```python
### Script Generated by Control Surface Studio for Python 2.7 (resorted to default: no)
from __future__ import division
import Live
from _Framework.ControlSurface import ControlSurface
from _Framework.Layer import Layer
from _Framework.DeviceComponent import DeviceComponent
from _Framework.MixerComponent import MixerComponent
from _Framework.SliderElement import SliderElement
from _Framework.TransportComponent import TransportComponent
from _Framework.InputControlElement import *
from _Framework.ButtonElement import ButtonElement
from _Framework.ButtonMatrixElement import ButtonMatrixElement
from _Framework.SessionComponent import SessionComponent
from _Framework.EncoderElement import *
from Launchpad.ConfigurableButtonElement import ConfigurableButtonElement
import time
from itertools import imap, chain
from _Framework.Util import find_if
import collections
try:
    from user import *
except ImportError:
    pass
class css_atcoperator_imported_1(ControlSurface):
    def __init__(self, c_instance):
        super(css_atcoperator_imported_1, self).__init__(c_instance)
        with self.component_guard():
            global _map_modes
            _map_modes = Live.MidiMap.MapMode
            self.current_track_offset = 0
            self.current_scene_offset = 0
            global mixer
            num_tracks = 128
            num_returns = 24
            if hasattr(self, 'modifierList'):
                self.modifierList()
            if hasattr(self, 'customLists'):
                self.customLists()
            self._settings()
            self._inputs()
            self.turn_inputs_off()
            self.mixer = MixerComponent(num_tracks, num_returns)
            global active_mode
            self.debug_on = False
            self.mode_list()
            self.set_active_mode(self.modes[0])
            self.listening_to_tracks()
            self.song().add_tracks_listener(self.listening_to_tracks)
            self.song().add_tracks_listener(self._on_tracks_changed)
            self.song().add_scenes_listener(self._on_scenes_changed)
            self.all_track_device_listeners()
            self.song().view.add_selected_parameter_listener(self._on_selected_parameter_changed)
            self.create_clip_slot_map()
            try:
                self.user = user(self)
            except:
                pass
            self.call_script_reaction(None, None, 'script_was_initialised')
    def modifierList(self):
        global modifiers
        self.modifiers = {}
        self.modifiers["m1"] = {"value": 0}
        self.modifiers["m2"] = {"value": 0}
        self.modifiers["m3"] = {"value": 0}
        self.modifiers["m4"] = {"value": 0}
        self.modifiers["m5"] = {"value": 0}
        self.modifiers["m6"] = {"value": 0}
        self.modifiers["m7"] = {"value": 0}
        self.modifiers["m8"] = {"value": 0}
        self.modifiers["m9"] = {"value": 0}
        self.modifiers["m10"] = {"value": 0}
        self.modifiers["m11"] = {"value": 0}
        self.modifiers["m12"] = {"value": 0}
        self.modifiers["m13"] = {"value": 0}
        self.modifiers["m14"] = {"value": 0}
        self.modifiers["m15"] = {"value": 0}
        self.modifiers["m16"] = {"value": 0}
        self.modifiers["m17"] = {"value": 0}
        self.modifiers["m18"] = {"value": 0}
        self.modifiers["m19"] = {"value": 0}
        self.modifiers["m20"] = {"value": 0}
    def customLists(self):
        global lists
        self.lists = {}
        self.lists["list1"] = {"value": []}
        self.lists["list2"] = {"value": []}
        self.lists["list3"] = {"value": []}
        self.lists["list4"] = {"value": []}
        self.lists["list5"] = {"value": []}
        self.lists["list6"] = {"value": []}
        self.lists["list7"] = {"value": []}
        self.lists["list8"] = {"value": []}
        self.lists["list9"] = {"value": []}
        self.lists["list10"] = {"value": []}
    def _settings(self):
        self.global_feedback = "default"
        self.global_feedback_active = True
        self.global_LED_on = 127
        self.global_LED_off = 0
        self.controller_LED_on = 127
        self.controller_LED_off = 0
        self.led_on = self.controller_LED_on
        self.led_off = self.controller_LED_off
    def mode_list(self):
        global modes
        self.mode_conf = 8367
        self.modes = {}
        self.modes[0] = "1"
    def _inputs(self):
        self.input_map = [
            "midi_note_ch_15_val_1",
            "midi_note_ch_15_val_2",
            "midi_note_ch_15_val_3",
            "midi_note_ch_15_val_4"]
        self.midi_note_ch_15_val_1 = ConfigurableButtonElement(True, MIDI_NOTE_TYPE, 15, 1)
        self.midi_note_ch_15_val_1.set_on_off_values(self.led_on, self.led_off)
        self.midi_note_ch_15_val_1.add_value_listener(self.placehold_listener, identify_sender= False)
        self.midi_note_ch_15_val_1.pre_val = 0
        self.midi_note_ch_15_val_1.cur_val = 0
        self.midi_note_ch_15_val_2 = ConfigurableButtonElement(True, MIDI_NOTE_TYPE, 15, 2)
        self.midi_note_ch_15_val_2.set_on_off_values(self.led_on, self.led_off)
        self.midi_note_ch_15_val_2.add_value_listener(self.placehold_listener, identify_sender= False)
        self.midi_note_ch_15_val_2.pre_val = 0
        self.midi_note_ch_15_val_2.cur_val = 0
        self.midi_note_ch_15_val_3 = ConfigurableButtonElement(True, MIDI_NOTE_TYPE, 15, 3)
        self.midi_note_ch_15_val_3.set_on_off_values(self.led_on, self.led_off)
        self.midi_note_ch_15_val_3.add_value_listener(self.placehold_listener, identify_sender= False)
        self.midi_note_ch_15_val_3.pre_val = 0
        self.midi_note_ch_15_val_3.cur_val = 0
        self.midi_note_ch_15_val_4 = ConfigurableButtonElement(True, MIDI_NOTE_TYPE, 15, 4)
        self.midi_note_ch_15_val_4.set_on_off_values(self.led_on, self.led_off)
        self.midi_note_ch_15_val_4.add_value_listener(self.placehold_listener, identify_sender= False)
        self.midi_note_ch_15_val_4.pre_val = 0
        self.midi_note_ch_15_val_4.cur_val = 0
    def _mode1(self):
        self.show_message("Mode 1 is active")
        num_tracks = 128
        num_scenes = 1
        track_offset = self.current_track_offset
        scene_offset = self.current_scene_offset
        combination_mode = "off"
        feedbackArr = {}
        feedbackArr["ClipRecording"] = None
        feedbackArr["ClipStarted"] = None
        feedbackArr["ClipStopped"] = None
        feedbackArr["ClipTriggeredPlay"] = None
        feedbackArr["ClipTriggeredRecord"] = None
        feedbackArr["NoScene"] = None
        feedbackArr["RecordButton"] = None
        feedbackArr["Scene"] = None
        feedbackArr["SceneTriggered"] = None
        feedbackArr["StopAllOff"] = None
        feedbackArr["StopAllOn"] = None
        feedbackArr["StopClip"] = None
        feedbackArr["StopClipTriggered"] = None
        feedbackArr["StopTrackPlaying"] = None
        feedbackArr["StopTrackStopped"] = None
        clips = []
        stop_all = self.midi_note_ch_15_val_1
        stop_tracks = []
        scene_launch = [self.midi_note_ch_15_val_2]
        self.session_box(num_tracks, num_scenes, track_offset, scene_offset, clips, stop_all, stop_tracks, scene_launch, feedbackArr, combination_mode)
        self.midi_note_ch_15_val_4.add_value_listener(self.midi_note_ch_15_val_4_mode1_listener,identify_sender= False)
        self.midi_note_ch_15_val_3.add_value_listener(self.midi_note_ch_15_val_3_mode1_listener,identify_sender= False)
        self.midi_note_ch_15_val_2.add_value_listener(self.
midi_note_ch_15_val_2_mode1_listener,identify_sender= False)
        self._mode1_configs()
        self._mode1_led_listeners()
    def _remove_mode1(self):
        self.show_message("Mode 1 is removed")
        self.turn_inputs_off()
        combination_mode = "off"
        self.remove_session_box(combination_mode)
        self.midi_note_ch_15_val_4.remove_value_listener(self.midi_note_ch_15_val_4_mode1_listener)
        self.midi_note_ch_15_val_3.remove_value_listener(self.midi_note_ch_15_val_3_mode1_listener)
        self.midi_note_ch_15_val_2.remove_value_listener(self.midi_note_ch_15_val_2_mode1_listener)
        self._remove_mode1_led_listeners()
    def midi_note_ch_15_val_4_mode1_listener(self, value):
        self.midi_note_ch_15_val_4.cur_val = value
        if not hasattr(self.midi_note_ch_15_val_4, "pre_val"):
            self.midi_note_ch_15_val_4.pre_val = None
        if not hasattr(self.midi_note_ch_15_val_4, "prev_press_time"):
            self.midi_note_ch_15_val_4.prev_press_time = time.time()
        self.pick_brain(self.session_box_navigation_next_id_3)
        self.midi_note_ch_15_val_4.pre_val = value
        self.midi_note_ch_15_val_4.prev_press_time = time.time()
    def midi_note_ch_15_val_3_mode1_listener(self, value):
        self.midi_note_ch_15_val_3.cur_val = value
        if not hasattr(self.midi_note_ch_15_val_3, "pre_val"):
            self.midi_note_ch_15_val_3.pre_val = None
        if not hasattr(self.midi_note_ch_15_val_3, "prev_press_time"):
            self.midi_note_ch_15_val_3.prev_press_time = time.time()
        self.pick_brain(self.session_box_navigation_prev_id_7)
        self.midi_note_ch_15_val_3.pre_val = value
        self.midi_note_ch_15_val_3.prev_press_time = time.time()
    def midi_note_ch_15_val_2_mode1_listener(self, value):
        self.midi_note_ch_15_val_2.cur_val = value
        if not hasattr(self.midi_note_ch_15_val_2, "pre_val"):
            self.midi_note_ch_15_val_2.pre_val = None
        if not hasattr(self.midi_note_ch_15_val_2, "prev_press_time"):
            self.midi_note_ch_15_val_2.prev_press_time = time.time()
        self.pick_brain(self.session_box_navigation_advance_id_5)
        self.midi_note_ch_15_val_2.pre_val = value
        self.midi_note_ch_15_val_2.prev_press_time = time.time()
    def _mode1_configs(self):
        self.mode_1_configs_map = [
            "session_box_navigation_next_id_3",
            "session_box_navigation_prev_id_7",
            "session_box_navigation_advance_id_5"]
        self.session_box_navigation_next_id_3 = {}
        self.session_box_navigation_next_id_3["attached_to"] = "midi_note_ch_15_val_4"
        self.session_box_navigation_next_id_3["module"] = "self"
        self.session_box_navigation_next_id_3["element"] = "scroll_sess_offset"
        self.session_box_navigation_next_id_3["output_type"] = "func"
        self.session_box_navigation_next_id_3["func_arg"] = "cnfg"
        self.session_box_navigation_next_id_3["tracks_scenes"] = "scenes"
        self.session_box_navigation_next_id_3["ui_listener"] = "offset"
        self.session_box_navigation_next_id_3["feedback_brain"] = "feedback_sessbox_nav"
        self.session_box_navigation_next_id_3["ctrl_type"] = "increment"
        self.session_box_navigation_next_id_3["enc_first"] = 127
        self.session_box_navigation_next_id_3["enc_second"] = 0
        self.session_box_navigation_next_id_3["steps"] = 1
        self.session_box_navigation_next_id_3["switch_type"] = "momentary"
        self.session_box_navigation_next_id_3["LED_mapping_type_needs_feedback"] = "1"
        self.session_box_navigation_next_id_3["LED_feedback"] = "default"
        self.session_box_navigation_next_id_3["LED_feedback_active"] = "1"
        self.session_box_navigation_next_id_3["LED_on"] = "127"
        self.session_box_navigation_next_id_3["LED_off"] = "0"
        self.session_box_navigation_next_id_3["LED_send_feedback_to_selected"] = ["midi_note_ch_15_val_4"]
        self.session_box_navigation_next_id_3["json_id"] = 3
        self.session_box_navigation_next_id_3["mapping_name"] = "Session Box Navigation NEXT"
        self.session_box_navigation_next_id_3["mapping_type"] = "Session Box Navigation"
        self.session_box_navigation_next_id_3["parent_json_id"] = 1
        self.session_box_navigation_next_id_3["parent_name"] = "mode_1_id_1"
        self.session_box_navigation_prev_id_7 = {}
        self.session_box_navigation_prev_id_7["attached_to"] = "midi_note_ch_15_val_3"
        self.session_box_navigation_prev_id_7["module"] = "self"
        self.session_box_navigation_prev_id_7["element"] = "scroll_sess_offset"
        self.session_box_navigation_prev_id_7["output_type"] = "func"
        self.session_box_navigation_prev_id_7["func_arg"] = "cnfg"
        self.session_box_navigation_prev_id_7["tracks_scenes"] = "scenes"
        self.session_box_navigation_prev_id_7["ui_listener"] = "offset"
        self.session_box_navigation_prev_id_7["feedback_brain"] =
"feedback_sessbox_nav"
        self.session_box_navigation_prev_id_7["ctrl_type"] = "decrement"
        self.session_box_navigation_prev_id_7["enc_first"] = 127
        self.session_box_navigation_prev_id_7["enc_second"] = 0
        self.session_box_navigation_prev_id_7["steps"] = 1
        self.session_box_navigation_prev_id_7["switch_type"] = "momentary"

        self.session_box_navigation_prev_id_7["LED_mapping_type_needs_feedback"] = "1"
        self.session_box_navigation_prev_id_7["LED_feedback"] = "default"
        self.session_box_navigation_prev_id_7["LED_feedback_active"] = "1"
        self.session_box_navigation_prev_id_7["LED_on"] = "127"
        self.session_box_navigation_prev_id_7["LED_off"] = "0"

        self.session_box_navigation_prev_id_7["LED_send_feedback_to_selected"] = ["midi_note_ch_15_val_3"]
        self.session_box_navigation_prev_id_7["json_id"] = 7
        self.session_box_navigation_prev_id_7["mapping_name"] = "Session Box Navigation PREV"
        self.session_box_navigation_prev_id_7["mapping_type"] = "Session Box Navigation"
        self.session_box_navigation_prev_id_7["parent_json_id"] = 1
