# PQE's Favorite Max Objects

The scariest thing about Max is the sheer number of functions available. I've seen it described as a truckload of refrigerator magnets hitting the side of a warehouse. My externals directory has 342 items in it, not counting Lobjects (114) and assorted interesting objects by other developers. But I take my mantra from Douglas Adams:

Don't Panic

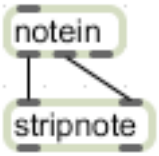
90% of the Max patches I have built use the same 50 basic objects. Beyond those, objects are either very specialized, only slightly different from the standard object, or can be duplicated by a few simple ones. The benefits of using specialized objects are often only cosmetic, resulting in cleaner windows. Don't underestimate that value, because a simple patch is easier to debug than a cluttered one. But making it work in the first place is important too.

Here, in no particular order, are the objects I use (and see used) the most.

## *Input*

**Basic MIDI input**. Almost all of my patches use one of my trusty MIDI controllers. Usually there's an argument to ctlin to watch a single control.

**Stripnote** gives only the note on messages. Don't forget to connect the inlets to the left and center outlets of notein.

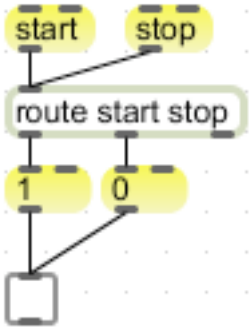


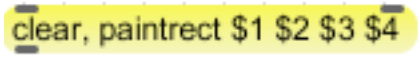

## *User Interface*

**Number boxes**. When I'm developing a patch, I use these everywhere. They let me inject test input at various stages and watch values change.

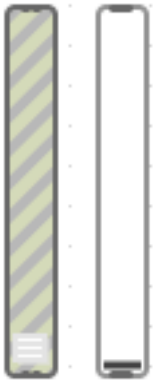

**Toggles** and **buttons** are nearly as common. They are easy to use when expanded a bit.


**Message boxe**s make the best buttons for starting actions. If the destination doesn't understand the message I want to use, they are easily converted to other messages with route:

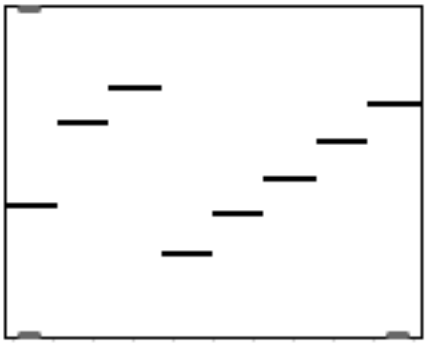



I'll often use a comma in a message box to get two messages from one input. The $1 etcetera items are tokens replaced by elements of an incoming list.



**Sliders**. When I'm going to control actions with a mouse, I use one of these. Note the color difference between the audio gain control (left) and the ordinary data slider. When I'm finishing a fancy application, I may replace the data slider with something prettier, but not often.



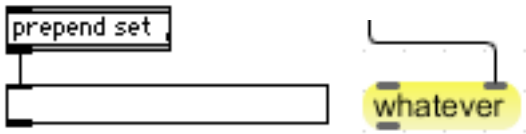
**Multi-slider** usually turns up when I'm building an audio gadget with a lot of channels. You have to use the cmd-I inspector to set these up.

**Kslider**. Even when I have a keyboard attached, I'll use one of these as a test input. I usually take it out of finished patches, though. You can't really play it.


**Key** reveals what is happening at the computer keyboard. The left outlet shows the acsii code of the key pressed, the center shows the Macintosh or Windows key code (regardless of shift, etc.) and the right center shows what the modifiers are up to. The far right outlet is new to version 5-- it has a unique number that is the same on Mac and Windows. You don't need a chart of ASCII codes or key codes. Just hang a number box on key and press the one you want to know. Gotcha-- if a number box has focus (it has a blinking cursor) in the patcher, key won't respond to typing on the number keys.

## *Display*

Most of the UI objects also work for displaying data. Slider and multi-slider nicely show current values, for instance. What I do the most is this:

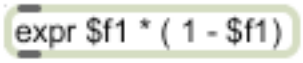



Prepend is like append, but before. The **prepend** set object sends the message set "whatever" to the message box, which will then display whatever it is. My patches are littered with these things when they are under development. We type "prepend set" so much that Max 5 message boxes have a right inlet that sets the message. Prepend set will gradually disappear from my patches and writings, but there are plenty of other uses for prepend.

