A Mercury live coding recap

Table des matières

Please, note that some examples or settings have been modified from the original tutorials.
@beryann.parker

// https:linktr.ee/beryann.parker

//Sample :

//a sound with time()

```
new sample harp_up time(1)

new sample kick_909 time(1/4)
```

// Try for example: 1/8, 3/16, 1/6, 5/32

ex :

```
new sample kick_909 time(3/8)
new sample hat_909 time(1/8)
new sample snare_909 time(1/2)
```

ex with offsets :

```
new sample kalimba_g time(1)
new sample kalimba_a time(1 3/16)
new sample kalimba_e time(1 3/8)
```

//List of beats :

//with a lists of booleans(true/false : 1 or O) :

```
list hatBeat [1 0 1 0 0 1 0]
new sample hat_909 time(1/16) play(hatBeat)

list tablaBeat [1 0 1 1 0]
new sample tabla_mid time(1/16) play(tablaBeat)
```

//List of sounds/beats :

```
list drumSounds [kick_909 hat_909 snare_909 hat_909]
new sample drumSounds time(1/8)
```

//Or :

```
list sounds [hat_909 snare_909 hat_909 tabla_mid tabla_hi]
list beat [1 0 1 0 1 1]
new sample sounds time(1/16) play(beat)
```

//Polyrhythm :

```
new sample kalimba_e time(1)
new sample kalimba_g time(1/3)
new sample kalimba_cis time(1/2)
new sample kalimba_a time(1/4)
```

//Chance :

//percentage of chance that the sound will play with « play() »

```
new sample hat_808 time(1/16) play(0.6)
```

//A list of chance :

```
list hatBeat [1 0.1 0.9 0.2]
new sample hat_909 time(1/16) play(hatBeat)
```

// plays 100%, 10%, 90% and 20% in a sequences

// Speed

//affects the playback of speed

```
list pitch [1 0.5 0.25 1 2]
new sample pluck_a time(1/8) speed(pitch)
```

//Soundscapes :

//with different time() and speed()

```
new sample gong_hi time(4) speed(0.3)
new sample bowl_hi time(3) speed(0.125)
new sample drone_cymbal time(7) speed(0.25)
```

//Shape :

//fade-in/fade-out in milliseconds

```
new sample harp_down time(1/3) shape(1 100)
new sample bowl_hi time(1) shape(1000 2)
```

//Or with a shape-list :

```
list fadeIn [2 2 500]
list fadeOut [20 20 100 20 500]
new sample harp_down time(1/4) shape(fadeIn fadeOut)
```

//Start Position :

//(0 to 1), (0.5 is halfway through the sound)

```
list positions [0.7 0.2 0.3 0.2 0.5 0.1]
```

```
new sample choir_01 time(1/8) start(positions) shape(1 100 1)
```

//Panning :

// from left(-1) to the center(0) to the right(1) or pan(random)

```
new sample violin_c time(1) pan(-1)
new sample pluck_e time(1 3/16) pan(1)
new sample bamboo_g time(1 4/16) pan(0)
new sample kick_909 time(1/4) pan(random)
```

//or with a list panning :

```
list panning [-1 -0.75 -0.5 -0.25 0 0.25 0.5 0.75 1]
new sample hat_808 time(1/16) pan(panning)
```

//Volume :

//A gain of 0 is off, a gain of 1 is the original volume of the soundfile, a gain of 2 is 2x louder (be careful!)

```
new sample scrape gain(0.6)
```

//Or a list : list dynamics [0.1 0.4 0.9 0.3 0.15 0.8 0.7 0.1]

//Time divide :

//timediv() allows you to subdivide the timing into smaller portions based on a list of integer numbers

```
new sample hat_808 time(1/8) timediv([1 1 2 1 3 1 4 1])
```

//Humanize :

//human() the argument is a delay range in milliseconds with which the trigger of the sound is randomly delayed or rushed

```
new sample hat_808 time(1/16) human(10)
new sample [kick_808_dist snare_808] time(1/2) human(100)
```

//Note and tune :

//it is also possible to change the pitch of a sample with the note() function, similar to the note function in a synth. In the case the sample is not recorded at c4 (midi 60, 261.626Hz) you'll have to set the tune() in order to get the correct pitch transposition.

```
new sample kalimba_a time(1/4) note([0 7 5 3] 2) tune(a3)
```

//Name instruments :

//instruments can have a name. Giving the instrument a name gives two advantages: Parameters can be set for the instrument on another line with `set <name>`, re-evaluating code will transfer the current count to the new instrument.This will preserve continuity while sequencing (long) lists, instead of hearing the counter reset every time code is evaluated

```
list notes spread(13 0 24)
list rhythm euclid(16 13)
new synth saw note(notes 0) time(1/16) play(rhythm) name(synthy)
set synthy fx(filter low 3000 0.3) fx(degrade 0.4) fx(delay)
```

//Synthetiser :

//new synth with time() gain() random() and lists

```
new synth saw time(1/8) gain(0.8) pan(random)
new synth sine
new synth square
new synth triangle
```

// Note :

//to choose a note to play for the synth we use the note() function. This function expects a number as a semitone (half-step) on the piano-keyboard. The numbers can be positive or negative.

```
new synth saw time(1) note(0)
new synth saw time(1 1/16) note(3)
new synth saw time(1 3/16) note(7)
```

//Melody :

```
list melody [0 7 12 19 15 12 5 3 7 5 2]
new synth saw time(1/16) note(melody)
```

//Octave :

//with a second argument in the note() function we can choose an octave offset. Stepping up one octave is the same as adding 12 to the semitone.

```
list theMelody [0 7 5 3 7 5 2]
list theOctaves [0 0 0 1 1 1 2 2 2]
new synth saw time(1/16) note(theMelody theOctaves)
```

//Scales :