        self.session_box_navigation_prev_id_7["parent_name"] = "mode_1_id_1"
        self.session_box_navigation_advance_id_5 = {}
        self.session_box_navigation_advance_id_5["attached_to"] = "midi_note_ch_15_val_2"
        self.session_box_navigation_advance_id_5["module"] = "self"
        self.session_box_navigation_advance_id_5["element"] = "scroll_sess_offset"
        self.session_box_navigation_advance_id_5["output_type"] = "func"
        self.session_box_navigation_advance_id_5["func_arg"] = "cnfg"
        self.session_box_navigation_advance_id_5["tracks_scenes"] = "scenes"
        self.session_box_navigation_advance_id_5["ui_listener"] = "offset"
        self.session_box_navigation_advance_id_5["feedback_brain"] = "feedback_sessbox_nav"
        self.session_box_navigation_advance_id_5["ctrl_type"] = "increment"
        self.session_box_navigation_advance_id_5["enc_first"] = 127
        self.session_box_navigation_advance_id_5["enc_second"] = 0
        self.session_box_navigation_advance_id_5["steps"] = 1
        self.session_box_navigation_advance_id_5["switch_type"] = "momentary"

        self.session_box_navigation_advance_id_5["LED_mapping_type_needs_feedback"] = "1"
        self.session_box_navigation_advance_id_5["LED_feedback"] =
"default"
        self.session_box_navigation_advance_id_5["LED_feedback_active"] = "1"
        self.session_box_navigation_advance_id_5["LED_on"] = "127"
        self.session_box_navigation_advance_id_5["LED_off"] = "0"
        self.session_box_navigation_advance_id_5["LED_send_feedback_to_selected"] = ["midi_note_ch_15_val_2"]
        self.session_box_navigation_advance_id_5["json_id"] = 5
        self.session_box_navigation_advance_id_5["mapping_name"] = "Session Box Navigation Advance"
        self.session_box_navigation_advance_id_5["mapping_type"] = "Session Box Navigation"
        self.session_box_navigation_advance_id_5["parent_json_id"] = 1
        self.session_box_navigation_advance_id_5["parent_name"] = "mode_1_id_1"
    def _mode1_led_listeners(self):
        try:
            self._mode1_fire_all_feedback()
        except:
            self.log("_mode1_led_listeners tried to call _mode1_fire_all_feedback but it does not exist")
        try:
            self.song().add_tracks_listener(self._all_tracks_listener)
        except:
            self.log("_mode1_led_listeners tried to call add_tracks_listener but it does not exist")
        try:
            self.all_track_device_listeners()
        except:
            self.log("_mode1_led_listeners tried to call all_track_device_listeners but it does not exist")
        try:
            self._mode1_ui_listeners()
        except:
            self.log("_mode1_led_listeners tried to call _mode1_ui_listeners but it does not exist")
        self.track_feedback(1)
        self.device_feedback(1)
        self.mode_device_bank_leds(1)
    def _remove_mode1_led_listeners(self):
        try:
            self.song().remove_tracks_listener(self._all_tracks_listener)
        except:
            self.log("_remove_mode1_led_listeners tried to call remove_tracks_listener but it does not exist")
        try:
            self._remove_all_track_device_listeners()
        except:
            self.log("_remove_mode1_led_listeners tried to call _remove_all_track_device_listeners but it does not exist")
        try:
            self._remove_mode1_ui_listeners()
        except:
            self.log("_remove_mode1_led_listeners tried to call _remove_mode1_ui_listeners but it does not exist")
    def _mode1_ui_listeners(self):
        try:
            self._session.add_offset_listener(self.session_box_navigation_next_id_3_led_listener)
        except:
            self.log("_mode1_ui_listeners: self._session does not exist")
        try:
            self._session.add_offset_listener(self.session_box_navigation_prev_id_7_led_listener)
        except:
            self.log("_mode1_ui_listeners: self._session does not exist")
        try:
            self._session.add_offset_listener(self.session_box_navigation_advance_id_5_led_listener)
        except:
            self.log("_mode1_ui_listeners: self._session does not exist")
    def _remove_mode1_ui_listeners(self):
        try:
            self._session.remove_offset_listener(self.session_box_navigation_next_id_3_led_listener)
        except:
            self.log("remove__mode1_ui_listeners: self._session does not exist")
        try:
            self._session.remove_offset_listener(self.session_box_navigation_prev_id_7_led_listener)
        except:
            self.log("remove__mode1_ui_listeners: self._session does not exist")
        try:
            self._session.remove_offset_listener(self.session_box_navigation_advance_id_5_led_listener)
        except:
            self.log("remove__mode1_ui_listeners: self._session does not
exist")
    def _mode1_fire_all_feedback(self):
        self.session_box_navigation_next_id_3_led_listener()
        self.session_box_navigation_prev_id_7_led_listener()
        self.session_box_navigation_advance_id_5_led_listener()
    def session_box_navigation_next_id_3_led_listener(self):
        self.feedback_brain(self.session_box_navigation_next_id_3)
    def session_box_navigation_prev_id_7_led_listener(self):
        self.feedback_brain(self.session_box_navigation_prev_id_7)
    def session_box_navigation_advance_id_5_led_listener(self):
        self.feedback_brain(self.session_box_navigation_advance_id_5)
    ################## CORE: Python 2.7 #################
    def get_value_from_ranges(self, a1, b2, c3, d4, e5, f6, g7, h8, i9, j10, k11):
        logging = a1
        steps = b2
        round_down = c3
        current_input_value = d4
        i = {}
        i["minimum"] = e5
        i["maximum"] = f6
        i["decimal_places"] = g7
        i["steps"] = steps
        i["distance"] = i["maximum"] - i["minimum"]
        i["speed"] = i["distance"] / i["steps"]
        inn = self.step_values(i)
        o = {}
        o["minimum"] = h8
        o["maximum"] = i9
        o["decimal_places"] = j10
        o["reverse_mode"] = k11
        o["steps"] = steps
        o["distance"] = o["maximum"] - o["minimum"]
        o["speed"] = o["distance"] / o["steps"]
        out = self.step_values(o)
        closest_inn = self.f_n(inn, current_input_value, round_down)
        relative_out_value = out[closest_inn['index']]
        ret = {}
        ret['in'] = inn
        ret['selected_in'] = closest_inn
        ret['out'] = out
        ret["selected_out"] = relative_out_value
        if(logging == True):
            if(round_down == False):
                rounding = "up"
                rou_symb = str(">")
            else:
                rounding = "down"
                rou_symb = str("<")
            log_arr = []
            log_arr.append("In: " + str(current_input_value) )
            log_arr.append("Out: " + str(ret["selected_out"]) )
            log_arr.append("Steps: " + str(steps) )
            log_arr.append("Rounding: " + str(rounding) )
            log_arr.append("Rev: " + str(o["reverse_mode"]) )
            log_str = ' '.join(log_arr)
            table_arr = []
            table_arr.append(str("<table class='rangeValueTable'>") )
            step_arr = []
            count = 0
            for item in ret['in']:
                if(count==ret['selected_in']['index']):
                    td = "<td class='slctd'>"
                else:
                    td = "<td>"
                step_arr.append(td + str(count) + "</td>")
                count = count + 1
            step_str = ''.join(step_arr)
            in_arr = []
            count = 0
            for item in ret['in']:
                td = "<td>"
                if(count==ret['selected_in']['index']):
                    td = "<td class='slctd'>"
                in_arr.append(td + str(item) + "</td>")
                count = count + 1
            in_str = ''.join(in_arr)

            out_arr = []
            count = 0
            for item in ret['out']:
                td = "<td>"
                if(count==ret['selected_in']['index']):
                    td = "<td class='slctd'>"
                out_arr.append(td + str(item) + "</td>")
                count = count + 1
            out_str = ''.join(out_arr)
            rev = ""
            if o["reverse_mode"] == True:
                rev = "(Rev)"

            table_arr.append(str("<tr><td class='hd'>Steps</td>" +
… step_str + "</tr>") )
            table_arr.append(str("<tr><td class='hd'>In (" + rou_symb +
… str(current_input_value) + ")</td>" + in_str + "</tr>") )
            table_arr.append(str("<tr><td class='hd'>Out " + rev + "</td>"
+ out_str + "</tr>") )
            table_arr.append(str("</table>") )
            table_str = ''.join(table_arr)
            self.log_message("csslog: " + str(table_str) )
        return ret["selected_out"]

    def f_n(self, array, current_val, round_down = True):
        i = 0
        nearest = {}
        nearest['index'] = None
        nearest['value'] = None
        prev_idx = i
        prev_val = array[0]
        for array_val in array:
            if array_val == current_val:
                nearest['index'] = i
                nearest['value'] = array_val
                break
            elif current_val > prev_val and current_val < array_val:
                if round_down is True:
                    nearest['index'] = prev_idx
                    nearest['value'] = prev_val
                else:
                    nearest['index'] = i
                    nearest['value'] = array_val
                break
            else:
                prev_val = array_val
                prev_idx = i
                i = i + 1
        return nearest;
    def placehold_listener(self, value):
        return
    def pick_brain(self, obj):
        cnfg = obj.copy()
        if cnfg["output_type"] == "val":
                self.val_brain(cnfg)
        elif cnfg["output_type"] == "func":
            self.func_brain(cnfg)
        elif cnfg["output_type"] == "bool":
            self.bool_brain(cnfg)
    def should_it_fire(self, cnfg):
        controller = getattr(self, cnfg["attached_to"])
        cnfg["value"] = controller.cur_val
        cnfg["pre_val"] = controller.pre_val
        cnfg["prev_press_time"] = controller.prev_press_time
        timenow = time.time()
        fire = 0;
        if (cnfg["ctrl_type"] == "on/off" or cnfg["ctrl_type"] == "increment" or cnfg["ctrl_type"] == "decrement"):
            if(cnfg["switch_type"] == "delay"):
                if((cnfg["value"] == cnfg["enc_second"]) and (timenow - cnfg["prev_press_time"]) > cnfg["delay_amount"]):
                    fire = 1;
            elif(cnfg["switch_type"] == "toggle"):
                if cnfg["value"] == cnfg["enc_first"] or cnfg["value"] == cnfg["enc_second"]:
                    fire = 1;
            elif (cnfg["switch_type"] == "momentary" and cnfg["value"] == cnfg["enc_first"]):
                fire = 1;
        elif cnfg["ctrl_type"] == "absolute":
            if cnfg["value"] >= cnfg["enc_first"] and cnfg["value"] <= cnfg["enc_second"]:
                fire = 1;
        elif cnfg["ctrl_type"] == "relative":
            if cnfg["value"] == cnfg["enc_first"] or cnfg["value"] == cnfg["enc_second"]:
                fire = 1;
        return fire
    def bool_brain(self, cnfg):
        method_to_call = getattr(eval(cnfg["module"]), cnfg["element"])
        fire = self.should_it_fire(cnfg)
        if fire == 1:
            if cnfg["element"] == "solo" and self.song().exclusive_solo:
                for index in range(len(self.song().tracks)):
                    self.song().tracks[index].solo = False
                for index in range(len(self.song().return_tracks)):
                    self.song().return_tracks[index].solo = False
            if cnfg["element"] == "arm" and self.song().exclusive_arm:
                for index in range(len(self.song().tracks)):
                    self.song().tracks[index].arm = False
            if method_to_call is False:
                setattr(eval(cnfg["module"]), cnfg["element"], True)
            else:
                setattr(eval(cnfg["module"]), cnfg["element"], False)
    def func_brain(self, cnfg):
        fire = self.should_it_fire(cnfg)
        if fire == 1:
            method_to_call = getattr(eval(cnfg["module"]), cnfg["element"])
            if cnfg["func_arg"] != "" and cnfg["func_arg"] != "cnfg":
                method_to_call(cnfg["func_arg"])
            elif cnfg["func_arg"] == "cnfg":
                method_to_call(cnfg)
            else:
                method_to_call()
    def val_brain(self, cnfg):
        try:
            cnfg["current_position"] = getattr(eval(cnfg["module"]), cnfg["element"])
        except:
            self.show_message("This control does not exist in your session")
            return
        self._parameter_to_map_to = eval(cnfg["module"])
        if cnfg["ctrl_type"] != "on/off" and hasattr(self._parameter_to_map_to, "max") and hasattr(self._parameter_to_map_to, "min"):
            param_range = self._parameter_to_map_to.max - self._parameter_to_map_to.min
            if cnfg.has_key("minimum"):
                usermin = cnfg["minimum"] / 100.;
                min_value = float(usermin * param_range)
                cnfg["minimum"] = min_value + self._parameter_to_map_to.min
            if cnfg.has_key("maximum") and cnfg["mapping_type"] != "On/Off":
                usermax = cnfg["maximum"] / 100.;
                max_value = float(usermax * param_range)
                cnfg["maximum"] = max_value + self._parameter_to_map_to.min
        controller = getattr(self, cnfg["attached_to"])
        cnfg["value"] = controller.cur_val
