Source code sample Python

Excerpt from the main class of a generative music composing engine.

```
# [...] marks repetitive parts or long sections with non-interesting material removed for demo-
# purposes

# CreateNotes.py is part of the master thesis of B.Sc. Marc A. Modrow titled ”Generative Music
# Scoring for Interactive Entertainment Software” written at the University Bremen (GER) in
# the program of  digital media in 2013.
# Copyright (c) 2013 Marc A. Modrow (mmodrow@uni-bremen.de)
#
# CreateNotes.py is used to create instances of the Pure Data wrapper for the NoteObject.py-
# Class. This class also contains all the algorithms needed for composing. This makes
# CreateNotes the heart of the composing engine.
#
# This is based on a script by Thomas Grill called simple.py that showed how to print and give
# to outlets text based on the input type from pyext to Pure Data.
# Original credits of sample file used as basis:
#
# py/pyext - python script objects for Pd and MaxMSP
#
# Copyright (c) 2002-2007 Thomas Grill (gr@grrrr.org)
# For information on usage and redistribution, and for a DISCLAIMER OF ALL
# WARRANTIES, see the file, ”license.txt” in this distribution.
#

try:
  import pyext
except:
  print ”ERROR: This script must be loaded by the Pd/Max pyext external”

from array import *
import random, math

import PyHelper

#################################################################
# Well, it creates notes and control messages :p
# It gives all the info needed to the dyn_note and dyn_ctl dynamic patchers.
# This is where all the magic happens
class CreateNotes(pyext._class):
  # number of inlets and outlets
  _inlets=1
  _outlets=3

  # variables

  # needed as the score has to be written a few ticks ahead of playing it back.
  # Measured in full notes.
  time_offset = 1.0/8.0
  # diagnose mode toggle
  create_notes_diag = 1

  # the name of all 12 notes in the chromatic circle starting with C
  chromatic_circle = ([”C”, ”Cis”, ”D”, ”Dis”, ”E”, ”F”, ”Fis”, ”G”, ”Gis”,
    ”A”, ”Ais”, ”B”])
  # the half-tone step width of a few of the most common scales
  # other scales can be set up manually
  scales = {’M’: [2, 2, 1, 2, 2, 2, 1],’nm’: [2, 1, 2, 2, 1, 2, 2], ’hm’: [2, 1, 2, 2, 1,3,1],
    ’mm’: [2, 1, 2, 2, 2, 2, 1], ’chr’: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
  # a selection of chords with their individual half-tone step widths retreived from
  # http://www.musictheory.net/lessons
  # in the notation read /o as half-diminished and o’s as diminished, +’s are augmentations,
  # trailing 7’s as a seventh-chord.
  # leading numbers are read as superscript_subscript, or just superscript if no _ is present,
  # following the chord name to mark inversions. E.g. 27 is a third inversion 7th chord.
  # all non-inverted begin with a 0 as they are all counted from the root note and not from
  # the previous note like the scales, ergo inverted chords start off with a 12.
  chords = {
    # simple triads
    ’M’: [0, 4, 7], ’m’: [0, 3, 9], ’+’: [0, 4, 8], ’o’: [0, 3, 6],
    # single inverted triads
    ’6M’: [12, 4, 7], ’6m’: [12, 3, 9], ’6+’: [12, 4, 8], ’6o’: [12, 3, 6],
    # double inverted triads
    ’6_4M’: [12, 16, 7], ’6_4m’: [12, 15, 9], ’6_4+’: [12, 16, 8], ’6_4o’: [12, 15, 6],
    # sevenths
    ’7’: [0, 4, 7, 10], ’M7’: [0, 4, 7, 11], ’m7’: [0, 3, 7, 10], ’/o7’: [0, 3, 6, 10],
    ’o7’: [0, 3, 6, 9],
    # [...]
    }
  # split up the chords so they are callable by type. To be called as chords[triads[A][B]] or
  # chords[sevenths[A][B]] whereas A is the inversion count and B is the chord number in the
  # class. This eases the randomisation of chord decision.
  triads = [['M', 'm', '+', 'o'], ['6M', '6m', '6+', '6o'], ['6_4M', '6_4m',
    '6_4+', '6_4o']]
  sevenths = [['7', 'M7', 'm7', '/o7', 'o7', 'mM7', '+M7', '+7'], ['6_57',
    '6_5M7', '6_5m7', '6_5/o7', '6_5o7', '6_5mM7', '6_5+M7', '6_5+7'], ['4_37',
    '4_3M7', '4_3m7', '4_3/o7', '4_3o7', '4_3mM7', '4_3+M7', '4_3+7'],
    ['27', '2M7', '2m7', '2/o7', '2o7', '2mM7', '2+M7', '2+7']]



  # methods for all inlets

  # class-specific diagnose mode toggle for console output in Pd
  def diag_1(self, toggle):
    if toggle == 0 or toggle == "false":
      CreateNotes.create_notes_diag = 0
      print "CreateNotes diagnose mode deactivated"
    elif toggle == 1 or toggle == "true":
      CreateNotes.create_notes_diag = 1
      print "CreateNotes diagnose mode activated"
    else:
      print "CreateNotes diagnose mode only accepts 0 or 1 as input. ", arg, "is invalid."