//you hear some notes twice, because some values between 0-12 don't fit in the scale, so they're shifted to the closest value that does belong to the scale
set scale major

```
list melody [0 1 2 3 4 5 6 7 8 9 10 11 12]
new synth saw time(1/16) note(melody)
```

//More scales :

```
list scales scaleNames()
print scales

set scale romanian_minor
list melody [12 11 10 9 8 7 6 5 4 3 2 1 0]
new synth square time(1/16) note(melody 1)
```

//Root :

//a scale starts at a specified note. This note is the root.By default this root is a 'c', but you can choose other roots like d, f#, gb etc.

```
set tempo 97
set scale minor_pentatonic gb
list melody [12 11 10 9 8 7 6 5 4 3 2 1 0]
new synth square time(1/16) note(melody 1)
```

// Synth Shape :

//the shape() method for the synth allows us to create a longer or shorter sound with a fade-in/fade-out. We can also set the shape(off) resulting in a never stopping sound.

```
new synth triangle time(1/4) shape(1 1/16) note(0 2)
new synth triangle time(1) shape(off)
new synth triangle time(1/2) shape(1/4 10) note(0 1)
```

//Synth Shape list :

// here the synth has an attack of 2 ms and a release depending on the list

```
list fadeOut [20 20 100 20 200 500]
new synth saw time(1/16) shape(2 fadeOut)
```

// Super Synth :

//the super synth is a synth that uses multiple waveforms at the same time with a small detuning to create a more richer sound

```
new synth saw time(1/16) super(2.78 10) shape(1 80 1)
```

//Filter :

//this filter removes high frequencies above 1200Hz (lowpass) and has a little resonance on the cutoff frequency. Resonance results in a whistling sounding effect

```
new synth saw time(1/16) fx(filter low 1200 0.6) shape(1 100)
```

//Highpass :

```
new synth saw time(1/16) fx(filter high 1200 5000) shape(1 100)
```

//Filter modulation1 :

//make sure the cutoffs are values between 50 and 18000 (Hz) and the resonance are values between 0 and 1

```
list cutoffs [200 400 700 1000]
list qs [0.3  0.5  0.8]
new synth saw time(1/16) fx(filter low cutoffs qs) shape(1 80)
```

//Filter modulation2 :

//we can make even more interesting filter modulations by adding a few extra arguments to the filter fx function. When doing that we can choose a low and high frequency range
between which a sinewave, sawtooth up or sawtooth down can modulate.The modulation time is designed by the division argument. arguments:
// fx(filter <type> <division> <low> <high> <resonance> <direction> <slope-curve>)

```
list nts spread(5 0 12)
new synth saw time(1/2) note(nts) shape(off) fx(filter low 1/1 100 4000 0.8)
```

// this filter modulates up and down once per bar between 100 and 4000 Hz with resonance 0.8

```
new synth saw time(1/16) note(nts 1) shape(off) fx(filter low 1/16 200 6000 0.6 0 0.05)
```

// this filter modulates down every 1/16 note from 6000 to 200 Hz with resonance of 0.6
// the 0.2 determins the slope curve the filter goes up/down with (try 0.05 and 1 to hear the difference)

//Note slide :

// With the 'slide()' function we can slide the pitch from one note to another (called portamento) in a specified amount of time (division or milliseconds)

```
set tempo 123
set scale minor b
list bassLine repeat([3 2 0 -7] 4)
new synth saw note(bassLine 0) shape(2 250) time(1/4) slide(1/2)
```

//PolySynth :

// The polySynth is a polyphonic synthesizer, this allows you to play overlapping notes or notes at the same time to generate chord progressions

```
set tempo 80
set scale major c
// a 2-dimensional list of notes plays all the notes at the same time
list chord [ [0 4 5 7 8 10 12] ]
new polySynth sine note(chord 2) time(1/2) shape(1 1/4)
```

//Chord Progression :

// With the chordsFromNumerals function you can generate a 2-dimensional list containing the notes for the chords you specify. Chords are numbered I to VII. Adding an 'm' makes them minor.It is also possible to add the 7, 9, 11 or 13th. We can set the scale to none to allow all the notes to be played

```
set tempo 100
set scale none
// a chord progression in the style of the 4-chord-song
list progression chordsFromNumerals([I IV V VIm])
// see in the console that the result is a 2d-list
print progression
new polySynth sine note(progression 2) time(1/1) shape(1 3/4 1/4)
```

// PolySample :

// is a polyphonic sample player, allowing you to play overlapping sounds and also play notes with those sounds.

```
set tempo 110
set scale none
```

// here the chord progression is repeated 4 times and flattened into a 1d list

```
list progression flat(repeat(chordsFromNumerals([I7 IV7 V7 VIm7]) 4))
```

// With the sample it is important to have voice stealing 'on' when the shape is turned off. Otherwise all the voices will stay busy because there is no "fade-out"

```
new polySample bamboo_g name(polySamp)
set polySamp note(progression 2) tune(67) time(1/16) shape(off) steal(on)
```

//FX Delay :

// The Delay effect creates echos of the sound that slowly fade away in a rhythmic pattern

// The delay can have arguments for the left/right delaytimes in division. A decimal-point value 0-1 determines the amount of fade-out. Try different delaytimes: 5/16, 3/4, 1/32, 1/12.Try changing the values in the function to hear what happens. It is also possible to modulate the parameters with lists

```
set tempo 100
new sample piano_g time(2) fx(delay 3/16 2/16 0.8)
```

//FX Distortion :

// fx(drive amount) - alias: distort

```
list amounts spread(5 10 50)
new loop house time(1) gain(0.6) fx(drive amounts)
```

//FX Reverb :

// The first decimal value determines the balance between original (dry) and reverb sound (wet). The second argument is the reverb time in seconds

```
new synth saw note(0 0) time(1) shape(1 1/4 1) fx(reverb 0.3 10)
new synth saw note([0 3 7] 1) time(1/4) shape(1 1/32) fx(reverb 0.5 3)
```

