Tidal tutorial

Welcome to the Tidal tutorial. Tidal is a mini-language for exploring pattern, designed for use in live coding performance. In this tutorial we'll step through different levels of abstraction, starting with sounds and filters, then sequences of sounds and filters, and moving up to functions for manipulating those sequences, and ending up looking at functions which manipulate other functions. Fun stuff!

1 Sounds and effects

With a bit of fiddling, Tidal can be used to pattern the input to any device which takes MIDI or Open Sound Control input, but the default is the Dirt software sampler. If you followed the install process, you should have Dirt installed and it should be running.

To test it, run the following by typing it into your text editor, holding down ctrl and pressing enter:

```
d1 $ sound "can"
```

You should be able to hear a repeating sample of someone hitting a can. Tidal is designed with repetitive dance music in mind, and will repeat the pattern forever, although you can build a great deal of variety in a single pattern, and also change it while it is running (i.e. live code).

The `can` in the above is the name of the sample you are playing. Well actually it is the name of a folder full of samples. You can find them in the `samples` subfolder of your dirt folder. You can specify a different sample by number, using the colon:

```
d1 $ sound "can:1"
```

Try some different numbers to hear all the different can samples that come with dirt.

Dirt comes with a wide range of samples to work with, here's some of them:

```
    flick sid can metal future gabba sn mouth co gretsch mt arp h cp
    cr newnotes bass crow hc tabla bass0 hh bass1 bass2 oc bass3 ho
    odx diphone2 house off ht tink perc bd industrial pluck trump
    printshort jazz voodoo birds3 procshort blip drum jvbass psr
    wobble drumtraks koy rave bottle kurt latibro rm sax lighter lt
    arpy feel less stab ul
```

Replace `can` with one of these to explore.

1.1 Silence

At this point you are probably reaching for a way to make a pattern stop. You can do this by setting the pattern to be silent like this:

```
d1 $ silence
```

Quicker still, you can just make everything silent by evaluating (again, with ctrl-enter) the word `hush`:

```
hush
```

This will become more useful later when we learn how to play more than one sound at once - `hush` will silence them all.

1.2 Effects

You can also apply a range of effects to change what your sound, er, sounds like. For example a vowel-like 'formant filter':

```
d1 $ sound "can:1" |+| vowel "a"
```

The `|+|` operator in the above is what binds the sound with the vowel parameter.

Try changing the "a" for other vowels. You can also play the sample faster, which makes it higher in pitch:

```
d1 $ sound "can:1" |+| speed "2"
```

Or slower:

```
d1 $ sound "can:1" |+| speed "0.5"
```

Or even backwards:

```
d1 $ sound "can:1" |+| speed "-1"
```

You can also apply several effects at the same time:

```
d1 $ sound "can:1" |+| vowel "a" |+| speed "-1"
```

Here is the full list of effects you can play with.

Name	Description
accelerate	a pattern of numbers that speed up (or slow down) samples while they play.
bandf	a pattern of numbers from 0 to 1. Sets the center frequency of the band-pass filter.
bandq	a pattern of numbers from 0 to 1. Sets the q-factor of the band-pass filter.
begin	a pattern of numbers from 0 to 1. Skips the beginning of each sample, e.g. 0.25 to cut off the first quarter from each sample.

