`	{metadata}	Returns a dict (metadata type → metadata value). If parameter is not specified returns all set metadata parameters.
`SetMetadata(metadataType, metadataValue)`	Bool	Sets metadata by given type and value. Returns True if successful.
`GetMediaId()`	string	Returns a unique ID name related to MediaPoolItem.
`AddMarker(frameId, color, name, note, duration)`	Bool	Creates a new marker at given frameId position and with given marker information.
`GetMarkers()`	{markers...}	Returns a dict (frameId → {information}) of all markers and dicts with their information. Example of output format: {96.0: {'color': 'Green', 'duration': 1.0, 'note': '', 'name': 'Marker 1'}, ...} In the above example - there is one 'Green' marker at offset 96 (position of the marker)
`DeleteMarkersByColor(color)`	Bool	Delete all markers of the specified color from the media pool item. "All" as argument deletes all color markers.
`DeleteMarkerAtFrame(frameNum)`	Bool	Delete marker at frame number from the media pool item.
`AddFlag(color)`	Bool	Adds a flag with given color (text).
`GetFlagList()`	[colors...]	Returns a list of flag colors assigned to the item.

Method	Return Type	Comment
`ClearFlags(color)`	Bool	Clears the flag of specified color from an item. If "All" argument is provided, all flags will be cleared.
`GetClipColor()`	string	Returns an item color as a string.
`SetClipColor(colorName)`	Bool	Sets color of an item based on the colorName (string).
`ClearClipColor()`	Bool	Clears clip color of an item.
`GetClipProperty(propertyName)`	{clipProperties}	Returns a dict (property name → property value) of an item. If no argument is provided, all clip properties will be returned. Check the section below for more information.
`SetClipProperty(propertyName, propertyValue)`	Bool	Sets into given propertyName (string) propertyValue (string). Check the section below for more information.

Timeline

Return to Basic Resolve API

Method	Return Type	Comment
`GetName()`	string	Returns user-defined name of the timeline.
`SetName(timelineName)`	Bool	Sets timeline name is timelineName (text) is unique.
`GetStartFrame()`	int	Returns frame number at the start of timeline.
`GetEndFrame()`	int	Returns frame number at the end of timeline.
`GetTrackCount(trackType)`	int	Returns a number of track based on specified track type ("audio", "video" or "subtitle").
`GetItemListInTrack(trackType, index)`	[items...]	Returns a list of Timeline items on the video or audio track (based on trackType) at specified index. 1 ⇐ index ⇐ GetTrackCount(trackType).
`AddMarker(frameId, color, name, note, duration)`	Bool	Creates a new marker at given frameId position and with given marker information.

Method	Return Type	Comment
`GetMarkers()`	{markers...}	Returns a dict (frameId → {information}) of all markers and dicts with their information. Example of output format: {96.0: {'color': 'Green', 'duration': 1.0, 'note': '', 'name': 'Marker 1'}, ...} In the above example - there is one 'Green' marker at offset 96 (position of the marker)
`DeleteMarkersByColor(color)`	Bool	Delete all markers of the specified color from the timeline. "All" as argument deletes all color markers.
`DeleteMarkerAtFrame(frameNum)`	Bool	Delete marker at frame number from the timeline.
`ApplyGradeFromDRX(path, gradeMode, item1, item2, ···)`	Bool	Loads a still from given file path (string) and applies grade to Timeline Items with gradeMode (int): 0 - "No keyframes", 1 - "Source Timecode aligned", 2 - "Start Frames aligned".
`ApplyGradeFromDRX(path, gradeMode, [items])`	Bool	Loads a still from given file path (string) and applies grade to Timeline Items with gradeMode (int): 0 - "No keyframes", 1 - "Source Timecode aligned", 2 - "Start Frames aligned".
`GetCurrentTimecode()`	string	Returns a string representing a timecode for current position of the timeline, while on Cut, Edit, Color and Deliver page.
`GetCurrentVideoItem()`	item	Returns current video timeline item.
`GetCurrentClipThumbnailImage()`	{thumbnailData}	Returns a dict (keys "width", "height", "format" and "data") with data containing raw thumbnail image data (RGB 8-bit image data encoded in base64 format) for current media in the Color Page. Example is provided in 6_get_current_media_thumbnail.py in Example folder.