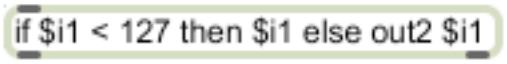
## *Math*


**Basic math objects**. These are all pretty much the same. The argument sets the type of the result and should give a reasonable behavior before anything arrives at the right inlet. The only oddity is the construction !- means input is subtracted from the argument rather than the other way around. The gotcha is don't send a list to this-- the second item of the list will replace the argument. Another gotcha is in creating these:


Make sure you have the space between the operator and the argument. There's no such thing as a +2 object, so that is a harmless error, but typing -2 gets you an int box with -2 in it and that won't do any subtraction.




**Expr** This replaces a lot of basic objects with a compact formula in a simple to understand manner. I get my math from reference books. (Even when I am certain how to do something, I check the method.) Most of the time, the example in the reference can be copied pretty directly. The gotcha? Type carefully. If you are editing one of these and leave a typo, all of the outlets will disappear and you'll have to remake all your connections. The construction $f1 is a token meaning a float at inlet 1.



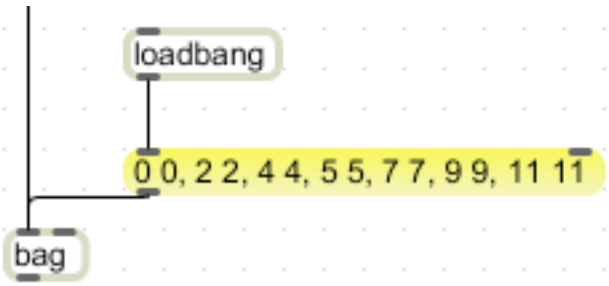

**If**. This is the decision maker. First is the condition. It can be pretty complicated, but I do most calculations in a separate expr. The true action is required (not much point otherwise). The optional else does not need to be sent out the right, but I usually do. The gotcha? You can't do any math in the true or false action statements. If you miss that as much as I do, check out L?[1].


**Random** is a good source of unpredictable numbers. It will produce a new value each time it is banged. The only gotcha is that the range of outputs is from 0 to one less than the argument. The essay Random Max explores random processes in detail.

## *Managing Data*

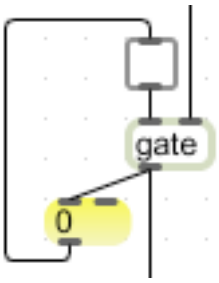

A common frustration with Max is to get a patch working, save it, but discover that it won't work after it is re-opened. This is generally due to uninitialized objects, parts of your patch that must be primed with certain values (such as metro rate) to work properly. **Loadbang** addresses this problem by outputting a bang when the patcher file is first opened. The bang can be used with message boxes to provide initial settings to any objects than don't take arguments. This example with **bag** is typical:

[1] That's an Lobject that implements the ? : form of code.

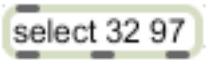



The gotcha with loadbang is that if you have several, you won't know which will bang first. The answer to that is to have only one attached to **send** objects that distribute the bang in a controlled order. **Loadmess** is a new version that will send more than a bang.

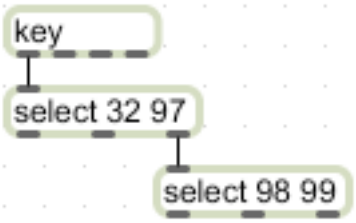

**Gate** and **switch** turn data streams off and otherwise control where messages go. I've always felt these names were backwards. Gotcha-- these are always off when the patch loads, so a loadbang is often needed to get things going. A worse problem is since the control inlet is on the left, you have to take measures to ensure the gate is properly opened or closed before the data arrives. (Leftswitch and Lgate avoid this issue.) I often use this arrangement to let one item through at a time:

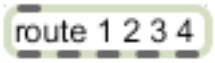



In some instances, the graphic versions of these are better, as they are open one way or another when the patch is loaded. The gotcha with these two is that the graphic indicator can be out of sync with what is actually happening.


**Select** or **sel** responds to input by banging the appropriate outlet. The none of the above outlet makes daisy chaining easy. I often use one with key to decode typist's actions:




**Route** is a lot like select, but it will pass messages through if they are appended to the selector. (The message "1 2 buckle my shoe" will produce "2 buckle my shoe" at the left outlet of this one.) I created **label** to have a complimentary function for this kind of action:


**Send** and **receive** (**s** and **r**) move data large distances without the need for long patch cords. You can even send from one window to another. The only gotcha-- if you have more than one receive matching a send, there is no way of knowing which one will get data first. Everything that is attached to the first receive will happen before anything attached to the other.