        cnfg["pre_val"] = controller.pre_val
        if cnfg.has_key("decimal_places"):
            cnfg["current_position"] = round(cnfg["current_position"], cnfg["decimal_places"])
        if cnfg["ctrl_type"] == "absolute":
            cnfg["steps"] = (cnfg["enc_second"] - cnfg["enc_first"])
        if cnfg["ctrl_type"] != "on/off":
            cnfg["distance"] = cnfg["maximum"] - cnfg["minimum"]
            cnfg["speed"] = cnfg["distance"] / cnfg["steps"]
            cnfg["step_values"] = self.step_values(cnfg)
            cnfg["velocity_seq"] = self._velocity_seq(cnfg)

        if int(cnfg["current_position"]) < int(cnfg["minimum"]) or int(cnfg["current_position"]) > int(cnfg["maximum"]):
            new_val = self.snap_to_max_min(cnfg)
        elif cnfg["ctrl_type"] == "absolute":
            new_val = self.absolute_decision(cnfg)
        elif cnfg["ctrl_type"] == "relative":
            new_val = self.relative_decision(cnfg)
        elif cnfg["ctrl_type"] == "on/off" or cnfg["ctrl_type"] ==
"increment" or cnfg["ctrl_type"] == "decrement":
                new_val = self.button_decision(cnfg)
        try:
            setattr(eval(cnfg["module"]), cnfg["element"], new_val)
        except:
            return
    def snap_to_max_min(self, cnfg):

        if(cnfg["enc_first"] < cnfg["enc_second"]):
            enc_lowest = cnfg["enc_first"];
            enc_highest = cnfg["enc_second"]
        else:
            enc_lowest = cnfg["enc_second"];
            enc_highest = cnfg["enc_first"]

        if cnfg["snap_to"] == True and (cnfg["value"] <= enc_lowest or cnfg["value"] >= enc_highest):
            if int(cnfg["current_position"]) < int(cnfg["minimum"]):
                new_val = cnfg["minimum"]
                self.log("snapped to min")
            elif int(cnfg["current_position"]) > int(cnfg["maximum"]):
                new_val = cnfg["maximum"]
                self.log("snapped to max")
        else:
            new_val = cnfg["current_position"]
            self.show_message("remotify: snapping is off for this control. Check min / max values")
        return new_val
    def step_values(self, cnfg):
        calc = []
        for i in range(0, cnfg["steps"] +1):
            val = (i * cnfg["speed"]) + cnfg["minimum"]
            if cnfg.has_key("decimal_places"):
                val = round(val, cnfg["decimal_places"])
                if cnfg["decimal_places"] is 0:
                    val = int(val)
            calc.append(val)
        if "reverse_mode" in cnfg and cnfg["reverse_mode"] is True:
            calc = list(reversed(calc))
        return calc
    def relative_decision(self, cnfg):
        fire = 0
        new_val = cnfg["current_position"]
        if cnfg["value"] == cnfg["enc_second"]:
            max_min = "max"
            fire = 1
        elif cnfg["value"] == cnfg["enc_first"]:
            max_min = "min"
```

```
                fire = 1
            if fire == 0:
                return new_val
            if cnfg["current_position"] in cnfg["step_values"]:
                current_pos_index =
cnfg["step_values"].index(cnfg["current_position"])

                feedback = current_pos_index / cnfg["steps"] * 127
                feedback = round(feedback, 0)
                method_to_call = getattr(self, cnfg["attached_to"])
                incr_index = current_pos_index + 1
                decr_index = current_pos_index - 1
                if max_min == "max" and incr_index < len(cnfg["step_values"]):
                    incr = cnfg["step_values"][incr_index]
                    while incr == cnfg["current_position"]:
                        incr_index = incr_index + 1
                        if incr_index < len(cnfg["step_values"]):
                            incr = cnfg["step_values"][incr_index]
                        else:
                            break
                    new_val = incr
                elif max_min == "min" and decr_index >= 0:
                    decr = cnfg["step_values"][decr_index]
                    new_val = decr
                return new_val
            else:
                new_val = self.step_in_line(cnfg, max_min)
                return new_val
            return new_val
        def percent_as_value(self, param, percentage):
            param =          eval(param)
            if hasattr(param, 'max') and hasattr(param, 'min'):
                param_range = param.max - param.min
                val = percentage * param_range / 100
                return val
            else:
                self.log("param does not have min and/or max attribute(s)")
        def button_decision(self, cnfg):
            new_val = cnfg["current_position"]
            fire = self.should_it_fire(cnfg)
            if fire == 0:
                return new_val;
            if cnfg["ctrl_type"] == "on/off":
                if(cnfg["switch_type"] == "toggle"):
                    if cnfg["value"] == cnfg["enc_first"]:
                        new_val = cnfg["maximum"]
                        return new_val
                    elif cnfg["value"] == cnfg["enc_second"]:
```

```
                    new_val = cnfg["minimum"]
                    return new_val
            elif(cnfg["switch_type"] == "momentary"):
                if(cnfg["current_position"] == cnfg["maximum"]):
                    new_val = cnfg["minimum"]
                else:
                    new_val = cnfg["maximum"]
                return new_val
            elif(cnfg["switch_type"] == "delay"):
                if(cnfg["current_position"] == cnfg["maximum"]):
                    new_val = cnfg["minimum"]
                elif (cnfg["current_position"] == cnfg["minimum"]):
                    new_val = cnfg["maximum"]
                return new_val
            else:
                self.log("neither momentary or toggle were set for on off button")
                return new_val
        if cnfg["current_position"] in cnfg["step_values"]:
            current_pos_index = cnfg["step_values"].index(cnfg["current_position"])
            incr_index = current_pos_index + 1
            decr_index = current_pos_index - 1
            if cnfg["ctrl_type"] ==  "increment" and incr_index < len(cnfg["step_values"]):
                incr = cnfg["step_values"][incr_index]
                new_val = incr
            elif cnfg["ctrl_type"] == "decrement" and decr_index >= 0:
                decr = cnfg["step_values"][decr_index]
                new_val = decr
            return new_val
        else:
            if cnfg["ctrl_type"] ==  "increment":
                max_min = "max"
            elif cnfg["ctrl_type"] == "decrement": max_min = "min"
            new_val = self.step_in_line(cnfg, max_min)
            return new_val
        return new_val
    def step_in_line(self, cnfg, max_min):
        previous = int()
        step_num = 0
        speed = 0
        for step_val in cnfg["step_values"]:
            step_num += 1
            if cnfg["current_position"] > previous and cnfg["current_position"] < step_val:
                if max_min == "min":
                    speed = cnfg["current_position"] - previous
                    new_val = previous
                elif max_min == "max":
                    speed = step_val - cnfg["current_position"]
                    new_val = step_val
                break
            previous = step_val
        return new_val
    def absolute_decision(self, cnfg):
        if(cnfg["enc_first"] > cnfg["enc_second"]):
            self.log("enc_first is higher than enc_second, needs to be lower")
        new_val = cnfg["current_position"]
        if cnfg["pre_val"] is None:
            return new_val
        ######### Get pre_val details from list values #########
        ######### ######### ######### ######## ######
        if cnfg["pre_val"] in cnfg["velocity_seq"]:
            cnfg["previous_step_num"] = cnfg["velocity_seq"].index(cnfg["pre_val"])
            cnfg["previous_step_value"] = cnfg["step_values"][cnfg["previous_step_num"]]
        else:
            cnfg["previous_step_value"] = None
        ######### get value details from list #########
        ######### ######### ######### ######### ######
        if cnfg["value"] in cnfg["velocity_seq"]:
            cnfg["step_num"] = cnfg["velocity_seq"].index(cnfg["value"])
            cnfg["step_value"] = cnfg["step_values"][cnfg["step_num"]]
        else:
            cnfg["step_num"] = None
            cnfg["step_value"] = None

        ######### MAX OR MIN ########
        ######### ######### #########
        if cnfg["reverse_mode"] is False:
            if cnfg["value"] > cnfg["pre_val"]: max_min = "max"
            elif cnfg["value"] < cnfg["pre_val"]: max_min = "min"
        elif cnfg["reverse_mode"] is True:
            if cnfg["value"] > cnfg["pre_val"]: max_min = "min"
            elif cnfg["value"] < cnfg["pre_val"]: max_min = "max"
        inside_outside = self.inside_outside_checks(cnfg)
        if inside_outside is not False:
            self.log("inside outside was not false")
            return inside_outside
        ######### straight assign or takeover #########
        ######### ######### ######### ######### #######
        if cnfg["previous_step_value"] == cnfg["current_position"]:
            new_val = cnfg["step_value"]
        elif cnfg["takeover_mode"] == "None":
            new_val = cnfg["step_value"]
        elif cnfg["takeover_mode"] == "Pickup":
            new_val = self.pickup(cnfg, max_min)
        elif cnfg["takeover_mode"] == "Value scaling": new_val = self.value_scaling(cnfg, max_min)
        else: self.log("nothing got decided")

        return new_val
    def inside_outside_checks(self, cnfg):
        new_val = cnfg["current_position"]
        if cnfg["reverse_mode"] is False:
            minimum = cnfg["minimum"]
            maximum = cnfg["maximum"]
        elif cnfg["reverse_mode"] is True:
            minimum = cnfg["maximum"]
            maximum = cnfg["minimum"]
        ######### was outside and is still outside ######
        ######### ######### ######### ######### #########
        if (cnfg["pre_val"] < cnfg["enc_first"] and cnfg["value"] < cnfg["enc_first"]):
            self.log("was below and still below")
            return new_val
        elif (cnfg["pre_val"] > cnfg["enc_second"] and cnfg["value"] > cnfg["enc_second"]):
            self.log("was above and still above")
            return new_val
        ## 1. Going Below
        if (cnfg["pre_val"] >= cnfg["enc_first"] and cnfg["value"] < cnfg["enc_first"]):
            self.log("going below enter")
            if cnfg["takeover_mode"] == "Pickup":
                if cnfg["reverse_mode"] is False and cnfg["current_position"] > cnfg["previous_step_value"]:
                    return new_val
                elif cnfg["reverse_mode"] is True and cnfg["current_position"] < cnfg["previous_step_value"]:
                    return new_val
            if cnfg["reverse_mode"] is False:
                new_val = minimum
                self.log("going below 1")
                return new_val
            elif cnfg["reverse_mode"] is True:
                new_val = minimum
                self.log("going below 2")
                return new_val
        ## 2. Going Above
        if (cnfg["pre_val"] <= cnfg["enc_second"] and cnfg["value"] >
cnfg["enc_second"]):
            if cnfg["takeover_mode"] == "Pickup":
                self.log("THIS SHOULD FIRE 1")
                if cnfg["reverse_mode"] is False and cnfg["current_position"] < cnfg["previous_step_value"]:
                    self.log("THIS SHOULD FIRE 2")
                    return new_val
                elif cnfg["reverse_mode"] is True and cnfg["current_position"] > cnfg["previous_step_value"]:
                    return new_val
            if cnfg["reverse_mode"] is False:
                new_val = maximum
                self.log("going above 1")
                return new_val
            elif cnfg["reverse_mode"] is True:
                new_val = maximum
                self.log("going above 2")
                return new_val
        #########  >>0<< Coming inside ########
        ######### ######### ######### #########
        if (cnfg["pre_val"] < cnfg["enc_first"] and cnfg["value"] >= cnfg["enc_first"]):
            self.log("come in from below")

        elif (cnfg["pre_val"] > cnfg["enc_second"] and cnfg["value"] <= cnfg["enc_second"]):
            self.log("coming in from above")
        return False
    def _velocity_seq(self,cnfg):
        number_of_steps = cnfg['enc_second'] - cnfg['enc_first']
        arr = []
        i = 0
        sequence_num = cnfg['enc_first']
        while i <= number_of_steps:
            arr.append(sequence_num)
            i += 1
            sequence_num += 1
        return arr
    def pickup(self, cnfg, max_min):
        new_val = cnfg["current_position"]
        found = False
        if cnfg["previous_step_value"] is None:
            self.log("just entered")

            if cnfg["reverse_mode"] is False:
                if cnfg["pre_val"] < cnfg["enc_first"] and cnfg["step_value"] > cnfg["current_position"]:
                    new_val = cnfg["step_value"]
                    found = True
                    self.log("pickup 1 found")
                elif cnfg["pre_val"] > cnfg["enc_second"] and cnfg["step_value"] < cnfg["current_position"]:
                    new_val = cnfg["step_value"]
                    found = True
                    self.log("pickup 2 found")
            elif cnfg["reverse_mode"] is True:
                if cnfg["pre_val"] < cnfg["enc_first"] and cnfg["step_value"] < cnfg["current_position"]:
                    new_val = cnfg["step_value"]
                    found = True
                    self.log("pickup 3 found")
