Csound and Python: An Introductory Tutorial on Algorithmic Score Generation

Marco Gasperini and Giuseppe Ernandez

Conservatorio "Antonio Scontrino" - Trapani
`gasperini@constp.it`
`giuseppeernandez@hotmail.it`

Abstract. The aim of the authors is to present a brief survey on the state of the art and a short tutorial for musicians on the chances given by the interaction between Csound and the Python programming languages regarding the field of algorithmic composition. Some examples using historical models have been developed in particular an environ-ment for generating John Cage's *Imaginary Lanscape n° 5* variants and another to simulate James Tenney'*Four Stochastic Studies* grammar.

Keywords: Python, Algorithmic Score Generation, Electronic Music Teaching, John Cage, James Tenney.

1 Introduction

After some time of relative stability and compatibility between operating systems, platforms, etc. the last ten years have seen several *revolutions* in IT technology and a musician may have lost itself somewhere sometime: from 32 bits to 64, from Csound 5 to Csound 6, from Python 2.7 to Python 3, from Intel MAC to M1 MAC, just to say a few that regard us most. So, let's try to make our point. The Python support in Csound was introduced since version n° 5 and it lets the user interoperate between the two languages by introducing new opcodes that could run Python scripts in the orchestra or in the score (or in both, by the way) of a CSD file. Also the Csound's libraries could be called too with their *classes, functions* and *methods* and all, which could be called from a Python shell to control active instances of Csound. There are many documents available on the internet and elsewhere on these topics but they may arise some issues to the users since they may refer to different versions of the libraries and opcodes, which may be incompatible and/or not cross-platforms. Until the last distribution of Csound 5 there was a document by Michael Gogins[1] , distributed with the binaries, meant to begin the first steps in managing the interactions between the two languages. Due to the aforementioned changes the writing is unfortunately no more of immediate utility, since some functionality described are no longer supported or have now changed. Even other documents available in the internet[2] refer to previous stages of the development and cannot be used directly (by who is not into the development branch) to be introduced to the chances offered by the interactions between the two languages. More recent documents, as

[1] the last version at the authors' disposal is dated 20 december 2009, the one given in the references dates 2006.

[2] For example Cabrera [2], Pinot [3]

the FLOSS manual page [4], require quite complex environments before getting started. An easier approach would be using the Python facilities offered by the CsoundQt IDE (along with its "Scripts" examples) that uses scripting to control a large part of the program's features thanks to its in-built **QuteCsound** object, available directly in the interpreter through objects that belong to the class called **q**. Unfortunately, not all CsoundQt distributions are built with PythonQt support. Since one of our aims is to be as simpler and universal as possible, even more for didactic reasons, we preferred to figure out how to use the Python functionality with Csound by keeping the most conservative and cross-platform system configuration, even by renouncing to some possible improvement given by more developed environments.

2 Algorithmic Score Generation with Python

The authors do not expect to give a complete survey of the Python language nor an introduction to it; for these purposes there is a lot of documentation on the internet and in many publications, starting from the Python's official documentation [5]. We would simply give some examples of its use to generate Csound's scores and, by doing this, to introduce some of the program's features. Moreover, we would remind in the following paper what is strictly necessary to follow our examples. In this article we assume a basic system set-up with *Csound 6, Python 3* or 2.7 (64 bits), and a Csound's IDE, like CsoundQt or Winxound, configured in order to have access to both binaries. Python functionalities are extended by way of *modules* that are not included in the program's core and are loaded by the user according to its needs through an import command; in our examples the following libraries are used:

- **sys** : system-specific parameters and functions;
- **os** : miscellaneous operating system interfaces;
- **random** : generate psudo-random numbers;
- **tkinter**: Python interface to Tcl/Tk. It is a graphical module to build user's inter-faces (GUI).

To check that the modules are in the system a simple script must be given to the interpreter: **import [library]**; if this command raises some errors it means that the module is not installed and must be downloaded. We remind, in order to correctly read the examples given in the article, that code indentation is part of the language syntax and that comments are preceded by a " ## " sign. Of course **print()** commands should be used throughout but they have been omit-ted here to save space.