//FX Shift :

// With the pitch shifter effect you can change the pitch of a sound in semitones. Allowing you to make melodies with a single sample. The first argument is the pitch shift in semitones which is also mapped according to the scale that is set

```
set scale harmonic_minor c
new sample chimes time(1/8) fx(shift [0 3 2 -1 7 5]) shape(500)
```

//Squash :

// The squash effect is also a type of distortion effect that compresses the sound a bit softer in the beginning but with higher values introduces some crunch also

// fx(squash amount)

```
list squashes [4 15 10 4 40]
new sample kick_909 time(1/16) fx(squash squashes) shape(1 500)
```

//FX All :

// It is possible to give multiple sounds the same effects by using the `set all` code. Note that this actually creates individual effects for all the sounds. Meaning they can all have individual modulation speed when using a list with arguments

```
new sample kick_909 time(1/2)
new sample hat_909 time(1/4 1/8)
new synth saw time(3/16) note(0 1) shape(1 1/8) super(3 0.0423)
set all fx(degrade)  fx(reverb 0.5 2)
```

//FX Filter :

// The filter is a simple filter where the first argument sets the type of the filter such as low(pass), band(pass) and hi(pass). The second argument sets the cutoff frequency with a number or list. The third argument changes the resonance (Q). NOTE: cutoff ramptime in milliseconds is deprecated, now use modulation filter or triggerFilter !!!

```
new synth saw shape(off) time(1/4) fx(filter low [8000 400 2000 200] [0.2 0.5 0.8])
```

//FX Trigger Filter :

// The trigger filter is a filter where the cutoff frequency is driven by an envelope (shape). The filter is triggered every time the note is played. Set the filter type: lowpass, highpass or bandpass. Set the attack and release times in milliseconds or relative to the tempo. Set the high and low frequency points between which the envelope moves.

```
new synth saw note(0 0) time(1/1) shape(off) fx(triggerFilter low 1/4 3/4 5000 100)
```

//FX Compressor :

// A compressor allows you to reduce the dynamic range of a synth or sample.The signals volume gets reduced by a specified ratio when it crosses the threshold. Set the threshold in dBFS (-100 to 0, default = -30).Set the compression ratio (20 to 0, default = 6). Set the attack and release time in milliseconds or relative to tempo

```
new loop amen time(1) fx(compress -20 5 5 50)
```

//FX Degrade :

// A Downsampling Chiptune effect. Downsamples the signal by a specified amount resulting in a lower samplerate, making it sound more like 8bit/chiptune

// fx(degrade amount) - alias: chip

```
list degrades spreadF(16 0.5 0.9)

new sample choir_o time(1/4) shape(off) fx(degrade degrades)
```

//FX Chorus/Double :

// A Chorus and Doubling effect. The Chorus creates 2 copies of the sound. One on the left and one right. These copies slowly change in time creating modulating effects that make the sound more wider. Use the parameters modulation time (division), depth (ms) and wetdry (0-1)

```
new synth saw note(spread(5 0 12) 1) time(1/4) shape(1 1/2) fx(chorus 8/1 40 0.5)
```

// The double effect does the same but removes the original from the center (wet = 1)

// Giving the effect of hearing 2 instruments.

```
new sample snare_909 time(1/2 1/4) speed(0.6) fx(double)
```

//Algorithmic Composition :

// This chapter discusses the algorithmic processes that can be used in Mercury. Mercury is heavily inspired by the composition technique Serialism. This technique approaches every musical parameter (rhythm, pitch, dynamics, etc) as an individual sequence of numbers. These sequences can then be transformed in many ways to extend and generate new musical material.

With algorithmic processes (functions) we can generate or transform lists. With the lists we can control the parameters in functions of instruments. By combining various list functions we can create more complex outputs.This chapter will cover most of the algorithmic methods, but many unique combinations are up to you to discover!

// A list is a collection of items (numbers, words) that has a unique name

```
list myValues [1 2 3.14 6.18 kick_909 snare_808]
```

// It is possible to print the content of a list. This is useful to view the result of list functions

```
print myValues
```

//Spread :

// Generate a list of n-length containing whole numbers starting at x and ending at y (excluding y)

//spread(size low high) useful for melodies and ascending/descending number sequences for modulation of for example note-length

```
list melody spread(7 0 12)
list length spread(7 500 20)
print melody length
```

```
[0 1 3 5 6 8 10]
[431 362 294 225 157 88 20]
```

```
new synth saw note(melody 1) time(1/16) shape(1 length)
```

//Spread Inclusive :

// Generate a list of n-length containing whole numbers starting at x and ending at y (including y)

// spreadInclusive(size low high)

// Useful for melodies and ascending/descending number sequences for modulation of for example note-length

```
list melody spreadInclusive(5 0 12)
list length spreadInclusive(7 500 20)
print melody length
```

```
[0 3 6 9 12]
[500 420 340 260 180 100 20]
```

```
new synth saw note(melody 1) time(1/16) shape(1 length)
```

//Spread Float :

// Generate a list of n-length containing floating-point numbers starting at x and ending at y (excluding y)

// spreadFloat(size low high) (alias: spreadF)

// spreadInclusiveFloat(size low high) (alias: spreadInclusiveF)

// Useful for parameters that are based on floating-point numbers such as sequences for modulation of for example gain and panning

```
list volume spreadF(16 0 1)

list panning spreadInclusiveF(7 -1 1)

print volume panning
```

```
[0 0.0625 0.125 0.1875 0.25 0.3125 0.375 0.4375 0.5 0.5625 0.625 0.6875 0.75 0.8125 0.875
0.9375]
[-1 -0.6666666666666667 -0.33333333333333337 0 0.33333333333333326 0.6666666666666665 1]
```

```
new synth saw note(0 1) time(1/16) shape(1 80) pan(panning) gain(volume)
```

//Fill :

// Fill a list with values and a number of repetitions. The values are provided in pairs of number, amount.

