# Aleatora I: Composing with Streams

Ian Clester
ijc@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, United States

## ABSTRACT

Computers are a powerful medium for musical compositions: with capabilities above and beyond the playback technologies of the past, they can play *computational compositions* that may not be fully determined until they are played. The accessibility of such compositions is currently limited: many existing languages are more oriented towards synthesis than composition, there is no convenient platform for distribution, and tools for non-programmers are lacking.

In this paper, I explore the first issue, compositional language design, by building a compositional framework for Python around the abstraction of lazy, effectful streams. Such streams provide a natural fit for computational compositions, as they capture the notions of sequential data computed on-demand which may come to an end. Thus, the streams support *parallel* composition (as in summing two signals together), *sequential* composition (as in concatenating two sections of a composition together), and *functional* composition (as in one stream pulling values from another). The framework includes tools for inspecting the structure of stream-based compositions.

## 1 INTRODUCTION

This semester, I began work on the first branch of a three-prong project. That project is **Aleatora**, a platform for creating and distributing *computational compositions*. Computational compositions are compositions expressed computationally, as programs, to be executed by a human or a machine. As the name of the project suggests, these compositions may be aleatoric in nature: deriving their final form partly from chance, or from some external state such as the weather, the stock market, what's playing on FM radio, or tweets.

Today, there are many obstacles to creating and distributing computational compositions. I see three main issues:

(1) Existing audio programming languages are oriented more towards instrument design and audio synthesis than musical composition.
(2) Distributing computational compositions is difficult for the composer and inconvenient for the listener. For the composer, the process requires packaging their code, potentially cross-compiling, and building common interface elements from scratch (play, pause, seek). For the listener, it often involves installing new software, running untrusted executables, or (in the best case) listening to WebAudio scripts with no consistent interface. There is no common platform for distributing computational compositions comparable to those available for rendered compositions (SoundCloud, Spotify, YouTube, Bandcamp, Freesound, ...).
(3) Creating computational compositions requires mastery of a programming language, even if the composer only needs a small subset of their functionality (e.g., specifying a weighted, directed graph of song sections to walk through randomly).

Correspondingly, over the course of this project I hope to address the following three research questions:

(1) What are the compositional implications of different computational abstractions? That is, what are the affordances provided by different means of mapping musical ideas into a computer program?
(2) How can computational compositions be distributed efficiently (both in terms of performance and usability), so as to lower the barrier to listening? (This is challenging because a single computational composition may have many possible outputs, or outputs that aren't finite—so distributing a static rendering of the composition cannot suffice in general.)
(3) How can musicians who do not code take advantage of the expressive possibilities offered by computational compositions?

In this paper, I begin my investigation of the first issue and research question, the design of a language for composition, by building a compositional framework for a general-purpose programming language (Python) using an unusual abstraction: lazy, effectful streams.

## 2 RELATED WORK

The most relevant related work comes from other audio programming languages. Most languages, such as Max/MSP [1], Pure Data [8], and SuperCollider [7] (and the many libraries built on top of it, including Sonic Pi and Overtone) completely separate synthesis and control. These systems divide them across two separate languages, with synthesis adopting a signal processing/circuit building/dataflow paradigm, and control adopting either the same paradigm (as in Max/PD) or an imperative style (as in SuperCollider and various libraries built on it).

ChucK [9] is an interesting case: as it is strongly-timed (down to the sample level, if desired), the "control" code can itself perform synthesis. However, unit generators are still given a special place in the language semantics and syntax, and the language encourages a mix of signal processing-style audio graph construction (facilitated by the ChucK operator =>) for synthesis and imperative code for control. That said, this work shares with ChucK the notion of programmable control rates (and gratuitous use of operator overloading).