TimelineItem

Return to Basic Resolve API

Method	Return Type	Comment
`GetName()`	string	Returns a name of the item.
`GetDuration()`	int	Returns a duration of item.
`GetEnd()`	int	Returns a position of end frame.
`GetFusionCompCount()`	int	Returns the number of Fusion compositions associated with the timeline item.
`GetFusionCompByIndex(compIndex)`	fusionComp	Returns Fusion composition object based on given index. 1 ⇐ compIndex ⇐ timelineItem.GetFusionCompCount()
`GetFusionCompNameList()`	[names...]	Returns a list of Fusion composition names associated with the timeline item.
`GetFusionCompByName(compName)`	fusionComp	Returns Fusion composition object based on given name.
`GetLeftOffset()`	int	Returns a maximum extension by frame for clip from left side.
`GetRightOffset()`	int	Returns a maximum extension by frame for clip from right side.
`GetStart()`	int	Returns a position of first frame.
`AddMarker(frameId, color, name, note, duration)`	Bool	Creates a new marker at given frameId position and with given marker information.
`GetMarkers()`	{markers...}	Returns a dict (frameId → {information}) of all markers and dicts with their information. Example of output format: {96.0: {'color': 'Green', 'duration': 1.0, 'note': '', 'name': 'Marker 1'}, ...} In the above example - there is one 'Green' marker at offset 96 (position of the marker)
`DeleteMarkersByColor(color)`	Bool	Delete all markers of the specified color from the timeline item. "All" as argument deletes all color markers.
`DeleteMarkerAtFrame(frameNum)`	Bool	Delete marker at frame number from the timeline item.
`AddFlag(color)`	Bool	Adds a flag with given color (text).
`GetFlagList()`	[colors...]	Returns a list of flag colors assigned to the item.
`ClearFlags(color)`	Bool	Clears the flag of specified color from an item. If "All" argument is provided, all flags will be cleared.
`GetClipColor()`	string	Returns an item color as a string.
`SetClipColor(colorName)`	Bool	Sets color of an item based on the colorName (string).

Method	Return Type	Comment
`ClearClipColor()`	Bool	Clears clip color of an item.
`AddFusionComp()`	fusionComp	Adds a new Fusion composition associated with the timeline item.
`ImportFusionComp(path)`	fusionComp	Imports Fusion composition from given file path by creating and adding a new composition for the item.
`ExportFusionComp(path, compIndex)`	Bool	Exports Fusion composition based on given index into provided file name path.
`DeleteFusionCompByName(compName)`	Bool	Deletes Fusion composition by provided name.
`LoadFusionCompByName(compName)`	fusionComp	Loads Fusion composition by provided name and sets it as active composition.
`RenameFusionCompByName(oldName, newName)`	Bool	Renames Fusion composition by provided name with new given name.
`AddVersion(versionName, versionType)`	Bool	Adds a new Version associated with the timeline item. versionType: 0 - local, 1 - remote.
`DeleteVersionByName(versionName, versionType)`	Bool	Deletes Version by provided name. versionType: 0 - local, 1 - remote.
`LoadVersionByName(versionName, versionType)`	Bool	Loads Version by provided name and sets it as active Version. versionType: 0 - local, 1 - remote.
`RenameVersionByName(oldName, newName, versionType)`	Bool	Renames Version by provided name with new given name. versionType: 0 - local, 1 - remote.
`GetMediaPoolItem()`	MediaPoolItem	Returns a corresponding to the timeline item media pool item if it exists.
`GetVersionNameList(versionType)`	[names...]	Returns a list of version names by provided versionType: 0 - local, 1 - remote.
`GetStereoConvergenceValues()`	{keyframes...}	Returns a dict (offset → value) of keyframe offsets and respective convergence values.
`GetStereoLeftFloatingWindowParams()`	{keyframes...}	For the LEFT eye → returns a dict (offset → dict) of keyframe offsets and respective floating window params. Value at particular offset includes the left, right, top and bottom floating window values.
`GetStereoRightFloatingWindowParams()`	{keyframes...}	For the RIGHT eye → returns a dict (offset → dict) of keyframe offsets and respective floating window params. Value at particular offset includes the left, right, top and bottom floating window values.