// fill(value1 amount1 value2 amount2 etc...)

// Useful for creating longer lists with a lot of duplicate values

```
list melody fill(0 4 7 2 3 4 12 2 9 4 3 2 2 2)

print melody

[0 0 0 0 7 7 3 3 3 3 12 12 9 9 9 9 3 3 2 2]

new synth sine note(melody 2) time(1/16)
```

//Euclidian Rhythm :

// Generate a euclidean rhythm. The algorithm evenly spaces n-beats over n-steps, return a list of 1's and 0's. Inspired by Godfried Toussaints famous paper "The Euclidean Algorithm Generates Traditional Musical Rhythms".

// euclidean(steps, beats, rotate) (alias: euclid)

// Useful for generating intricate rhythmical patterns with the play() function

```
set tempo 130

list rhythm1 euclid(16 5)

list rhythm2 euclid(8 5)

print rhythm1 rhythm2

[1 0 0 0 1 0 0 1 0 0 1 0 0 1 0 0]

[1 0 1 0 1 1 0 1]


new sample bongo time(1/8) play(rhythm1)
new sample bowl_mid time(1/8) play(rhythm2)
```

// Hexadecimal Rhythm :

// Hexadecimal beats make use of hexadecimal values (0 - f) that are a base-16 number system. Because one digit in a base-16 number system has 16 possible values (0 - 15) these can be converted to 4 bits that therefore can be seen as groups of 4 16th notes. These hexadecimal values will then represent any permutation of 1's and 0's in a 4 bit number, where 0 = 0 0 0 0, 7 = 0 1 1 1, b = 1 0 1 1, f = 1 1 1 1 and all possible values in between. Useful for generating intricate rhythmical patterns with the play() function

```
list rhythm1 hex('f02c')

list rhythm2 hex('094a')

print rhythm1 rhythm


new sample bongo time(1/16) play(rhythm1)

new sample bongo_lo time(1/16) play(rhythm2)
```

//Fibonacci Numbers :

// Generate a list of Fibonacci numbers F[n] = F[n-1] + F[n-2]. The fibonacci sequence is famous because it's numbers pop up in nature in many places and when you divide any number in the sequence by its previous number it converges towards the golden ratio (phi, 1.618)

// fibonacci(amount)

// fibonacci(amount, starting number)

// Useful as a starting point for melodic content or modulation of parameters

```
list melody fibonacci(9)

list cutoff fibonacci(8 12)

print melody cutoff

[0 1 1 2 3 5 8 13 21]

[144 233 377 610 987 1597 2584 4181]


new synth saw note(melody 1) time(1/8) shape(off) fx(filter cutoff)
```

// Pisano Periods :

// Generate Pisano periods from the Fibonacci sequence. The pisano period is a result of applying a modulo (%) operation on the Fibonacci sequence

// F[n] = (F[n-1] + F[n-2]) mod a. The length of the period differs per modulus value, but the sequence will always repeat.

// Useful as a starting point for melodic content or modulation of parameters

```
list melody pisano(13)

print melody

[0 1 1 2 3 5 8 0 8 8 3 11 1 12 0 12 12 11 10 8 5 0 5 5 10 2 12 1]

new synth sine note(melody 2) time(1/16)
```

// Translate Time :

// Generate 2-dimensional arrays of relative notevalues that can be used as chord information or flattened to generate melodic progressions based on chords

//chordsFromNumerals(list)

// Convert a chord progression from roman numerals to semitones

// alias: makeChords()

```
print chordsFromNumerals([I IIm IVsus2 V7 VIm9])
```

// Convert a chord progression from chordnames to semitones

```
print chordsFromNames([C Dm Fsus2 G7 Am9])

set scale chromatic c

list chords flatten(chordsFromNumerals(repeat([I7 IV7sus2 IIm7 V7] 2)))

print chords

[[0 4 7] [2 5 9] [5 7 0] [7 11 2 5] [9 0 4 7 11]]

[[0 4 7] [2 5 9] [5 7 0] [7 11 2 5] [9 0 4 7 11]]

[0 4 7 10 0 4 7 10 0 0 2 5 9 0 2 5 9 0 7 11 2 5 7 11 2 5]
```

```
new synth saw note(chords 1) time(1/16)
```

// Random :

// Generate a list of n-length containing random values of whole numbers between low and high value (excluding high). Note that any time you evalute the code the random numbers are different. See "randomseed" for a solution to this issue.

// random(size low high)

// Useful to generate random sequences for melodies and any parameter

```
list melody random(8 0 12)
list cutoff random(5 300 4000)
list length random(4 80 300)
print melody cutoff length
[9 2 10 6 11 9 3 7]
[1499 3737 666 1871 1378]
[147 275 237 150]
```

```
new synth saw note(melody 1) time(1/8) shape(1 length) fx(filter cutoff)
```

// Random Seed :

// Set the seed for the Random Number Generator (RNG). A value of 0 sets to unpredictable seeding. Seeding the RNG results in predictable pseudo random numbers that give the same result every time the code is evaluated.

// randomSeed anyValue

// Useful to fix generated random numbers. Try both seeds and hear the difference

```
set randomSeed 4738
set randomSeed 7385
```

```
list melody random(5 0 12)
list cutoff random(5 100 2000)
list rhythm random(16 0 2)

print melody cutoff rhythm

new synth saw note(melody 1) time(1/16) play(rhythm) fx(filter cutoff)
```

//Drunk :

// Generate a list of n-length containing random values of whole numbers between low and high value (excluding high). Every random number is based on the previous number and generated within a specified step-range.