                elif cnfg["pre_val"] > cnfg["enc_second"] and cnfg["step_value"] > cnfg["current_position"]:
                    new_val = cnfg["step_value"]
                    found = True
                    self.log("pickup 4 found")

        else:
            self.log("we were already in here")

            if cnfg["previous_step_value"] < cnfg["current_position"] and cnfg["step_value"] > cnfg["current_position"]:
                new_val = cnfg["step_value"]
                found = True
                self.log("pickup 4 found")
            elif cnfg["previous_step_value"] > cnfg["current_position"] and cnfg["step_value"] < cnfg["current_position"] :
                new_val = cnfg["step_value"]
                found = True
                self.log("pickup 5 found")
            else:
                self.log("waiting for pickup")
        if found is False:
            msg = "remotify says: waiting for pickup " + str(cnfg["step_value"]) + " >> " + str(cnfg["current_position"])
            self.show_message(msg)
        return new_val
        step_num = cnfg["step_num"]
        step_value = cnfg["step_value"]
        remaining_steps = cnfg["steps"] - step_num
        new_val = cnfg["current_position"]
        distance_to_max = cnfg["maximum"] - cnfg["current_position"]
        distance_to_min = cnfg["current_position"] - cnfg["minimum"]
        speed_to_max = 0
        speed_to_min = 0
        if cnfg["current_position"] >= cnfg["minimum"] and
```

```
cnfg["current_position"] <= cnfg["maximum"]:
            if max_min == "max" and distance_to_max > 0:
                if cnfg["reverse_mode"] is False and remaining_steps > 0: speed_to_max = distance_to_max / remaining_steps
                elif cnfg["reverse_mode"] is True and step_num > 0: speed_to_max = distance_to_max / step_num
                if speed_to_max is not 0: new_val = speed_to_max + cnfg["current_position"]
            elif max_min == "min" and distance_to_min > 0:
                if cnfg["reverse_mode"] is False and step_num > 0: speed_to_min = distance_to_min / step_num
                elif cnfg["reverse_mode"] is True and remaining_steps > 0: speed_to_min = distance_to_min / remaining_steps
                if speed_to_min is not 0: new_val = cnfg["current_position"] - speed_to_min
        return new_val
    def value_scaling(self, cnfg, max_min):
        step_num = cnfg["step_num"]
        step_value = cnfg["step_value"]
        remaining_steps = cnfg["steps"] - step_num
        new_val = cnfg["current_position"]
        distance_to_max = cnfg["maximum"] - cnfg["current_position"]
        distance_to_min = cnfg["current_position"] - cnfg["minimum"]
        speed_to_max = 0
        speed_to_min = 0
        if cnfg["current_position"] >= cnfg["minimum"] and cnfg["current_position"] <= cnfg["maximum"]:
            if max_min == "max" and distance_to_max > 0:
                if cnfg["reverse_mode"] is False and remaining_steps > 0: speed_to_max = distance_to_max / remaining_steps
                elif cnfg["reverse_mode"] is True and step_num > 0: speed_to_max = distance_to_max / step_num
                if speed_to_max is not 0: new_val = speed_to_max + cnfg["current_position"]
            elif max_min == "min" and distance_to_min > 0:
                if cnfg["reverse_mode"] is False and step_num > 0: speed_to_min = distance_to_min / step_num
                elif cnfg["reverse_mode"] is True and remaining_steps > 0: speed_to_min = distance_to_min / remaining_steps
                if speed_to_min is not 0: new_val = cnfg["current_position"] - speed_to_min
        return new_val
    def track_num(self, track_num):
        if ((hasattr(self, '_session')) and (self._session is not None)):
            track_num = track_num + self._session._track_offset
        else:
            track_num = track_num
        return track_num
    def scene_num(self, scene_num):
        if ((hasattr(self, '_session')) and (self._session is not None)):
            scene_num = scene_num + self._session._scene_offset
        else:
            scene_num = scene_num
        return scene_num
    def log_cnfg_settings(self, cnfg):
        for i in cnfg:
            text = i + ": " + str(cnfg[i])
            self.log(text)
    def dump(self, obj):
        for attr in dir(obj):
            try:
                self.log_message("csslog: %s" % (attr))
            except:
                self.log_message("next")
    def log(self, msg):
        if self.debug_on is True:
            self.log_message("csslog:" + str(msg))
    def pret(self, ugly):
        for key,value in sorted(ugly.items()):
            self.log_message(key)
            self.log_message(value)
            self.log_message("")

    ################## Extra Functions: Python 2.7 #################
    def get_list(self, list_name):
        try:
            if list_name in self.lists:
                return self.lists[list_name]["value"]
            else:
                self.log_message('csslog: The custom list "' +
str(list_name) + '" does not exist')
                return False
        except Exception as e:
            self.log_message('csslog: There was an error getting a custom
list with "get_list", ' + str(e))
            return False
    def get_list_length(self, list_name):
        theList = self.get_list(list_name)
        if theList is False:
            return False
        return len(theList)
    def get_list_item(self, list_name, item_num):
        try:
            theList = self.get_list(list_name)
            if theList is False:
                return False
```

```python

            list_len = self.get_list_length(list_name)
            if list_len is False:
                return
            if list_len >= item_num:
                return theList[item_num - 1]
            else:
                self.log_message('csslog: Custom list "' + str(list_name) + '" does not have ' + str(item_num) + ' items')
                return False
        except Exception as e:
            self.log_message('csslog: There was an error in "get_list_item"', str(e))
            return False
    def add_to_list(self, list_name, value_to_add, position):
        try:
            theList = self.get_list(list_name)
            if theList is False:
                return False
            list_len = self.get_list_length(list_name)
            if position is None or position > list_len:
                position = list_len
            theList.insert(position, value_to_add)
        except Exception as e:
            self.log_message('csslog: There was an error in "add_to_list()", ' + str(e))
            return False;
    def remove_from_list(self, list_name, position):
        try:
            theList = self.get_list(list_name)
            if theList is False:
                return False
            list_len = self.get_list_length(list_name)
            if list_len == 0:
                self.log_message("csslog: Nothing to delete from list '" + str(list_name) + "' as it's already empty")
                return
            if position > list_len:
                self.log_message("csslog: Custom list '" + str(list_name) + "' does not contain " + str(position) + " items")
                return False
            if position is None:
                position = list_len
            theList.pop(position)
        except Exception as e:
            self.log_message('csslog: There was an error in "remove_from_list", ' + str(e))
            return False;
    def clear_list(self, list_name):
        try:
            theList = self.get_list(list_name)
            if theList is False:
                return False
            del theList[:]
        except Exception as e:
            self.log_message('csslog: There was an error in "clear_list", ' + str(e))
            return False;
    def get_num_of_tracks(self, track_slug):
        try:
            s = "self.song()." + track_slug
            s = eval(s)
            return len(s)
        except:
            self.log_message('There was an error in get_num_of_tracks()')
            return -1

    def get_num_of_scenes(self):
        try:
            s = "self.song().scenes"
            s = eval(s)
            return len(s)
        except:
            self.log_message('There was an error in get_num_of_scenes()')
            return -1

    def get_num_of_devices(self, track_slug):
        try:
            s = "self.song()." + track_slug + ".devices"
            s = eval(s)
            return len(s)
        except:
            self.log_message('There was an error in get_num_of_devices()')
            return -1

    def get_selected_track_num(self):
        track = self.song().view.selected_track
        track = self.tuple_index(self.song().tracks, track)
        return track

    def get_selected_scene_num(self):
        scene = self.song().view.selected_scene
        scene = self.tuple_index(self.song().scenes, scene)
        return scene

    def get_selected_device_num(self, track_slug):
        try:
            device_list = "self.song()." + track_slug + ".devices"
            selected_device = "self.song()." + track_slug + ".view.selected_device"
            s = self.tuple_index(eval(device_list), eval(selected_device))
            if(s == False and s != 0):
                s = -1
            return s
        except:
            self.log_message('csslog: There was an error in get_num_of_devices()')
            return -1
    def get_active_mode_id(self):
        global active_mode
        return active_mode
    def get_sessbox_track_offset(self):
        if hasattr(self, '_session') and self._session is not None:
            return self._session._track_offset
        else:
            return -1
    def get_sessbox_scene_offset(self):
        if hasattr(self, '_session') and self._session is not None:
            return self._session._scene_offset
        else:
            return -1
    def get_sessbox_last_track_number(self):
        if hasattr(self, '_session') and self._session is not None:
            last_track = self._session._track_offset + self._session.width()
            return last_track
        else:
            return -1
    def get_sessbox_last_scene_number(self):
        if hasattr(self, '_session') and self._session is not None:
            last_scene = self._session._scene_offset + self._session.height()
            return last_scene
        else:
            return -1
    def get_sessbox_width(self):
        if hasattr(self, '_session') and self._session is not None:
            return self._session.width()
        else:
            return -1
    def get_sessbox_height(self):
        if hasattr(self, '_session') and self._session is not None:
            return self._session.height()
        else:
            return -1

    def get_sessbox_is_active(self):
        if hasattr(self, '_session') and self._session is not None:
            return True
        else:
            return False
    def set_highlighted_track(self, n):
        self.song().view.selected_track = self.song().tracks[n]

    def set_highlighted_scene(self, n):
        self.song().view.selected_scene = self.song().scenes[n]
    def set_sessionbox_offsets(self, track_offset, scene_offset):
        if hasattr(self, '_session') and self._session is not None:
            self._session.set_offsets(track_offset, scene_offset)
    def set_sessionbox_combo_mode(self, combo):
        if hasattr(self, '_session') and self._session is not None:
            if combo == True:
                self._session._link
            elif combo == False:
                self._session._unlink
    def _quantizeDict(self):
        grid_setting = str(self.song().view.highlighted_clip_slot.clip.view.grid_quantization)
        is_it_triplet = self.song().view.highlighted_clip_slot.clip.view.grid_is_triplet
        if (is_it_triplet is True):
            grid_setting += "_triplet"
        RecordingQuantization = Live.Song.RecordingQuantization
        quantDict = {}
        quantDict["g_thirtysecond"] = RecordingQuantization.rec_q_thirtysecond
        quantDict["g_sixteenth"] = RecordingQuantization.rec_q_sixtenth
        quantDict["g_eighth"] = RecordingQuantization.rec_q_eight
        quantDict["g_quarter"] = RecordingQuantization.rec_q_quarter
        quantDict["g_eighth_triplet"] = RecordingQuantization.rec_q_eight_triplet
        quantDict["g_sixteenth_triplet"] = RecordingQuantization.rec_q_sixtenth_triplet
        return quantDict[grid_setting];
    def _arm_follow_track_selection(self):
        for track in self.song().tracks:
            if track.can_be_armed:
                track.arm = False
        if self.song().view.selected_track.can_be_armed:
            self.song().view.selected_track.arm = True
    def turn_inputs_off(self):
        send_feedback = False
        if hasattr(self, "global_feedback"):
            if self.global_feedback == "custom":
                if self.global_feedback_active == True:
                    send_feedback = True
            elif hasattr(self, "controller_LED_on") and hasattr(self, "controller_LED_off"):
                send_feedback = True
        if send_feedback == True:
            for input_name in self.input_map:
                input_ctrl = getattr(self, input_name)
                input_ctrl.send_value(self.led_off)
    def feedback_brain(self, obj):
        cnfg = obj.copy()
        try:
            method_to_call = getattr(self, cnfg["feedback_brain"])
            method_to_call(cnfg)
        except:
            return
    def feedback_bool(self, feedback_to):
        control =   eval("self." + str(feedback_to["attached_to"]))
        param =         eval(feedback_to["module"] + "." + feedback_to["ui_listener"])
