Tidal: the guide

You've installed TidalCycles and (Super)Dirt, maybe even made a few sounds, but now you're ready to get to business and start really learning. This guide will help you get started with simple patterns and walk you through all the way to complex compositions.

> Why not play with the code as you read, running your own experiments by changing the examples, and seeing where they take you?

» Creating Rhythmic Sequences

Sequences

Play a Single Sample

Tidal starts with nine connections to the SuperDirt synthesiser, named from `d1` to `d9` (if you're using the 'classic' dirt, then instead use `c1` to `c9`). Here's a minimal example, that plays a bass drum every cycle:

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

> Evaluate the above code in the Atom (or Emacs) editor by pressing `Ctrl+Enter`

In the code above, `sound` tells us we're making a pattern of sound samples, and `"bd"` is a pattern that contains a single sound. `bd` is a sample of a bass drum. Samples live inside the `/samples` folder which came with SuperDirt, and each sub-folder under `/samples` corresponds to a sample name (like `bd`).

> To find the SuperDirt samples on your system, run the command `Quarks.folder` in SuperCollider (paste it in, press shift-enter), and the 'quark' plugin path will be shown in the postwindow on the right. You'll find the samples in the Dirt-Samples subfolder.

We can pick a different sample in the `bd` folder by adding a colon (`:`) then a number. For example this picks the fourth bass drum (it counts from zero, so `:3` gives you the fourth sound in the folder):

```
d1 $ sound "bd:3"
```

If you specify a number greater than the number of samples in a folder, then Tidal just "wraps" around back to the first sample again (it starts counting at zero, e.g. in a folder with five samples, "bd:5" would play "bd:0").

Sequences From Multiple Samples

Putting things in quotes actually defines a sequence. For example, the following gives you a pattern of bass drum then snare:

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

When you run the code above, you are replacing the previous pattern with another one on-the-fly. Congratulations, you're live coding.

> You can find (and if you like, add to) the samples in the samples folder inside the Dirt folder. They're in 'wav' format.

Playing More Than One Sequence

The easiest way to play multiple sequences at the same time is to use two or more connections to the synthesizer:

```
d1 $ sound "bd sn"

d2 $ sound "hh hh hh hh"

d3 $ sound "arpy"
```

> NOTE: each connection must be evaluated separately in your text editor. That is, you must press `Ctrl+Enter` three times, once for each line above. Make sure that there is a blank line between them each pattern, or Tidal will evaluate them together and get confused (if you want to evaluate just one line, you can press shift-enter).

What is a Cycle?

A cycle is the main “loop” of time in Tidal. The cycle repeats forever in the background, even when you’ve stopped samples from playing. The cycle’s duration always stays the same unless you modify it with `cps` or `bps`, we’ll cover this later.

> Note that this omniprescent cyclic looping doesn’t necessary constrain you, for example it’s common to stretch a pattern outside of a single loop, and vary patterns from one loop to the next. We’ll see several ways to do this later, as well.

All of the samples inside of a pattern get squashed into a single cycle. The patterns below all loop over the same amount of time:

```
d1 $ sound "bd sn"
d1 $ sound "bd sn hh cp mt arpy drum"
d1 $ sound "bd sn hh cp mt arpy drum odx bd arpy bass2 feel future"
```

Note how the more steps you add to the pattern, the faster it goes to fit them all in. No matter how many samples you put in a pattern in this way, they will always be distributed evenly within a single cycle.

» Silence

Silence

An empty pattern is defined as `silence`, so if you want to ‘switch off’ a pattern, you can just set it to that:

```
d1 silence
```

If you want to set all the connections (from `d1` to `d9`) to be silent at once, there’s a single-word shortcut for that:

```
hush
```

You can also isolate a single connection and silence all others with the `solo` function:

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

» Patterns Within Patterns

Pattern Groups

Use Tidal's *square braces* syntax to create a pattern grouping:

```
d1 $ sound "[bd sn sn] cp"
```

Square braces allow several events to be played inside of a single step. You can think of the above pattern as having two steps, with the first step broken down into a subpattern, which has three steps. Practically, this means you can create denser sub-divisions of samples:

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

You can even nest groups inside groups to create very dense and complex patterns:

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

Layering (Polyrhythms) Instead of Grouping

You can also layer up several loops, by using commas to separate the different parts:

```
d1 $ sound "[bd bd bd, sn cp sn cp]"
```

This would play the sequence `bd bd bd` at the same time as `sn cp sn cp`. Note that the first sequence only has three events, and the second one has four. Because tidal ensures both loops fit inside the same cyclic duration, you end up with a polyrhythm.

You can layer any number of patterns to create many polyrhythms:

```
d1 $ sound "[bd bd bd, sn cp sn cp, arpy arpy, odx]"
```

And of course you can use groupings inside of the layers:

```
d1 $ sound "[bd bd bd, [sn sn] cp, arpy [arpy [arpy arpy] arpy arpy], odx]"
```

» Pattern Repetition and Speed

Repetition

There are two short-hand symbols you can use inside patterns to speed things up or slow things down: `*` and `/`. You could think of these like multiplication and division.