Nyquist [5] is this work's closest relative conceptually (if not syntactically). This work shares with Nyquist several crucial features:

- An interactive environment based on a general-purpose language with a REPL (Lisp for Nyquist, Python for Aleatora)

- No distinction between the "score" and "orchestra", or "control" and "synthesis"
- Sounds as first-class values, with operators to combine and manipulate them directly
- Streams ("signals" in Nyquist) as lazy linked-lists, which may be finite or infinite
- Support for arbitrarily mixed sample rates

These features are sufficiently rare that they are already enough to ensure that Nyquist is this work's closest intellectual relative. But this work also differs from Nyquist in a few essential ways:

- It can generate and play audio in real-time. This is in part due to the use of Python, which collects most garbage as soon as possible via reference counting, mitigating the impact of slow collection pauses mentioned by Dannenberg.[4]
- It does not memoize streams by default, so the computation may take a different path and yield a different sequence the next time through (due to side-effects).
- It handles external events (e.g. MIDI messages, UI interactions, tweets) by conceptualizing them as ("impure" or side-effectful) streams of data, treated the same as any other stream.
- It provides convenient syntax for common operations by overloading operators in the host language. (This is less of an option for Nyquist, as syntactic uniformity is typical of Lisps.)

And this work does not deal with some issues that were important to Nyquist, such as block computation, logical start and stop times, and behavioral abstraction for context-dependent transformations.

Looking further afield, to computer science and programming languages generally, there is important related work on the stream abstraction. A classic reference is the "Streams" section of Abelson and Sussman's *Structure and Interpretation of Computer Programs* [2]. A more refined and complete realization of the same ideas exists in SRFI-41 [3], a common extension to the Scheme programming language. In both SICP and SRFI-41, streams are memoized, and mixing streams with side effects is discouraged. This is similar to Haskell's native List type, which is naturally lazy and prohibits side effects. A more recent development, Haskell's Pipes library [6] offers a streaming abstraction that elegantly combines laziness and effects. Pipes makes extensive use of Haskell's robust static type system and offers complexity well beyond what I consider here, but the core abstraction is similar to that used in this work.

# 3 DESIGN AND IMPLEMENTATION

## 3.1 Form Factor

The first design question to answer was what form this compositional language should take. I opted to build on top of an existing general-purpose programming language, to further the goals of accessibility (ideally the user does not have to learn a whole new language, and picking a popular language means many potential users already have knowledge useful for my system) and power (building on a general-purpose language means the user need not resort to external tools as soon as they want to do something I did not expect, and it saves me from reinventing the wheel).

With that decided, the remaining question was what language to choose as the base. For the sake of accessibility, I narrowed the options down to Python (which is popular and has an excellent ecosystem) and JavaScript (which runs on anything with a web browser). Ultimately, I chose Python due to its support for operator overloading, ease of interacting with the outside world, and personal preference. However, the choice was close, and I may revisit JavaScript in the future when I consider the problem of distributing computational compositions.

## 3.2 Basic Concepts

The basic concept in the library is the stream. A stream is a sequence of values generated on-demand. When a programmer asks for the next element of a stream, the element is computed, and the programmer is presented with that element and the rest of the stream (as yet unevaluated). In other words, after each element is consumed, the computation is suspended until the next element is consumed. Streams operate by returning a tuple of (next element, rest of stream). Streams may terminate by returning an instance of Return, which indicates that there are no further values. The Return instance may contain one final value.[1] The Stream type is a function that returns `(any, Stream)` or `Return(any)`.

Note that, unlike Python's generators (which also lazily generate sequences), streams are immutable[2] and can be played back from any earlier point, assuming the programmer saved a reference to it. Since streams are not memoized by default, the second playback may produce a different sequence of results than the first.

Unfortunately, this distinction means that Python's generators, which are easy to define using `yield`, cannot implement most streams. Instead, basic streams tend to have a slightly convoluted definition involving chains of closures. For example, defining an infinite stream of natural numbers as a generator could look like:

```
def count(start=0):
    while True:
        yield start
        start += 1
```

As straight-line imperative code, this is pretty readable, and it's obvious at a glance that this generator goes on forever. By contrast, the actual stream definition is the concise but potentially more confusing:

```
@raw_stream
def count(start=0):
    return lambda: (start, count(start+1))
```