2.1 An Introductory Tutorial

One of the first benefits we found in using the Python language is its versatility in algorithmic score generation. We will give here a brief introduction on the ways to get it working and then give some more complex examples, based on historical compositions, to give some hints to those who are new to the language. In order to be used for writing a Csound score the Python binary must be executed before the ***CsScore*** end tab, by means of the Code 2.1.1.

Code 2.1.1 Calling the Python interpreter:

```
<CsScore bin="python">
[...]
</CsScore>
```

The Python code must be written in the space between the opening and the ending ***CsScore*** tabs in order to be fed to the interpreter to generate the Csound score. A statement which is always needed is the one for loading system's functions through the **import** command. The rest of the script shown in code 2.1.2 opens a temporary file and assigns it to a **sco** variable, on which a **write** method will later be applied, creates a string with variables fields and a **for** loop that is the actual algorithm: generate a 12 notes chromatic scale, starting from C4, each sound a second long and whose amplitudes increase by 1000 at each step. The resulting score is temporary and is deleted at the end of the run.

Code 2.1.2 Basic score generation algorithm[3]*:*

```
import sys                       # imports system's functions
sco = open(sys.argv[1], "w")     # open temporary file to write
i_statem = "i 1 {} 1 {} {} \n"   # notes' string for instr 1
for i in range(0,12):            # score gen loop
    # write strings
   sco.write(i_statem.format(i,1000+i*1000,8+i/100.))
```

A basic instrument to test the previous Python script is given in code 2.1.3: it needs five parametric fields from the score. The model of a single note statement for this instrument is contained in the **i˙statement** variable in code 2.1.2, a string composed of two constant fields (p1 and p3; instrument identifier and sound duration) and three variable ones, p2, p3 and p5 (start time, amplitude and pitch).

Code 2.1.3 A simple Csound instrument:

```
instr 1
    aEnv cosseg 0, 0.1, 1, p3-0.2, 1, 0.1, 0
    aSig poscil3 p4, cpspch(p5)
    out aSig*aEnv
endin
```

The actual score generation process, consisting in applying the **write** method to the **sco** object, is contained in a **for** loop which repeats itself twelve times with an increasing (from 0 to 11) i index. When the interpreter ends the loop, it gives Csound the temporary score to play the **instrument** 1 in real time or writing audio to a file. It may be of some use to read the generated score in order to verify the results and to eventually adjust them manually. In order to do so we must modify the script as in code 2.1.4 (with some other minor adjustements) to generate a **test.sco** file. Note the " += " form to append (or accumulate) the results to any object.

Code 2.1.4 An improved Python script:

```
<CsScore bin="python">
import sys              # imports system's functions
```

[3] The code is adapted from Lazzarini et al.[6]

```
amp = 1000            # initial amplitude (0db = 32768!)
oct = 8            # starting octave
out_file = \test.sco"            # output file
sco_tmp = open(sys.argv[1], 'w+') # open a temporary file
sco = open(out_file, 'w+')     # open SCO file
score = ""     # score
i_statem = "i 1 {} 1 {} {} \n"      # notes' string for instr 1
for i in range(0,12):     # score gen loop
    # append to 'score'
    score += i_statem.format(i,amp+i*amp,oct+i/100.)
sco_tmp.write(score)
sco.write(score)
sco.close
</CsScore>
```

The modified score may be then included as a score macro in a CSD file containing the orchestra defined in code 2.1.3 through an **##include "test.sco"** command. As the algorithm grows complex it may be of some use to have it contained in an external PY file, while keeping the musically relevant variables in the CSD. The script will be placed inside a **test˙lib.py** file (in the same directory as the CSD) as a *function*: Python's functions are objects defined by the user through a **def name([arg],...)**: statement, whose body is not executed until they are called and may have arguments.