Name	Description
coarse	fake-resampling, a pattern of numbers for lowering the sample rate, i.e. 1 for original 2 for half, 3 for a third and so on.
crush	bit crushing, a pattern of numbers from 1 for drastic reduction in bit-depth to 16 for barely no reduction.
cutoff	a pattern of numbers from 0 to 1. Applies the cutoff frequency of the low-pass filter.
delay	a pattern of numbers from 0 to 1. Sets the level of the delay signal.
delayfeedback	a pattern of numbers from 0 to 1. Sets the amount of delay feedback.
delaytime	a pattern of numbers from 0 to 1. Sets the length of the delay.
end	the same as begin, but cuts the end off samples, shortening them; e.g. 0.75 to cut off the last quarter of each sample.
gain	a pattern of numbers that specify volume. Values less than 1 make the sound quieter. Values greater than 1 make the sound louder.
hcutoff	a pattern of numbers from 0 to 1. Applies the cutoff frequency of the high-pass filter.
hresonance	a pattern of numbers from 0 to 1. Applies the resonance of the high-pass filter.
pan	a pattern of numbers between 0 and 1, from left to right (assuming stereo)
resonance	a pattern of numbers from 0 to 1. Applies the resonance of the low-pass filter.
shape	wave shaping distortion, a pattern of numbers from 0 for no distortion up to 1 for loads of distortion (watch your speakers!)
sound	a pattern of strings representing sound sample names (required)
speed	a pattern of numbers from 0 to 1, which changes the speed of sample playback, i.e. a cheap way of changing pitch
unit	a pattern of words specifying the unit that the `speed` parameter is expressed in. Can either be 'rate' (the default, percentage playback rate), 'cycle' (cycle/n), or 'secs' (n seconds)
vowel	formant filter to make things sound like vowels, a pattern of either a, e, i, o or u. Use a rest (~) for no effect.

2 Continuous patterns

There are some 'continuous' patterns built in, which allow you to apply things like sinewaves to parameters which take patterns of numbers.

For example, to apply a sinewave to the `pan` parameter:

```
d1 $ sound "bd*16" |+| pan sine1
```

`sine1` is a sinewave pattern which travels from 0 to 1 and back again over a cycle. There is also `tri1`, `saw1` and `square1` for triangular, sawtooth and square waves respectively. If you miss off the '1' from the end of these you get a pattern from -1 to 1 instead, which can sometimes be useful. We'll see how to manipulate these patterns to go to any range later on.

3 Sequences

You're probably bored of hearing the same sample over and over by now, let's quickly move on to sequences. Tidal sequences allow you to string samples together, stretch the sequences out and stack them up in a variety of interesting ways, as well as start mixing in randomisation.

You can make a tidal cycle with more than one sample just like this:

```
d1 $ sound "drum drum:1"
```

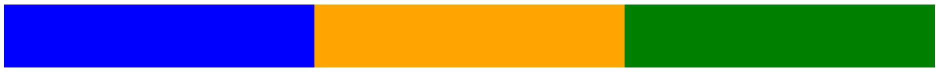
Kick and snare forever!

You'll notice that however many things you put into a Tidal pattern, it still takes up the same amount of time. For example the following fits three sounds into the same cycle duration:

```
d1 $ sound "drum drum:1 can"
```

By the way, from time to time I will illustrate concepts with patterns of colour, for example:

```
"blue orange green"
```



You can understand the above in terms of a sound pattern that reads from left to right, i.e. the horizontal axis is time.

The ~ symbol represents a rest, or pause, e.g.:

```
d1 $ sound "drum drum:1 ~"
```

You can play around with some more off-kilter patterns, for example this one which has seven steps in it:

```
d1 $ sound "drum ~ can ~ ~ drum:1 ~"
```

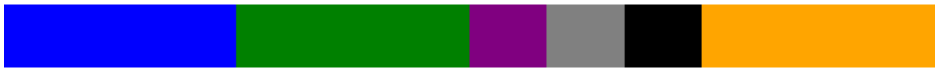
3.1 Subdividing sequences

You can take one step in a pattern and subdivide it into substeps, for example in the following the three `can` samples are played inside the same amount of time that each `drum` sample does:

```
d1 $ sound "drum drum [can can:4 can:5] drum"
```

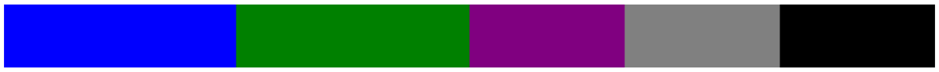
Again, here's the visual equivalent, which makes clear that an event is broken down into three 'subevents':

```
"blue green [purple grey black] orange"
```