  # the trigger-message updates the current timestamp and triggers loop  methods were
  # applicable. Gets called for every instrument automatically by Pd
  def trigger_1(self, timestamp):
    self.general_settings['current_timestamp'] = timestamp
    if self.general_settings['active']:
      if self.progression_settings['active'] and (self.progression_settings['next_timestamp'] <= timestamp or self.progression_settings['next_timestamp'] == -1):
        self.progression()
      if self.kpc_settings['active'] and (self.kpc_settings['next_timestamp'] <= timestamp or self.kpc_settings['next_timestamp'] == -1):
        self.key_phrase_composition()
      if self.metronome_settings['active']:
        self.metronome()
      if self.bin_subdiv_settings['active'] and (self.bin_subdiv_settings['next_timestamp'] <= timestamp or self.bin_subdiv_settings['next_timestamp'] == -1):
        self.bin_subdiv()
    self.general_settings['prev_timestamp'] = self.general_settings['current_timestamp']

  # stochastic binary subdivision as described in the thesis in the scoring-chapter. It is
  # triggered by the trigger_1()-method mostly good for drums,
  # but applicable on melody as well
  def bin_subdiv(self):
    if self.create_notes_diag:
        print "bin subdiv"
    # gets the current time from the last update (happens in the trigger-method)
    timestamp = self.general_settings['current_timestamp']# - self.time_offset
    # prepare an empty note stack buffer and call the recursive method.
    self.bin_subdiv_settings['notestack'] = []
    self.bin_subdiv_rec(self.bin_subdiv_settings['bound_dur_long'], timestamp)
    # some naive chromatic scale running for melody instruments
    if self.general_settings['channel'] != 10:
      i = 30
      for note in self.bin_subdiv_settings['notestack']:
        note[0] = self.legal[i%len(self.legal)]
        i += 1
    # Sets the next time for this method to be activated. This is after the runtime of the
    # maximum allowed note length - the amount of time covered by one call.
    self.bin_subdiv_settings['next_timestamp'] = self.general_settings['current_timestamp'] + (1.0/self.bin_subdiv_settings['bound_dur_long'])
    if self.create_notes_diag:
      print "next timestamp = ", self.bin_subdiv_settings['next_timestamp'], "equals: ", self.general_settings['current_timestamp'], "+ 1/", self.bin_subdiv_settings['bound_dur_long']
    # create the actual notes from the note stack
    self.notes_from_stack(timestamp, self.bin_subdiv_settings['notestack'] )

  # this is the recursive function to work through the given time frame
  # by dividing it into a binary tree of durations of a given depth.
  # usually called by bin_subdiv()
  def bin_subdiv_rec(self, dur, timestamp):
    # is further division possible and does chance allow it?
    if dur < self.bin_subdiv_settings['bound_dur_short'] and random.random() <= self.bin_subdiv_settings['subdiv_probability']:
      # half the duration (it is seen as 1/dur so doubling the value  halves the duration)
      # and call two new instances for the given timestamp and duration
      dur *= 2
      self.bin_subdiv_rec(dur, timestamp)
      self.bin_subdiv_rec(dur, timestamp + 1.0/dur)

    # no further division shall be done, create notes instead.
    else:
      # base velocity
      vel = 60

      # melody instuments
      if self.general_settings['channel'] != 10:
        # small chance for a rest
        if random.random() <=0.1:
          vel = 0
          if self.create_notes_diag:
              print "Including pause"
        # create a note. The pitch will be assigned in bin_subdiv()
        self.bin_subdiv_settings['notestack'].append([60, vel, self.general_settings['channel'], dur, timestamp + self.time_offset])
        if self.create_notes_diag:
          print "Melody notes generated at dur = ", dur

      # this is for channel 10 = percussion
      else:
        # as the scale is not applied to this these are raw MIDI pitches
        bass_drum = [35, 36]
        snare = [37, 38, 40]
        hihat = [42, 44, 46]
        tom = [41, 43, 45, 47, 48, 50]
        ride = [51, 53, 59]
        crash = [49, 57]
        effect_cymbal = [53, 54, 55, 56]
        # find the position within the measure
        current_beat = self.exact_current_beat(timestamp)
        output = []
        if self.create_notes_diag:
          print "current_beat = ", current_beat, ", measure length = ", self.general_settings['measure_length'], ", timestamp = ", timestamp

        # adjust velocity and create notes. for each position in the measure
        # each voice has a different probability.
        #
        # the down-beat:
        if current_beat == 0:
          if random.random() <=0.9:
            vel = random.randrange(80, 127)
          if random.random() <=0.8:
            output.append(random.choice(bass_drum))
          if random.random() <=0.2:
            output.append(random.choice(snare))
          if random.random() <=0.6:
            output.append(random.choice(hihat))
          if random.random() <=0.6:
            output.append(random.choice(tom))
          if random.random() <=0.4:
            output.append(random.choice(ride))
          if random.random() <=0.4:
            output.append(random.choice(crash))
          if random.random() <=0.3:
            output.append(random.choice(effect_cymbal))