In this example, the function `count()` returns a function, which, when called, returns the current count and the next function (the result of `count(start+1)`. Essentially, this is a chain of closures, with `start` incremented in each successive closure, and with each closure creating the next one in the series when called.[3] The decorator (`@raw_stream`) wraps this function in the `Stream` class, which

[1] The Return value is important for implementing certain operations such as a lazy `split`. In terms of Pipes, this is essential for implementing the monadic bind (`>>=`)
[2] However, the computation represented by a stream may rely on mutable data.
[3] Each closure is also freed as soon as iteration proceeds to the next one due to reference-counting, unless the programmer keeps a reference around.

enables the programmer to iterate over streams like any other sequence and defines some important operators discussed in the next section.

## 3.3 Composing Streams

Often, the programmer will not need to define basic ("raw") streams directly, because they can instead build them by combining simpler streams. The most fundamental stream operators are concatenation (») and mixing (+). These correspond aurally to the basic notions of playing one thing after another, and playing two things at the same time, respectively.

`a » b` constructs a stream that will yield values from `a` until it runs out, and then will yield values from `b` until it runs out (and then `a » b` itself will run out). This operator is made possible by the fact that streams (unlike signals in many synthesis languages) can "run out": the language supports the concept of ending. If `a` is an infinite stream, `a » b` will never get to `b`. An performance important detail is that » is associative, and any amount of chaining, in any order (e.g. `a » (b » c)` or `(a » b) » c`), will flatten to the same internal representation and incur the same amount of overhead. Also, once the concatenation operator gets down to the final stream (that is, when `a » b` exhausts `a` and reaches `b`), it simply returns `b` as "the rest of the stream", eliminating the operator overhead (and simplifying the audio graph).

`a + b` constructs a stream that will yield values from `a` summed with values from `b`, element-wise. `a + b` will keep producing values until *both* `a` and `b` have been exhausted. If, for example, `a` finishes first, `a + b` will continue as if it were followed by silence (infinite zeros), returning the remainder of `b`. As with », + is associative, and it will flatten nested additions (`a + (b + c)` = `(a + b) + c` = `a + b + c`) to reduce overhead. Also like », + disposes of exhausted streams and, once it gets down to a single remaining stream, simply returns it as "the rest of the stream" to eliminate the extra layer of indirection.

Aside from + and », streams support the common arithmetic operators `-` (binary and unary), `*`, `/`, `//`, `%`, `pow`, and `abs`. All of these operate lazily in an element-wise way. The binary operators (aside from ») allow the second operand to be another stream or a constant. Streams also offer the `.map()` method, which returns a new stream with a function applied to every element.

Lastly, streams support slicing. In Python, sequences can be sliced with the syntax `seq[start:stop:step]`, where `start`, `stop`, and `:step` (including the colon) are optional.[4] With this syntax, the programmer may slice elements off the start or end of a sequence, or skip over elements at regular intervals. Streams also support this operation. Floating point values of start, stop, or step (illegal for built-in sequences) are interpreted as seconds and automatically converted to samples. This syntax is especially convenient for cutting infinite streams down to size; for instance, one can create a stream of `a` followed by `b` with a one-second gap between via `a » silence[:1.0] » b`, where `silence` is an infinite stream of zeroes.

Many streams can be built out of other streams using these operators, without the need to define a raw stream as a chain of functions. For example, the built-in stream `osc` is defined in terms of `count` and `.map()`:

```
@stream
def osc(freq):
    return count().map(lambda t:
        sin(2*pi*t*freq/SAMPLE_RATE))
```

The last way to compose streams is by passing streams as arguments to other streams. For example, the function `fm_osc()` takes a stream of frequencies as an argument, and returns a stream that uses the input stream to modulate a sine wave. So, for example, `fm_osc(const(440))` will yield the same values as `osc(440)`, while `fm_osc(440 + osc(0.2)*20)` will slowly oscillate between 420 and 460 Hz, `fm_osc(440 + osc(440))` will produce something more spectrally complex, and `fm_osc(440 + 440*fm_osc(440 + osc(0.2)*20))` will go wild.