As you can see the square brackets give the start and end of a subdivision. Here's another example, which has two events, which are broken down into different numbers of sub events:

```
"[blue green] [purple grey black]"
```

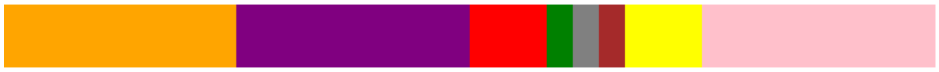


Actually you can keep going, and subdivide a step within a subdivision:

```
d1 $ sound "drum drum [can [can:4 can:6 can:3] can:5] drum"
```

Again, the visual equivalent:

```
"orange purple [red [green grey brown] yellow] pink"
```



3.2 Layering up patterns

Square brackets also allow you to specify more than one subpattern, by separating them with a comma:

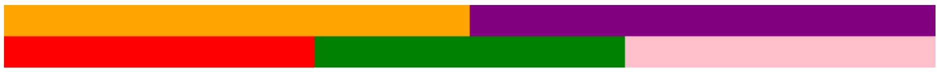
```
d1 $ sound "drum [can cp, can bd can:5]"
```

As you can hear, the two patterns are layered up. Because they are different lengths (one with two sounds, the other with three), you can get an interesting polyrhythmic effect. You can hear this better if you just have a single subdivision like this:

```
d1 $ sound "[can cp, can bd can:5]"
```

We can visualise this by stacking up the different part of the patterns, which makes clear how they co-occur:

```
"[orange purple, red green pink]"
```

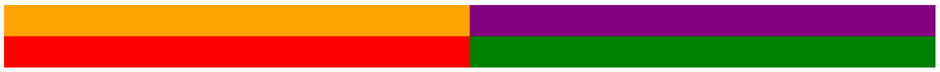


If you use curly brackets rather than square brackets the subpatterns are layered up in a different way, so that the sounds inside align, and the different lengths of patterns seem to roll over one another:

```
d1 $ sound "{can can:2, can bd can:5}"
```

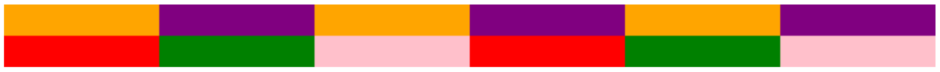
Here's what that looks like:

```
"{orange purple, red green pink}"
```



The problem with the above is that the pattern is structured over several 'cycles' (in this case, three), and we can only see the first cycle. Lets jump a bit ahead and use the `density` function to pack more cycles in:

```
density 3 "{orange purple, red green pink}"
```



Again, you can layer up more than one of these subpatterns:

```
d1 $ sound "[can cp, can bd can:5, arpy arpy:2 ~ arpy:4 arpy:5]"
```

And subdivide further:

```
d1 $ sound "{[can can] cp, can bd can:5, arpy arpy:2 ~ [arpy:4 arpy:5] arpy:5}"
```

This can already start getting very complex, and we haven't even got on to functions yet!

3.3 Sequencing niceties and tricks

Staying with sequences for a bit longer, there are a couple of other things you can do.

3.3.1 Repetition and division

If you want to repeat the same sample several times, you can use `*` to specify how many times. For example this:

```
d1 $ sound "bd [can can can]"
```

Can be written like this:

```
d1 $ sound "bd can*3"
```

When live coding saving a little bit of typing helps a lot. You can experiment with high numbers to make some strange sounds:

```
d1 $ sound "bd can*32 bd can*16"
```

The above pattern plays the samples so quickly that your ears can't hear the individual sounds any more, and instead you hear it as an audio frequency, i.e. a musical note.