        # each full beat (number counts)
        # [...]
        # 8th off the beat ("and"s)
        # [...]
        # 16th off the 8th ("e"s and "de"s)
        # [...]
        # everything below 16th level not on anything above
        # [...]
        if self.create_notes_diag:
          print "Drum notes generated for ", output, " at dur = ", dur
        # append the gathered output data to the note stack for bin_subdiv().
        self.bin_subdiv_settings['notestack'].append([output, vel, self.general_settings['channel'], dur, timestamp + self.time_offset])

  # wrapper for set_bin_subdiv for Pd
  def set_bin_subdiv_1(self, *args):
    self.set_bin_subdiv(args)

  # sets bin_subdiv settings ;)
  # when setting active to true it also sets up an appropriate starting time (next full note)
  def set_bin_subdiv(self, args):
    # checks whether the args are consisting of a valid variable name and an integer
    if len(args) == 2 and PyHelper.isNumber(args[1]) and str(args[0]) in self.bin_subdiv_settings:
      self.bin_subdiv_settings[str(args[0])] = args[1]
      # additionally checks for the variable name being active and the value being one
      # to prepare a proper starting time for the next full note to hit the crucial beats
      if str(args[0]) == "active" and args[1] == 1:
          self.bin_subdiv_settings['next_timestamp'] = round(self.general_settings['current_timestamp']+0.5)
          print "bin_subdiv set to start at ", self.bin_subdiv_settings['next_timestamp'], ", which is beat#", self.exact_current_beat(self.bin_subdiv_settings['next_timestamp'])
      if self.create_notes_diag:
        print "This CreateNotes' bin_subdiv settings['", args[0], "'] is set to", self.bin_subdiv_settings[str(args[0])]
```

```python

  # tells the exact position in the current bar. if a timestamp is handed it is used instead
  # of the one stored in general_settings['current_timestamp']
  def exact_current_beat(self, timestamp = -1):
    if timestamp == -1:
      timestamp = self.general_settings['current_timestamp']
    current_beat = (4 * timestamp)%self.general_settings['measure_length']
    return current_beat

  # tells the last quater note passed. if a timestamp is handed it is used instead of the one
  # stored in general_settings['current_timestamp']
  def passed_quater_beat(self, timestamp = -1):
    if timestamp == -1:
      timestamp = self.general_settings['current_timestamp']
    last_quater = int(4 * timestamp)%self.general_settings['measure_length']
    return last_quater

  # tells the exact position in the current bar at the previous tick. if a timestamp is handed
  # it is used instead of the one stored in general_settings['current_timestamp']
  def exact_previous_beat(self, timestamp = -1):
    if timestamp == -1:
      timestamp = self.general_settings['prev_timestamp']
    last_beat = (4 * (timestamp))%self.general_settings['measure_length']
    return last_beat

  # tells the bar number the timer currently is within. if a timestamp is handed it is used
  # instead of the one stored in general_settings['current_timestamp']
  def current_bar(self, timestamp = -1):
    if timestamp == -1:
      timestamp = self.general_settings['current_timestamp']
    return int(timestamp / self.general_settings['measure_length'])

  # a basic metronome. It only returns the current measure and beat to outlet 2.
  # depending on the setting in framework.pd/metronome that will be passed as
  # chn. 10 note information or rendered into the Pd-DAC directly.
  def metronome(self):
    # prep
    last_beat = self.exact_previous_beat()
    current_beat = self.exact_current_beat()
    last_quater = self.passed_quater_beat()
    # print "metro: last_beat: ", last_beat, " current_beat: ", current_beat, " last_quater: ", last_quater
    # calculate
    if (last_beat < last_quater or last_beat > current_beat) and current_beat >= last_quater:
      if (last_quater+1)%self.general_settings['measure_length'] == 0:
        self.metronome_settings['total_measures'] += 1
      # sending out the data
      self._outlet(2, [self.metronome_settings['total_measures'], (last_quater + 1)%self.general_settings['measure_length']])
      if self.create_notes_diag:
        print "Metronome: Beat #", (last_quater + 1)%self.general_settings['measure_length'], "in bar # ", self.metronome_settings['total_measures'], "(last beat:", last_beat, ", current_beat: ", current_beat, ", last_quater:", last_quater, ")"
    # setting up the current timestamp as previous for the next call
    self.metronome_settings['prev_timestamp'] = self.general_settings['current_timestamp']

  # wrapper for set_metro for Pd
  def set_metro_1(self, *args):
    self.set_metro(args)

  # sets metronome settings ;)
  def set_metro(self, args):
    # checks whether the args are consisting of a valid variable name and an integer
    if len(args) == 2 and PyHelper.isNumber(args[1]) and str(args[0]) in self.metronome_settings:
      self.metronome_settings[str(args[0])] = args[1]
      if self.create_notes_diag:
        print "This CreateNotes' metronome settings['", args[0], "'] is set to", self.metronome_settings[str(args[0])]

  # gets settings ;)
  def get_metro_1(self, *args):
    if len(args) == 0 or (len(args) == 1 and args[0] == 0):
      print self.metronome_settings.items()
    elif len(args) == 1 and self.metronome_settings.has_key(str(args[0])):
      print self.metronome_settings[str(args[0])]
    else:
      print "Didn't know what to do with ", args

  # wrapper for progression for Pd
  def progression_1(self, i):
    if self.progression_settings['active'] and self.progression_settings['next_timestamp'] == i or self.progression_settings['next_timestamp'] == -1:
      self.general_settings['current_timestamp'] = i
      progression()

  # usually launched by trigger(). for functionality look at the implementation chapter
  # of the thesis this belongs to
  def progression(self):
    if self.create_notes_diag:
        print "progression"
    # set up the variables
    # current note duration by density
    notes_per_full = math.pow(2, random.randint(self.progression_settings['bound_dur_long'], self.progression_settings['bound_dur_short']))
    self.progression_settings['duration'] = 1.0/notes_per_full
    current_beat = self.exact_current_beat()
    # if set duration would cross the bar line cut duration to the bar line
    if current_beat + self.progression_settings['duration'] > self.general_settings['measure_length']:
      self.progression_settings['duration'] = self.general_settings['measure_length'] - current_beat
    self.progression_settings['next_timestamp'] = self.general_settings['current_timestamp'] + self.progression_settings['duration']
    if self.create_notes_diag:
        print "next progression timestamp = ", self.progression_settings['next_timestamp']
    self.progression_settings['prev_timestamp'] = self.general_settings['current_timestamp']
    # define a pitch for the next note
    self.progression_settings['note_buff'] = self.legal[(self.progression_settings['note_buff'] + random.randint(self.progression_settings['bound_step_low'], self.progression_settings['bound_step_up'])) % len(self.legal)]
    # define a velocity for the next note
    vel_buff = 127 - self.progression_settings['note_buff'] + random.randint(-10, 10)
    # do create the note with a 5% chance of a rest
    if random.random() < 0.95:
      # amount of notes to be generated at this timestamp
      amount = 1
      # chance of a chord of at least 2 notes
      if random.random() < self.progression_settings['chord_probability']:
        amount = 2
        # chance of a chord of at least 3 notes
        if random.random() < self.progression_settings['chord_probability']:
          amount = 3
          # chance of a chord of 4 notes
          if random.random() < self.progression_settings['chord_probability']:
            amount = 4
            if self.create_notes_diag:
              print "Seventh chord!"