Code 2.1.5 The score generating algorithm as a function:

```
import sys     # imports system's functions
[...] # and all others needed
# define function's name and arguments
def score_gen(amp,oct,outFile):
    sco_tmp = open(sys.argv[1], 'w+')
    [...]
```

The **score˙gen** function has three arguments that must be passed by the calling in-stance: linear amplitude, base octave and output file name. We have to keep track of the indentation level of the code that must be kept in order to avoid errors. The CSD file thus will just contain the **import** statement to load the compositional library and the function's call with its arguments, as in Code 2.1.6.

Code 2.1.6 The "definitive" CSD score script:

```
<CsScore bin="python">
from test_lib import *  # import all functions from test_lib.py
amp = 1000 # initial amplitude (0db = 32768!)
oct = 8 # starting octave
out_file = "test.sco" # output file
# Call the score_gen function from the test_lib library
score_gen(amp,oct,out_file)
</CsScore>
```

In some Mac OS system the Python's current working directory may not be the same as the Csound's one so it may not find the **test˙lib.py** file; to avoid this error a full path to the external library must be given, as in code 2.1.7.

Code 2.1.7 Mac OS script:

```
<CsScore bin="python /Volumes/full/path/to/library/">
from test_lib import *
[...]
```

It would in some case be useful to have a graphic interface to insert the musically relevant variables, instead of a textual one. In these cases we can resort to the **tkinter** module; a typical way of doing this is inserting the following command line in the **test˙lib.py** file: **import tkinter as tk** (the module is imported with a sort of nick-name). Then a global **amp˙value** variable and the graphic module (assigned to the object **root**) are initialized. There are two new function definitions: (1) **startGUI**: opens a TK window to draw widgets into; (2) **renderAll**: reads values (in the example the linear amplitude value only) from the interface and pass them to the **score˙gen** function (note the default values assigned to the latter in the **def** statement).

Code 2.1.8 score˙gen with GUI:

```
amp_value = "Null"
root = tk.Tk()  # oggetto tk
def startGUI():
    root.mainloop()
def score_gen(amp,oct=8,outFile="test.sco")
[...]
def renderAll():
    global amp_value
    amp_value = int(amp_entry.get())
    score_gen(amp_value)
    root.quit()
```

The actual basic interface consists of an entry field and a button to trigger the **renderAll** function.

Code 2.1.9 Tkinter GUI:

```
amp_entry = tk.Entry(root)
amp_entry.grid(row = 0, column = 0)
exit_button = tk.Button(root,text= "Render", command = renderAll)
exit_button.grid(row = 1, column = 0)
```

Other widgets are available from Tkinter to improve the appearance and usability: we just wanted to give some hints from where to start.

2.2 Imaginary Landscape n°5 (J. Cage)

As it is known from the score [7] and at least one detailed formal analysis [8] *Imaginary Landscape n° 5* (1952), by John Cage, one of the first tape music pieces in history, was composed combining deterministic and random procedures. It is a musical recordings collage with a 5 parts formal structure, each part being composed of 5 sections ("the rhythmic structure is 5x5"); the duration's ratios are ruled by a numeric sequence: 1-3-4-5-2. Each one of the 25 sections is characterized by a certain polyphonic degree (called "density" by

Cage), ranging from 1 to 8. Sounds and pauses duration (besides several other parameters) are randomly chosen (albeit not completely). We have no space here to look deeper in the compositional technique, that has been analyzed in detail in our Electronic Music course in Trapani's Music Conservatory bringing forth the formalization of an algorithm called *IL5* to generate new variants from almost the same principles.

Code 2.2.1 IL5 internal loop:

```
for i in range(0,5):                    # SEQUENCES
    seq_ratio = Tseq_ratio[part][i]
    seq_dur = seq_ratio*base_dur*part_ratio
    dur_mem = 0 # time counter
    seq_int = [0]*8 # 8 voices interruptors
    seq_den = D_seq[part][i]         # sequence density
    seq_voc = sample(voices,seq_den)    # active voices
for i in range(0,seq_den):          # write voices' current status
    voice = seq_voc[i]
    seq_int[voice-1] = 1
for i in range(0,8):                # VOICES 1-8
    voice_indx = i
    voice_gen(voice_indx,dur_mem,seq_dur,seq_int)
```

In code 2.1.10 the internal loop to generate sounds and pauses in each sequence can be seen: the **seq˙ratio** object reads from a predefined 5x5 matrix the actual time ratio for each sequence, then the actual voices to play are selected (randomly chosen through the **sample** method of the **random** module) according to the polyphonic degree determined by another external 5x5 matrix; in the first loop the active status for sounding voices is written, while in the second the **voice˙gen** function is called 8 times to actually generate the active voices content (inactive ones will internally be skipped by reading their status in the **seq˙int** list).