        ctrl_on =   self.feedback_which_ctrl_on_off(feedback_to, "on")
        ctrl_off =  self.feedback_which_ctrl_on_off(feedback_to, "off")
        if(feedback_to["mapping_type"] == "Mute"):
            if param == False:
                send_val = ctrl_on
            elif param == True:
                send_val = ctrl_off
        else:
            if param == True:
                send_val = ctrl_on
            elif param == False:
                send_val = ctrl_off
        self.feedback_handler(feedback_to, send_val)
    def feedback_on_off(self, feedback_to):
        param =         eval(feedback_to["module"])
        ctrl_on =   self.feedback_which_ctrl_on_off(feedback_to, "on")
        ctrl_off =  self.feedback_which_ctrl_on_off(feedback_to, "off")
        param_value = round(param.value,2)
        mapping_type = str(feedback_to["mapping_type"])
        if feedback_to.has_key("maximum") and feedback_to.has_key("minimum"):
            max_val = feedback_to["maximum"]
            min_val = feedback_to["minimum"]
        elif hasattr(param, "max") and hasattr(param, "min"):
            max_val = param.max
            max_val = round(max_val,2)
            min_val = param.min
            min_val = round(min_val,2)
        else:
            self.log_message(str(param) + " does not have a max/min param")
            return
        send_val = None
        if param_value == max_val:
            send_val = ctrl_on
        elif param_value == min_val:
            send_val = ctrl_off
        if send_val is not None:
            self.feedback_handler(feedback_to, send_val)
        else:
            return
    def feedback_increment(self, feedback_to):
        control =   eval("self." + str(feedback_to["attached_to"]))
        param =         eval(feedback_to["module"])
        mapping_type = str(feedback_to["mapping_type"])
        ctrl_on =   self.feedback_which_ctrl_on_off(feedback_to, "on")
        ctrl_off =  self.feedback_which_ctrl_on_off(feedback_to, "off")
        snapping = feedback_to["snap_to"]
        mapping_type = str(feedback_to["mapping_type"])
        if feedback_to.has_key("maximum") and feedback_to.has_key("minimum"):
            max_val = feedback_to["maximum"]
            min_val = feedback_to["minimum"]
            if mapping_type != "On/Off":
                max_val = self.percent_as_value(feedback_to["module"], feedback_to["maximum"])
                min_val = self.percent_as_value(feedback_to["module"], feedback_to["minimum"])
        elif hasattr(param, "max") and hasattr(param, "min"):
            max_val = param.max
            min_val = param.min
        else:
            self.log_message(str(param) + " does not have a max/min param")
            return
        if snapping == False and param.value < min_val:
            send_val = ctrl_off
        elif param.value < max_val:
            send_val = ctrl_on
        else:
            send_val = ctrl_off
        self.feedback_handler(feedback_to, send_val)
    def feedback_decrement(self, feedback_to):
        control =   eval("self." + str(feedback_to["attached_to"]))
        param =         eval(feedback_to["module"])
        mapping_type = str(feedback_to["mapping_type"])
        ctrl_on =   self.feedback_which_ctrl_on_off(feedback_to, "on")
        ctrl_off =  self.feedback_which_ctrl_on_off(feedback_to, "off")
        snapping = feedback_to["snap_to"]
        if feedback_to.has_key("maximum") and feedback_to.has_key("minimum"):
            max_val = feedback_to["maximum"]
            min_val = feedback_to["minimum"]
            if mapping_type != "On/Off":
                max_val = self.percent_as_value(feedback_to["module"], feedback_to["maximum"])
                min_val = self.percent_as_value(feedback_to["module"], feedback_to["minimum"])
        elif hasattr(param, "max") and hasattr(param, "min"):
            max_val = param.max
            min_val = param.min
        else:
            self.log_message(str(param) + " does not have a max/min param")
            return
        if snapping == False and param.value > max_val:
            send_val = ctrl_off
        elif param.value > min_val:
            send_val = ctrl_on
        else:
            send_val = ctrl_off
        self.feedback_handler(feedback_to, send_val)
    def feedback_which_ctrl_on_off(self, feedback_to, on_off):
        if feedback_to["LED_feedback"] == "default":
            ctrl_on = self.led_on
            ctrl_off = self.led_off
        elif feedback_to["LED_feedback"] == "custom":
            if feedback_to["ctrl_type"] == "on/off" or feedback_to["ctrl_type"] == "increment" or feedback_to["ctrl_type"] == "decrement":
                ctrl_on = feedback_to["LED_on"]
                ctrl_off = feedback_to["LED_off"]
            elif feedback_to["ctrl_type"] == "absolute" or feedback_to["ctrl_type"] == "relative":
                ctrl_on = feedback_to["enc_first"]
                ctrl_off = feedback_to["enc_second"]
        if on_off == "on":
            value = ctrl_on
        elif on_off == "off":
            value = ctrl_off
        return value;
    def feedback_range(self, feedback_to):
        if feedback_to['ctrl_type'] == "on/off":
            self.feedback_on_off(feedback_to)
        elif feedback_to['ctrl_type'] == "increment":
            self.feedback_increment(feedback_to)
        elif feedback_to['ctrl_type'] == "decrement":
            self.feedback_decrement(feedback_to)
        control =   eval("self." + str(feedback_to["attached_to"]))
        param =         eval(feedback_to["module"])
        ctrl_min =  feedback_to["minimum"]
        ctrl_max =  feedback_to["maximum"]
        ctrl_type = feedback_to["ctrl_type"]
        default_ctrl_first = 0
        default_ctrl_last = 127
        if ctrl_type == "relative":
            crl_reverse = False
            ctrl_first = 0
            ctrl_last = 127
        else:
            crl_reverse = feedback_to["reverse_mode"]
            ctrl_first = feedback_to["enc_first"]
            ctrl_last = feedback_to["enc_second"]
        param_range = param.max - param.min
        orig_param_range = param.max - param.min
        param_range = ctrl_max * orig_param_range / 100
        ctrl_min_as_val = ctrl_min * orig_param_range / 100
        param_range = param_range - ctrl_min_as_val
        param_value = param.value - ctrl_min_as_val

        if orig_param_range == 2.0 and param.min == -1.0:
            param_value = param_value + 1
        percentage_control_is_at = param_value / param_range * 100
        ctrl_range = ctrl_last - ctrl_first
        percentage_of_ctrl_range = ctrl_range * percentage_control_is_at / 100 + ctrl_first
        percentage_of_ctrl_range = round(percentage_of_ctrl_range,0)
        if crl_reverse == True:
            percentage_of_ctrl_range = ctrl_range - percentage_of_ctrl_range
        self.feedback_handler(feedback_to, percentage_of_ctrl_range)
    def feedback_a_b_crossfade_assign(self, feedback_to):
        assigned_val = eval(str(feedback_to['parent_track']) + ".mixer_device.crossfade_assign")
        if(assigned_val == 0):
            send_val = feedback_to["LED_on"]
        elif(assigned_val == 1):
            send_val = feedback_to["LED_off"]
        elif(assigned_val == 2):
            send_val = feedback_to["LED_assigned_to_b"]
```

```python
        else:
            send_val = 0
        self.feedback_handler(feedback_to, send_val)
    def feedback_handler(self, config, send_val):
        send_feedback = False
        if config.has_key("LED_feedback"):
            if config["LED_feedback"] == "custom":
                if config["LED_feedback_active"] == "1" or config["LED_feedback_active"] == "true":
                    send_feedback = True
            elif hasattr(self, "global_feedback"):
                if self.global_feedback == "custom":
                    if self.global_feedback_active == True:
                        send_feedback = True
                elif hasattr(self, "controller_LED_on") and hasattr(self, "controller_LED_off"):
                    send_feedback = True
            if send_feedback == True:
                if config["LED_feedback"] == "custom":
                    for item in config["LED_send_feedback_to_selected"]:
                        feedback_control =  eval("self." + str(item))
                        feedback_control.send_value(send_val)
                else:
                    control =   eval("self." + str(config["attached_to"]))
                    control.send_value(send_val)
            else:
                self.log("feedback_handler says 'not sending led feedback'")
    def sess_highlight_banking_calculate(self, feedback_to, num_of_tracks_scenes, offset_is_at):
        ctrl_first = feedback_to["enc_first"]
        ctrl_last = feedback_to["enc_second"]
        ctrl_range = ctrl_last - ctrl_first
        if feedback_to['ctrl_type'] == "absolute" or feedback_to['ctrl_type'] == "relative":
            percentage_control_is_at = offset_is_at / num_of_tracks_scenes * 100
            velocity_val = ctrl_range * percentage_control_is_at / 100 + ctrl_first
            velocity_val = int(velocity_val)
        elif feedback_to['ctrl_type'] == "on/off" or feedback_to['ctrl_type'] == "increment":
            if offset_is_at == num_of_tracks_scenes:
                velocity_val = feedback_to["LED_on"]
            else:
                velocity_val = feedback_to["LED_off"]
        elif feedback_to['ctrl_type'] == "decrement":
            if offset_is_at == 0:
                velocity_val = feedback_to["LED_off"]
            else:
                velocity_val = feedback_to["LED_on"]
        if feedback_to['ctrl_type'] == "absolute" and feedback_to["reverse_mode"] == True:
            velocity_val = ctrl_range - velocity_val
        self.feedback_handler(feedback_to, velocity_val)
    def feedback_scroll_mode_selector(self, feedback_to):
        global active_mode
        num_of_tracks_scenes = len(self.modes) - 1
        count = 0
        for mode_num in self.modes.values():
            if mode_num == active_mode:
                offset_is_at = count
                break
            count += 1
        self.sess_highlight_banking_calculate(feedback_to, num_of_tracks_scenes, offset_is_at)
    def feedback_scroll_mode_selector_select(self, feedback_to):
        global active_mode
        mode_to_select = int(feedback_to["func_arg"])
        if int(active_mode) == mode_to_select:
            self.feedback_handler(feedback_to, feedback_to["LED_on"])
        else:
            self.feedback_handler(feedback_to, feedback_to["LED_off"])
    def feedback_param_banking_select(self, feedback_to):
        if type(feedback_to["banking_number"]) == str:
            banking_number = self.get_modifier_value(feedback_to["banking_number"])
        else:
            banking_number = feedback_to["banking_number"] - 1
        parent_device_id = feedback_to["parent_device_id"]
        offset_is_at = getattr(self, "device_id_" + str(parent_device_id) + "_active_bank")
        if banking_number == offset_is_at:
            self.feedback_handler(feedback_to, feedback_to["LED_on"])
        else:
            self.feedback_handler(feedback_to, feedback_to["LED_off"])
    def feedback_param_banking(self, feedback_to):
        self.log_message("scroll banking fired")
        parent_device_id = feedback_to["parent_device_id"]
        bank_array = getattr(self, "device_id_" + str(parent_device_id) + "_banks")
        num_of_tracks_scenes = len(bank_array) - 1
        offset_is_at = getattr(self, "device_id_" + str(parent_device_id) + "_active_bank")
        self.sess_highlight_banking_calculate(feedback_to, num_of_tracks_scenes, offset_is_at)
    def feedback_highlight_nav_select(self, feedback_to):
        tracks_or_scenes = feedback_to["tracks_scenes"]
        tracks_scene_num = int(feedback_to["highlight_number"])
        if tracks_or_scenes == "tracks":
            offset_is_at = int(self.selected_track_idx()) - 1
        elif tracks_or_scenes == "scenes":
            offset_is_at = int(self.selected_scene_idx()) - 1
        if tracks_scene_num == offset_is_at:
            self.feedback_handler(feedback_to, feedback_to["LED_on"])
        else:
            self.feedback_handler(feedback_to, feedback_to["LED_off"])
    def feedback_highlight_nav(self, feedback_to):
        tracks_or_scenes = feedback_to["tracks_scenes"]
        if tracks_or_scenes == "tracks":
            offset_is_at = int(self.selected_track_idx()) - 1
            num_of_tracks_scenes = int(len(self.song().tracks)) - 1
        elif tracks_or_scenes == "scenes":
            offset_is_at = int(self.selected_scene_idx()) - 1
            num_of_tracks_scenes = int(len(self.song().scenes)) - 1
        self.sess_highlight_banking_calculate(feedback_to, num_of_tracks_scenes, offset_is_at)
    def feedback_sessbox_nav_select(self, feedback_to):
        try:
            self._session
        except:
            self.show_message("There's no Session Box to select for feedback")
            return
        tracks_scene_num = int(feedback_to["highlight_number"])
        tracks_or_scenes = feedback_to["tracks_scenes"]
        if tracks_or_scenes == "tracks":
            offset_is_at = int(self._session.track_offset())
        elif tracks_or_scenes == "scenes":
            offset_is_at = int(self._session.scene_offset())
        if tracks_scene_num == offset_is_at:
            self.feedback_handler(feedback_to, feedback_to["LED_on"])
        else:
            self.feedback_handler(feedback_to, feedback_to["LED_off"])
    def feedback_sessbox_nav(self, feedback_to):
        try:
            self._session
        except:
            self.show_message("There's no Session Box to scroll for feedback sir.")
