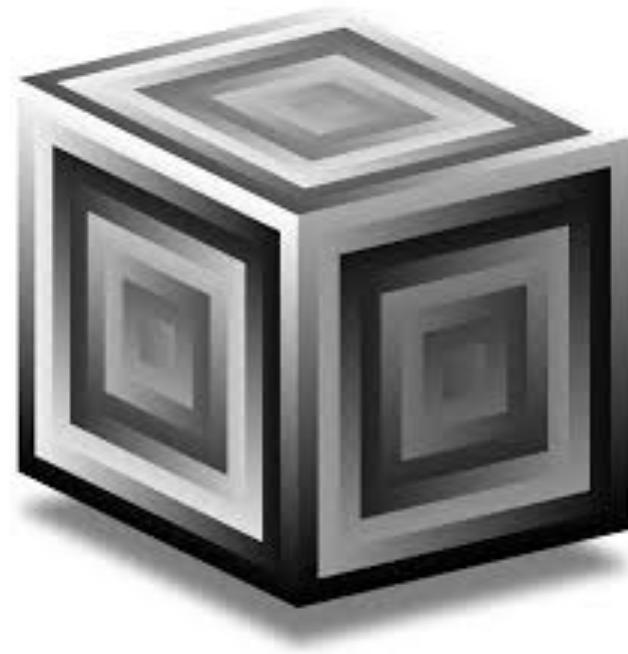


Streams, Events, Patterns

# Topics Addressed

- Streams
- Routines
- Music with Routines
- Patterns
- Events
- Pbind
- Musical Examples



# Streams

- Every object in SuperCollider is a **stream**.
  - A stream is a sequence of values that are obtained by successive calls to the method **.next**.
  - All streams respond to two methods: **.next** and **.reset**
    - The **.next** method yields the next value in the sequence.
    - The **.reset** method resets the sequence to its original starting value.
- Streams can be finite or infinite in length. When a finite stream reaches the end of its values, it returns **nil**.

```
// some examples of streams
7.next; // yields the number seven
"hi".next; // yields the string "hi"
a = [1, 2, 3];
a.next; // yields the entire array, not individual elements
f = {10.rand};
f.next; // yields the function, not evaluation of the function
```

# Routines

- Functions are seemingly ill-suited for musical events because the body of a function is evaluated entirely upon the call to it. If a function, let's say, described more than one musical event, we would have no means for controlling when those events were supposed to occur.
- Routines offer a means of starting and stopping functions, making them more suited for controlling musical time/events.
- Routines are a special type of stream.
- Routines take a function as an argument and respond to several key methods *within* the body of that function.
  - With a call to **.next** on a routine, the function begins evaluation until it finds the method **.yield** *within* the function. The function is then suspended and the object of the **.yield** method is returned.
  - With subsequent calls to **.next** on the routine, execution resumes where the function was last suspended and then is suspended again upon encountering the next yield method. If the function completes, then the routine is stopped and can be restarted from the beginning if the routine is called with the **.reset** method.

# Simple Routines

```
(  
  r = Routine({  
    1.yield; ←  
    2.yield; ←  
    3.yield; ←  
  });  
)  
  
r.next;  
r.reset;
```

1 <sup>st</sup> call to next;
2 <sup>nd</sup> call to next;
3 <sup>rd</sup> call to next;
> 4 <sup>th</sup> call; Returns nil

The `yield` method pauses execution of the function passed to the routine and returns the receiver of the `yield` method (generally a number but could be any object). Subsequent calls to `next` resume execution until the next `yield`/end of function.

```
(  
  r = Routine({  
    3.do({arg i; i.yield});  
  });  
)  
  
r.next;  
r.reset;
```

The `yield` method can go inside of loops. The example above is identical to the one on the left. The method `reset` will start the routine over at the beginning of the function.

# Counter Example

Write a short routine that counts up from the number one by one for each call to `.next`.

```
(  
r = Routine({  
  var count = 0;  
  while({true}, {  
    count = count + 1;  
    count.yield;  
  });  
});  
)
```

# Scheduling Routines

- Routines provides the tools for a lazy evaluation of a function that can pause/start with the methods **.yield**, **.next** and **.reset**.
- The **.next** method controls the progression within the body of the function. How can we control the timing of when the **.next** method is called?
- Answer: clocks!
  - SuperCollider provides several different clocks that can be used to schedule different routines, streams, and other objects.
    - The class `SystemClock` is the main clock that the computer runs at.
    - The class `TempoClock` can be used to control musical events in terms of beats/tempo
  - We will use `SystemClock` initially as the main way of controlling time.