If you have a pattern that has a repeat that isn't a subpattern, like this:

```
d1 $ sound "bd can can can"
```

You can repeat successive events with `!`:

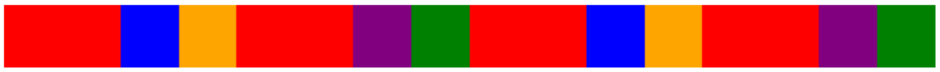
```
d1 $ sound "bd can ! !"
```

You can also 'slow down' a subpattern, for example this plays the `[bd arpy sn:2 arpy:2]` at half the speed:

```
d1 $ sound "bd [bd arpy sn:2 arpy:2]/2"
```

That is, the first cycle you get `bd [bd arpy]` and the second time around you get `bd [sn:2 arpy:2]`. This is a little bit difficult to understand, but basically if you don't get through a whole subpattern during one cycle, it carries on where it left off the next one. This is worth looking at a colour pattern:

```
density 4 "red [blue orange purple green]/2"
```



Again we have used `density` to pack in more cycles (in this case 4) so you can see the changes from one cycle to the next.

You can get some strange things going on by for example repeating four thirds of a subpattern per cycle:

```
d1 $ sound "bd [bd arpy sn:2 arpy:2]*4/3"
```

If you like strange time signatures, hopefully you will be having fun with this already.

3.3.2 Random drops

If you only want something to happen sometimes, you can put a question mark after it:

```
d1 $ sound "bd can? bd sn"
```

In the above, the can sample will only play on average 50% of the time. If you add a question mark to a subpattern, it applies separately to each element of the subpattern. For example in the following sometimes you get no can sounds, sometimes just the first or second, and sometimes both:

```
d1 $ sound "bd [can can:4]? bd sn"
```

3.3.3 Enter Bjorklund (and Euclid)

If you give two numbers in parenthesis after an element in a pattern, then Tidal will distribute the first number of sounds equally across the second number of steps:

```
d1 $ sound "can(5,8)"
```

But then, it isn't possible to distrute three elements equally across eight discrete steps, but the algorithm does the best it can. The result is a slightly funky bell pattern. Try this one:

```
d1 $ sound "can(5,8)"
```

This uses "Bjorklund's algorithm", which wasn't made for music but for an application in nuclear physics, which is exciting. More exciting still is that it is very similar in structure to the one of the first known algorithms written in Euclid's book of elements in 300 BC. You can read more about this in the paper The Euclidean Algorithm Generates Traditional Musical Rhythms by Toussaint. Examples from this paper are included below, although some require rotation to start on a particular beat - see the paper for full details and references.

Pattern	Description
(2,5)	A thirteenth century Persian rhythm called Khafif-e-ramal.
(3,4)	The archetypal pattern of the Cumbia from Colombia, as well as a Calypso rhythm from Trinidad.
(3,5)	When started on the second onset, is another thirteenth century Persian rhythm by the name of Khafif-e-ramal, as well as a Rumanian folk-dance rhythm.
(3,7)	A Ruchenitza rhythm used in a Bulgarian folk-dance.
(3,8)	The Cuban tresillo pattern.
(4,7)	Another Ruchenitza Bulgarian folk-dance rhythm.
(4,9)	The Aksak rhythm of Turkey.
(4,11)	The metric pattern used by Frank Zappa in his piece titled Outside Now.
(5,6)	Yields the York-Samai pattern, a popular Arab rhythm, when started on the second onset.
(5,7)	The Nawakhat pattern, another popular Arab rhythm.
(5,8)	The Cuban cinquillo pattern.
(5,9)	A popular Arab rhythm called Agsag-Samai.
(5,11)	The metric pattern used by Moussorgsky in Pictures at an Exhibition.
(5,12)	The Venda clapping pattern of a South African children's song.
(5,16)	The Bossa-Nova rhythm necklace of Brazil.
(7,8)	A typical rhythm played on the Bendir (frame drum).

Pattern	Description
(7,12)	A common West African bell pattern.
(7,16)	A Samba rhythm necklace from Brazil.
(9,16)	A rhythm necklace used in the Central African Republic.
(11,24)	A rhythm necklace of the Aka Pygmies of Central Africa.
(13,24)	Another rhythm necklace of the Aka Pygmies of the upper Sangha.