Use the `*` symbol to make a pattern, or part of a pattern, repeat as many times as you'd like:

```
d1 $ sound "bd*2"
```

> This is the same as doing `d1 $ sound "bd bd"`

The code above uses `*2` to make a sample play twice.

You can use the `/` symbol to make a part of a pattern slow down, or occur less often:

```
d1 $ sound "bd/2"
```

The code above uses `/2` to make a sample play half as often, or once every 2nd cycle.

Using different numbers works as you'd expect:

```
d1 $ sound "bd*3" -- plays the bd sample three times each cycle
d1 $ sound "bd/3" -- plays the bd samples only once each third cycle
```

Using * and / on Groups

You can apply the `*` and `/` symbols on groups of patterns:

```
d1 $ sound "[bd sn]*2 cp"
d1 $ sound "[bd sn] cp/2"
d1 $ sound "[[bd sn] cp]*2" -- speeds up the entire pattern by 2
d1 $ sound "[[bd sn] cp]/2" -- slows down the entire pattern by 2
```

You can also use the symbols on nested groups to create more complex rhythms:

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

» Modifying Sequences With Functions

Transformation

Tidal comes into its own when you start building things up with functions which transform the patterns in various ways.

For example, `rev` reverses a pattern:

```
d1 $ rev (sound "[bd bd] [bd [sn [sn sn] sn] sn]")
```

That's not so exciting, but things get more interesting when this is used in combination with another function. For example `every` takes two parameters: a number, a function and a pattern to apply the function to. The number specifies how often the function is applied to the pattern. For example, the following reverses the pattern every fourth repetition:

```
d1 $ every 4 (rev) (sound "bd*2 [bd [sn sn*2 sn] sn]")
```

> It takes a while to get used to how we're using parenthesis here. In the previous example, `rev` takes one parameter, a pattern, and we had to 'wrap up' the pattern `sound "[bd bd] [bd [sn [sn sn] sn] sn]"` in brackets to pass it to `rev`. In the above example `every` takes three parameters: a number, a function and a pattern. We had to wrap up the pattern as before, but also `rev` in order to give it to `every`. This should become clearer with practice.

You can also slow down or speed up the playback of a pattern, this makes it a quarter of the speed:

```
d1 $ slow 4 $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

And this four times the speed:

```
d1 $ density 4 $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

> Note that `slow 0.25` would do exactly the same as `density 4`.

Again, this can be applied selectively:

```
d1 $ every 4 (density 4) $ sound "bd*2 [bd [sn sn*2 sn] sn]"
```

> Note again the use of parenthesis, around `density 4`. This is needed, to group together the function `density` with its parameter 4, before being passed as a parameter to the function `every`.

In the examples above, the `sound` function takes a pattern of sample names, and turns it into a pattern of synthesiser triggers. This might take a while to fully understand, but the important thing to remember is that "it's patterns all the way down". In this case, this means that you can operate on the inner pattern of sample names, instead of the outer pattern of synthesiser triggers that `sound` gives you:

```
d1 $ sound (every 4 (density 4) "bd*2 [bd [sn sn*2 sn] sn]")
```

Where are all the functions?

There are many types of functions that help you change patterns. Some of them re-order sequences, some alter time, some provide conditional logic, and some can help compose more complex patterns.

All of the functions available in Tidal can be found on the Reference page (/functions.html).

» Using Effects

Effects

TidalCycles has a number of effects that you can apply to sounds. Some of them do simple things like change volume, and others do more complex things like add delay or distortion.

You use an effect by adding the `#` operator between your sound pattern and the effect:

```
d1 $ sound "bd*4" # gain "0.5"
```

The above code decreases the volume of the “bd” sample by 50%.

You can chain multiple effects together, separating them again with the `#` operator:

```
d1 $ sound "bd*4" # gain "0.5" # delay "0.5"
```

The code above decreases the volume by 50% and also applies a “delay” effect at a level of 0.5.

Effects are patterns too

You may notice that the values of effects are specified in double quotes. This means that you can pattern the effect values too:

```
d1 $ sound "bd*4" # gain "1 0.8 0.5 0.7"
```

Effect patterns follow all the same grouping rules as sound patterns:

```
d1 $ sound "bd*4 sn*4" # gain "[[1 0.8]*2 [0.5 0.7]]/2"
```

And you can also apply functions to effect patterns:

```
d1 $ sound "bd*4" # gain (every 3 (rev) $ "1 0.8 0.5 0.7")
```

> Like with the `sound` example earlier, you must use parenthesis after `gain` in order to specify a function on the `gain` pattern.

Effect pattern order

You can specify the effect before the sound pattern:

```
d1 $ gain "1 0.8 0.5 0.7" # sound "bd"
```

The order that you put things matters; the structure of the pattern is given by the pattern on the left of the `#`. In this case, only one `bd` sound is given, but you hear four, because the structure comes from the `gain` pattern on the left.

Modifying effect values

The `#` operator is just a shortcut to a longer form of operator called `|=|`. The `|=|` operator means something special about combining patterns, but we'll cover that later. All you need to know right now is that `|=|` will set an effect's value equal to a pattern.