# System Clock

- The important class method for system clock is **.sched**
  - 1<sup>st</sup> argument: an offset to when the object should be scheduled
  - 2<sup>nd</sup> argument: an object
- The scheduling with the system clock works such that the **.awake** method of the object in the 2<sup>nd</sup> argument will be called after the initial offset of time in the 1<sup>st</sup> argument. For functions, the **.awake** method is equivalent to **.value**, and for routines it is equivalent to **.next**.
  - If the value returned by the **.awake** method on the object is a number, then the object will be rescheduled for another call to **.awake** as an offset based on that number.
  - If the value returned by the **.awake** method on the object is **nil**, then the object will not be rescheduled. Other non-numbers will also fail to reschedule the object.



# Simple System Clock Examples

```
SystemClock.sched(2.0, {  
  "2.0 seconds later".postln;  
  nil;  
});
```

The **.awake** method is called on the function after two seconds. The function is not rescheduled because **nil** was returned.

```
SystemClock.sched(0.0, {  
  arg time;  
  time.postln;  
  1.0  
});
```

The **.awake** method is called on the function after 0 seconds. When functions are the 2<sup>nd</sup> argument, the time from the clock can be passed as an argument. This function will be rescheduled every one second.

# Second Counter Example

Schedule a simple counter on the System Clock that counts up from one by one every second

```
(  
var count = 0;  
SystemClock.sched(0, {  
    count = count + 1;  
    count.println;  
    1  
});  
)  
  
SystemClock.clear;
```

Using `SystemClock.clear` will clear the System Clock scheduler and stop the counting.

# Clocks and Routines

- The **.awake** method is equivalent to the **.next** method for routines.
- In the top routine, "hi" gets printed only once because the initially call to **.awake** posts "hi" and then yields 0.5. The clock reschedules the routine after 0.5 seconds and the routine yields **nil** preventing the routine from being rescheduled again.
- In the bottom routine, the same code is wrapped in a while loop that never breaks. The routine continually yields 0.5, causing it to be constantly scheduled.

```
SystemClock.sched(0.0,  
  Routine({  
    // Only prints "hi" once  
    "hi".postln;  
    0.5.yield;  
  })  
);
```

```
SystemClock.sched(0.0,  
  Routine({  
    while(true,  
      {"hi".postln; 0.5.yield}  
    );  
  })  
);
```

# Loop Method

SuperCollider offers a syntactic shortcut for the “while true” loop. The class `Function` has an instance method `.loop`. You will see this shortcut often in SC code with routines.

```
SystemClock.sched(0.0,  
  Routine({  
    while(  
      {true},  
      {"hi".postln; 0.5.yield}  
    );  
  })  
);
```

```
SystemClock.sched(0.0,  
  Routine({  
    {"hi".postln; 0.5.yield}.loop;  
  })  
);
```



SAME!

# Tempo Clock

- The class `TempoClock` is very similar to `SystemClock` except that it schedules based on beats per minute (i.e., tempo).
- SuperCollider has a default `TempoClock` set to 60bpm (beats per minute) accessible with `TempoClock.default`.
  - Note that the default `TempoClock` can be changed to an entirely new tempo clock or its properties like bpm can be changed with any of the available instance methods for `TempoClock`.
- We can also create our own Tempo Clocks.

# Tempo Clock Examples

- We can schedule functions just liked we did with `SystemClock` except here we will do it with beats.
- Two important methods:
  - `.schedAbs` – Schedules at a specific beat after startup. Like beat 81, for example.
  - `.sched` – Schedules at a delta beat away from the current clock time

```
(  
var tempo = 120; // In bpm  
t = TempoClock(tempo/60);  
t.schedAbs(4, {  
    "This is beat 4!".postln;  
    nil  
});  
)
```

```
(  
("Initial beat:" ++ t.beats).postln;  
t.sched(4, {  
    ("This is beat:" ++ t.beats).postln;  
    nil  
})  
)
```

# .play method

The **.play** method for routines is a syntactic shortcut to eliminate the need for explicitly using `TempoClock.sched` to schedule routines.

```
TempoClock.default.sched(0.0,  
  Routine({  
    {"hi".postln; 0.5.yield}.loop;  
  })  
);
```

```
Routine({  
  {"hi".postln; 0.5.yield}.loop;  
}).play;
```

SAME!