4 Functions

Up until now we have mostly only been working with building sequences, although this has included polyrhythm, and some simple algorithmic manipulation that is built into Tidal's pattern syntax. Now though it is time to start climbing up the layers of abstraction to see what we can find on the way.

First, lets have a closer look at the functions we have been using so far.

4.1 Sending patterns to Dirt

`d1` is a function that takes a pattern as input, and sends it to dirt. By default there are ten of them defined, from `d1` to `d10`, which allows you to start and stop multiple patterns at once.

For example, try running each of the following four lines in turn:

```
d1 $ sound "bd sn"

d2 $ sound "arpy arpy:2 arpy"

d1 $ silence

d2 $ silence
```

The first line will start a bass drum - snare pattern, the second start a slightly tuneful pattern, then the third swaps the first pattern with silence (so it stops), and the four swaps the second with silence (so everything is silent).

It's important to notice here that the Tidal code you type in and run with `ctrl-enter` is changing patterns which are running in the background. The running patterns don't change while you are editing the code, until you hit `ctrl-enter` again. There is a disconnect between code and process that might take a little getting used to.

4.2 The dollar `$`

You might wonder what that dollar symbol `$` is doing. If you are not wondering this, you are safe to skip this explanation.

The dollar actually does almost nothing; it simply takes everything on its right hand side, and gives it to the function on the left. If we take it away in the following example, we get an error:

```
d1 sound "bd sn"
```

That's because Tidal[1] reads from left to right, and so gives the `sound` to `d1`, before `sound` has taken `"bd sn"` as input, which results in confusion. We could get the right behaviour a different way, using parenthesis:

```
d1 (sound "bd sn")
```

[1]well, Tidal's underlying language, Haskell

This makes sure sound gets its pattern before it is passed on to d1. The dollar is convenient though because you don't have to match the closing bracket, which can get fiddly when you have lots of patterns embedded in each other.

```
d1 $ sound "bd sn"
```

Anyway, lets escape this syntactical diversion.

4.3 Layering up patterns with `stack`

You can play several patterns at once with the `stack` function, giving a list of patterns by wrapping them in square brackets and separating with commas. This is rather similar to the sequencing syntax we saw earlier, but takes place in the outside world of functions.

```
d1 $ stack [sound "bd sn:2" |+| vowel "[a e o]/2",
            sound "casio casio:1 casio:2*2"
           ]
```

4.4 Sticking patterns end-to-end with `cat` and `slowcat`

If you replace `stack` with `cat`, the patterns will be stuck one after another instead of on top of one another:

```
d1 $ cat [sound "bd sn:2" |+| vowel "[a o]/2",
          sound "casio casio:1 casio:2*2"
         ]
```

The `cat` function squeezes all the patterns into the space of one, but `slowcat` will maintain the speed of playback:

```
d1 $ slowcat [sound "bd sn:2" |+| vowel "[a o]/2",
              sound "casio casio:1 casio:2*2"
             ]
```

4.5 Slowing down and speeding up patterns with `slow` and `density`

Simply slowing patterns down substantially changes their character, sometimes in quite suprising ways. Use `slow` to slow down a pattern:

```
d1 $ slow 2 $ sound "bd ~ sn bd ~ [~ bd]/2 [sn [~ bd]/2] ~"
```

And our friend `density` to speed it up again.

```
d1 $ density 2 $ sound "bd ~ sn bd ~ [~ bd]/2 [sn [~ bd]/2] ~"
```

Play around with the numbers, and note that `density 0.5` is actually the same as `slow 2`.