// drunk(size step low high)

```
// Useful to generate random sequences for melodies and any parameter

list melody drunk(16 2 0 12)

list cutoff drunk(16 1000 300 4000)

list length drunk(16 50 80 500)

print melody cutoff length
```

```
[6 7 7 7 6 5 7 8 8 10 9 8 7 9 8 8]

[2000 2487 2259 1661 2014 2223 3121 2214 2952 1979 2550 1782 883 439 899 671]

[295 254 219 220 256 283 307 341 336 384 385 422 418 446 399 430]
```

```
new synth saw note(melody 1) time(1/16) shape(1 length) fx(filter cutoff)
```

//Clave :

// Generate random clave patterns. The output is a binary list that represents a rhythm, where 1's represent onsets and 0's rests. A clave pattern can be found in musical genres such as salsa, mambo, rumba, raggae, raggaeton and more. It is usually defined as an alternation of a beat with 1 or 2 rests

// clave(size)

// Useful to generate random rhythms

```
set tempo 130
```

```
list rhythm1 clave(8)
list rhythm2 clave(8)
print rhythm1 rhythm2

[1 0 0 1 0 1 0 0]
[1 0 1 0 1 0 0 1]

new sample bongo time(1/16) play(rhythm1)
new sample bongo_lo time(1/16) play(rhythm2)
```

//Coin :

// Generate a list of n-length containing random coin tosses .The resulting list only contains 0's and 1's

// coin(size)

// Useful to generate random rhythms

```
set tempo 130
list rhythm1 coin(8)
list rhythm2 coin(8)
print rhythm1 rhythm2

[1 1 1 1 1 1 0 0]
[0 1 1 1 0 1 0 0]

new sample bongo time(1/16) play(rhythm1)
new sample bongo_lo time(1/16) play(rhythm2)
```

//Choose :

// Use Choose to randomly select items from a defined list.Useful to generate random lists with predefined options. For example for melodies or sample selections

// choose(size list)

```
list melody choose(16 [-1 0 2 3 6 7])
list beat choose(8 [kick_808 hat_808 snare_808])
print melody beat
```

```
[-1 -1 3 2 7 -1 -1 -1 3 2 3 2 0 6 3 0]
[kick_808 hat_808 h
```

//Shuffle :

// Shuffle the contents of a list. The shuffle order is also controlled by the random seed. Useful to generate random orders of a predefined list. For example for melodies or sample selections

// shuffle(list) - alias: scramble

```
list melody shuffle([-1 0 2 3 6 7 9 12])
list beat shuffle([kick_808 hat_808 snare_808 tabla_hi_short])
print melody beat
```

```
[0 6 9 7 12 2 -1 3]
[hat_808 snare_808 kick_808 tabla_hi_short]
```

```
new synth sine note(melody 2) time(1/16)
new sample beat time(1/8)
```

//Reverse :

// Reverse the contents of a list. Useful to generate a reversed version of a list like, for example a melody

//Reverse(list) - alias: rev

```
list melody [0 2 3 7 9 12]
list revMelody reverse(melody)
```

```
new synth saw note(melody 1) time(1/4)

new synth saw note(revMelody 2) time(1/4 1/8)
```

//Palindrome :

// Reverse the contents of a list and append it to the original list to create a palindrome. Useful to extend a melody or list starting from a single phrase

// palinedrome(list) - alias: palin, mirror

```
list melody palindrome([0 2 3 7 9 12])

print melody
```

```
[0 2 3 7 9 12 12 9 7 3 2 0]
```

```
new synth saw note(melody 1) time(1/16)
```

//Invert :

// Invert the contents of a list. This is done by looking at the highest and lowest value in the list and flipping every number to the opposite side. Useful to transform a melodic phrase to generate new musical material

// invert(list) - alias: flip

```
list melody [0 3 2 7 9 7 12 14]

list inv invert(melody)

print melody inv
```

```
[0 3 2 7 9 7 12 14]

[14 11 12 7 5 7 2 0]
```

```
new synth saw note(melody 1) time(1/4)
new synth saw note(inv 0) time(1/4 1/8)
```

//Join:

// Join to or more lists together into one longer list. Useful to transform a melodic phrase to generate new musical material

// join(list1 list2 ... list-n) - alias: combine

```
list melody [0 3 7 12]
list inv invert(melody)
list shuf shuffle(melody)
list joined join(melody inv shuf shuf)
print melody inv shuf joined
```

```
[0 3 7 12]
[12 9 5 0]
[12 0 3 7]
[0 3 7 12 12 9 5 0 12 0 3 7 12 0 3 7]
```

```
new synth sine note(joined 2) time(1/16)
```

//Repeat :

// Repeat the content of a list a specified amount of times. The amount can be specified as another list resulting in different repetitions per value from the input list.

// repeat(list amount)

```
list melody [0 3 7 12 9]
list repeats repeat(melody [4 2 3 5 10 1])
print melody repeats
```

```
[0 3 7 12 9]
```

```
[0  3  7  12  9  8]
[0  0  0  0  3  3  7  7  7  7  7  12  12  12  9  9  9  9  9  9  9  9  9  9  8]
```

```
new synth saw note(repeats 1) time(1/16)
```

//Lace:

// Interleave (lace/zip) the contents of two or more lists. The longest list is preserved but other lists are not repeated in the interleaving process

// lace(list1 list2 ... list-n) - alias: zip

```
list melody [0 3 7 9]
list melody2 add(melody 12)
list melody3 [24 27]
list laced lace(melody melody2 melody3)
print melody melody2 melody3 laced
```

```
[0 3 7 9]
[12 15 19 21]
[24 27]
[0 12 24 3 15 27 7 19 9 21]
```

```
new synth saw note(laced 1) time(1/16)
```

//Lookup :

// Lookup any items from a list based on the numbers in another list. The numbers represent the index (0-based) and wrap on the list length. Useful to reorder content of a list

// lookup(indeces items)

```
list sounds [kick_808 hat_808 snare_808]
list pattern [0 1 1 1 0 1 2 1]
list beat lookup(pattern sounds)
print beat
```

```
[kick_808 hat_808 hat_808 hat_808 kick_808 hat_808 snare_808 hat_808]
```

```
new sample beat time(1/16)
```