Note that you can use some other clock other than the default tempo clock as well by passing the clock into the first argument of **.play**.

# Music with Routines

- Routines can be used to control many different kinds of musical events.
- We will build a relatively simple example based on “Understanding Streams, Patterns, and Events” from the help documents. We need three key parts:
  - A synth sound to be played back
  - A routine to control what note to play back
  - A routine to control the rate of notes (i.e., tempo/rhythm)
- Especially with the rate of notes, routines are an ideal choice for controlling sound through time.



# Music with Routines

THE  
SYNTH

```
(
  SynthDef(\buzzSaw, {
    arg out = 0, freq = 440;
    var sig, env;
    env = EnvGen.kr(Env.perc, levelScale: 0.3, doneAction: 2);
    sig = RLPF.ar(
      in: LFSaw.ar(freq, 0, env),
      freq: LFNoise1.kr(1, 36, 110).midicps,
      rq: 0.1
    );
    4.do({ sig = AllpassN.ar(sig, 0.05, [0.05.rand, 0.05.rand], 4) });
    Out.ar(out, sig);
  }).add;
)
```

- Creates a buzzy sawtooth sound with the default percussion envelope. The partials of the sawtooth are fluctuated by resonant lowpass filter whose cutoff frequency moves up and down in a linear fashion based on the output of `LFNoise1.kr`. The result is passed through four stereo allpass filters in series to create a little reverberance.
- The class `LFNoise1` creates linearly interpolated ramps between randomized values in the range of -1 to 1. Compare `{LFNoise0.ar(5000)}.plot` and `{LFNoise1.ar(5000)}.plot`.

## NOTE ROUTINE



# Music with Routines

```
~noteRoutine = Routine({
  {
    // Low arpeggio
    if(0.5.coin, {
      24.yield;
      31.yield;
      36.yield;
      43.yield;
      48.yield;
      55.yield;
    });

    // Varying arpeggio
    rrand(2, 5).do({
      60.yield;
      #[63, 65].choose.yield;
      67.yield;
      #[70, 72, 74].choose.yield;
    });

    // Higher notes
    rrand(3, 9).do({#[74, 75, 77, 79, 81].choose.yield});
  }.loop;
});
```

- The routine here describes a randomized choice of notes that are yielded with subsequent calls to `.next`. The notes are encoded as midi pitch numbers.
- The `.coin` method chooses true based on a probability of choosing true (given by the receiver)
- The notation `#[ ]` describes a literal array. This is an array that cannot be changed after it has been declared but is very efficient.
- The method `.choose` when the receiver is an array randomly chooses an element from the array
- The method `.rrand` chooses an integer between a low value and high value, inclusive.

# Music with Routines

```
(  
  Routine({  
    var dur = 1/8; // Play a note every eighth of a second  
    {  
      Synth(\buzzSaw, [\freq, ~noteRoutine.next.midicps]);  
      dur.yield;  
    }.loop;  
  }).play;  
)
```

PLAYBACK  
ROUTINE



- This routine governs the selection of the notes from the previous routine and sets the rate of playback to an eighth of a second. Remember the play method sets the routine to be rescheduled by the value yielded (which is always 1/8 in this routine).
- Each scheduling of the routine generates exactly one synth which selects the note to be played from the note routine (previous slide) and converts the note from a MIDI pitch number to a frequency in hertz.

# Patterns

- Patterns are closely related to streams in SuperCollider.
  - Remember with streams, a stream is defined as any object in SuperCollider that has a **.next** and **.reset** method. All objects in SuperCollider are streams.
  - Patterns are defined as any object that has a **.asStream** method. You can think of them as templates that can be converted into streams. All objects in SuperCollider are patterns.
- The **.asStream** method converts a pattern into a stream. All patterns are meant to be converted into streams and lazily evaluated through the **.next** method.
- By default, most objects when used as the receiver to **.asStream** return itself as a stream (remember all objects are patterns so this is acceptable). There are objects, however, that have many more applications when used with **.asStream** and are not trivial intellectual exercises. We will focus on those.