4.6 Reversal with `rev`

The `rev` function reverses every cycle in a pattern:

```
d1 $ rev $ sound "bd ~ sn bd ~ [~ bd]/2 [sn [~ bd]/2] ~"
```

4.7 chop

The chop function chops each sample into the given number of bits:

```
d1 $ chop 16 $ sound "bd ~ sn bd ~ [~ bd]/2 [sn [~ bd]/2] ~"
```

This makes it sounds really granulated. It sounds stranger if you reverse it after the chop:

```
d1 $ rev $ chop 16 $ sound "bd ~ sn bd ~ [~ bd]/2 [sn [~ bd]/2] ~"
```

Due to the use of `$`, this is working from right to left; first it makes the sequence, then it passes the sequence to `chop 16`, then passes that to `rev`, and finally out to the Dirt synth using `d1`. If we swap the order of the chop and the rev, it sounds different:

```
d1 $ chop 16 $ rev $ sound "bd ~ sn bd ~ [~ bd]/2 [sn [~ bd]/2] ~"
```

That's because it's doing the reverse first, then chopping the samples up after, so the bits don't end up reversed. (I hope that makes sense).

Chop works particularly well for longer samples:

```
d1 $ rev $ slow 4 $ chop 16 $ sound "breaks125"
```

This gets a lot more fun with meta functions.

5 Meta-functions

There are a lot more functions than the above, but before looking at some more of them lets jump up a level to look at some meta-functions.

5.1 every

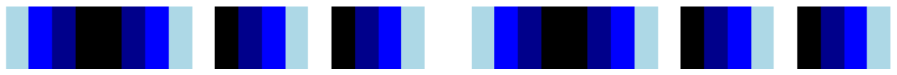
By meta-functions I mean functions which take other functions as input. For example, what if we didn't want to reverse a pattern every time, but only every other time?

```
d1 $ every 2 rev $ sound "bd can sn can:4"
```

Instead of applying `rev` directly to `sound "bd can sn can:4"`, the above passes `rev` to `every`, telling it to apply it every `2` repetitions. Try changing `2` to `4` for a very different feel.

Lets have a look at a visual example:

```
density 8 $ every 4 rev "black darkblue blue lightblue white"
```

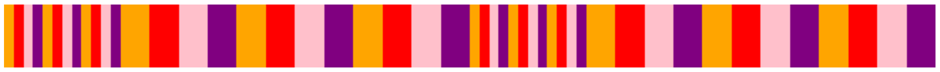


This works with other functions that work on patterns. Here's how you make a pattern twice as dense every four repetitions:

```
d1 $ every 4 (density 2) $ sound "bd can sn can:4"
```

... and visually:

```
density 8 $ every 4 (density 3) "orange red pink purple"
```



Note that we have to wrap `density 2` in parenthesis, to package it up to pass to `every` as a function that can be selectively applied to `"bd can sn can:4"`. If this doesn't make sense, get a feel for it by playing around with it, and content yourself with the fact that this technique involves something called *currying*, so cna't be all bad.

Lets try this with some longer samples:

```
d1 $ every 2 (density 1.5) $ every 3 (rev) $ slow 4 $ chop 16 $ sound "breaks125"
```

5.2 sometimes

The `sometimes` function works a bit like `every`, except it sometimes applies the given function, and sometimes doesn't, in an unpredictable manner.

```
d1 $ sometimes (density 2) $ sound "bd can*2 sn can:4"
```

There are similar functions `often` and `almostAlways` which apply the function more often than not, and `rarely` and `almostNever`, which apply the function less often.

You can always stack functions up, for example this works:

```
d1 $ sometimes (density 2) $ every 4 (rev) $ sound "bd can sn can:4"
```

In general, Tidal gets most interesting when you take simple parts and combine them in this way.

5.3 jux

The `jux` metafunction applies the given function in just the one channel or speaker. For example, in the following the given pattern is reversed in one of the speakers, and played normally in the other:

```
d1 $ jux rev $ sound "bd sn*2 can [~ arpy]"
```

In this one the pattern is played 25% faster in one speaker than the other:

```
d1 $ jux (density 1.25) $ sound "arpy:2 arpy:4 arpy:1 [~ arpy]"
```

5.4 weave

Weave is a strange one, which takes different synth parameters and overlays them, offset against each other, on top of a base pattern. Ok, this needs an example:

```
d1 $ weave 16 (pan sine1)
  [sound "bd sn", sound "arpy ~ arpy:3", sound "can ~ ~ can:4"]
```

In the above all three patterns have the `pan sine1` parameter applied, but are spaced out around the cycle of the `pan`, which is also stretched out over `16` cycles. As a result, the three patterns move around each other, following each other around the speakers. This is especially nice if you're running Dirt in multichannel mode, i.e. with more than two speakers.