          elif self.create_notes_diag:
            print "Triad chord!"
        elif self.create_notes_diag:
          print "Secundian chord!"
      # create a chord from the amount of notes and the base note
      chord = self.random_chord(self.progression_settings['note_buff'], amount)
      if self.create_notes_diag:
        print [chord, vel_buff, self.general_settings['channel'], notes_per_full, self.general_settings['current_timestamp'] + self.time_offset]
      # actually send the note to Pd
      self.note([chord, vel_buff, self.general_settings['channel'], notes_per_full, self.general_settings['current_timestamp'] + self.time_offset])

  # wrapper for set_progression for Pd
  def set_prog_1(self, *args):
    self.set_prog(args)

  # sets progression settings ;)
  def set_prog(self, args):
    # checks whether the args are consisting of a valid variable name and an integer
    if len(args) == 2 and PyHelper.isNumber(args[1]) and str(args[0]) in self.progression_settings:
      self.progression_settings[str(args[0])] = args[1]
      if self.create_notes_diag:
        print "This CreateNotes progression settings['", args[0], "'] is set to", self.progression_settings[str(args[0])]
    # otherwise it is supposed to be a bulk-change
    else:
      if len(args)>0:
        self.general_settings['channel'] = args[0] if PyHelper.isNumber(args[0]) else 0
        if len(args)>1:
          self.progression_settings['chord_probability'] = args[1] if PyHelper.isNumber(args[1]) else 0
          if len(args)>3:
            self.progression_settings['bound_step_low'] = args[2] if PyHelper.isNumber(args[2]) else 0
            self.progression_settings['bound_step_up'] = args[3] if PyHelper.isNumber(args[3]) else 0
            if len(args)>5:
              self.general_settings['bound_valid_low'] = args[4] if PyHelper.isNumber(args[4]) else 0
              self.general_settings['bound_valid_up'] = args[5] if PyHelper.isNumber(args[5]) else 0
              if len(args)>7:
                self.progression_settings['bound_dur_long'] = args[6] if PyHelper.isNumber(args[6]) else 0
                self.progression_settings['bound_dur_short'] = args[7] if PyHelper.isNumber(args[7]) else 0
                if len(args)>8:
                  self.general_settings['measure_length'] = args[8] if PyHelper.isNumber(args[8]) else
```

```
                    0
      if self.create_notes_diag:
        print "CreateNotes' progression_settings is set to: ", self.progression_settings

  # gets settings ;)
  def get_prog_1(self, *args):
    if len(args) == 0 or (len(args) == 1 and args[0] == 0):
      print self.progression_settings.items()
    elif len(args) == 1 and self.progression_settings.has_key(str(args[0])):
      print self.progression_settings[str(args[0])]
    else:
      print "Didn't know what to do with ", args

  # wrapper for key_phrase_composition for Pd
  def key_phrase_composition_1(self, i):
    if self.kpc_settings['active'] and self.kpc_settings['next_timestamp'] == i or self.kpc_settings['next_timestamp'] == -1:
      self.general_settings['current_timestamp'] = i
      key_phrase_composition()

  # usually launched by trigger(). for functionality look at the implementation chapter
  # of the thesis this belongs to
  def key_phrase_composition(self):
    # div0-error catch:
    if self.general_settings['measure_length'] <= 0:
      self.general_settings['measure_length'] = 1
    if self.create_notes_diag:
        print "kpc"
    time_diff = 0
    # on start/ after reset
    if self.kpc_settings['next_keyphrase'] == -1 or self.kpc_settings['next_keyphrase'] <= self.general_settings['current_timestamp']:
      # set the aspects of the next key phrase
      self.kpc_settings['next_keyphrase'] = self.general_settings['current_timestamp'] + random.randint(self.kpc_settings['key_dist_low'], self.kpc_settings['key_dist_up'])
      time_diff = self.kpc_settings['next_keyphrase'] - self.general_settings['current_timestamp']
      rand = random.randint(time_diff * -1, time_diff) if time_diff * -1 < time_diff else random.randint(time_diff, time_diff * -1)
      self.kpc_settings['keyphrase_target'] = self.kpc_settings['note_buff'] + rand*self.kpc_settings['mobility']
    # define the next keyframe time and target
    time_left = self.kpc_settings['next_keyphrase'] - self.general_settings['current_timestamp']
    # make sure to have a target that differs from the current note; otherwise the next
    # phrase will be rather dull
    while self.kpc_settings['keyphrase_target'] == self.kpc_settings['note_buff']:
      self.kpc_settings['keyphrase_target'] = self.kpc_settings['note_buff'] + random.randint(time_diff * -1, time_diff)*self.kpc_settings['mobility']
    dist_left = self.kpc_settings['keyphrase_target'] - self.kpc_settings['note_buff']
    tendency = time_left * self.general_settings['measure_length'] / dist_left
    notes_per_full = math.pow(2, random.randint(self.kpc_settings['bound_dur_long'], self.kpc_settings['bound_dur_short']))
    # set up the variables
    self.kpc_settings['duration'] = 1.0/notes_per_full
    current_beat = self.exact_current_beat()
    # catch and correct if the next note would pass the bar line
    if current_beat + self.kpc_settings['duration'] > self.general_settings['measure_length']:
      self.kpc_settings['duration'] = self.general_settings['measure_length'] - current_beat