            return
        tracks_or_scenes = feedback_to["tracks_scenes"]
        if tracks_or_scenes == "tracks":
            offset_is_at = int(self._session.track_offset())
            num_of_tracks_scenes = int(len(self.song().tracks)) - 1
        elif tracks_or_scenes == "scenes":
            offset_is_at = int(self._session.scene_offset())
            num_of_tracks_scenes = int(len(self.song().scenes)) - 1
        self.sess_highlight_banking_calculate(feedback_to, num_of_tracks_scenes, offset_is_at)
    def feedback_tempo(self, feedback_to):
        control =   eval("self." + str(feedback_to["attached_to"]))
        param =         eval(feedback_to["module"])
        ctrl_min =  feedback_to["minimum"]
        ctrl_max =  feedback_to["maximum"]
        ctrl_type = feedback_to["ctrl_type"]
        ctrl_first = feedback_to["enc_first"]
        ctrl_last = feedback_to["enc_second"]
        default_ctrl_first = 0
        default_ctrl_last = 127
        crl_reverse = feedback_to["reverse_mode"]
        param_range = ctrl_max - ctrl_min
        param =         eval(feedback_to["module"] + "." + feedback_to["ui_listener"])
        zero = ctrl_min
        if param < ctrl_min or param > ctrl_max:
            self.log("tempo is outside ctrl_min / ctrl_max")
        else:
            zerod_param = param - zero
            percentage_control_is_at = zerod_param / param_range * 100
        ctrl_range = ctrl_last - ctrl_first
        percentage_of_ctrl_range = ctrl_range * percentage_control_is_at / 100 + ctrl_first
        if crl_reverse == True:
            percentage_of_ctrl_range = ctrl_range - percentage_of_ctrl_range
        self.feedback_handler(feedback_to, percentage_of_ctrl_range)
    def mode_device_bank_leds(self, mode_id):
        config_map = "mode_" + str(mode_id) + "_configs_map"
        config_map = getattr(self, config_map)
        for config_name in config_map:
            config = getattr(self, config_name)
            if config["mapping_type"] == "Parameter Bank":
                parent_id = config["parent_json_id"]
                bank_names_array_name = "device_id_" + str(parent_id) + "_banks"
                active_bank_name = "device_id_" + str(parent_id) + "_active_bank"
                bank_names_array = getattr(self, bank_names_array_name)
                active_bank = getattr(self, active_bank_name)
                for index, bank_name in enumerate(bank_names_array):
                    if bank_name == config_name:
```

```python
                        if index == active_bank:
                            led_on = config["LED_on"]
                            self.feedback_handler(config, led_on)
                        else:
                            led_off = config["LED_off"]
                            self.feedback_handler(config, led_off)
    def bank_led_feedback(self, parent_device_id):
        global active_mode
        device = "device_id_" + str(parent_device_id);
        device_bank_array = getattr(self, device + "_banks")
        active_bank_idx = getattr(self, device + "_active_bank")
        device_bank_params = getattr(self, device + "_bank_parameters_" + str(active_bank_idx))
        for index, val in enumerate(device_bank_array):
            bank_cnfg = getattr(self, val)
            bank_cnfg["LED_feedback"] = "custom";
            if index == active_bank_idx:
                    if bank_cnfg.has_key("LED_on"):
                        led_on = bank_cnfg["LED_on"]
                        self.feedback_handler(bank_cnfg, led_on)
            else:
                if bank_cnfg.has_key("LED_off"):
                    led_off = bank_cnfg["LED_off"]
                    self.feedback_handler(bank_cnfg, led_off)

        remove_mode = getattr(self, "_remove_mode" + active_mode + "_ui_listeners")
        remove_mode()
        activate_mode = getattr(self, "_mode" + active_mode + "_ui_listeners")
        activate_mode()
        for param in device_bank_params:
            fire_param_feedback = getattr(self, param + "_led_listener")
            fire_param_feedback()
    def device_feedback(self, mode_id=None):
        if (mode_id == None):
            global active_mode
            mode_id = active_mode
        config_map = "mode_" + str(mode_id) + "_configs_map"
        config_map = getattr(self, config_map)
        for config_name in config_map:
            config = getattr(self, config_name)
            if config.has_key("mapping_type") and config["mapping_type"] == "Device":
                led_on = config["LED_on"]
                led_off = config["LED_off"]
                try:
                    device = eval(config["module"])
                except:
                    self.feedback_handler(config, led_off)
                    return
                find = config["module"].find("selected_track")
                if find >= 0:
                    selected_device = self.song().view.selected_track.view.selected_device
                    if device == selected_device:
                        self.feedback_handler(config, led_on)
                    else:
                        self.feedback_handler(config, led_off)
                else:
                    for parent_name in config_map:
                        parent_config = getattr(self, parent_name)
                        if parent_config["json_id"] == config["parent_json_id"]:
                            parent_track = parent_config["module"]
                            break
                    tracks_selected_device = eval(parent_track + ".view.selected_device")
                    if device == tracks_selected_device:
                        self.feedback_handler(config, led_on)
                    else:
                        self.feedback_handler(config, led_off)
    def _on_selected_track_changed(self):
        global active_mode, prev_active_mode, modes
        self.log("selected track changed")
        remove_modex_led_listeners = "_remove_mode" + active_mode + "_led_listeners"
        add_modex_led_listeners = "_mode" + active_mode + "_led_listeners"
        if(hasattr(self, remove_modex_led_listeners)):
            mode_to_call = getattr(self, remove_modex_led_listeners)
            mode_to_call()
        if(hasattr(self, add_modex_led_listeners)):
            mode_to_call = getattr(self, add_modex_led_listeners)
            mode_to_call()
            self.track_feedback()
            self.device_feedback()
        self.refresh_state()
    def track_feedback(self, mode_id=None):
        if (mode_id == None):
            global active_mode
            mode_id = active_mode
        config_map = "mode_" + str(mode_id) + "_configs_map"
        config_map = getattr(self, config_map)
        selected_track = self.song().view.selected_track
        for config_name in config_map:
            config = getattr(self, config_name)
            if config.has_key("mapping_type") and config["mapping_type"] == "Track":
                led_on = config["LED_on"]
                led_off = config["LED_off"]
                try:
                    track = eval(config["module"])
                except:
                    self.feedback_handler(config, led_off)
                    return
                if track == selected_track:
                    self.feedback_handler(config, led_on)
                else:
                    self.feedback_handler(config, led_off)
    def create_clip_slot_map(self):
        num_of_tracks = int(len(self.song().tracks))
        num_of_scenes = int(len(self.song().scenes))
        for track in xrange(0,num_of_tracks):
            for scene in xrange(0,num_of_scenes):
                if(not self.song().tracks[track].clip_slots[scene].has_clip_has_listener(self._on_clip_added_removed)):
                    try:
                        self.song().tracks[track].clip_slots[scene].add_has_clip_listener(self._on_clip_added_removed)
                    except:
                        pass
    def _on_clip_added_removed(self):
        global active_mode
        self.log("a clip has been added or removed")
        updated_by = "_on_clip_added_removed"
        self._remove_custom_lom_listeners_handler(active_mode, updated_by)
        self._add_custom_lom_listeners_handler(active_mode, updated_by)
    def _on_tracks_changed(self):
        global active_mode
        self.log("tracks changed")
        updated_by = "_on_tracks_changed"
        self._remove_custom_lom_listeners_handler(active_mode, updated_by)
        self._add_custom_lom_listeners_handler(active_mode, updated_by)
        self.all_track_device_listeners()
        self.create_clip_slot_map()
    def _on_scenes_changed(self):
        global active_mode
        self.log("scenes changed")
        updated_by = "_on_scenes_changed"
        self._remove_custom_lom_listeners_handler(active_mode, updated_by)
        self._add_custom_lom_listeners_handler(active_mode, updated_by)
        self.create_clip_slot_map()
```

```python
    def _on_devices_changed(self):
        global active_mode, prev_active_mode, modes
        self.log("devices changed")
        updated_by = "_on_devices_changed"
        self._remove_custom_lom_listeners_handler(active_mode, updated_by)
        self._add_custom_lom_listeners_handler(active_mode, updated_by)
        try:
            mode_to_call = getattr(self, "_remove_mode" + active_mode + "_led_listeners")
            mode_to_call()
            mode_to_call = getattr(self, "_mode" + active_mode + "_led_listeners")
            mode_to_call()
        except:
            pass
    def _on_selected_device_changed(self):
        global active_mode, prev_active_mode, modes
        self.log("selected device changed")
        try:
            mode_to_call = getattr(self, "_remove_mode" + active_mode + "_led_listeners")
            mode_to_call()
            mode_to_call = getattr(self, "_mode" + active_mode + "_led_listeners")
            mode_to_call()
            self.device_feedback()
            self.refresh_state()
        except:
            pass
    def _on_selected_parameter_changed(self):
        global active_mode
        self.log("selected parameter changed")
        if(hasattr(self.song().view.selected_parameter, "canonical_parent") and hasattr(self.song().view.selected_parameter.canonical_parent, "type")):
            updated_by = "_on_selected_parameter_changed"
            self._remove_custom_lom_listeners_handler(active_mode, updated_by)
            self._add_custom_lom_listeners_handler(active_mode, updated_by)
    def _on_selected_scene_changed(self):
        global active_mode, prev_active_mode, modes
        self.log("selected scene changed")
        remove_modex_led_listeners = "_remove_mode" + active_mode + "_led_listeners"
        add_modex_led_listeners = "_mode" + active_mode + "_led_listeners"
        if(hasattr(self, remove_modex_led_listeners)):
            mode_to_call = getattr(self, remove_modex_led_listeners)
            mode_to_call()
        if(hasattr(self, add_modex_led_listeners)):
            mode_to_call = getattr(self, add_modex_led_listeners)
            mode_to_call()
        self.refresh_state()
    def _all_tracks_listener(self):
        global active_mode, prev_active_mode, modes
        self.log("mode 1 tracks listener")
        mode_to_call = getattr(self, "_remove_mode" + active_mode + "_led_listeners")
        mode_to_call()
        mode_to_call = getattr(self, "_mode" + active_mode + "_led_listeners")
        mode_to_call()
    def all_track_device_listeners(self):
        numtracks = len(self.song().tracks)
        for index in range(numtracks):
            try:
                self.song().tracks[index].view.add_selected_device_listener(self._on_selected_device_changed)
                self.song().tracks[index].add_devices_listener(self._on_devices_changed)
            except:
                pass
        num_returns = len(self.song().return_tracks)
        for index in range(num_returns):
            try:
                self.song().return_tracks[index].view.add_selected_device_listener(self._on_selected_device_changed)
                self.song().return_tracks[index].add_devices_listener(self._on_devices_changed)
            except:
                pass
        try:
            self.song().master_track.view.add_selected_device_listener(self._on_selected_device_changed)
            self.song().master_track.add_devices_listener(self._on_devices_changed)
        except:
            pass
    def _remove_all_track_device_listeners(self):
        numtracks = len(self.song().tracks)
        for index in range(numtracks):
            try:
```

```
self.song().tracks[index].view.remove_selected_device_listener(self._on_selected_device_changed)
self.song().tracks[index].remove_devices_listener(self._on_devices_changed)
            except:
                pass
        num_returns = len(self.song().return_tracks)
        for index in range(num_returns):
            try:
self.song().return_tracks[index].view.remove_selected_device_listener(self._on_selected_device_changed)
self.song().return_tracks[index].remove_devices_listener(self._on_devices_changed)
            except:
                pass
        try:
self.song().master_track.view.remove_selected_device_listener(self._on_selected_device_changed)
self.song().master_track.remove_devices_listener(self._on_devices_changed)
        except:
            pass
    ################################################
    ############# Extra Functions ##################
    ################################################
    def scroll_through_devices(self, cnfg):
        NavDirection = Live.Application.Application.View.NavDirection
        if cnfg["ctrl_type"] == "absolute":
            if cnfg["value"] > cnfg["pre_val"]:
                if cnfg["reverse_mode"] is False:
                    goto = "right"
                elif cnfg["reverse_mode"] is True:
                    goto = "left"
                times = 1;
            elif cnfg["value"] < cnfg["pre_val"]:
                if cnfg["reverse_mode"] is False:
                    goto = "left"
                elif cnfg["reverse_mode"] is True:
                    goto = "right"
                times = 1;
        elif cnfg["ctrl_type"] == "relative":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = "left"
        times = cnfg["steps"];
      elif cnfg["enc_second"] == cnfg["value"]:
        goto = "right"
        times = cnfg["steps"];
    elif cnfg["ctrl_type"] == "on/off":
      if cnfg["enc_first"] == cnfg["value"]:
          goto = "right"
      elif cnfg["enc_second"] == cnfg["value"]:
          goto = "right"
    elif cnfg["ctrl_type"] == "increment":
      if cnfg["enc_first"] == cnfg["value"]:
        goto = "right"
        times = cnfg["steps"];
    elif cnfg["ctrl_type"] == "decrement":
      if cnfg["enc_first"] == cnfg["value"]:
        goto = "left"
        times = cnfg["steps"];
    if goto == "right":
      for x in range(0, times):
        self._scroll_device_chain(NavDirection.right)
    elif goto == "left":
      for x in range(0, times):
        self._scroll_device_chain(NavDirection.left)
  def _scroll_device_chain(self, direction):
    view = self.application().view
    if not view.is_view_visible('Detail') or not view.is_view_visible('Detail/DeviceChain'):
      view.show_view('Detail')
      view.show_view('Detail/DeviceChain')
    else:
      view.scroll_view(direction, 'Detail/DeviceChain', False)
  def selected_device_idx(self):
    self._device = self.song().view.selected_track.view.selected_device
    return self.tuple_index(self.song().view.selected_track.devices, self._device)
  def selected_track_idx(self):
    self._track = self.song().view.selected_track
    self._track_num = self.tuple_index(self.song().tracks, self._track)
    self._track_num = self._track_num + 1
    return self._track_num
  def selected_scene_idx(self):
    self._scene = self.song().view.selected_scene
    self._scene_num = self.tuple_index(self.song().scenes, self._scene)
    self._scene_num = self._scene_num + 1
    return self._scene_num
```

```python
    def tuple_index(self, tuple, obj):
        for i in xrange(0, len(tuple)):
            if (tuple[i] == obj):
                return i
        return(False)
    def select_a_device(self, cnfg):
        parent_track = cnfg["parent_track"]
        device_chain = cnfg["device_chain"]
        chain_selector = "self.song().view.selected_track" + device_chain
        try:
            self.song().view.selected_track = eval(parent_track)
            try:
                self.song().view.select_device(eval(chain_selector))
            except IndexError:
                self.show_message("Device you are trying to select does not exist on track.")