2.3 Four Stochastic Studies, 1962 (J. Tenney)

Formally less defined than *Imaginary Landscape n° 5*, these studies[4] by James Tenney are probably the first composition in history to use algorithmic procedures to digitally synthesize sound structures. The focus of this composition was based on the *clang* concept [9][10], derived from *Gestalt* psychology: what are the mechanisms that in our perception unite or separate a stream of sounds? One factor is of course timbre, others are either pitch, rhythmic or loudness proximity, each of which has its own *grammar* in this composition musical model: we give, in code 2.3.1, the instruction with which timbre unity is obtained in each clang.

Code 2.3.1 Timbre selection for clangs:

[4] This is how Tenney remember this composition in his 1969 writing [9]. However the official Youtube channel (and no other sound sources are available to the authors since in the Selected Works 1961-1969 New World Records CD it is not included) contains only a 1962 computer piece titled Five Stochastic Studies that seems to be the one referenced in the later writing.

```
for i in range(0,sections): # SECTIONS
    section=i
    clangCount=randint(1,5) # clang per section
    waveStack=sample(timbreFts,clangCount)
```

In code 2.3.1 we can see how each section in the piece is randomly composed of up to five clangs (a formal unity composed of a certain number of single sounds) by way of the **randint** method from the **random** module; then an equal number of different wave-forms are randomly selected (by way of the **sample** method) from a **timbreFts** list and later on each one of these timbre will be assigned to a single clang through another nested **for** loop.

3 Conclusion

Using Python as a scripting tool to generate Csound's scores in algorithmic composition courses proved to be, after initial assessment, reliable on all the systems tested (Mac OS, Windows and Linux) and a very fluent language to rapidly develop musical patterns. Several historical formal models have been repeated apart from those described above, particularly from the Tenney catalog, compositions like *Analogue ## 1* (1961), *Dialogue* (1963), *Phases* (*for Edgard Varèse*) (1963) and *Ergodos II* (1964). Other ways of using the two languages have not been thoroughly explored for the reasons given in the Introduction, but it seems of great interest the possibility to manage Csound instances through Python for live-coding with the **ctcsound** module that replaced the previous **csnd** and **csnd6**, and will be the next step of employ and research.

References

1. Gogins, M.: "A Csound Tutorial" site,
   `http://dream.cs.bath.ac.uk/Csound-archive/OldReleases/csound5/csound5.07/tutorial_5.07.pdf`
2. Cabrera, A.: Using Python Inside Csound. Csound Journal, Issue 6 (2007)
3. Pinot, F.: Real-time Coding Using the Python API: Score Events. Csound Journal, Issue 14 (2011)
4. Csound FLOSS manual site,
   `https://flossmanual.csound.com/csound-and-other-programming-languages/python-and-csound`
5. Python documentation site,
   `https://docs.python.org/3.8/index.html`
6. Lazzarini, V. et al. Csound. A Sound and Music Computing System. Springer International Pub-lishing, Switzerland, 2016
7. Cage, J.: Imaginary Landscape n. 5. Henmar Press, New York, 1961
8. Di Scipio, A.: Una macchina organizzativa indifferente: "Imaginary Landscape n. 5" di John Cage. In Sonus – Materiali per la musica moderna e contemporanea, fascicolo 19 Agosto 1999.
9. Tenney, J.: Computer Music Experiences 1961-64. Electronic Music Reports ##1, Utrecht: Insti-tute of Sonology, 1969.
10. Tenney, J.: META/HODOS and META Meta/Hodos, Hanover (NH) Frog Peak Music, 1988.