    self.kpc_settings['next_timestamp'] = self.general_settings['current_timestamp'] + self.kpc_settings['duration']
    self.kpc_settings['prev_timestamp'] = self.general_settings['current_timestamp']
    if self.create_notes_diag:
        print self.kpc_settings['next_timestamp'], tendency
    # make sure that the scale has been set and choose a pitch
    if not len(self.legal) == 0:
      rand = random.uniform(self.kpc_settings['bound_step_low'] * tendency, self.kpc_settings['bound_step_up'] * tendency)
      self.kpc_settings['note_buff'] = self.legal[(self.kpc_settings['note_buff'] + int(rand)) % len(self.legal)]
      if self.create_notes_diag:
        print self.kpc_settings['note_buff'], len(self.legal), rand
    ########
    # from here on it is similar to progression - basic chord creation
    ########
    vel_buff = 127 - self.kpc_settings['note_buff'] + random.randint(-10, 10)

    # do create the note with a 5% chance of a rest
    if random.random() < 0.95:
      # amount of notes to be generated at this timestamp
      amount = 1
      # chance of a chord of at least 2 notes
      if random.random() < self.kpc_settings['chord_probability']:
        amount = 2
        # chance of a chord of at least 3 notes
        if random.random() < self.kpc_settings['chord_probability']:
          amount = 3
          # chance of a chord of 4 notes
          if random.random() < self.kpc_settings['chord_probability']:
            amount = 4
            if self.create_notes_diag:
              print "Seventh chord!"
          elif self.create_notes_diag:
            print "Triad chord!"
        elif self.create_notes_diag:
          print "Secundian chord!"
      # create a chord from the amount of notes and the base note
      chord = self.random_chord(self.kpc_settings['note_buff'], amount)
      if self.create_notes_diag:
        print [chord, vel_buff, self.general_settings['channel'], notes_per_full, self.general_settings['current_timestamp'] + self.time_offset]
      # actually send the note to Pd
      self.note([chord, vel_buff, self.general_settings['channel'], notes_per_full, self.general_settings['current_timestamp'] + self.time_offset])

  # wrapper for set_kpc for Pd
  def set_kpc_1(self, *args):
    self.set_kpc(args)

  # sets kpc settings ;)
  def set_kpc(self, args):
    # checks whether the args are consisting of a valid variable name and an integer
    if len(args) == 2 and PyHelper.isNumber(args[1]) and str(args[0]) in self.kpc_settings:
      self.kpc_settings[str(args[0])] = args[1]
      if self.create_notes_diag:
        print "This CreateNotes' kpc settings['", args[0], "'] is set to", self.kpc_settings[str(args[0])]
    # otherwise it is supposed to be a bulk-change
    else:
      if len(args)>0:
        self.general_settings['channel'] = args[0] if PyHelper.isNumber(args[0]) else 0
        if len(args)>1:
          self.kpc_settings['chord_probability'] = args[1] if PyHelper.isNumber(args[1]) else 0
          if len(args)>3:
            self.kpc_settings['bound_step_low'] = args[2] if PyHelper.isNumber(args[2]) else 0
            self.kpc_settings['bound_step_up'] = args[3] if PyHelper.isNumber(args[3]) else 0
            if len(args)>5:
              # [...]
      if self.create_notes_diag:
        print "CreateNotes' kpc_settings is set to: ", self.kpc_settings

  # gets settings ;)
  def get_kpc_1(self, *args):
    if len(args) == 0 or (len(args) == 1 and args[0] == 0):
      print self.kpc_settings.items()
    elif len(args) == 1 and self.kpc_settings.has_key(str(args[0])):
      print self.kpc_settings[str(args[0])]
    else:
      print "Didn't know what to do with ", args

  # wrapper for note for Pd
  def note_1(self, *values):
    self.note(*values)

  # here actually the note making magic happens as this constructs the note creating message.
  def note(self, *values):
    import types
    # under certain circumstances it happens that the note is an array in
    # the first element of another array. This compensates for that.
    if len(values) == 1 and type(values[0]) in (types.TupleType, types.ListType):
      values = values[0]

    # if values[0] is a tuple we effectively have a chord
    # if values[0] is an empty tuple it is correctly ignored ;)
    if type(values[0]) in (types.TupleType, types.ListType):
        for pitch in values[0]:
          self.note([pitch, values[1], values[2] if len(values) == 5 else self.general_settings['channel'], values[3] if len(values) == 5 else values[2], values[4] if len(values)==5 else values[3]])
        # break this execution for a new call otherwise there will be type errors
        # quasi-recursion makes things much easier here
        return
    # initiate all needed varaibles
    pitch = 0
    vel = 0
    chn = 0
    dur = 0
    timestamp = 0
    # a note of 4 elements has no channel. This should be default!
    # general_settings['channel'] is used then
    if len(values) == 4:
      pitch = values[0]
      vel = values[1]
      chn = self.general_settings['channel']
      dur = values[2]
      timestamp = values[3]
    # if the note has 5 elements the third is interpreted as channel
    elif len(values) == 5:
      pitch = values[0]
      vel = values[1]
      chn = values[2]
      dur = values[3]
      timestamp = values[4]
    # if it has neither 4 nor 5 elements it is incomplete or overlong and thus corrupt
    else:
      print len(values), "is not a valid number of arguments to create a note message."