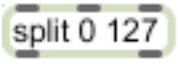

**Int** and **float** are for temporary storage of numbers. Generally this is necessary when right to left operations make it difficult to present the right numbers at the right time for some calculation. If you send the number to the right inlet, int and float will hold the numbers until they are banged. If you just type a number into an object box, int or float are what you get.


**Coll** is the general purpose data collection. Look at Max & Chords for some examples of how I use it. Gotcha-- be sure to open the inspector and check "save data with patcher".

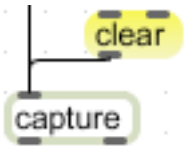
But the truth is I avoid coll if I can. Most simple sets of data can be stored in a list, and **llist**[2] can be used to access them like this:

[2] Fans of Terry Pratchett will recognize the Llamadosian spelling of Llist and a few other lobjects that start with L. I should have done that from the beginning, but I was young then. Hence Label, Loop and Like.


(Items in lists are accessed by index numbers, which start at 0. So the fourth item in the list has an index of 3.)

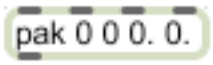


**Split** replaces a complicated if statement to solve a common problem. If a number is within an approved range (inclusive) it is sent on. Otherwise, it is sent elsewhere.

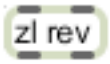


**Capture** is mostly a debugging tool. Many processes happen so fast that number boxes and message boxes cannot keep up. Capture keeps all the data that arrives until you give it the clear message.

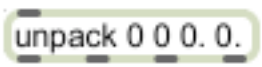
## *List management*



I usually use pack to put together a list. It will only send output when the left value is received. Pak will output every time any input changes. This is handy sometimes, but it can result in a lot of spurious calculations. Gotcha-- the arguments not only set the number of inlets and the default contents of the output list, they set the type of every item. The Lobjects are full of specialized list builders and converters but I still use these a lot.



These list processors are derived from James McCartney's list ops, which predate the Lobjects. McCartney went on to write SuperCollider, and decided not to support these objects through the various Max versions. Eventually, David Zicarelli added most of them to the standard set. These are all zl objects, but the argument changes the function. Many classic Lobject functions are now available in a zl version.



**Unpack** takes lists apart again. Again, the arguments set the type of the output.

These take lists apart, sending one item out after another. The difference is **iter** runs as fast as possible, **unlist** (an Lobject) synchronizes each item with a bang.

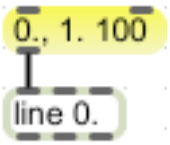
Thresh is the opposite of iter. Individual items received in a close group are combined into a list. The argument sets the timing gap that will trigger list output. (The help file is misleading.) Gotcha-- only 255 items can be collected. Any more than that without a break will be lost. Lcatch deals with this by sending multiple lists. (A list can only be 255 items long.)

## *Timing*

**Metro** is the mother of all repetitive action. Effective use relies on finesse in setting the timing period. See Max & Rhythm for details. Qmetro is a variant used with graphics, especially jitter patches. Whereas metro is quite insistent that its actions be done immediately, qmetro is more relaxed, allowing MIDI processing and user interaction to happen first.

**Timer** is a simple stopwatch. To get the time, send the message "time" to the **date** object.

**Line** generates a series of numbers that make a smooth transition to a target value. You send it a message consisting of the target and time in millisecond it should take. The argument sets the starting value, but after it's been used, the start value is the last output. To make ramps use a compound message like this:



There is a second argument (third list member) that sets how often line puts out while it is changing. This should be adjusted to match the times used for the ramps.

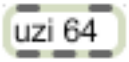
**Counter** counts messages. It's full of gotchas--

- It starts with zero, so it shows one less than the number of messages received.
- If you give it an argument it counts up to and including that value, so counter 10 gives 11 counts per cycle.
- Two arguments set start and end, so counter 1 1000000 will count things the way you would expect. After it hits the end, the next number out is the start.