However, you can also change an effect value on the other side of a pattern:

```
d1 $ (|=| speed "2") $ sound "arpy*4" |=| speed "1"
```

In the code above, the left-most effect overrides the original effect that was specified on the right. In this case, `speed` will always equal 2.

You can do this conditionally:

```
d1 $ every 2 (|=| speed "2") $ sound "arpy*4" |=| speed "1"
```

There are other types of operators that allow you to perform arithmetic:

```
|+|
|-|
|*|
|/|
```

For example, using `|+|` will perform an *addition* operation and *add* to an original value:

```
d1 $ every 2 (|+| speed "1") $ sound "arpy*4" |=| speed "1"
```

> The code above results in a speed of “2” every other cycle.

The following will multiply values:

```
d1 $ every 2 (|*| speed "1.5") $ sound "arpy*4" |=| speed "1"
```

More complex patterns and chaining can be done, and with any effect, of course:

```
d1 $ every 3 (|-| up "3") $ every 2 (|+| up "5") $ sound "arpy*4" |=| up "0 2 4 5"
```

Some Common Effects

Here is a quick list of some effects you can use in Tidal (the full list is available in the Reference section):

- gain (changes volume, values from 0 to 1)
- pan (pans sound right and left, values from 0 to 1)
- shape (a type of amplifier, values from 0 to 1)
- vowel (a vowel formant filter, values include a, e, i, o, and u)
- speed (changes playback speed of a sample, see below)

» Sample Playback Speed (and Pitch)

Speed

You can change the playback speed of a sample in TidalCycles by using the `speed` effect. You can use `speed` to change pitches, to create a weird effect, or to match the length of a sample to a specific period of the cycle time.

You can set a sample's speed by using the `speed` effect with a number.

- `speed "1"` plays a sample at its original speed
- `speed "0.5"` plays a sample at half of its original speed
- `speed "2"` plays a sample at double its original speed

```
d1 $ sound "arpy" # speed "1"
d1 $ sound "arpy" # speed "0.5"
d1 $ sound "arpy" # speed "2"
```

Just like other effects, you can specify a pattern for speed:

```
d1 $ sound "arpy*4" # speed "1 0.5 2 1.5"
```

You can also reverse a sample by specifying negative values:

```
d1 $ sound "arpy*4" # speed "-1 -0.5 -2 -1.5"
```

Play a sample at multiple speeds simultaneously

Use the pattern grouping syntax with a comma to cause `speed` to play a sample back at multiple speeds at the same time:

```
d1 $ sound "arpy" # speed "[1, 1.5]"
d1 $ sound "arpy*4" # speed "[1 0.5, 1.5 2 3 4]"
```

12-tone scale speeds

You can also use the `up` function to change playback speed. `up` is a shortcut effect that matches speeds to half steps on a 12-tone scale. For example, the following plays a chromatic scale:

```
d1 $ sound "arpy*12" # up "0 1 2 3 4 5 6 7 8 9 10 11"
```

> You can also use the `run` function to create an incrementing pattern of integers: `d1 $ sound "arpy*12" # up (run 12)`. The `run` function will be discussed later.

» Euclidean Sequences

Bjorklund

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 "bd(5,8)"
```

You can also use the `e` function to do this. `e` takes the same two arguments as what is used in the parenthesis above:

```
d1 $ e 5 8 $ sound "bd"
```

You can use the parenthesis notation within a single element of a pattern:

```
d1 $ sound "bd(3,8) sn*2"
d1 $ sound "bd(3,8) sn(5,8)"
```

You can also add a third parameter, which 'rotates' the pattern so it starts on a different step:

```
d1 $ sound "bd(5,8,2)"
```

You can also use the `e` function to apply a Euclidean algorithm over a complex pattern, although the structure of that pattern will be lost:

```
d1 $ e 3 8 $ sound "bd*2 [sn cp]"
```

In the above, three sounds are picked from the pattern on the right according to the structure given by the `e 3 8`. It ends up picking two `bd` sounds, a `cp` and missing the `sn` entirely.

As a bonus, it is possible to pattern the parameters within the parenthesis, for example to alternate between 3 and 5 elements:

```
d1 $ sound "bd([5 3]/2,8)"
```

These types of sequences use “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 (http://cgm.cs.mcgill.ca/~godfried/publications/banff.pdf) by Toussaint. Some examples from this paper are included below, including rotation in some cases.

- (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.
- (3,5,2) : 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.
- (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).
- (7,12) : A common West African bell pattern.
- (7,16,14) : A Samba rhythm necklace from Brazil.
- (9,16) : A rhythm necklace used in the Central African Republic.
- (11,24,14) : A rhythm necklace of the Aka Pygmies of Central Africa.
- (13,24,5) : Another rhythm necklace of the Aka Pygmies of the upper Sangha.

» Tempo

Tempo

If you've made it this far without changing the tempo in all these examples, then you're probably ready to change it up.

Tidal's core unit of time is cycles per second. It can be set with the `cps` function:

```
cps 1
```

You can execute `cps` just like a pattern (using Shift+Enter in your editor).

`cps` accepts a positive numeric value that can include a decimal:

```
cps 1.5
cps 0.75
cps 10
```

Setting BPM

Tidal also includes a helper function called `bps` to set "beats per second". To set beats-per-minute, call `bps` with your bpm value, divided by 60:

```
-- sets a tempo of 170 BPM:
bps (120/60)

Or you might want to divide it by 120 instead, to create a pattern twice as
long (or half the speed, depending on how you think about it:

-- sets a tempo of 100 BPM:
bps (100/120)
```

» The Run Function

Run

There is a special utility function called `run` which will return a pattern of integers up to a specified maximum. You can use `run` with effects to aid in automatically generating a linear pattern:

```
d1 $ sound "arpy*8" # up (run 8)
d1 $ sound "arpy*8" # speed (run 8)
```

In the above we're specifying the number of sounds twice - in the `sound` pattern as well as the `up` or `speed` pattern. There's actually a neat way of only having to specify this once, simply by switching them round, so the effect parameter is on the left:

```
d1 $ up (run 8) # sound "arpy"
```

This works because TidalCycles always takes the structure of a pattern from the parameter that's on the left. We usually want the structure to come from the `sound` parameter, but not always.

Because `run` returns a pattern, you can apply functions to its result:

```
d1 $ sound "arpy*8" # up (every 2 (rev) $ run 8)
```

For a more practical example of using `run` , read below about selecting samples from folders.

» (Algorithmically) Selecting Samples

Sample Selection

The `sound` parameter we've been using up to now can actually be broken into two seperate parameters, making it easy to select samples with a pattern. These parameters are `s` that gives the name of the sample set, and `n` which gives the number of the sample within that set. For example, the following two patterns do exactly the same:

```
d1 $ sound "arpy:0 arpy:2 arpy:3"
d1 $ n "0 2 3" # s "arpy"
```

It's possible to break the `sound` parameter into two different patterns, namely `s` that gives the name of the sample set, and `n` which gives the index of the sample within that set. For example, the following two patterns are the same:

```
d1 $ sound "arpy:0 arpy:2 arpy:3"
d1 $ n "0 2 3" # s "arpy"
```

This allows us to separate the sample folder name from the index inside the folder, possibly with surprising results!

There is also special function called `samples` that lets you do the same using the `sound` parameter.

```
d1 $ sound $ samples "drum*4" "0 1 2 3"

-- the code above equals this:
d1 $ sound "drum:0 drum:1 drum:2 drum:3"
```

Whether you use `n` and `s` together, or `sound` with `samples` is up to you, although you might find the former to be more flexible.

Remember the `run` function? Since `run` generates a pattern of integers, it can be used with `n` to automatically "run" through the sample indices of a folder:

```
d1 $ n (run 4) # s "drum"
d1 $ sound $ samples "drum*4" (run 4) -- or with samples
```

And of course you can specify a different pattern of sample names:

```
d1 $ s "drum arpy cp hh" # n (run 10)
```

Again, by swapping the order of the `s` and `n` parameters, you can hear the difference between taking the structure from one or the other:

```
d1 $ n (run 10) # s "drum arpy cp hh"
```

> NOTE: if you specify a run value that is greater than the number of samples in a folder, then the higher number index will "wrap" to the beginning of the samples in the folder (just like with the colon notation).

You might sometimes see the samples function wrapped in parenthesis:

```
d1 $ sound (samples "drum arpy cp hh" (run 10))
```

» Combining Types of Patterns

Combining Patterns

Ok, remember when we started adding effects:

```
d1 $ sound "bd sn drum arpy" # pan "0 1 0.25 0.75"
```

What we're actually doing in the code above is *combining two patterns together*: the `sound` pattern, and the `pan` pattern. The special pipe operators (`|=|, |+|, |-|, |*|, |/|`), allow us to combine two patterns. Remember that `#` is shorthand for `|=|` .

We can actually swap sides and it sounds the same:

```
d1 $ pan "0 1 0.25 0.75" # sound "bd sn drum arpy"
```

As we touched on earlier, the main thing to know when combining patterns like this is that the left-most pattern determines the rhythmic structure of the result. Removing one of the elements from the `pan` pattern on the left results in a cycle with three samples played:

```
d1 $ pan "0 1 0.25" # sound "bd sn drum arpy"
```

In the code above, the `pan` pattern determines the rhythm because it is the left-most pattern. The `sound` pattern now only determines what samples are played at what time. The sound pattern gets *mapped* onto the pan pattern.

You might be wondering how TidalCycles decides which sound values get matched with which pan values in the above. (If not, there is no need to read the rest of this paragraph just now!) The rule is, for each value in the pattern on the left, values from the right are matched where the start (or *onset*) of the left value, fall within the timespan of the value on the right. For example, the second `pan` value of `1` starts one third into its pattern, and the second `sound` value of `sn` starts one quarter into its pattern, and ends at the halfway point. Because the former onset (one third) falls inside the timespan of the latter timespan (from one quarter until one half), they are matched. The timespan of `arpy` doesn't contain any onsets from the `pan` pattern, and so it doesn't match with anything, and isn't played.

The rule described above may seem like a lot to keep in mind while composing patterns, but in practice there is no need. Our advice is to not worry, write some patterns and get a feel for how they fit together.

Anyway, this composition of pattern parameters allows us to do some unique things:

```
d1 $ up "0 0*2 0*4 1" # sound "[arpy, bass2, bd]"
```

Above, the `sound` pattern is merely specifying three samples to play on every note. Both the rhythm and pitch of these notes is defined by the `up` pattern.

» Oscillation with Continuous Patterns

Continuous Patterns

So far we've only been working with *discrete* patterns, by which we mean patterns which containing events which begin and end. Tidal also supports *continuous* patterns which instead vary continually over time. You can create continuous patterns using functions which give sine, saw, triangle, and square waves:

```
d1 $ sound "bd*16" # pan sine1
```

The code above uses the `sine1` function to generate a sine wave oscillation of values between 0 and 1 for the `pan` values, so the bass drum moves smoothly between the left and right speakers.

In addition to `sine1`, there is also a `sine` function. What is the difference?

- `sine` produces values between -1 and 1
- `sine1` produces values between 0 and 1

Thus, the "1" suffix means only positive values.

In addition to the `sine` / `sine1` functions, Tidal also has `saw` / `saw1`, `tri` / `tri1`, and `square` / `square1`.

Just like discrete patterns, you can control the speed of continuous patterns with `slow` or `density`:

```
d1 $ sound "bd*16" # pan (slow 8 $ saw1)
d1 $ sound "bd*8 sn*8" # pan (density 1.75 $ tri1)
d1 $ sound "bd*8 sn*8" # speed (density 2 $ tri)
```

You can also combine them in different ways:

```
d1 $ sound "bd*16" # pan (slowcat [sine1, saw1, square1, tri1])
d1 $ sound "sn:2*16" # (speed $ scale 0.5 3 sine1) |*| (speed $ slow 4 saw1)
```

Scaling Oscillation

You can tell the oscillation functions to scale themselves and oscillate between two values:

```
d1 $ sound "bd*8 sn*8" # speed (scale 1 3 $ tri1)
d1 $ sound "bd*8 sn*8" # speed (slow 4 $ scale 1 3 $ tri1)
```

You can also scale to negative values, but make sure to wrap negative values in parens (otherwise the interpreter thinks you're trying to subtract `2` from something):

```
d1 $ sound "bd*8 sn*8" # speed (scale (-2) 3 $ tri1)
```

This technique works well for a slow low-pass filter cutoff:

```
d1 $ sound "hh*32" # cutoff (scale 300 1000 $ slow 4 $ sine1) # resonance "0.4"
d1 $ sound "hh*32" # cutoff (scale 0.001 0.1 $ slow 4 $ sine1) # resonance "0.1"
```

> NOTE 1: If you're using SuperDirt to produce sound (the default install choice), `cutoff` is specified in Hertz, as in the first example above. If you're using the older "classic" dirt, then you will instead need to specify cutoff between 0 and 1, as in the second example.

> NOTE 2: Despite the fact that the oscillator functions produce continuous values, you still need to map them to discrete sound events.

» Rests

Rests

So far we have produced patterns that keep producing more and more sound. What if you want a rest, or gap of silence, in your pattern? You can use the “tilde” `~` character to do so:

```
d1 $ sound "bd bd ~ bd"
```

Think of the `~` as an ‘empty’ step in a sequence, that just produces silence.

» Polymeters

Polymeter

We talked about *polyrhythms* earlier, but Tidal can also produce *polymeter* sequences. A polymeter pattern is one where two patterns have different sequence lengths, but share the same pulse or tempo.

You use curly brace syntax to create a polymeter rhythm:

```
d1 $ sound "{bd hh sn cp, arpy bass2 drum notes can}"
```

The code above results in a five-note rhythm being played at the pulse of a four-note rhythm. If you switch the groups around, it results in a four-note rhythm over a five-note rhythm:

```
d1 $ sound "{arpy bass2 drum notes can, bd hh sn cp}"
```

Sometimes you might want to create an odd polymeter rhythm without having to explicitly create a base rhythm. You *could* do this with rests:

```
d1 $ sound "{~ ~ ~ ~, arpy bass2 drum notes can}"
```

But a more efficient way is to use the `%` symbol after the closing curly brace to specify the number of notes in the base pulse:

```
d1 $ sound "{arpy bass2 drum notes can}%4"

-- the above is the same as this:
d1 $ sound "{~ ~ ~ ~, arpy bass2 drum notes can}"
```

If “polymeter” sounds a bit confusing, there’s a good explanation here: http://music.stackexchange.com/questions/10488/polymeter-vs-polyrhythm (http://music.stackexchange.com/questions/10488/polymeter-vs-polyrhythm)

» Shifting Time

Shifting Time

You can use the `~>` and `<~` functions to shift patterns forwards or backwards in time, respectively. With each of these functions, you can specify an amount, in cycle units.