    if self.create_notes_diag:
      print "Generating note: pitch=", pitch, "=", self.chromatic_circle[pitch%12], "vel=", vel, "chn=", chn, "dur=", dur, "timestamp=", timestamp

    # if vel is 0 it is a rest.
    if vel > 0:
      # otherwise hand it to Pd
      self._outlet(1, pitch, vel, chn, dur, timestamp)

  # creates notes from a stack-array
  # the elements from the stack are: pitch, vel, chn, dur, timestamp
  # first element may be an array. this is resolved in note()
  def notes_from_stack(self, timestamp, stack):
    import types
    if 4 <= len(stack[0]) <= 5:
      for item in stack:
        # if vel is 0 it is a rest.
        if item[1] > 0:
          self.note([item[0], item[1], item[2], item[3], timestamp if len(item)==4 else item[4]])
          if self.create_notes_diag:
            print "note to create from stack: ", [item[0], item[1], item[2], item[3], timestamp if len(item)==4 else item[4]]
        timestamp += 1.0/(item[3])
    else:
      print "Notes are supposed to have 4 or 5 arguments for a stack-operation, not ", len(stack), "!"

  # wrapper for control for Pd
  def control_1(self, *values):
    self.control(values)

  # here actually the control message making magic happens as this constructs the control
  # message creating message.
  def control(self, *values):
    import types
    # under certain circumstances it happens that the note is an array in
    # the first element of another array. This compensates for that.
    if len(values) == 1 and type(values[0]) in (types.TupleType, types.ListType):
      values = values[0]
    # there is no stack-execution for control messages. make individual calls for each one

    # initiate all needed varaibles
    ctl = 0
    value = 0
    chn = 0
    timestamp = 0
    # a control message of 3 elements has no channel. This should be default!
    # general_settings['channel'] is used then
    if len(values) == 3:
      ctl = values[0]
      value = values[1]
      chn = self.general_settings['channel']
      timestamp = values[2]
    # if the note has 4 elements the third is interpreted as channel
    elif len(values) == 4:
      ctl = values[0]
      value = values[1]
      chn = values[2]
      timestamp = values[3]
    else:
      print len(values), "is not a valid number of arguments to create a control message."
    if self.create_notes_diag:
      print "Generating control: ctl=", ctl, "value=", value, "chn=", chn, "timestamp=", timestamp
    # there are no rests for control messages - everything gets passed
    self._outlet(3, ctl, value, chn, timestamp)

  # creates control messages from a stack-array
  # the elements from the stack are: ctl, value, chn, timestamp
  def control_from_stack(self, timestamp, *stack):
    import types
    for item in stack[0]:
      if len(item) == 3:
        self.control(item[0], item[1], item[2], timestamp)
      elif len(item) == 2:
        self.control(item[0], item[1], self.general_settings['channel'], timestamp)

  # wrapper for scale for Pd
  def scale_1(self, name, *stepargs):
    self.scale(name, str(stepargs))

  # this method defines all the legal tones for the chosen key in self.legal[]
  # name is an int from 0-11 corresponding to the half-tones from C to B or an upper-case note
  # name as string.
  # stepargs is either a series of half-tone steps that add up to 12 or a scale name as
  # defined in self.scales
  def scale(self, name, *stepargs):
    # regular expression package
    import re
    self.legal = []
    # exclude percussion instruments - they do not need scales
    if self.general_settings['channel'] != 10:
      if self.create_notes_diag:
        print "Defining scale:", name, ", ", stepargs, "len(steps):", len(stepargs)
      # validate name from the chromatic circle if it's no number
      if not PyHelper.isNumber(name):
        try:
          name = self.chromatic_circle.index(name)
        except ValueError:
          name = random.randint(0, 11)
          if self.create_notes_diag:
            print "Name is neither a number nor the proper name of a note... I just pick one. I choose ", self.chromatic_circle[name]
      # buffer for the step array
      steps = []
      # if the stepargs' length is one and it contains anything but numbers, commas,
      # whitespaces and brackets is't most likely actually a string
      if len(stepargs) == 1 and not re.match("^[0-9\,\(\)\ ]*$", stepargs[0]):
        scalename = ""
        # sometimes the name of the scale comes in with "decorations" - cutting them off
        if  len(stepargs[0])>3 and "Symbol" in str(stepargs[0]):
          scalename = stepargs[0][9:-3]
        else:
          scalename = stepargs[0]
        if self.create_notes_diag:
          print "The scale name is: ", scalename
        # use a predefined set of stepwidths from self.scales named by scalename
        if str(scalename) in self.scales:
          steps = self.scales[str(scalename)]
          if self.create_notes_diag:
            print "Set scale to ", self.chromatic_circle[name], str(scalename), "."