The definition of `fm_osc` (as a raw stream) is given below. Note how it constructs a chain of closures and iterates over its input stream through explicit calls (including handling the termination condition).

```
@raw_stream
def fm_osc(freq_stream, phase=0):
    def closure():
        result = freq_stream()
        if isinstance(result, Return):
            return result
        freq, next_stream = result
        return (sin(phase),
                fm_osc(next_stream,
                       phase + 2*pi*freq/SAMPLE_RATE))
    return closure
```

`fm_osc` iterates through its input stream at the same rate it outputs samples, but this is not true of all streams. For example, `glide(freq_stream, hold_time, transition_time)` also takes a stream of frequencies, but it holds each frequency for `hold_time` and then interpolates between frequencies for `transition_time`. Therefore, it only needs to ask for the next element of `freq_stream` once every `hold_time + transition_time`. If `hold_time` and `transition_time` are both one second, then whatever code is executed by `freq_stream`, it will only be executed once every two seconds—in a sense, `glide` defines its own "control rate" by how often it pulls values from (and thereby resumes the computation of) `freq_stream`.

`resample(stream, advance_stream)` is an even more interesting in this regard. This function returns a stream that resamples `stream` at a variable rate, determined by `advance_stream`. Each time `resample` is asked for a sample, it asks `advance_stream` for a value representing how many (fractional) samples to advance forwards in `stream`. Thus, `resample` pull values from (and compute) `stream` at a variable rate dictated by `advance_stream`. `resample(stream, const(0.5))` runs the computation of `stream` at half the "usual rate", `resample(stream, const(2))` runs it twice as fast, and `resample(stream, osc(0.1)+1)` runs it at a rate that oscillates between 0x and 2x speed over time.

[4] `start` defaults to 0, the start of the sequence, `stop` defaults to the end of the sequence, and `step` defaults to 1

### 3.4 Workflows

There are plenty of ways to compose streams, but all of this composition is for naught without a way to hear the results. The `audio` module provides functions for this purpose. `audio.run()` plays back a stream non-interactively; the function blocks until the stream is finished. `audio.play()` plays back a stream interactively, without blocking—this makes it suitable for use in the REPL. At any time, the currently playing stream can be replaced by calling `play()` again, and calling it without arguments will the current stream with `silence`. Volume can be adjusted during playback (often important during demos) via `audio.volume()`. `play()` and `run()` both play streams back in real-time.

For rendering a composition to a file, there's `wav.save()`. `wav.load()` and `wav.load_mono()` go the other direction, loading audio files into buffers than can then be converted into streams via `list_to_stream()`.

Real-time playback is useful when working on a composition, but it can stutter and stall as compositions grow in complexity. Python is reasonably fast, especially with PyPy, but it is a dynamic, interpreted language. Also, all computation is done one stream element at a time. Rates may vary, but streams do not operate in blocks, further limiting performance.

That said, there are strategies to work around these issues and improve performance. Two easy options are memoization and freezing. By default, all streams are unmemoized so they can produce different results on playback (streams may be aleatoric). But if a stream is known to be deterministic, or if the composer wishes to preserve a particular path from later usage, they can use the `memoize()` or `freeze()` functions. `memoize()` returns a memoized stream which will store results upon evaluation and return the stored values upon replay (rather than recomputing them). `freeze()` evaluates the entire stream immediately and produces a new, frozen stream consisting of the results. Note that `freeze()` will hang forever if called on an infinite stream, while `memoize()` will not.

By applying these tools judiciously and using the included `prof` module to profile streams embedded in compositions, the composer can better maintain real-time performance as their composition's complexity increases. Maintaining real-time performance is particularly valuable because of Python's interactive REPL, which allows the user to construct and playback different sounds at runtime; a workflow that is even more convenient with editor integration for evaluating lines of code from a file.