```
d1 $ (0.25 <~) $ sound "bd*2 cp*2 hh sn"
d1 $ (0.25 ~>) $ sound "bd*2 cp*2 hh sn"
```

The above code shifts the patterns over by one quarter of a cycle.

You can hear this shifting effect best when applying it conditionally. For example, the below shifts the pattern every third cycle:

```
d1 $ every 3 (0.25 <~) $ sound "bd*2 cp*2 hh sn"
d1 $ every 3 (0.25 ~>) $ sound "bd*2 cp*2 hh sn"
```

You can shift patterns as little or as much as you’d like:

```
d1 $ every 3 (0.0625 <~) $ sound "bd*2 cp*2 hh sn"
d1 $ every 3 (1000 ~>) $ sound "bd*2 cp*2 hh sn"
d1 $ every 3 (1000.125 ~>) $ sound "bd*2 cp*2 hh sn"
```

However, in the above case every cycle is the same, so you won’t here a difference between shifting it `1` or `1000` cycles.

You can also express time as a fraction, for example `1%4` instead of `0.25`. However, due to the way Haskell’s parser works, you’ll need to put this in parenthesis:

```
d1 $ every 3 ((1%4) <~) $ sound "bd*2 cp*2 hh sn"
```

» Introducing Randomness

Randomness

Tidal can produce random patterns of integers and decimals. It can also introduce randomness into patterns by removing random events.

Random Decimal Patterns

You can use the `rand` function to create a random value between 0 and 1. This is useful for effects:

```
d1 $ sound "arpy*4" # pan (rand)
```

As with `run` and all numeric patterns, the values that `rand` give you can be scaled, for example the below gives random numbers between `0.25` and `0.75`:

```
d1 $ sound "arpy*4" # pan (scale 0.25 0.75 $ rand)
```

Random Integer Patterns

Use the `irand` function to create a random integer up to a given maximum. The most common usage of `irand` is to produce a random pattern of sample indices (similar to `run`):

```
d1 $ s "arpy*8" # n (irand 30)
```

The code above randomly chooses from 30 samples in the “arpy” folder.

> Hairy detail: `rand` and `irand` are actually *continuous* patterns, which in practical terms means they have infinite detail - you can treat them as pure information! As with all patterns they are also deterministic, stateless functions of time, so that if you retriggered a pattern from the same logical time point, exactly the same numbers would be produced. Furthermore, if you use a `rand` or `irand` in two different places, you would get the same ‘random’ pattern - if this isn’t what you want, you can simply shift or slow down time a little for one of them, e.g. `slow 0.3 rand`.

Removing or “Degrading” Pattern events

Tidal has a few ways to randomly remove events from patterns. You can use the shorthand `?` symbol if you want to give an event a 50/50 chance of happening or not on every cycle:

```
d1 $ sound "bd?"
```

In the code above, the “bd” sample has a 50% chance of being played on every cycle.

You can add the `?` after the completion of any event or group in a pattern:

```
d1 $ sound "bd*16?"
d1 $ sound "bd sn? cp hh?"
d1 $ sound "[bd sn cp hh]?"
```

The `?` symbol is shorthand for the `degrade` function. The two lines below are equivalent:

```
d1 $ sound "bd*16?"
d1 $ degrade $ sound "bd*16"
```

Related to `degrade` is the `degradeBy` function, where you can specify the probability (from 0 to 1) that events will be removed from a pattern:

```
d1 $ degradeBy 0.25 $ sound "bd*16"
```

There is also `sometimesBy` , which executes a function based on a probability:

```
d1 $ sometimesBy 0.75 (slow 2) $ sound "bd*16"
```

The code above has a 75% chance of calling `slow 2` on every event in the pattern.

There are other aliases for `sometimesBy` :

```
sometimes = sometimesBy 0.5
often = sometimesBy 0.75
rarely = sometimesBy 0.25
almostNever = sometimesBy 0.1
almostAlways = sometimesBy 0.9

d1 $ sometimes (density 2) $ sound "bd*8"
d1 $ rarely (density 2) $ sound "bd*8"
```

» Creating Variation in Patterns

Variation

You can create a lot of cyclic variations in patterns by layering conditional logic:

```
d1 $ every 5 (|+| speed "0.5") $ every 4 (0.25 <~) $ every 3 (rev) $
   sound "bd sn arpy*2 cp"
   # speed "[1 1.25 0.75 -1.5]/3"
```

In addition to `every` you can also use the `whenmod` conditional function. `whenmod` takes two parameters; it executes a function when the remainder of the current loop number divided by the first parameter is less than the second parameter.

For example, the following will play a pattern normally for cycles 1-6, then play it in reverse for cycles 7-8. Then normally again for six cycles, then in reverse for two, and so on:

```
d1 $ whenmod 8 6 (rev) $ sound "bd*2 arpy*2 cp hh*22"
```

» Creating "Fills" and using "const"

Fills

You can think of a “fill” as a change to a regular pattern that happens regularly. e.g. every 4 cycles do “xya”, or every 8 cycles do “abc”.

We've already been using `every` and `whenmod` to do pattern function fills:

```
d1 $ every 8 (rev) $ every 4 (density 2) $ sound "bd hh sn cp"
d1 $ whenmod 16 14 (# speed "2") $ sound "bd arpy*2 cp bass2"
```

However, what if you wanted to conditionally replace the pattern with a new one? You can use the `const` function to completely replace a playing pattern.

Let's start with a trivial example where we use `const` to replace an entire pattern all the time:

```
d1 $ const (sound "arpy*3") $ sound "bd sn cp hh"
```

In the code above, we've completely replaced the "bd sn cp hh" pattern with an "arpy" pattern. `const` specifies the new pattern.

We can conditionally apply `const` using `every` or `whenmod` :

```
d1 $ whenmod 8 6 (const $ sound "arpy(3,8) bd*4") $ sound "bd sn bass2 sn"
d1 $ every 12 (const $ sound "bd*4 sn*2") $ sound "bd sn bass2 sn"
```

» Composing Multi-Part Patterns

Composing_patterns

There are a few ways that you can compose new patterns from multiple other patterns. You can concatenate or "append" patterns in serial, or you can "stack" them and play them together in parallel.

Concatenating patterns in serial

You can use the `cat` function to add patterns one after 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. The more patterns you add to the list, the faster each pattern will be played so that all patterns can fit into a single cycle.

```
d1 $ cat [sound "bd sn:2" # vowel "[a o]/2",
          sound "casio casio:1 casio:2*2",
          sound "drum drum:2 drum:3 drum:4*2"
         ]
```

`slowcat` will maintain the original playback speed of the patterns:

```
d1 $ slowcat [sound "bd sn:2" # vowel "[a o]/2",
              sound "casio casio:1 casio:2*2",
              sound "drum drum:2 drum:3 drum:4*2"
             ]
```

`slowcat` is a great way to create a linear sequence of patterns (a sequence of sequences), giving a larger form to multiple patterns.

There's also `randcat`, which will play a random pattern from the list.

Playing patterns together in parallel

The `stack` function takes a list of patterns and combines them into a new pattern by playing all of the patterns in the list simultaneously.

```
d1 $ stack [
  sound "bd bd*2",
  sound "hh*2 [sn cp] cp future*4",
  sound (samples "arpy*8" (run 16))
]
```

This is useful if you want to apply functions or effects on the entire stack:

```
d1 $ every 4 (slow 2) $ whenmod 5 3 (# speed "0.75 1.5") $ stack [
  sound "bd bd*2",
  sound "hh*2 [sn cp] cp future*4",
  sound (samples "arpy*8" (run 16))
] # speed "[[1 0.8], [1.5 2]*2]/3"
```

» Truncating samples with "cut"

Cut

So far, all of our examples have used short samples. However, maybe you've experimented with some long samples. Maybe you've noticed that really long samples can cause a lot of bleed and unwanted sound.

With Tidal's `cut` effect, you can "choke" a sound and stop it from playing when a new sample is triggered.

Consider the following example where we have a pattern of "arpy" sounds, played at a low speed, so there is a lot of bleed into each sample:

```
d1 $ sound $ samples "arpy*8" (run 8) # speed "0.25"
```

We can stop this bleed by using `cut` and assigning the pattern a cut group of "1":

```
d1 $ sound $ samples "arpy*8" (run 8) # speed "0.25" # cut "1"
```

No more bleed!

You can use any number for the cut group.

Cut groups are global, to the Tidal process, so if you have two Dirt connections, use two different cut group values to make sure the patterns don't choke each other:

```
d1 $ sound $ samples "arpy*8" (run 8) # speed "0.25" # cut "1"
d2 $ sound $ samples "bass2*6" (run 6) # speed "0.5" # cut "2"
```

This also works in a `stack` :

```
d1 $ stack [
   sound $ samples "arpy*8" (run 8) # speed "0.25" # cut "1",
   sound $ samples "bass2*6" (run 6) # speed "0.5" # cut "2" ]
```

» Transitions Between Patterns

Transitions

Changing the pattern on a channel takes effect (almost) immediately. This may not be what you want, especially when performing live!

That's why Tidal allows you to choose a transition that will introduce another pattern, eventually replacing the current one.

For every Dirt channel, there's a transition channel that accepts a transition function and a new pattern.

So instead of directly sending the new pattern to d1, we'll send it to the corresponding transition channel t1 and give it a nice transition function:

```
d1 $ sound (samples "hc*8" (iter 4 $ run 4))
t1 anticipate $ sound (samples "bd(3,8)" (run 3))
```

To transition from here, simply change the pattern within t1, and in this case also change the transition function:

```
t1 (xfadeIn 16) $ sound "bd(5,8)"
```

The above will fade over 16 cycles from the former pattern to the given new one.

Apart from anticipate and xfadeIn there are a lot more transition functions e.g. some that will force you to keep changing your patterns to avoid repetitive performances…

» Samples

Samples

If you're using SuperDirt, all the default samples can be found in the `Dirt-Samples` folder - you can open it by running `Quarks.gui` in SuperCollider, clicking on "Dirt-Samples" and then "open folder". If you're using classic dirt, look in its `samples` subfolder. Here's some you could try:

```
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
```

Each one is a folder containing one or more wav files. For example when you put `bd:1` in a sequence, you're picking up the second wav file in the `bd` folder. If you ask for the ninth sample and there are only seven in the folder, it'll wrap around and play the second one.

If you want to add your own samples, just create a new folder in the samples folder, and put `wav` files in it.

» Synths

Synths

SuperDirt is created with SuperCollider, a fantastic synthesis engine and language with huge sonic possibilities. You can trigger custom SuperCollider synths from TidalCycles in much the same way as you trigger samples. For example:

```
d1 $ midinote "60 62*2" # s "supersaw"
```

The above plays note 60 and 62 of the MIDI scale, using the `midinote` parameter. You can alternatively specify notes by name, using `n` :

```
d1 $ n "c5 d5*2" # s "supersaw"
```

You can also specify note numbers with `n` , but where `0` is middle c (rather than `60` with `midinote`).

```
d1 $ n "0 5" # s "supersaw"
```

The default sustain length is a bit long so the sounds will overlap, you can adjust this using the `sustain` parameter

```
d1 $ n "c5 d5*2" # s "supersaw" # sustain "0.4 0.2"
```

Many example synths can be found in the `default-synths.scd` file in the `SuperDirt/synths` folder. These include:

- a series of tutorials: `tutorial1`, `tutorial2`, `tutorial3`, `tutorial4`, `tutorial5`
- examples of modulating with the cursor or sound input: `pmsin`, `in`, `inr`
- physical modeling synths: `supermandolin`, `supergong`, `superpiano`, `superhex`
- a basic synth drumkit: `superkick`, `superhat`, `supersnare`, `superclap`, `super808`
- four analogue-style synths: `supersquare`, `supersaw`, `superpwm`, `supercomparator`
- two digital-style synths: `superchip`, `supernoise`

To find the SuperDirt folder, simply run `Quarks.folder` in supercollider. The full folder location should appear in the postwindow (which is usually in the bottom right).

Many of the above synths accept additional Tidal Parameters or interpret the usual parameters in a slightly different way. For complete documentation, see `default-synths.scd`, but here are some examples to try:

```
d1 $ jux (# accelerate "-0.1") $ s "supermandolin*8" # midinote "[80!6 78]/8"
  # sustain "1 0.25 2 1"
```

```
d1 $ midinote (slow 2 $ fmap ((+50) . (*7)) $ run 8) # s "supergong" # decay "[1 0.2]/4"
  # voice "[0.5 0]/8" # sustain (slow 16 $ scale 5 0.5 $ saw1)
```

```
d1 $ sound "superhat:0*8" # sustain "0.125!6 1.2" # accelerate "[0.5 -0.5]/4"
```

```
d1 $ s "super808 supersnare" # n (irand 5)
  # voice "0.2" # decay "[2 0.5]/4" # accelerate "-0.1" # sustain "0.5" # speed "[0.5 2]/
4"
```

```
d1 $ n (slow 8 "[[c5 e5 g5 c6]*4 [b4 e5 g5 b5]*4]") # s "superpiano"
  # velocity "[1.20 0.9 0.8 1]"

d2 $ n (slow 8 $ fmap (+24) "[[c4,e4,g4,c5]*4 [e4,g4,b5,e5]*4]") # s "superpiano"
  # velocity (slow 8 $ scale 0.8 1.1 sine1) # sustain "8"
```

```
d1 $ n "[c2 e3 g4 c5 c4 c3]/3" # s "[superpwm supersaw supersquare]/24" # sustain "0.5"
  # voice "0.9" # semitone "7.9" # resonance "0.3" # lfo "3" # pitch1 "0.5" # speed "0.25
 1"
```

```
d1 $ every 16 (density 24 . (|+| midinote "24") . (# sustain "0.3") . (# attack "0.05"))
  $ s "supercomparator/4" # midinote (fmap (+24) $ irand 24)
  # sustain "8" # attack "0.5" # hold "4" # release "4"
  # voice "0.5" # resonance "0.9" # lfo "1" # speed "30" # pitch1 "4"
```

```
d1 $ n "[c2 e3 g4 c5 c4 c3]*4/3" # s "superchip" # sustain "0.1"
  # pitch2 "[1.2 1.5 2 3]" # pitch3 "[1.44 2.25 4 9]"
  # voice (slow 4 "0 0.25 0.5 0.75") # slide "[0 0.1]/8" # speed "-4"

d2 $ every 4 (echo (negate 3/32)) $ n "c5*4" # s "supernoise"
  # accelerate "-2" # speed "1" # sustain "0.1 ! ! 1" # voice "0.0"
```

```
d1 $ s "supernoise/8" # midinote (fmap (+30) $ irand 10) # sustain "8"
 # accelerate "0.5" # voice "0.5" # pitch1 "0.15" # slide "-0.5" # resonance "0.7"
 # attack "1" # release "20" # room "0.9" # size "0.9" # orbit "1"
```

This is all quite new and under ongoing development, but you can read about modifying and adding your own synths to SuperDirt at its github repository (https://github.com/musikinformatik/SuperDirt).