        # the input is no valid name thus a chromatic scale is chosen as fallback
        else:
          if self.create_notes_diag:
            print "Input non intelligable. Falling back to ", self.chromatic_circle[name], "chromatic scale."
          steps = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

      # use what you are given
      # is stepargs consisting only of numbers and Pd-specific list-decorations?
      elif len(stepargs) == 1 and re.match("^[0-9\,\(\)\ ]*$", stepargs[0]):
        # clean up the decorations and build a proper array from the string
        for i in stepargs[0]:
          if re.match("[0-9]", i):
            steps.append(int(i))
      # else it already is a proper array that can be used right away
      else:
        steps = stepargs
      # this section is only relevant in diagnose mode so it got into that if section
      if self.create_notes_diag:
        print "steps are: ", steps
        # check if the scale adds up to a full octave. If it does not nothing changes in
        # terms of execution, but if diagnose mode is on a warning is displayed as it
        # seems highly unlikely that it has been done intentionally
        c = 0
        for i in steps:
          c += i
        if not c == 12:
          if self.create_notes_diag:
            print "Something went wrong. The steps should add up to 12, but did add up to", c, ". Is that intended?"
      # i1 (short for index 1) is where the selection starts. As MIDI pitch 0 is C-1 (5
      # octaves below middle C) it can be counted from there on
      # i2 (short for index 2) is the length of the next step in the scale
      # both are iterating over the course of the upcoming loop
      i1 = name
      i2 = 0
      # this stores all allowed note indices before pushing them to the class space array
      legalbuffer = []
      # run until the upper boundary of the allowed MIDI space is reached
      while i1 <= self.general_settings[’bound_valid_up’]:
        # only append notes that are at least as high as the lowest allowed note
        if i1 >= self.general_settings[’bound_valid_low’]:
          legalbuffer.append(i1)
        # follow the scale regardless of the lower boundary
        # you will get there soon enough
        # traverse the pitches according to the next point int the scale
        i1 +=  steps[i2]
        # loop over the length of the steps array
        i2 = (i2+1)%(len(steps))
      # after the loop all the notes collected within it get written to self.legal
      self.legal = legalbuffer
      # here the allowed tone names are retrieved
      # as they are ownly needed for diagnose mode all the section is put into it.
      if self.create_notes_diag:
        tone_names_allowed = ””
        sum = 0
        i = 0
        while i < len(steps):
          sum = 0
          # the step widths need to be added for the index of each note to arise
          for j in range(i):
            sum += steps[j]
          tone_names_allowed += ” ” + self.chromatic_circle[(sum+name)%12]
          i += 1
        print ”Following notes are allowed:”, tone_names_allowed
    # all the way back: This is for channel 10 (percussions) only. In general MIDI
    # percussions use only the pitches 35 through 81, so all of those are legal
    else:
      for i in range(35, 81):
        self.legal.append(i)
      if self.create_notes_diag:
        print (”This is an instrument on channel 10. Channel 10 is meant for percussion ”
          ”instruments. They don’t need conventional scales. I better skip this.”)

  # wrapper for chord for Pd
  # directly creates a chord as notes for the current timestamp + offset
  def chord_1 (self, rootnote, chordname):
    self.note([self.chord(rootnote, str(chordname)), 50, self.general_settings[’channel’], 8, self.
        general_settings[’current_timestamp’] + self.time_offset])

  # purely for testing custom chords from within Pd
  def chord_test_1(self, rootnote):
    self.note([self.chord(rootnote, [0, 2, 3]), 50, self.general_settings[’channel’], 8, self.
        general_settings[’current_timestamp’] + self.time_offset])

  # this method creates a chord.
  # it expects the root note of the chord (int or string) and its name (string or int-array
  # with half-tone-steps for custom chords) and returns the pitches in plain MIDI values
  def chord(self, rootnote, chordname):
    chord = []
    import types
    if self.create_notes_diag:
      print ”rootnote is a : ”, type(rootnote), ” chordname is a: ”, type(chordname)
    # if the rootnote is no int
    if not PyHelper.isNumber(rootnote):
      # transform string to int
      if PyHelper.isString(rootnote):
        rootnote = self.chromatic_circle.index(rootnote)
      # transform other (e.g. symbol) to int
      elif str(rootnote) in self.chromatic_circle:
        rootnote = self.chromatic_circle.index(str(rootnote))
    if self.create_notes_diag:
      print ”chord rootnote: ”, rootnote
    # if the rootnote is no valid midi note number this cannot work
    if PyHelper.isNumber(rootnote) and rootnote >= 0 and rootnote <= 127:
      # the chordname can be given either as string name or...
      if PyHelper.isString(chordname):
        if str(chordname) in self.chords:
          for i in self.chords[str(chordname)]:
            chord.append(rootnote + i)
        else:
          print ”You specified a non-legit chord name.”
      # ...as list of halftone steps
      elif isinstance(chordname, list):
        if self.create_notes_diag:
          print len(chordname), chordname
        for i in chordname:
          if PyHelper.isNumber(i):
```

```
          chord.append(rootnote + i)
      else:
        print "It seems something went wrong when naming the chord."
      if self.create_notes_diag:
        print "The chord I build consists of ", chord, "."