Method	Return Type	Comment
`SetLUT(nodeIndex, lutPath)`	Bool	Sets LUT on the node mapping the node index provided, 1 ⇐ nodeIndex ⇐ total number of nodes. The lutPath can be a relative path or absolute path. The operation will be successful for valid lut paths that Resolve has already discovered.
`SetCDL([CDL map])`	Bool	Keys of map are: "NodeIndex", "Slope", "Offset", "Power", "Saturation", where 1 ⇐ NodeIndex ⇐ total number of nodes. Example python code - SetCDL({"NodeIndex" : "1", "Slope" : "0.5 0.4 0.2", "Offset" : "0.4 0.3 0.2", "Power" : "0.6 0.7 0.8", "Saturation" : "0.65"})
`AddTake(mediaPoolItem, startFrame, endFrame)`	Bool	Adds a new take to take selector. It will initialise this timeline item as take selector if it's not already one. Arguments startFrame and endFrame are optional, and if not specified the entire clip will be added.
`GetSelectedTakeIndex()`	int	Returns the index of currently selected take, or 0 if the clip is not a take selector.
`GetTakesCount()`	int	Returns the number of takes in take selector, or 0 if the clip is not a take selector.
`GetTakeByIndex(idx)`	{takeInfo...}	Returns a dict (keys "startFrame", "endFrame" and "mediaPoolItem") with take info for specified index.
`DeleteTakeByIndex(idx)`	Bool	Deletes a take by index, 1 ⇐ idx ⇐ number of takes.
`SelectTakeByIndex(idx)`	Bool	Selects a take by index, 1 ⇐ idx ⇐ number of takes.
`FinalizeTake()`	Bool	Finalizes take selection.
`CopyGrades([tgtTimelineItems])`	Bool	Copies grade to all the items in tgtTimelineItems list. Returns true on success and false if any error occured.

List and Dict Data Structures

Beside primitive data types, Resolve's Python API mainly uses list and dict data structures. Lists are denoted by [...] and dicts are denoted by { ... } above. As Lua does not support list and dict data structures, the Lua API implements "list" as a table with indices, e.g. { [1] = listValue1, [2] = listValue2, ... }. Similarly the Lua API implements "dict" as a table with the dictionary key as first element, e.g. { [dictKey1] = dictValue1, [dictKey2] = dictValue2, ... }.

Looking up Project and Clip properties

This section covers additional notes for the functions "Project:GetSetting", "Project:SetSetting", "MediaPoolItem:GetClipProperty" and "MediaPoolItem:SetClipProperty". These functions are used to get and set properties otherwise available to the user through the Project Settings and the Clip Attributes dialogs.

The functions follow a key-value pair format, where each property is identified by a key (the settingName or propertyName parameter) and possesses a value (typically a text value). Keys and values are designed to be easily correlated with parameter names and values in the Resolve UI. Explicitly enumerated values for some parameters are listed below.

Some properties may be read only - these include intrinsic clip properties like date created or sample rate, and properties that can be disabled in specific application contexts (e.g. custom colorspaces in an ACES workflow, or output sizing parameters when behavior is set to match timeline)

Getting Values

Invoke "Project:GetSetting" or "MediaPoolItem:GetClipProperty" with the appropriate property key. To get a snapshot of all queryable properties (keys and values), you can call "Project:GetSetting" or "MediaPoolItem:GetClipProperty" without parameters (or with a NoneType or a blank property key). Using specific keys to query individual properties will be faster. Note that getting a property using an invalid key will return a trivial result.