# Pattern Objects: Pseq

- SC has a number of great, non-trivial pattern objects that can be converted to streams.
- The class `Pseq` – a sequential list of items. `nil` is provided when the list is exhausted.
  - 1<sup>st</sup> argument: a list
  - 2<sup>nd</sup> argument: the number of repetitions. Provide `inf` for infinite repeats. Default is 1.
  - 3<sup>rd</sup> argument: initial offset into the list. Default is 0.

```
(  
// prints out 1, 2, 3, 4, nil  
var pattern, stream;  
pattern = Pseq.new([1, 2, 3, 4]);  
stream = pattern.asStream;  
5.do({stream.next.postln});  
)
```

```
(  
// prints out 2, 3, 4, 1, 2, 3, 4, 1, ...  
// (capped at 50 items)  
var pattern, stream;  
pattern = Pseq.new([1, 2, 3, 4], inf, 1);  
stream = pattern.asStream;  
50.do({stream.next.postln});  
)
```

# Pattern Objects: Pser

- The class **Pser** – a sequential list of items.
  - 1<sup>st</sup> argument: a list
  - 2<sup>nd</sup> argument: the number of items. Wraps if the number of items > list length
  - 3<sup>rd</sup> argument: initial offset into the list. Default is 0.
- Pser is like Pseq except the second argument determines the **number of items** as opposed to the number of repeats.

```
(  
// prints out 1, 2, 3, 4, 1, 2  
var pattern, stream;  
pattern = Pser.new([1, 2, 3, 4], 6);  
stream = pattern.asStream;  
6.do({stream.next.postln});  
)
```

```
(  
// prints out 2, 3, 4, 1, 2, 3,  
var pattern, stream;  
pattern = Pser.new([1, 2, 3, 4], 6, 1);  
stream = pattern.asStream;  
6.do({stream.next.postln});  
)
```

# Pattern Objects: Prand

- The class `Prand` – choose a random item from a list of objects
  - 1<sup>st</sup> argument: a list
  - 2<sup>nd</sup> argument: the number of items
- If the calls to `.next` exceed the number of items, then `nil` is returned.

```
(  
  var pattern, stream;  
  pattern = Prand.new([1, 2, 3, 4, 5], 6);  
  stream = pattern.asStream;  
  7.do({stream.next.postln});  
)
```

# Pattern Objects: Pseries

- The class `Pseries` – create an arithmetic series of numbers from a starting point with a step value.
  - 1<sup>st</sup> argument: starting number
  - 2<sup>nd</sup> argument: step value
  - 3<sup>rd</sup> argument: number of values in the series

```
(  
  // print out 1, 4, 7, nil  
  var pattern, stream;  
  pattern = Pseries.new(1, 3, 3);  
  stream = pattern.asStream;  
  4.do({stream.next.postln});  
)
```



# Functions in Patterns

```
(  
var pattern, stream;  
pattern = Pseries.new(rrand(1, 10), 3, 5);  
stream = pattern.asStream;  
4.do({stream.next.postln});  
stream.reset;  
4.do({stream.next.postln});  
)
```

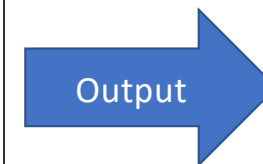
```
(  
var pattern, stream;  
pattern = Pseries.new({rrand(1, 10)}, 3, 5);  
stream = pattern.asStream;  
4.do({stream.next.postln});  
stream.reset;  
4.do({stream.next.postln});  
)
```

- Consider the two examples above. The one on the right wraps the random start value for the `Pseries` object in a function while the one on the left does not.
- Compare the output of the two snippets of code. The one on the left always has the same random number when the stream is reset. The one on the right reevaluates the random number when the stream is reset.
- Why? When the code on the left creates the `Pseries` object, the random number is evaluated and a number is passed, creating a fixed starting number. When a function is passed in, as on the right, the function serves as the starting value. In the source code for `Pseries` and many other patterns, values are often created with the `.value` method which for a number will yield the same result and for a function will evaluate the function anew each time.

# Nested Patterns

- Patterns can be embedded in other patterns.
- If a Pattern encounters another Pattern in its list, it embeds that pattern in its output. That is, it creates a stream on that pattern and iterates that pattern until it ends before moving on.
- Documentation is unclear about which arguments of which patterns can take other patterns. For example, you can use a pattern for `Pseries'` **step** but not **start** value. Look at source code for answers.

```
(  
var pattern, stream;  
pattern = Pseq.new([1, Pseq.new([100, 200], 2), 3], 3);  
stream = pattern.asStream;  
19.do({ stream.next.postln; });  
)
```



```
1  
100  
200  
100  
200  
3  
1  
100  
200  
100  
200  
3  
1  
100  
200  
100  
200  
3  
nil
```

# Organizing Sound: Events

- The class `Event` is a special dictionary that embeds all the information for a synth to be played on a server.
  - Recall that a `SynthDef` is a definition that encodes on the server the connections between any number of unit generators to output sound. It also provides names to input arguments that can be used to control that sound in real time.
  - We can use events to encapsulate the information for all the input arguments of synths on the server.
- Like many objects in SuperCollider, events respond to the `.play` message/method, providing a concise syntax for producing sound.