    # if already the rootnote was invalid
    else:
      print "It appears I tried to build a chord on a non-legitimate rootnote (", rootnote, "). Sorry for failing."
    return chord

  # returns a random chord starting with the given root note (see chord() for format) and the
  # amount of notes the chord shall include
  def random_chord(self, rootnote, amount):
    chordlist = []
    # hardly a chord
    if amount == 1:
      chordlist = rootnote
    # not yet a chord technically, but already perfect intervals can be applied to sound ok
    # most of the time
    elif amount == 2:
      chordlist = [rootnote, rootnote+random.choice([5, 7, 12])]
    # here comes the chord()-method into play
    elif amount == 3:
      chordlist = self.chord(rootnote, self.chords[random.choice(random.choice(self.triads))])
    elif amount == 4:
      chordlist = self.chord(rootnote, self.chords[random.choice(random.choice(self.sevenths))])
    # more than 4 notes are not prepared (pentachords are too seldom for it to be feasable)
    # and 0 or negative amounts don't make any sense
    else:
      print "amount of notes(", amount, ") for this chord is either too low or high."
    return chordlist

  # wrapper for set_general for Pd
  def set_general_1(self, *args):
    self.set_general(args)

  # gets general settings ;)
  def set_general(self, args):
    # checks whether the args are consisting of a valid variable name and an integer
    if len(args) == 2 and PyHelper.isNumber(args[1]) and str(args[0]) in self.general_settings:
      self.general_settings[str(args[0])] = args[1]
      if self.create_notes_diag:
        print "This CreateNotes' general settings['", args[0], "'] is set to", self.general_settings[str(args[0])]

  # creates a single note from a list - mostly for testing
  def list_1(self, *f):
    if len(f) >=5:
      self.note(f[0], f[1], f[2], f[3], f[4])

  # creates a line from Zimmer's "He's a Pirate"
  # needs the measure and beat to play from the tune. Is not run by trigger_1()
  def pirate_1(self, measure, beat):
    timestamp = self.general_settings['current_timestamp'] + self.time_offset
    c = self.general_settings['channel']
    notestack = [
        [
        [[[69],64,c,16], [72,64,c,16], [74,64,c,8]],
        [[77,64,c,16], [79,64,c,16], [76,64,c,8]]
        ],

        [
        [[[76],64,c,8], [74,64,c,16], [72,64,c,16]],
        [[74,64,c,8], [74,64,c,16], [76,64,c,16]]
        ],

        [
        [[[77],64,c,8], [77,64,c,8]],
        [[74,64,c,8], [0,0,c,8]]
        ]
      ]
    index = measure%len(notestack)
    self.notes_from_stack(timestamp, notestack[index] [beat]  )

  # creates a line from Evanescence's "My Immortal". Is not run by trigger_1()
  def immortal_1(self):
    # rest of 1/8
    timestamp = self.general_settings['current_timestamp'] + self.time_offset
    c = self.general_settings['channel']
    notestack = [[0,0,c,8], [64,64,c,8], [64,64,c,8], [62,64,c,8], [61,64,c,8], [59,64,c,16], [61,64,c,16.0/3.0], [61,64,c,8]]
    self.notes_from_stack(timestamp, notestack)

```

```python
  # Creates a chromatic linear progression of 16 notes over 2 bars in 8ths
  # Is not run by trigger_1(). Creates a bunch at a time, nondynamically and deterministic.
  def linear_1(self):
    timestamp = self.general_settings['current_timestamp'] + self.time_offset
    if self.create_notes_diag:
        print "linear from: ", timestamp
    notes_per_full = 8.0
    full_notes = 2.0
    c = self.general_settings['channel']
    for i in range(int(notes_per_full * full_notes)):
      self.note([90-i, 120-(i%notes_per_full)*8 ,c , notes_per_full, timestamp + i/notes_per_full])


  # Constructor - sets the values put in as args to the quintuple etc
  # args are supposed for the progession_1 method and they mean:
  # arg[0]: the channel this object refers to
  # arg[1]: the probability of chords (0...1) (higher chord rate also means bigger chords)
  # arg[2,3]: the max scale step width up and down
  # arg[4,5]: valid note range lower and upper boundary
  # arg[5,6]: note duration range lower and upper boundary (2^x), 0...6 is legal
  def __init__(self,*args):
    self.general_settings = {'active': 1, 'current_timestamp': -1.0, 'bound_valid_low': 22, 'bound_valid_up': 107, 'channel': 1, 'measure_length': 4}
    # these variables are object-space-variables for the progression method
    self.progression_settings = {'active': 0, 'bound_step_low': 0, 'bound_step_up': 0, 'bound_dur_long': 0, 'bound_dur_short': 0, 'chord_probability': 0, 'prev_timestamp': 0.0, 'next_timestamp': -1.0, 'duration': -1.0, 'note_buff': 0}
    # these variables are object-space-variables for the
    # key_phrase_composition method
    self.kpc_settings = {'active': 0, 'bound_step_low': -10, 'bound_step_up': 10,  'bound_dur_long': 1, 'bound_dur_short': 6, 'chord_probability': 0.1, 'prev_timestamp': 0.0, 'next_timestamp': -1.0, 'duration': -1.0, 'note_buff': 0, 'next_keyphrase': -1, 'keyphrase_target': 0, 'key_dist_low': 1, 'key_dist_up': 3, 'mobility': 10}
    # these variables are object-space-variable's for the metronome method
    self.metronome_settings = {'active': 0, 'prev_timestamp': -1.0, 'total_measures': 0}
    self.bin_subdiv_settings = {'active': 0, 'next_timestamp': -1.0, 'bound_dur_long': 2, 'bound_dur_short': 16, 'subdiv_probability': 0.9, 'notestack': []}
    self.legal = []
    self.set_prog(args)
    self.scale("C", "hm")
```

"./Marc A Modrow source code sample Python.txt"