Setting Values

Invoke "Project:SetSetting" or "MediaPoolItem:SetClipProperty" with the appropriate property key and a valid value. When setting a parameter, please check the return value to ensure the success of the operation. You can troubleshoot the validity of keys and values by setting the desired result from the UI and checking property snapshots before and after the change.

The following Project properties have specifically enumerated values:

superScale

The property value is an enumerated integer between 0 and 3 with these meanings: 0=Auto, 1=no scaling, and 2, 3 and 4 represent the Super Scale multipliers 2x, 3x and 4x.
Affects:

- x = Project:GetSetting('superScale') and Project:SetSetting('superScale', x)

timelineFrameRate

The property value is one of the frame rates available to the user in project settings under "Timeline frame rate" option. Drop Frame can be configured for supported frame rates by appending the frame rate with "DF", e.g. "29.97 DF" will enable drop frame and "29.97" will disable drop frame
Affects:

- x = Project:GetSetting('timelineFrameRate') and Project:SetSetting('timelineFrameRate', x)

The following Clip properties have specifically enumerated values:

superScale

The property value is an enumerated integer between 1 and 3 with these meanings: 1=no scaling, and 2, 3 and 4 represent the Super Scale multipliers 2x, 3x and 4x. Affects:

- x = MediaPoolItem:GetClipProperty('Super Scale') and MediaPoolItem:SetClipProperty('Super Scale', x)

Deprecated Resolve API Functions

The following API functions are deprecated.

ProjectManager

Method	Return Type	Comment
`GetProjectsInCurrentFolder()`	{project names...}	Returns a dict of project names in current folder.
`GetFoldersInCurrentFolder()`	{folder names...}	Returns a dict of folder names in current folder.

Project

Method	Return Type	Comment
`GetPresets()`	{presets...}	Returns a dict of presets and their information.
`GetRenderJobs()`	{render jobs...}	Returns a dict of render jobs and their information.
`GetRenderPresets()`	{presets...}	Returns a dict of render presets and their information.

MediaStorage

Method	Return Type	Comment
`GetMountedVolumes()`	{paths...}	Returns a dict of folder paths corresponding to mounted volumes displayed in Resolve's Media Storage.
`GetSubFolders(folderPath)`	{paths...}	Returns a dict of folder paths in the given absolute folder path.
`GetFiles(folderPath)`	{paths...}	Returns a dict of media and file listings in the given absolute folder path. Note that media listings may be logically consolidated entries.

Method	Return Type	Comment
`AddItemsToMediaPool(item1, item2, ⋯)`	{clips...}	Adds specified file/folder paths from Media Storage into current Media Pool folder. Input is one or more file/folder paths. Returns a dict of the MediaPoolItems created.
`AddItemsToMediaPool([items⋯])`	{clips...}	Adds specified file/folder paths from Media Storage into current Media Pool folder. Input is an array of file/folder paths. Returns a dict of the MediaPoolItems created.

Folder

Method	Return Type	Comment
`GetClips()`	{clips...}	Returns a dict of clips (items) within the folder.
`GetSubFolders()`	{folders...}	Returns a dict of subfolders in the folder.

MediaPoolItem

Method	Return Type	Comment
`GetFlags()`	{colors...}	Returns a dict of flag colors assigned to the item.

Timeline

Method	Return Type	Comment
`GetItemsInTrack(trackType, index)`	{items...}	Returns a dict of Timeline items on the video or audio track (based on trackType) at specified

TimelineItem

Method	Return Type	Comment
`GetFusionCompNames()`	{names...}	Returns a dict of Fusion composition names associated with the timeline item.
`GetFlags()`	{colors...}	Returns a dict of flag colors assigned to the item.
`GetVersionNames(versionType)`	{names...}	Returns a dict of version names by provided versionType: 0 - local, 1 - remote.