# Creating Events

- An event is created by simply using key-value pairs enclosed in parentheses.
  - A simple event: (`freq: 400`)
  - Key-value pairs are separated by colons. Note that I didn't need a `\` or quotes for `freq`. Alphanumeric characters beginning with a letter are assumed to be symbols.
  - This encodes the information for a synth with a frequency of 400. No sound is produced... yet.
- To play the event, we can use the `.play` method on the event. This initiates several actions:
  - Creates a synth on the server. The user can provide an instrument. Otherwise a default instrument is provided (a simple sine wave)
  - Passes in the arguments based on the key-value pairs in the events to the synth.
  - Plays the synth immediately.

# A Simple Event

- Here we will make a simple event based on the `\buzzSaw` SynthDef from earlier slides.
- We will need to provide the name of the instrument and, optionally, the frequency of the note. We could also change the outbus number since that is a parameter to the SynthDef but its default value of 0 is perfectly fine for our purposes.

```
(  
e = (\instrument: \buzzSaw, \freq: 400);  
e.class.postln; // Verify it is an Event  
e.play;  
)
```

# Events Are Dictionaries

- Events, inherited from Identity Dictionary, contain three “levels” of dictionaries. When a value is requested via a key, an Event first looks at the user defined pairings, then at the user defined default pairings (called proto), and finally SC defined default pairings (called parent) to find the key.
- Normally, an event has proto and parent set to `nil`.
  - However, when an event is called with the play method, SuperCollider **adds** a default parent dictionary that is required for the playing/creating of the synth. The play method calls from the parent dictionary the key `\play` whose value is function. That function is evaluated to play your note. Therefore, the parent dictionary is **required** in order for your event to be played on the server.
  - Once the play method is used on an event, the default parent dictionary is now the parent dictionary for the event. You can see the key-value pairings by using the method `.parent`.
  - This default parent dictionary provided by SuperCollider is very powerful and provides many conveniences to easily create interesting music.

# A Simple Event

```
(
e = (\instrument: \buzzSaw, \freq: 400);
e.class.postln; // Verify it is an Event
e.play;
)

e.parent; // See the parent dictionary of the Event
e[\play]; // The function necessary to play an event
e[\freq]; // The frequency specified by the user.
e[\instrument]; // The instrument specified by the user.
e[\amp]; // The default amplitude found in the parent dictionary
           // Will do nothing because amp is not a parameter to our synth
```

Note: you can use the class method **.default** to make an event with the default parent dictionary from the moment of instantiation. You can also establish your own parent dictionary though I recommend leaving it as is. Use a proto dictionary instead.

# Pbind

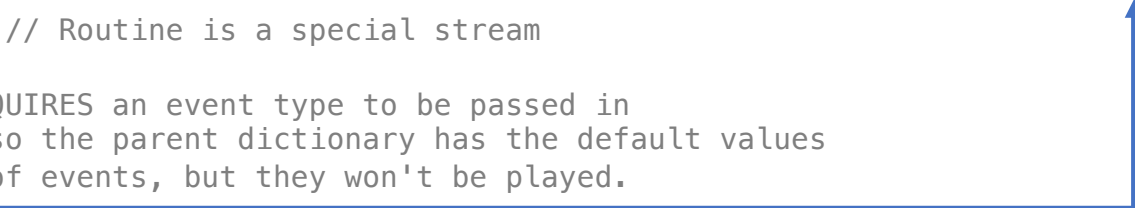
- All the patterns we have seen up to this point have returned single values with `.next`. We will now look at event patterns (i.e., patterns that return events).
- The class `Pbind` is one of the core classes for event patterns.
- It binds key symbols with value objects.
- The class `Pbind` takes a series of comma-separated key/values and creates a pattern of events using these key/value pairs.
- A `Pbind` object is often converted into a stream of events using its instance method `.asStream`, returning a routine that yields events.
  - The routine that is returned **requires** an `inval` argument of a type of event.
  - Generally, the default `Event` should be passed in as an argument.
- Like many objects in SC, `Pbind` has a `.play` instance method which returns an `EventStreamPlayer` object which handles the creation of a stream but also schedules the events for playing on the server.
  - You will generally want to use this method for playing the event patterns of a `Pbind`.