//Clone :

// Duplicate the contents of a list a specified amount of time. But add a value (offset) to every duplication based on the the value in another list. Useful to generate melodic progressions

```
set scale minor c
list notes [0 3 7 5]
list clones repeat(notes 4)
list melody clone(notes clones)
print notes clones melody
```

```
[0 3 7 5]
[0 0 0 0 3 3 3 3 7 7 7 7 5 5 5 5]
[0 3 7 5 0 3 7 5 0 3 7 5 0 3 7 5 3 6 10 8 3 6 10 8 3 6 10
8 3 6 10 8 7 10 14 12 7 10 14 12 7 10 14 12 7 10 14 12 5
8 12 10 5 8 12 10 5 8 12 10 5 8 12 10]
```

```
new synth sine note(melody 2) time(1/16)
```

//Merge :

// With the merge function you can combine multiple lists into a 2-dimensional list. This can be useful when you want to generate some chords for example.

```
set scale minor d
// different rows of notes are generated with various functions
```

```
list row1 spread(6 0 12)

list row2 cosine(6 3 7 19)

list row3 drunk(6 3 12 24)

// the rows are merged into a 2d list and every item is repeated 4 times

list chords repeat(merge(row1 row2 row3) 4)

print chords
```

```
[[0 19 15] [0 19 15] [0 19 15] [0 19 15] [2 7 14] [2 7 14] [2 7 14] [2 7 14] [4 19 14] [4 19 14] [4 19
14] [4 19 14] [6 7 12] [6 7 12] [6 7 12] [6 7 12] [8 19 15] [8 19 15] [8 19 15] [8 19 15] [10 7 14] [10
7 14] [10 7 14] [10 7 14]]
```

```
new polySynth saw note(chords 1) time(1/4) shape(1/6 100) fx(filter) fx(chorus)
```

//Add sub div mul :

// It is possible to perform basic arithmetic on entire lists with a single scalare, or adding 2 lists together add(list1 list2/scalar) sub() div() mul() - alias: subtract, divide, multiply

```
list melody [0 3 7 3 5 7 9]

list melody2 add(melody 12)

print melody melody2
```

```
[0 3 7 3 5 7 9]

[12 15 19 15 17 19 21]
```

```
new synth saw note(melody) time(1/4)

new synth saw note(melody2) time(1/4 1/8)
```

//Normalize :

// Normalize the content of a list. When normalizing a list the lowest value will become 0 and the highest value 1.All the values in between will be scaled in ratio to the lowest and highest value.

// normalize(list) - alias: norm()

```
list psn pisano(7)

list psnNorm normalize(psn)

print pattern psnNorm
```

```
pattern

[0 0.16666666666666666 0.16666666666666666 0.3333333333333333 0.5 0.8333333333333334
0.16666666666666666 1 0 1 1 0.8333333333333334 0.6666666666666666 0.3333333333333333 1
0.16666666666666666]
```

```
new sample hat_808 time(1/16) gain(psnNorm)
```

//Translate notes :

// You can use various methods to translate between different note notation formats such as Midi, Notenames and Frequency

// Convert Array or Int as midi-number to midi-notenames (alias: mton)

```
print midiToNote([60 63 67 69 57 65])
```

```
[c4 eb4 g4 a4 a3 f4]
```

// Convert midi-pitches to frequency (A4 = 440 Hz) (alias: mtof)

```
print midiToFreq([60 63 67 69 57 65])
```

```
[261.6255653005986 311.1269837220809 391.99543598174927
440 220 349.2282314330039]
```

// Convert Array of String as midi-notenames to midi-pitch (alias: ntom)

```
print noteToMidi(['c4' 'eb4' 'g4' 'a4' 'a3' 'f4'])
```

```
[60 63 67 69 57 65]
```

// Convert midi-notenames to frequency (A4 = 440 Hz) (alias: ntof)

```
print noteToFreq(['c4' 'eb4' 'g4' 'a4' 'a3' 'f4'])
```

```
[261.6255653005986 311.1269837220809 391.99543598174927
440 220 349.2282314330039]
```

// Convert frequency to nearest midi note (alias: ftom)

```
print freqToMidi([261 311 391 440 220 349])
```

```
[60 63 67 69 57 65]
```

// Set detune flag to true to get floating midi output for pitchbend

```
print freqToMidi([261 311 391 440 220 349] true)
```

```
[59.9585553965427 62.99293267927131 66.95598100539232 69
57 64.98868215221991]
```

// Convert frequency to nearest midi note name (alias: fton)

```
print freqToNote([261 311 391 440 220 349])
```

```
[c4 eb4 g4 a4 a3 f4]
```

//Translate relative :

// You can use translate relative values/names to midi numbers or frequencies easily with the relative and chroma methods

// Convert relative semitone values to midi-numbers (alias: rtom)

// specify the octave as second argument (default = 'C4' = 4 => 48)

```
print relativeToMidi([-12 -9 -5 0 4 7 2 5 9] 'c4')
```

```
//[48 51 55 60 64 67 62 65 69]
```

// Convert relative semitone values to frequency (A4 = 440 Hz) (alias: rtof) and specify the octave as second argument (default = 'C4' = 4 => 48)

```
print relativeToFreq([-12 -9 -5 0 4 7 2 5 9] 'c4')
```

```
//[130.8127826502993 155.56349186104046 195.99771799087463 261.6255653005986
329.6275569128699 391.99543598174927 293.6647679174076 349.2282314330039 440]
```

// Convert a chroma value to a relative note number (alias: ctor), can also include octave offsets with -/+, case-insensitive

```
print chromaToRelative(['C' 'Eb' 'G' 'Ab' 'A+' 'F-'])
//[0 3 7 8 21 -7]
```