### 3.5 Inspection and Introspection

One of the benefits of visual languages like Max/MSP and Pure Data is the ability to see the structure of a patch directly, rather than having to parse it out from source code. This ability is even more crucial in this framework, as the synthesis graph can change dynamically over time as streams reach their ends. To aid understanding and debugging, I have emphasized *inspectability* in the design of my framework. Streams can be inspected to see their parameters, which may include other streams (as in concatenation, mixing, mapping, and so on).

Inspection can happen in a few different ways. Simply calling `str()` on a stream will provide a formatted string; for example, `str(osc(440)[:1.0])` returns the string `'osc(freq=440)[:44100:]'`. If we inspect the rest of the stream after the first element, we get that `str(osc(440)[:1.0]()[1])` returns `'osc(freq=440)[:44099:]'`; the value in the slice has changed because we have consumed one element (so the stop-point of the slice is one sample closer).

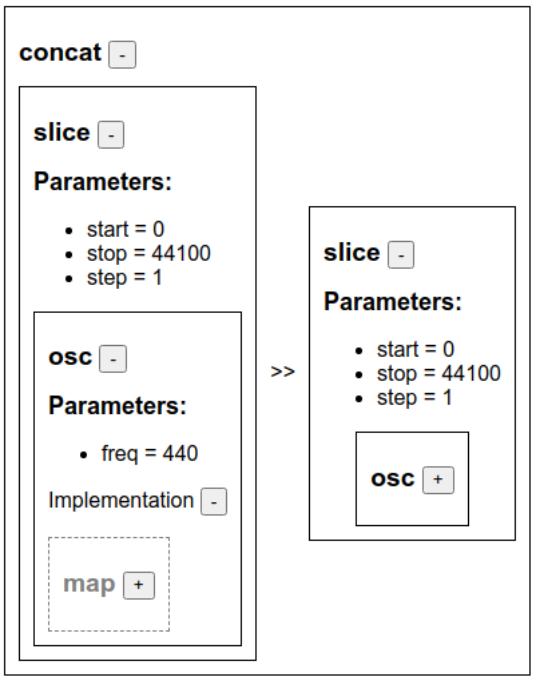


**Figure 1: Inspection and partial expansion of the composition `osc(440)[:1.0] » osc(660)[:1.0]`.**

Calling `.inspect()` on a stream will provide more detailed information as a dictionary. The function `graph.inspect()` does this recursively, inspecting all substreams while avoiding cycles, and generates an expandable HTML page that depicts the stream graph (as in 1). This page is served locally via a Python webserver, and it can be embedded inside a text editor. Streams built out of other streams (like `osc`) can be further expanded to show their implementation.

What's more, inspection is mostly automatic. Earlier versions required inspection functions to be tediously defined for each stream. Using Python's introspection capabilities, as in the standard library's `inspect` module, we can instead generate a fitting function automatically using the function signature. Thus, inspection comes for "free" (via the `@raw_stream` and `@stream` function decorators), with only special Stream subclasses[5] requiring manual implementations.

## 4 DISCUSSION

In this paper, I have presented a framework for musical composition, implemented as a Python library, built on the abstraction of lazy, effectful streams. In this section, I would like to step back and consider the compositional implications of some of these features.

[5] e.g. `ConcatStream` and `MixStream`, which are used to implement » and + and have variable numbers of child streams.

First, this framework eschews the division between score and orchestra, between control and synthesis. This choice is not because those divisions are useless; indeed, they are often useful. Rather, it's because the divisions should be chosen by the composer. If a composer's musical models do not mesh with the language designer's, the composer should not have to awkwardly work in two disparate languages across the divide. The composer, as programmer, should be free to devise the abstractions that best represent their composition, and the language should support that. My framework offers this capability by starting at the sample level and working up. The composer can go back all the way down to the sample level if they want, without leaving the language, and they can build their own preferred abstractions on top of it. My code provides some of examples of where boundaries could be drawn: functions like `glide`, `basic_sequencer`, `interp`, `events_to_stream`, and `beat` show how streams of higher-level information can be converted down to sample-level streams. But if those aren't useful for someone else using this framework, they can write their own functions and still benefit from much of the library.