# Pbind Example

```
~eventPattern = Pbind(  
  \instrument, \buzzSaw,  
  \freq, Pseq([440, 880]),  
);
```

```
(\instrument: \buzzSaw, \freq: 440)  
(\instrument: \buzzSaw, \freq: 880)  
nil
```



```
~routine = ~eventPattern.asStream; // Routine is a special stream
```

```
// The routine created by Pbind REQUIRES an event type to be passed in  
// Here we pass the default event so the parent dictionary has the default values  
// Note that we produced a stream of events, but they won't be played.
```

```
~routine.next(Event.default);
```

```
~routine.reset; // Reset so we can play the events
```

```
~routine.next(Event.default).play; // Call the play method on each event
```

```
// Play the pattern. Speed is determined by the symbol \dur. Default of 1 beat.  
// The method .play on a Pbind converts it to a stream (i.e., a routine) and then  
// sends that information to an EventStreamPlayer which plays the events.
```

```
~eventPattern.play;
```

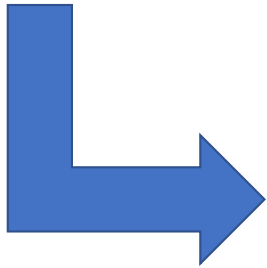
```
// Equivalent to ~eventPattern.play
```

```
~esp = EventStreamPlayer(~eventPattern.asStream, Event.default);
```

```
~esp.play;
```

# Simple Pbind Examples

```
SynthDef(\harpsichord, {  
  arg out = 0, freq = 440, amp = 0.1, gate = 1;  
  var sig, env;  
  env = EnvGen.ar(Env.adsr, gate, doneAction: 2);  
  sig = env * amp * Pulse.ar(freq, 0.25, 0.75);  
  Out.ar(out, sig ! 2);  
}).add;
```



```
p = Pbind(  
  \instrument, \harpsichord,  
  \degree, Pseq([  
    // Need functions otherwise same value  
    Pseries({rrand(0, 7)}, 1, {rrand(0, 8)}),  
    Pseries({rrand(0, 7)}, 1, {rrand(0, 8)}),  
    Pseries({rrand(0, 7)}, -2, {rrand(0, 8)})  
  ], inf),  
  \octave, 4, // Equivalent to C4 or middle C  
  \root, 3, // 3 semitones up from C (i.e., Eb)  
  \scale, [0, 2, 3, 5, 7, {rrand(8, 9)}, 11], // Harmonic/Melodic minor  
  \dur, 0.25  
).play;
```

# Pdef

- The class `Pdef` creates a reference to a pattern, allowing it to be changed mid-stream.
  - This is wonderful for real-time performance like a DJ would do
- The class `Pdef` takes two arguments:
  - 1<sup>st</sup> argument: a name for the pattern,
  - 2<sup>nd</sup> argument: a pattern (usually a `Pbind` for our purposes)
- All `Pdef` references can be found with the class method `.all` which lists all the key/value pairs of name and pattern
- See lecture code example for drum set instruments and several `Pdef` to control the two hi-hat, kick and snare drum sounds.

# Quantization

- Both `Pdef` and `Pbind` and many other classes in SuperCollider have a quantization method or parameter with the method `.play`.
- With `Pdef`, quantization allows for real-time changes to happen in certain moments relative to the tempo clock that `.play` uses.
  - Note you can pass your own tempo clock to `.play`.
- In the drum example in the lecture code, the quantization is set to 4 such that all changes happen on beats that are integer multiples of 4 (i.e., at the beginning of a 4/4 measure)
  - Note that the quantization is set with the method `.quant_`
  - The underscore is a shorthand for a setter method where you want to change an internal parameter.

# Further Reading

- The world of patterns, streams, and events in SuperCollider is daunting and massive. There are **many** more powerful tools with these classes that can produce wonderful musical results.
- Suggested Reading:
  - Understanding Events, Patterns, and
  - Pattern Guide
  - Pattern Guide Cookbook
  - Documentation under the class Pattern