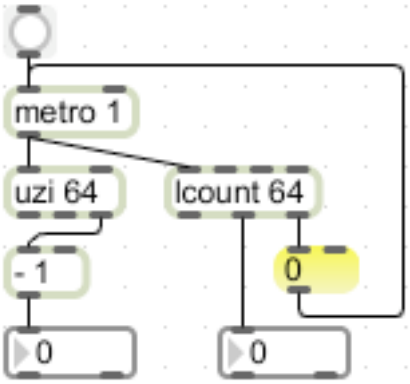
- The overflow and underflow (it can count backwards) outlets produce a 1 followed by a 0, so extra logic is needed to decode this to cascade counters.
- There is a carry count outlet, which works when counting up but not down.

I generally use my own version, called **Lcount**.


The **trigger** (**t**) object sorts out problems of precedence. When a message is received, the outlets will fire in a nice right to left order. Furthermore, the output will be converted according to the arguments in trigger. In this instance, a number would be sent as a float from the right, then an int from the center, then the left will bang. You can put messages in there too.




**Uzi** is the tool for repeating processes. With an argument of 64, one bang in causes 64 bangs out as fast as possible. The right outlet is the most useful, because it numbers the bangs. Avoid hanging one uzi on another. It's easy to create a long process (64x64 = 4096) that clogs up the computer, or at least makes mouse and display actions unresponsive. I prefer to combine uzi with metro and lcount like this:




The - 1 box deals with my pet peeve with uzi. The numbers out the right count from 1 rather than 0, which is need for most iterative processes.

## *Encapsulations*

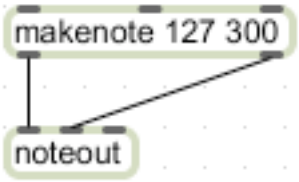

Subpatchers, which confusingly are called **patcher** (**p**) in an object box are a way to hide complex (not to say messy) parts of a patch. They have another advantage-- if key parts of an algorithm are encapsulated like this, you can copy and modify them, then use "Paste Replace" to try out different methods. Gotcha-- the name does not mean anything. If you copy this and modify one of the copies, the original is untouched. If you want that behavior, you must save the encapsulated section as a file.


**Poly~** is an encapsulation of multiple copies of a subpatcher. Even though it has the tilde, it's not dependent on MSP running. They are great for synthesis of

course, but I've also found applications in Jitter, where I often want many copies of a GL object.

## Output


**Makenote** and **noteout** go together like peanut butter and jelly. You see this at the bottom of practically every patch that uses MIDI:




The gotcha with makenote is its default arguments, which are 0. If you don't include a velocity argument the velocity of 0 will be a note off. If you don't include a duration, the note will be too short to be heard.
The gotcha with noteout is what happens when you change the MIDI channel. If you do this while a note is sounding, the note off will go to the wrong channel. My suggestion is don't change channels with noteout. Instead, use a makenote noteout pair for each channel you need. If you want to change channels on the fly, use Lnote instead of makenote.

## Basic MSP items

Audio output is of course essential to MSP. It's easy to start audio with the **ezdac~** button, but you should also be familiar with the effect of the startwindow message. Use **adc~** when you need more than two channels of playback. Details on all of these objects are in the BasicMSP essay.

The **audio input** as defined in DSP setup. I usually follow this with a

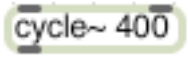
**meter~**.


**Sfplay~** and **Sfrecord~** are the way to play and record audio files. These only work with aiff and wav files. For MP3s use a buffer~. I hardly ever make a patch with sfrecord~ in it-- the help file does most of my recording.

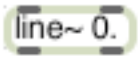

The functions based on buffer~ are central to live looping and other types of sample playback. You can play an MP3 file by importing it to a buffer~. The gotcha? Accurately typing the name of the buffer~. Capitalization matters.


**Tapin~**and **tapout~** are basic to delay effects. Just remember, the argument to tapin~ sets the maximum delay.


I don't do much math with audio signals, but the ***~** is essential. It's the equivalent of a volume control. I always put a **gain~** slider at the bottom of the patch and after inputs, but use *~ everywhere else. I use 0. for the argument when I want the patch to start with no sound.




**Cycle~** is the heart of synthesis. I also like to use the help file to test the audio output.




**Line~** is the basic envelope generator. Use with *~ and various messages.


There are a lot of good filters in MSP, but these are the ones I use the most, probably because they have the most synthetic sound.


There's nothing like banks of oscillators and filters to give the "computer music" sound. These are the heart of many of my favorites.