        except IndexError:
            self.show_message("Track does not exist for the device you are selecting.")
    def a_b_crossfade_assign(self, cnfg):
        assignment_type = cnfg['assignment_type'];
        if(assignment_type == "Scroll"):
            goto = self.scroll_a_b_assign(cnfg);
            if goto > 2:
                goto = 2
        elif cnfg["enc_first"] == cnfg["value"]:
            if assignment_type == "Select A":
                goto = 0
            elif assignment_type == "Select None":
                goto = 1
            elif assignment_type == "Select B":
                goto = 2
            else:
                goto = 0
        setattr(eval(str(cnfg['parent_track']) + ".mixer_device"), "crossfade_assign", goto)
    def scroll_a_b_assign(self, cnfg):
        should_it_fire = self.should_it_fire(cnfg)
        if(should_it_fire != 1):
            return
        current_assigned_value = eval(str(cnfg['parent_track']) + ".mixer_device.crossfade_assign")
        length = 3
        if cnfg["ctrl_type"] == "absolute":
            divider = (cnfg["enc_second"] - cnfg["enc_first"]) / length
            goto = int(cnfg["value"] / divider)
            if cnfg["reverse_mode"] is True:
                if(goto >= 2):
                    goto = 0
                elif(goto == 0):
                    goto = 2
            goto = int(goto)
        elif cnfg["ctrl_type"] == "relative":
            self.log_message("csslog: relative");
            if cnfg["enc_first"] == cnfg["value"] and current_assigned_value > 0:
                goto = current_assigned_value - 1
            elif cnfg["enc_second"] == cnfg["value"] and current_assigned_value < 2:
                goto = current_assigned_value + 1
        elif cnfg["ctrl_type"] == "on/off":
            if current_assigned_value < 2:
                goto = current_assigned_value + 1
            elif current_assigned_value >= 2:
                goto = 0
        elif cnfg["ctrl_type"] == "increment":
            if current_assigned_value < 2:
                goto = current_assigned_value + 1
            else:
                goto = current_assigned_value
        elif cnfg["ctrl_type"] == "decrement":
            if current_assigned_value > 0:
                goto = current_assigned_value - 1
            else:
                goto = current_assigned_value
        return int(goto)
    def scroll_highlight(self, cnfg):
        if cnfg["tracks_scenes"] == "tracks":
            length = len(self.song().tracks) + len(self.song().return_tracks)
            selected = self.selected_track_idx() - 1
        elif cnfg["tracks_scenes"] == "scenes":
            length = len(self.song().scenes)
            selected = self.selected_scene_idx() - 1
        else:
            self.log("scroll_highlight error, tracks_scenes was not set")
        if cnfg["ctrl_type"] == "absolute":
            divider = (cnfg["enc_second"] - cnfg["enc_first"]) / length
            if cnfg["reverse_mode"] is False:
                goto = cnfg["value"] / divider
            elif cnfg["reverse_mode"] is True:
                goto = (divider * length) / cnfg["value"]
            goto = int(goto)
        elif cnfg["ctrl_type"] == "relative":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = selected - cnfg["steps"]
            elif cnfg["enc_second"] == cnfg["value"]:
                goto = selected + cnfg["steps"]
        elif cnfg["ctrl_type"] == "on/off":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = length
            elif cnfg["enc_second"] == cnfg["value"]:
                goto = 0
        elif cnfg["ctrl_type"] == "increment":
            goto = selected + cnfg["steps"]
        elif cnfg["ctrl_type"] == "decrement":
            goto = selected - cnfg["steps"]
        if goto <= length and goto >= 0 and goto != selected:
            cnfg["highlight_number"] = goto
            self.select_highlight(cnfg)
    def select_sess_offset(self, cnfg):
        try:
            self._session
        except:
            self.show_message("There's no Session Box to select, buddy.")
            return
        tracks_scenes = cnfg["tracks_scenes"]
        track_offset = self._session.track_offset()
        scene_offset = self._session.scene_offset()
        if type(cnfg["highlight_number"]) == str:
            change_to = self.get_modifier_value(cnfg["highlight_number"])
        else:
            change_to = cnfg["highlight_number"]
        if tracks_scenes == "tracks":
            track_offset = change_to
        elif tracks_scenes == "scenes":
            scene_offset = change_to
        try:
            self._session.set_offsets(track_offset, scene_offset)
            self._session._reassign_scenes()
            self.set_highlighting_session_component(self._session)
            self.refresh_state()
            self.call_script_reaction(active_mode, None, 
'session_box_position')
        except:
            self.show_message("unable to move session box there.")
    def scroll_sess_offset(self, cnfg):
        try:
            self._session
        except:
            self.show_message("There's no Session Box to scroll, buddy.")
            return
        tracks_scenes = cnfg["tracks_scenes"]
        track_offset = self._session.track_offset()
        scene_offset = self._session.scene_offset()
        if cnfg["tracks_scenes"] == "tracks":
            length = len(self.song().tracks)
            selected = track_offset
        elif cnfg["tracks_scenes"] == "scenes":
            length = len(self.song().scenes)
            selected = scene_offset
        else:
            self.log("scroll_sess_offset error, tracks_scenes was not set")
        if cnfg["ctrl_type"] == "absolute":
            divider = (cnfg["enc_second"] - cnfg["enc_first"]) / length
            goto = cnfg["value"] / divider
            if cnfg["reverse_mode"] is True:
                goto = length - goto
            goto = int(goto)
        elif cnfg["ctrl_type"] == "relative":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = selected - cnfg["steps"]
            elif cnfg["enc_second"] == cnfg["value"]:
                goto = selected + cnfg["steps"]
        elif cnfg["ctrl_type"] == "on/off":
            if cnfg["enc_first"] == cnfg["value"] or cnfg["enc_second"] == cnfg["value"]:
                if selected != 0 and selected != length - 1:
                    goto = length - 1
                elif selected == 0:
                    goto = length - 1
                else:
                    goto = 0
        elif cnfg["ctrl_type"] == "increment":
            goto = selected + cnfg["steps"]
        elif cnfg["ctrl_type"] == "decrement":
            goto = selected - cnfg["steps"]
        if(goto < 0):
            goto = 0
        if cnfg["tracks_scenes"] == "tracks":
            track_offset = goto
        elif cnfg["tracks_scenes"] == "scenes":
            scene_offset = goto
        try:
            self._session.set_offsets(track_offset, scene_offset)
            self._session._reassign_scenes()
            self.set_highlighting_session_component(self._session)
            self.refresh_state()
            self.call_script_reaction(active_mode, None, 'session_box_position')
        except:
```

```python
            self.show_message("unable to move session box there.")
    def get_tracks_array(self):
        tracks_array = []
        count = 0
        for index in range(len(self.song().tracks)):
            tracks_array.append(self.song().tracks[count])
            count = count+1
        count = 0
        for index in range(len(self.song().return_tracks)):
            tracks_array.append(self.song().return_tracks[count])
            count = count+1
        tracks_array.append(self.song().master_track)
        return tracks_array
    def select_highlight(self, cnfg):
        tracks_scenes = cnfg["tracks_scenes"]
        if type(cnfg["highlight_number"]) == str:
            change_to = self.get_modifier_value(cnfg["highlight_number"])
        else:
            change_to = cnfg["highlight_number"]
        if tracks_scenes == "tracks":
            num_of_tracks_scenes = len(self.song().tracks) + len(self.song().return_tracks) + 1
        elif tracks_scenes == "scenes":
            num_of_tracks_scenes = len(self.song().scenes)
        if num_of_tracks_scenes >= change_to + 1:
            if tracks_scenes == "tracks":
                all_tracks_arr = self.get_tracks_array()
                self.song().view.selected_track = all_tracks_arr[change_to]
            elif tracks_scenes == "scenes":
                self.song().view.selected_scene = self.song().scenes[change_to]
        else:
            self.show_message("Your Session doesn't have " + str(change_to + 1) + " " + tracks_scenes)
    def scroll_active_device_bank(self, cnfg):
        device_id = cnfg["parent_device_id"]
        device = "device_id_" + str(device_id);
        active_bank = getattr(self, device + "_active_bank")
        banks = getattr(self, device + "_banks")
        length = len(banks) - 1
        if cnfg["ctrl_type"] == "absolute":
            divider = (cnfg["enc_second"] - cnfg["enc_first"]) / length
            if cnfg["reverse_mode"] is False:
                goto = cnfg["value"] / divider
            elif cnfg["reverse_mode"] is True:
                goto = (divider * length) / cnfg["value"]
            goto = int(goto)
        elif cnfg["ctrl_type"] == "relative":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = active_bank - 1
            elif cnfg["enc_second"] == cnfg["value"]:
                goto = active_bank + 1
        elif cnfg["ctrl_type"] == "on/off":
            if cnfg["switch_type"] == "toggle":
                if cnfg["enc_first"] == cnfg["value"]:
                    goto = length
                elif cnfg["enc_second"] == cnfg["value"]:
                    goto = 0
            elif active_bank == length:
                goto = 0
            else:
                goto = length
        elif cnfg["ctrl_type"] == "increment":
                goto = active_bank + 1
        elif cnfg["ctrl_type"] == "decrement":
                goto = active_bank - 1
        if goto <= length and goto >= 0 and goto != active_bank:
            cnfg["banking_number"] = goto + 1
            self.change_active_device_bank(cnfg)
    def change_active_device_bank(self, cnfg):
        global active_mode
        device_id = cnfg["parent_device_id"]
        if type(cnfg["banking_number"]) == str:
            change_to_bank = self.get_modifier_value(cnfg["banking_number"])
        else:
            change_to_bank = cnfg["banking_number"] - 1

        device = "device_id_" + str(device_id);
        bank_names = getattr(self, device + "_bank_names")
        length = len(bank_names) - 1;
        if change_to_bank <= length:
            setattr(self, device + "_active_bank", change_to_bank)
            self.bank_led_feedback(cnfg["parent_json_id"]);
            self.show_message("changed active bank to: " + bank_names[change_to_bank])
        elif change_to_bank > length:
            self.show_message("device does not have " + str(change_to_bank + 1) + " parameter banks set")
        fire_all_mode_feedback = getattr(self, "_mode" + active_mode + "_fire_all_feedback")
        fire_all_mode_feedback()
    def session_box(self, num_tracks, num_scenes, track_offset, scene_offset, clips, stop_all, stop_tracks, scene_launch, feedbackArr, combination_mode):
        self._session = SessionComponent(num_tracks, num_scenes)
        self._session.set_offsets(track_offset, scene_offset)
        self._session.add_offset_listener(self._on_session_offset_changes, identify_sender= False)
        self._session._reassign_scenes()
        self.set_highlighting_session_component(self._session)
        if clips:
            self._grid = ButtonMatrixElement(rows=[clips[(index*num_tracks):(index*num_tracks)+num_tracks] for index in range(num_scenes)])
            self._session.set_clip_launch_buttons(self._grid)
        if stop_all:
            self._session.set_stop_all_clips_button(stop_all)
        if stop_tracks:
            self._session.set_stop_track_clip_buttons(tuple(stop_tracks))
        if scene_launch:
            scene_launch_buttons = ButtonMatrixElement(rows=[scene_launch])
            self._session.set_scene_launch_buttons(scene_launch_buttons)
            self._session.set_stop_clip_triggered_value(feedbackArr["StopClipTriggered"])
            self._session.set_stop_clip_value(feedbackArr["StopClip"])
        for scene_index in range(num_scenes):
            scene = self._session.scene(scene_index)
            scene.set_scene_value(feedbackArr["Scene"])
            scene.set_no_scene_value(feedbackArr["NoScene"])
            scene.set_triggered_value(feedbackArr["SceneTriggered"])
            for track_index in range(num_tracks):
                clip_slot = scene.clip_slot(track_index)
                clip_slot.set_triggered_to_play_value(feedbackArr["ClipTriggeredPlay"])
                clip_slot.set_triggered_to_record_value(feedbackArr["ClipTriggeredRecord"])
                clip_slot.set_record_button_value(feedbackArr["RecordButton"])
                clip_slot.set_stopped_value(feedbackArr["ClipStopped"])
                clip_slot.set_started_value(feedbackArr["ClipStarted"])
                clip_slot.set_recording_value(feedbackArr["ClipRecording"])
            for index in range(len(stop_tracks)):
                stop_track_button = stop_tracks[index]
                if feedbackArr["StopTrackPlaying"] and feedbackArr["StopTrackStopped"]:
                    stop_track_button.set_on_off_values(feedbackArr["StopTrackPlaying"], feedbackArr["StopTrackStopped"])
            if stop_all:
                if feedbackArr["StopAllOn"] and feedbackArr["StopAllOff"]:
                    stop_all.set_on_off_values(feedbackArr["StopAllOn"], feedbackArr["StopAllOff"])