Second, this framework uses an abstraction that is fairly unusual in the realm of synthesis; Nyquist is the only project I am aware that does something similar. Streams are essentially an augmentation of the dataflow paradigm which is very common in synthesis: the augmentation is that flows may cease. Viewed this way, this begs the question: is the feature worth its weight? Do the benefits justify the added conceptual complexity? I believe they do. It is natural to think of music sequentially, concatenatively: "first this happens, and then that happens", "there's this section, and then the bridge". Allowing streams to have endings—and thus to be occur in sequence—captures this basic notion and enshrines it in a core abstraction. The consequence is that the low-level graph is dynamic, naturally evolving over the course of a composition as streams come to their ends, reflecting the way the music changes over time as its past falls away.

A follow-up question: is arbitrary replayability worth its weight? If streams were mutable, only going one way, they could be implemented using Python generators, avoiding convoluted chains of closures, and performance would likely improve. This question is a bit trickier; as of yet, I have only begun to experiment with explicit replaying and aleatoric pathways in my own compositions. However, I believe this feature (and the true, chain-of-thunks streams) has some side benefits for the framework design. In particular, it means that streams can permanently redirect to another stream, with zero overhead, by simply returning that other stream as "the rest of the stream". This makes the implementation of + and especially » more elegant *and* more efficient. Additionally, it saves us from the interface inconsistency that would likely result from a generator-based framework, where some functions would take generators while others, like `cycle`, would need to take "generator-makers" (functions returning generators, which are not themselves generators) in order to restart the generator from the beginning.

## 5 NEXT STEPS

First, I would like to build on the web-based inspector to make a more complete composition companion. I plan to make the interaction two-way by adding controls from the web page back to the Python process (e.g. to advance the composition), and I want to integrate the inspector with the profiler. Also, there is often compositional data that is easier to input graphically rather than textually (for example, Max's `function` object), and this framework could benefit from a way to input such data (perhaps via the web interface) and interact with it in code.

Next, I want to ensure this is usable. Part of that is integration: I want people to use what I have built, and an important (and oft overlooked) part of that is integrating with lots of other things that people already use. To that end, I plan to build out some basic functionality for interacting with MIDI (hardware and files), VSTs, and networks/web APIs. The other part is user testing: I focused on building out a platform this semester, but getting feedback from other musicians and programmers will be essential to building something they can actually use.

Lastly, looking further ahead, the two remaining prongs—distribution and tools for non-programmers—await. At this early stage, it is not clear if it will make sense to take this framework, or at least this implementation of it, along for the ride. Particularly for distribution, it may turn out that JavaScript is the better choice after all, and I may port some of this there. Or it might turn out that some existing language is sufficiently compelling that I switch to it for the remaining work. Regardless, I am confident what I have learned from this work, about design and composition, will live on.

## REFERENCES


[1] [n.d.]. Max/MSP. https://cycling74.com/products/max

[2] Harold Abelson and Gerald Jay Sussman. 1996. 3.5 Streams. In *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts. https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html

[3] Philip L. Bewig. 2007. SRFI-41: Streams. https://srfi.schemers.org/srfi-41/srfi-41.html

[4] Roger B. Dannenberg. 1997. The Implementation of Nyquist, A Sound Synthesis Language. *Computer Music Journal* 21, 3 (1997), 71–82. https://doi.org/10.2307/3681015 Publisher: The MIT Press.

[5] Roger B. Dannenberg. 1997. Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. *Computer Music Journal* 21, 3 (1997), 50–60. https://doi.org/10.2307/3681013 Publisher: The MIT Press.

[6] Gabriel Gonzalez. 2020. pipes: Compositional pipelines. https://hackage.haskell.org/package/pipes.

[7] James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal* 26 (Dec. 2002), 61–68. https://doi.org/10.1162/014892602320991383

[8] Miller S. Puckette. [n.d.]. Pure Data. https://puredata.info/

[9] Ge Wang and Perry R. Cook. 2003. ChucK: A Concurrent, On-the-fly, Audio Programming Language. In *Proceedings of the International Computer Music Conference (ICMC)*. Singapore.