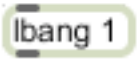
## *Fundamental Lobjects*

I started writing objects to make Max simpler, not more complex, but with more than 120 in the set, I have to admit to contributing to feature creep. They all seemed like a good idea at the time but many are seldom used. A few are so useful that cycling '74 has added their own versions to the standard objects and I keep mine only for backwards compatibility[3]. As I look over my own patches, I see these the most often:

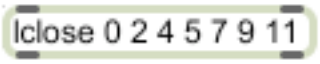
[3] Someone suggested to me that the zl objects were derived by reverse engineering the Lobjects. This works if you know that Elsea is pronounced "L-Z". It's not true, but loadmess, pak, and vexpr were definitely preceded by Lobjects that did the same thing.

My basic list math objects were never the only way to do member by member math, but I like the way the names and the arguments work together. Ladd 1 2 3 4 will ad those numbers to the first four members of a list, whereas Ladd 1 will add 1 to each member. All Lobjects also automatically adjust number types to the data coming in.

Setting bits in bytes is easily done with expressions like (($i1 & 1) << 4 ) | ($i2 & 0xF3)). Enough said?

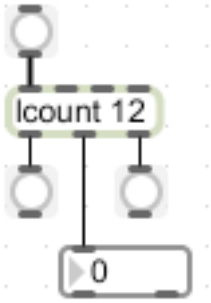


**Lbang** sends its arguments at load time. I wrote Lbang because I got tired of typing loadbang and adding a separate message box for initializations. Loadmess has recently appeared in the standard objects, so I guess it's time for lbang to retire.

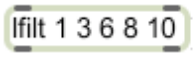


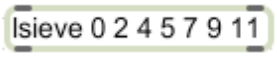
**Lclose** does the trick I've been teaching with bag-- gives an approved number nearest the input value. The gotcha is the actual approved values are sent from the right outlet. The left outlet provides the positions of the results in the stored list. I do this to be compatible with other objects that provide position and value.

Lcount came from frustration at trying to make counter do anything beyond simple counting. I got my start in circuit design, and a chip called the 74193 was a real powerhouse for building sequencers and clock displays. Lcount works much the same way. I admit I got carried away, but since jitter came out I have found it invaluable to be able to count to 2pi in increments of 0.1. The gotcha is the count value comes from the center outlet.



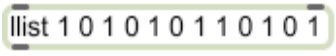
Lcount counts from the start (defalut 0) to one before the end value. This is the way most computer programs count-- counting 0 through 11 gives 12 events. The count comes from the center outlet. The right outlet bangs when the count hits the end, and the left outlet bangs when the count starts over. (Lcount will do other tricks like count by fractional increments.)



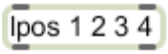


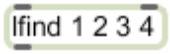
These route data based on its value. With standard objects, you'd need a split for each argument, and they would be awkward to change.

Zl join is great at combining two lists, but can't handle three. Lifo stands for "last in, first out", so multiple lists will be put together in the order they appear in the patcher (left to right). Link does first in, first out like join, but you can combine as many pieces (up to 255 elements) in the right as you want to. Both can take arguments to begin the output.

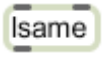


I often get suggestions for objects, and this was one of the best. You can store a list in a message box with prepend set, but that takes a lot of room. Zl reg stores lists in the same manner, but does not have the get individual members feature.





These tell where things are in long lists. That's important in analysis (see Max & chords) and fuzzy logic. **Lfind** finds single values (with interpolation) and **Lpos** finds lists.

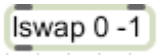


These compare lists in various ways. Lsame gives a 1 if the two lists are identical. If they are similar, L== will give a fractional answer, 0.5 if half the items match. Like routes lists according to the beginning.

Lnth does not process lists, it simply lets only every 3rd (or whatever) one through. You can do this with counter and gate, but lnth includes some tricks for compound counting. I use it to generate rhythms.

**Lreg** really has only one practical application, but it's important. It will easily keep track of what notes are currently playing.

Max has never been good at string manipulation-- the construction of words and phrases from ASCII values. The concept of symbols in Max makes for very efficient computation, but leads to a situation where the digits 123 could mean the value 0ne hundred twenty three or three characters from the keyboard. Spell and atoi handle conversion from symbols to ASCII just fine, but itoa always turns digits into symbols. Lstring has more flexibility in how digits are handled. You can learn more in the essay Max & ASCII.



**Lswap** reorders lists according to a template. It defaults to reversing, but you can ask for individual items, ranges, even make items appear twice.

The template 0 3 * will pull out every third item. The templates are changeable, so this can compose an entire piece.