You can flip things round so that the base is the `sound` pattern, and the patterns you're applying to different moving parts of it are effects:

```
d1 $ weave 16 (sound "arpy arpy:7 arpy:3")
  [vowel "a e i", vowel "o i", vowel "a i e o", speed "2 4 ~ 2 1"]
```

6 Functions part 2

Now we've seen how meta-functions can make use of functions, lets have a look at more of the latter.

6.1 Rotation with ~> and <~

The ~> operator 'rotates' a pattern by the given amount in cycles, for example this shifts the pattern forward in time by a quarter of a cycle:

```
d1 $ 0.25 ~> sound "arpy arpy:1 arpy:2 arpy:3"
```

Predictably the <~ operator does the same, but in the other direction:

```
d1 $ 0.25 <~ sound "arpy arpy:1 arpy:2 arpy:3"
```

Unless another pattern is playing at the same time, you can only hear the difference when you change the number, which you perceive as a skipping back and forth. This is again where those metafunctions come in:

```
d1 $ every 4 (0.25 <~) $ sound "arpy arpy:1 arpy:2 arpy:3"
```

In the above, the pattern skips every 4th cycle. Again, because the output of all these functions is a pattern, they can be used as input to another function:

```
d1 $ jux ((1/8) ~>) $ every 4 (0.25 <~) $ sound "arpy*3 arpy:1*2 arpy:4 [~ arpy:3]"
```

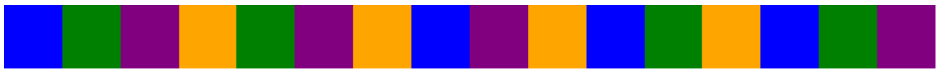
6.2 Compound rotation with `iter`

For a given `n`, the `iter` function shifts a pattern to the left by `1/n` steps every cycle. An example speaks wonders:

```
d1 $ iter 4 $ sound "arpy:1 arpy:2 arpy:3 arpy:4"
```

Here's the visual equivalent:

```
density 4 $ iter 4 $ "blue green purple orange"
```



This works well with `jux`:

```
d1 $ jux (iter 8) $ sound "arpy:1 arpy:2 arpy:3 arpy:4"
```

6.3 Scaling number patterns

The `scale` function is handy for taking a pattern like `sine1` which goes from `0` to `1`, and making it go to a different range, in the below example from `1` to `1.5`.

```
d1 $ jux (iter 4) $ sound "arpy arpy:2*2"
  |+| speed (slow 4 $ scale 1 1.5 sine1)
```

Using scale is particularly important for the `shape` parameter, because if you give that numbers which are too high (i.e. close to 1) it gets very loud. In the below we cap it at 0.8.

```
d1 $ jux (iter 4) $ sound "drum drum:1*2"
  |+| shape (slow 4 $ scale 0 0.8 sine1)
```

6.4 Picking samples

You can have one pattern for sample names, and another of sample numbers, and combine them to pick which sample you want.

```
d1 $ sound (samples "drum arpy newnotes" "0 1 2")
```

This isn't very exciting until you start manipulating the patterns before combining them, and as in the below adding a bit more pattern manipulation on top:

```
d1 $ jux (density 2) $ sound (samples "drum can can" (slow 2.5 "0 1 2 4 5 6"))
```

7 That's it for now

There is more, which you can dig out by exploring the Tidal website, Tidal screencasts (e.g. on youtube and vimeo) and the Tidal patterns that people have shared. You can also sign up to the Tidal forum at http://lurk.org/groups/tidal and join the community discussion there. Have fun!