// Convert ratios to relative midi-cents (alias: rtoc)

```
print ratioToCent([2/1 3/2 4/3 5/4 9/8])
```

```
//[1200 701.9550008653874 498.04499913461245 386.3137138648348 203.91000173077484]
```

// Translate Time :

// With the methods below you can translate between various time formats such as milliseconds, divisions and ratios. By default the global tempo is used, but you can set a specific tempo with an additional argument

```
// divisionToMs(list tempo)

set tempo 120
// convert beat division strings to milliseconds with global bpm (alias: dtom)
print divisionToMs([1/4 1/8 3/16 1/4 1/6 2])

//[500 250 375 500 333.3333333333333 4000]

// use a local bpm argument
print divisionToMs([1/4 1/8 3/16 1/4 1/6 2] 100)

//[600 300 450 600 400 4800]

// also converts ratios from floating point
print divisionToMs([0.25 0.125 0.1875 0.25 0.1667 2] 100)

//[600 300 450 600 400.08 4800]

// convert beat division strings to beat ratio floats (alias: dtor)
print divisionToRatio([1/4 1/8 3/16 1/4 1/6 2])

//[0.25 0.125 0.1875 0.25 0.16666666666666666 2]
```

// External Connections :

// It is possible to extend Mercury with external software, hardware, and other tools via MIDI, OSC and Microphone inputs. In this chapter we'll go over these different possibilities step-by-step

//MIDI Note :

// We can also output MIDI notes from the browser to external instruments or applications on the computer via WebMidi. This allows us to use Mecruy for generative music while other applications are handling the sound

// this will send a midi-note to the default device a gain of 1 is a velocity of 127, 0.8 = 101

```
new midi default time(1/8) note([0 3 7] 2) gain(0.8)
```

//MIDI Note :

// We can also output MIDI notes from the browser to external instruments or applications on the computer via WebMidi. This allows us to use Mecruy for generative music while other applications are handling the sound

```
new midi default time(1/8) note([0 3 7] 2) gain(0.8)
```

// this will send a midi-note to the default device

// a gain of 1 is a velocity of 127, 0.8 = 101

// MIDI Duration :

// With the length() method we can change the duration. The duration of a midi-note is determined by the interval between the note-on and note-off message in milliseconds

```
new midi default time(1/8) note(0 2) length([50 100 150 200])
```

// the list in length() will send different durations

//MIDI Channel :

// We can change the midi-channel with the out() method

```
new midi default time(1/8) note(0 0) out(10)
```

// channel 10 is the default drum channel in general midi

// MIDI Chord :

// By setting the chord() method to 'on' we can output

// multiple notes at the same time to create a chord

```
new midi default time(1) note([[0 3 7 11]] 2) chord(on)
```

// here the chord is created by making a list on the first

// place within another list (otherwise the notes would be

// played in order over time)

// MIDI Control Change :

// With the cc() method we can send control change messages to the same device, to control other parameters. Add multiple cc() methods to make different messages

```
new midi default time(1/16) note(0 0) name(myMidi)

set myMidi cc(10 [50 100]) cc(20 random(4 0 127))
```

// Above are two cc() methods, one to control number 10, and

// one to 20, both with different values to change.

//MIDI Pitchbend :

// With the bend() method you can send pitchbend messages to the same device. You can only send one pitchbend message at a time per channel. The pitchbend range is -1.0 to 1.0, where 0 is no pitchbend. The output is hiresolution 14bit 0-16383

```
set tempo 130

list pitchbender spreadF(16 -0.3 0.3)

new midi default time(1/16) note(0 2) bend(pitchbender)
```

// Above is one bend() output, with 16 values ramping from -0.3 to 0.3, send every 16th note.

//MIDI Program Change :

// With the program() method you can send program change messages to the same device on the specified channel. The program change is an whole number between 0 and 127. The changes can be sequenced as a list.

```
set tempo 130

list notes spread(5 0 12)

list changes [0 10 20]

new midi default time(1/8) note(notes 1) out(1) program(changes)
```

// change program changes 0, 10 and 20 to channel 1

//Input Microphone :

// You can access the default microphone as input in the code. WATCH OUT when not using headphones! You can get very loud feedback if the output of the speakers goes directly back into the microphone

```
new input default gain(0)
```

// the gain is set at 0 to make sure there is no feedback

// carefully increment the gain so you can hear the microphone

// if your microphone is not the default then try: in1, in2, in3, etc... (default = in0)

//Input Functions :

// Similar to the sample and synth you can use functions to adjust the incoming sound in real time. For example this is a stutter effect where the sound is only let through when the envelope is triggered

```
set tempo 120
new input default shape(1 1/32 1) time(1/16) gain(1)
```

//Input FX :

// It is also possible to add FX to the incoming microphone sound such as distortion, delay, pitchshifting, filtering, etc...

```
set tempo 100

new input default name(mic)
set mic time(1/16) shape(1 1/32 1) gain(1)
set mic fx(shift -12) fx(degrade 0.5) fx(delay 2/16 3/16 0.9)
```

//Open Sound Control (OSC) :

// It is possible to send OSC Messages to Mercury instruments. However this is only possible when running Mercury via a localhost. Go to https://github.com/tmhglnd/mercury-playground#-install and follow the instructions to run mercury locally.

// Now send osc-messages from other applications to ip: 127.0.0.1 (or localhost) at port: 8000

// for example control the note, length, volume and filter of a sawtooth synth the osc-addresses are written as strings ''

```
set tempo 100
new synth saw name(syn0) time(1/16)
	set syn0 note('/synth/note') shape(1 '/synth/length') gain('/synth/vol')
	set syn0 fx(filter low '/synth/cutoff' 0.5)
```

//OSC Output :

// With the new osc instrument you can send osc messages at addresses at a specific time interval/rhythm. The functions you type are used as osc addresses. The name of the instrument is the base of the address. This instrument only works when running Mercury via a localhost.