        if combination_mode == "on":
            self._session._link()
        self.refresh_state()
    def _on_session_offset_changes(self):
        global active_mode
        updated_by = "_on_session_offset_changes"
        self._remove_custom_lom_listeners_handler(active_mode, updated_by)
        self._add_custom_lom_listeners_handler(active_mode, updated_by)
        self.log("sessionbox offset changed")
        try:
            remove_mode = getattr(self, "_remove_mode" + active_mode + "_led_listeners")
            remove_mode()
            activate_mode = getattr(self, "_mode" + active_mode + "_led_listeners")
            activate_mode()
        except:
            self.log("_on_session_offset_changes: could not remove / add led_listeners")
            return;
    def remove_session_box(self, combination_mode):
        if hasattr(self, "_session"):
            self.current_track_offset = self._session._track_offset
            self.current_scene_offset = self._session._scene_offset
            self._session.set_clip_launch_buttons(None)
            self.set_highlighting_session_component(None)
            self._session.set_stop_all_clips_button(None)
            self._session.set_stop_track_clip_buttons(None)
            self._session.set_scene_launch_buttons(None)
            if combination_mode == "on":
                self._session._unlink()
            self._session = None
    def scroll_modes(self, cnfg):
        controller = getattr(self, cnfg["attached_to"])
        cnfg["value"] = controller.cur_val
        if cnfg["ctrl_type"] == "absolute":
            divider = (cnfg["enc_second"] - cnfg["enc_first"]) / (len(self.modes) - 1)
            if cnfg["reverse_mode"] is False:
                goto = cnfg["value"] / divider
            elif cnfg["reverse_mode"] is True:
                length = len(self.modes) - 1
                goto = (divider * length) / cnfg["value"]
            goto = int(goto)
```

```python
        elif cnfg["ctrl_type"] == "relative":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = self.key_num - 1
            elif cnfg["enc_second"] == cnfg["value"]:
                goto = self.key_num + 1
        elif cnfg["ctrl_type"] == "on/off":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = len(self.modes) - 1
            elif cnfg["enc_second"] == cnfg["value"]:
                goto = 0
        elif cnfg["ctrl_type"] == "increment":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = self.key_num + 1
        elif cnfg["ctrl_type"] == "decrement":
            if cnfg["enc_first"] == cnfg["value"]:
                goto = self.key_num - 1
        if goto <= len(self.modes) and goto >= 0 and active_mode != self.modes[goto]:
            self.set_active_mode(self.modes[goto])
    def listening_to_tracks(self):
        global active_mode
        self.remove_listening_to_tracks()
        for index in range(len(self.song().tracks)):
            _track = self.song().tracks[index]
            if _track.can_be_armed and hasattr(self, "_mode" + active_mode + "_arm_listener"):
                _track.add_arm_listener(getattr(self, "_mode" + active_mode + "_arm_listener"))
            if hasattr(self, "_mode" + active_mode + "_mute_listener"):
                _track.add_mute_listener(getattr(self, "_mode" + active_mode + "_mute_listener"))
            if hasattr(self, "_mode" + active_mode + "_solo_listener"):
                _track.add_solo_listener(getattr(self, "_mode" + active_mode + "_solo_listener"))
            if hasattr(self, "_mode" + active_mode + "_volume_listener"):
                _track.mixer_device.volume.add_value_listener(getattr(self, "_mode" + active_mode + "_volume_listener"))
            if hasattr(self, "_mode" + active_mode + "_panning_listener"):
                _track.mixer_device.panning.add_value_listener(getattr(self, "_mode" + active_mode + "_panning_listener"))
            if hasattr(self, "_mode" + active_mode + "_send_listener"):
                for send_index in range(len(_track.mixer_device.sends)):
                    _track.mixer_device.sends[send_index].add_value_listener(getattr(self, "_mode" + active_mode + "_send_listener"))
        for index in range(len(self.song().return_tracks)):
            _return_track = self.song().return_tracks[index]
            if hasattr(self, "_mode" + active_mode + "_mute_listener"):
                _return_track.add_mute_listener(getattr(self, "_mode" + active_mode + "_mute_listener"))
            if hasattr(self, "_mode" + active_mode + "_solo_listener"):
                _return_track.add_solo_listener(getattr(self, "_mode" + active_mode + "_solo_listener"))
            if hasattr(self, "_mode" + active_mode + "_volume_listener"):
                _return_track.mixer_device.volume.add_value_listener(getattr(self, "_mode" + active_mode + "_volume_listener"))
            if hasattr(self, "_mode" + active_mode + "_panning_listener"):
                _return_track.mixer_device.panning.add_value_listener(getattr(self, "_mode" + active_mode + "_panning_listener"))
            if hasattr(self, "_mode" + active_mode + "_send_listener"):
                for send_index in range(len(_return_track.mixer_device.sends)):
                    _return_track.mixer_device.sends[send_index].add_value_listener(getattr(self, "_mode" + active_mode + "_send_listener"))
        _master = self.song().master_track
        if hasattr(self, "_mode" + active_mode + "_volume_listener"):
            _master.mixer_device.volume.add_value_listener(getattr(self, "_mode" + active_mode + "_volume_listener"))
        if hasattr(self, "_mode" + active_mode + "_panning_listener"):
            _master.mixer_device.panning.add_value_listener(getattr(self, "_mode" + active_mode + "_panning_listener"))
    def remove_listening_to_tracks(self):
        global active_mode
        for index in range(len(self.song().tracks)):
            _track = self.song().tracks[index]
            if hasattr(self, "_mode" + active_mode + "_arm_listener"):
                if _track.arm_has_listener(getattr(self, "_mode" + active_mode + "_arm_listener")):
                    _track.remove_arm_listener(getattr(self, "_mode" + active_mode + "_arm_listener"))
            if hasattr(self, "_mode" + active_mode + "_mute_listener"):
                if _track.mute_has_listener(getattr(self, "_mode" + active_mode + "_mute_listener")):
                    _track.remove_mute_listener(getattr(self, "_mode" + active_mode + "_mute_listener"))
            if hasattr(self, "_mode" + active_mode + "_solo_listener"):
                if _track.solo_has_listener(getattr(self, "_mode" + active_mode + "_solo_listener")):
                    _track.remove_solo_listener(getattr(self, "_mode" + active_mode + "_solo_listener"))
            if hasattr(self, "_mode" + active_mode + "_volume_listener"):
                if _track.mixer_device.volume.value_has_listener(getattr(self, "_mode" + active_mode + "_volume_listener")):
                    _track.mixer_device.volume.remove_value_listener(getattr(self, "_mode" + active_mode + "_volume_listener"))
            if hasattr(self, "_mode" + active_mode + "_panning_listener"):
                if _track.mixer_device.panning.value_has_listener(getattr(self, "_mode" + active_mode + "_panning_listener")):
                    _track.mixer_device.panning.remove_value_listener(getattr(self, "_mode" + active_mode + "_panning_listener"))
            if hasattr(self, "_mode" + active_mode + "_send_listener"):
                for send_index in range(len(_track.mixer_device.sends)):
                    if _track.mixer_device.sends[send_index].value_has_listener(getattr(self, "_mode" + active_mode + "_send_listener")):
                        _track.mixer_device.sends[send_index].remove_value_listener(getattr(self, "_mode" + active_mode + "_send_listener"))
        for index in range(len(self.song().return_tracks)):
            _return_track = self.song().return_tracks[index]
            if hasattr(self, "_mode" + active_mode + "_mute_listener"):
                if _return_track.mute_has_listener(getattr(self, "_mode" + active_mode + "_mute_listener")):
                    _return_track.remove_mute_listener(getattr(self, "_mode" + active_mode + "_mute_listener"))
            if hasattr(self, "_mode" + active_mode + "_solo_listener"):
                if _return_track.solo_has_listener(getattr(self, "_mode" + active_mode + "_solo_listener")):
                    _return_track.remove_solo_listener(getattr(self, "_mode" + active_mode + "_solo_listener"))
            if hasattr(self, "_mode" + active_mode + "_volume_listener"):
                if _return_track.mixer_device.volume.value_has_listener(getattr(self, "_mode" + active_mode + "_volume_listener")):
                    _return_track.mixer_device.volume.remove_value_listener(getattr(self, "_mode" + active_mode + "_volume_listener"))
            if hasattr(self, "_mode" + active_mode + "_panning_listener"):
                if _return_track.mixer_device.panning.value_has_listener(getattr(self, "_mode" + active_mode + "_panning_listener")):
                    _return_track.mixer_device.panning.remove_value_listener(getattr(self, "_mode" + active_mode + "_panning_listener"))
            if hasattr(self, "_mode" + active_mode + "_send_listener"):
```

```python
                for send_index in range(len(_return_track.mixer_device.sends)):
                    if _return_track.mixer_device.sends[send_index].value_has_listener(getattr(self, "_mode" + active_mode + "_send_listener")):
                        _return_track.mixer_device.sends[send_index].remove_value_listener(getattr(self, "_mode" + active_mode + "_send_listener"))
        _master = self.song().master_track
        if hasattr(self, "_mode" + active_mode + "_volume_listener"):
            if _master.mixer_device.volume.value_has_listener(getattr(self, "_mode" + active_mode + "_volume_listener")):
                _master.mixer_device.volume.remove_value_listener(getattr(self, "_mode" + active_mode + "_volume_listener"))
        if hasattr(self, "_mode" + active_mode + "_panning_listener"):
            if _master.mixer_device.panning.value_has_listener(getattr(self, "_mode" + active_mode + "_panning_listener")):
                _master.mixer_device.panning.remove_value_listener(getattr(self, "_mode" + active_mode + "_panning_listener"))
    def set_active_mode(self, activate_new_mode):
        global active_mode, prev_active_mode, modes

        for number, mode_id in self.modes.items():
            if mode_id == activate_new_mode:
                self.key_num = mode_id
        if(activate_new_mode == "Previous Mode"):
            if 'prev_active_mode' not in globals():
                self.show_message("No previous mode is set yet.")
            else:
                remove_mode = getattr(self, "_remove_mode" + active_mode)
                remove_mode()
                activate_new_mode = prev_active_mode
                prev_active_mode = active_mode
                self.call_script_reaction(prev_active_mode, None, 'mode_is_deactivated')
                active_mode = activate_new_mode
                mode_to_call = getattr(self, "_mode" + activate_new_mode)
                mode_to_call()
                self.call_script_reaction(activate_new_mode, None, 'mode_is_activated')
        else:
            if 'active_mode' in globals():
                remove_mode = getattr(self, "_remove_mode" + active_mode)
                remove_mode()
                prev_active_mode = active_mode
                self.call_script_reaction(prev_active_mode, None, 'mode_is_deactivated')
            active_mode = activate_new_mode
            mode_to_call = getattr(self, "_mode" + activate_new_mode)
            mode_to_call()
            self.call_script_reaction(activate_new_mode, None, 'mode_is_activated')
    def target_by_name(self, target_list, name):
        matches = filter(lambda t: t.display_name == name, target_list)
        if matches:
            return matches[0]
        return
    def _add_custom_lom_listeners_handler(self, mode_number, updated_by=False):
        self.log("custom lom listeners refreshed")
        name_string = "_mode" + str(mode_number) + "_custom_lom_listeners"
        if hasattr(self, name_string):
            try:
                mode_to_call = getattr(self, name_string)
                mode_to_call(updated_by)
            except:
                self.log_message("csslog: unable to run " + name_string)
                pass
    def _remove_custom_lom_listeners_handler(self, mode_number, updated_by=False):
        name_string = "_remove_mode" + str(mode_number) + "_custom_lom_listeners"
        if hasattr(self, name_string):
            try:
                mode_to_call = getattr(self, name_string)
                mode_to_call(updated_by)
            except:
                self.log_message("csslog: unable to run " + name_string)
                pass
    def get_modifier_value(self, mod_name):
        return self.modifiers[mod_name]["value"]
    def set_modifier_value(self, mod_name, contents):
        global active_mode
        self.modifiers[mod_name]["value"] = contents
        self.call_script_reaction(active_mode, mod_name, "modifier_was_updated")
    def call_script_reaction(self, mode_id, param2, reaction_name):
        one = "";
        two = "";
        three = "";
        if(mode_id!=None):
            one = "_mode_" + str(mode_id)
```

```python
        if(param2!=None):
            two = "_" + str(param2)
        if(reaction_name!=None):
            three = "_" + str(reaction_name)
        reaction_method = one + two + three
        if hasattr(self, reaction_method):
            getattr(self, reaction_method)()
    def disconnect(self):
        self.call_script_reaction(None, None, 'script_is_disconnected')
        super(css_atcoperator_imported_1, self).disconnect()
```