// Follow the instructions on: https://tmhglnd.github.io/mercury/docs/getting-started#-without-internet (some programming experience with NodeJS is expected)

// Receive OSC messages at port 2440

```
set tempo 100

new osc base name(send) time(1/16)

set send play(euclid(16 11)) // send messages rhythmically

set send trigger(1) msg(spread(5 0 12)) // arguments can be lists
```

// expected messages received at port 2440:

// /base/trigger 1

// /base/msg 4

// /base/trigger 1

// /base/msg 7

// ...

// some function words are not allowed to be used as osc address

// because they are used for the core functionality of the instrument:

// time, play, timediv, name

//Hydra Visuals :

// We can store hydra sketches as a string in a list by creating a list of multiple sketches the instrument will cycle through them for every trigger

```
list hydras ['osc(10,0.1,2).out()' 'osc(20,-0.5,5).out()' 'osc(5,1,12).out()']

new sample kick_min time(1/16) play([1 0 0 1 0]) visual(hydras)
```

//View List :

// It is possible to view the content of a list in a visual manner. By using 'view' the list will be normalized (0-1) and displayed as grayscale values across the screen. The length of the list will determine the amount of squares displayed

```
list myVals reverse(randomF(10 1 1000))

view myVals
```

//List RGB :

// If you merge() three lists you can display Red, Green and Blue values seperately. Make sure the lists have the same length in order to fill all pixels equally. All lists are normalized together, so if one list has very large numbers, the other lists will become very dark

```
list cos mod(cosine(3034 50 0 7) 3)
list sin mod(sine(3034 40 0 7) 3)
list cos2 mod(cosine(3034 20 0 20) 3)
view merge(cos sin cos2)
```

//View hydra :

// You can access the canvas of the displayed list by initializing a hydra source: s0.init({src: listView})

```
list cos mod(cosine(2000 60 20) 2)
view cos

list hydraVisual ['s0.init({src: listView}); src(s0).modulate(noise([2,5,10]), [0.05, 0.1]).out()']
new sample hat_909 time(1/4) visual(hydraVisual)
```

//Samples Freesound :

// It is possible to load soundfiles via links from freesound by create a list for every sample you can specify the sound name and the url

```
list s1 [ snare_short 'https://cdn.freesound.org/previews/671/671221_3797507-lq.mp3' ]
list s2 [ psykick 'https://cdn.freesound.org/previews/145/145778_2101444-lq.mp3' ]
list s3 [ hat_short 'https://cdn.freesound.org/previews/222/222058_1676145-lq.mp3' ]
```

// load the samples with 'set samples'. This function only needs to be evaluated once! afterwards you can comment it.

```
set samples s1 s2 s3
```

// The sounds can be used with the name specified in the list. Loading the samples takes some time so better first load the sounds and then uncomment this code so you can use it and listen

```
// new sample psykick time(1/4)
// new sample snare_short time(1/16) play(euclid(7 3)) gain(0.5)
// new sample hat_short time(1/4 1/8) gain(1.3)
```

```
sound added as: snare_short
sound added as: psykick
sound added as: hat_short
```

//OSC Messages :

// It is possible to send Open Sound Control (OSC) Messages to Mercury instruments. However this is only possible when running Mercury via a localhost. Go to https://github.com/tmhglnd/mercury-playground#-install and follow the instructions

// Now send osc-messages from other applications to ip: 127.0.0.1 (or localhost) and port: 8000

// for example control the note, length, volume and filter of a sawtooth synth

```
set tempo 100
new synth saw name(syn0) time(1/16)
set syn0 note('/synth/note') shape(1 '/synth/length') gain('/synth/vol')
set syn0 fx(filter low '/synth/cutoff' 0.5)
```

//Samples Github :

// Loading samples from a github raw file is also possible. If you know the location of the file you can change the url by starting with: https://raw.githubusercontent.com/ followed by the user/repo/branch/path

// comment this line when the sound is loaded

```
set samples [ housekick
'https://raw.githubusercontent.com/tmhglnd/mercury/master/mercury_ide/media/
samples/drums/kick/kick_house.wav' ]
```

// uncomment this line after loading the sounds

```
// new sample housekick time(1/4)
```

```
sound added as: housekick
```

//Samples JSON :

// Loading more than one sample is also possible via a .json file.The .json file can be stored on the computer and selected when click 'add sounds'. Or it can be stored on github and used via the raw file link as argument to 'set samples'

// A .json file could look like this:

```
// {
//	"snare_short" : "https://cdn.freesound.org/previews/671/671221_3797507-lq.mp3",
//	"psykick" : "https://cdn.freesound.org/previews/145/145778_2101444-lq.mp3",
//	"hat_short" : "https://cdn.freesound.org/previews/222/222058_1676145-
lq.mp3"
// }

// If the base url is the same for all the sounds you can add that as a separate
key:value in the dictionairy

// {
//	"snare_short" : "671/671221_3797507-lq.mp3",
//	"psykick" : "145/145778_2101444-lq.mp3",
//	"hat_short" : "222/222058_1676145-lq.mp3",
//	"_base" : "https://cdn.freesound.org/previews/"
// }
```

//Granulation Basic :

```
// Simple granular timestretching is achieved by quickly playing short grains of a
sample changing the offset of the playback point and detuning the grains

set tempo 120

// a position ramp up-down over 500 values
list pos sineFloat(500 1 0 0.5)

// small detuning for every sample playback
list detune randomFloat(500 0.92 1.08)

// try different sample names
new sample bongo name(grain)
```

```
// a fast time interval and a short attack and release time for fade-in/out
set grain time(1/32) shape(20 20)
// set the start position from the list and random panning
set grain start(pos) pan(random)
// set the playback speed and gain
set grain speed(detune) gain(1.2)
// add some effects
set grain fx(delay 2/32 3/32 0.9) fx(reverb 0.4 5)
```