HOWTO Install & Load External Objects in Pd

by Alexandre Torres Porres

Index

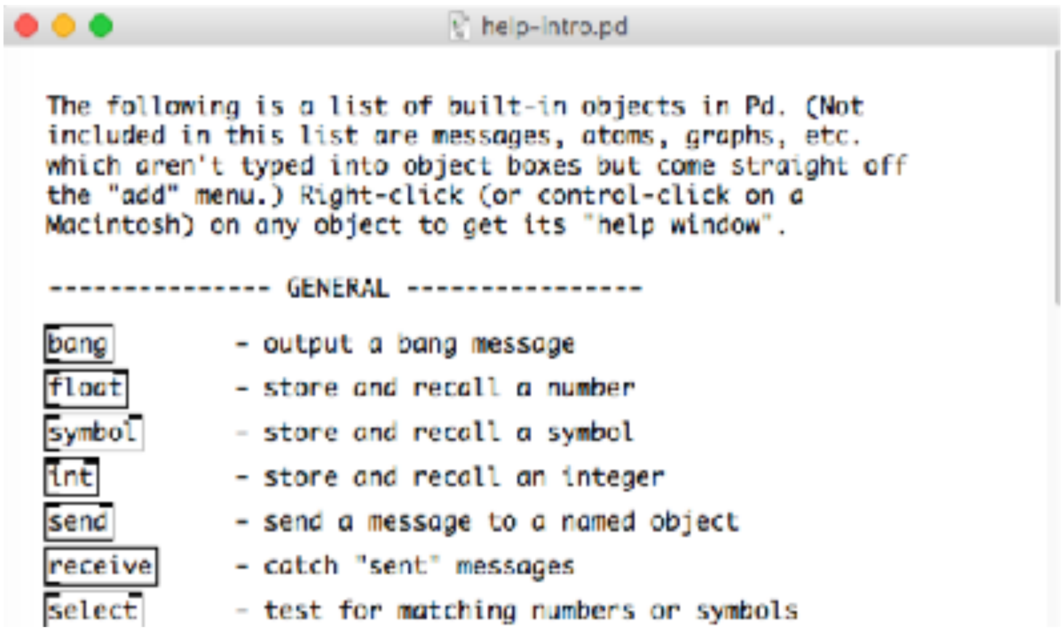
Chapter 1) Introduction: Defining External Objects & Libraries

1.1 - What are: Vanilla Objects, Internals & Externals?

Basically, internals are the objects that come as part of the Pd Vanilla[1] binary, whereas external objects are not! Besides internals, Pd Vanilla also comes with a few "extra" objects that are not part of its binary. Therefore, Vanilla objects (the built-in objects in Pd) include internals and externals.

Nonetheless, "externals" most commonly refer to objects that do not come in the Pd Vanilla distribution, meaning that usually you have to download these objects and install them properly so they can be loaded into Pd patches.

To get a full list of the Vanilla objects, go to the **Help** menu and then select **List of Objects**, or alternatively right click on an empty spot of a patch's canvas and select "help" - this loads the help-intro.pd file.


help-intro.pd

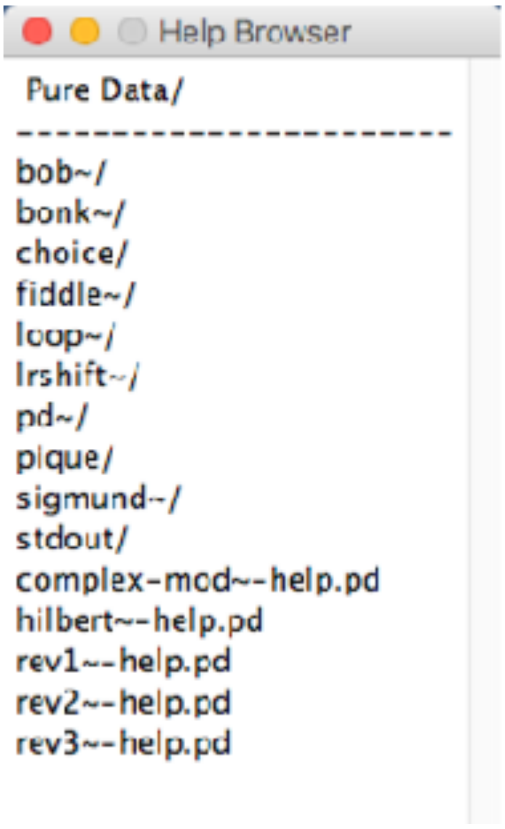
The following is a list of built-in objects in Pd. (Not included in this list are messages, atoms, graphs, etc. which aren't typed into object boxes but come straight off the "add" menu.) Right-click (or control-click on a Macintosh) on any object to get its "help window".

```
--------------- GENERAL ----------------

bang          - output a bang message
float         - store and recall a number
symbol        - store and recall a symbol
int           - store and recall an integer
send          - send a message to a named object
receive       - catch "sent" messages
select        - test for matching numbers or symbols
```

Figure 1 - Excerpt from list of Vanilla objects (help-intro.pd file)

[1] Pd Vanilla is the main distribution of Pure Data, provided by Miller Puckette <http://msp.ucsd.edu/software.html>.



```
Pure Data/
-----------------------
bob~/
bonk~/
choice/
fiddle~/
loop~/
lrshift~/
pd~/
pique/
sigmund~/
stdout/
complex-mod~-help.pd
hilbert~-help.pd
rev1~-help.pd
rev2~-help.pd
rev3~-help.pd
```

Figure 2 - Extra objects from Pd Vanilla

The extra objects, which reside in a folder named “extra” inside the Pd application, are pointed at the very end of the “help-intro.pd” file but can be viewed in the Help Browser menu (Help => Browser). See Figure2 above:

1.2 - What are the Types of External Objects?

An object in Pd can be either a Pd file (an abstraction) or a compiled binary (note that a binary can contain only one or several external objects, as discussed further on). More details about these two options below.

1.2.1 Compiled objects:

These are Pd objects compiled to binaries from programming code (like in C or C++). They have to be compiled for your operating system, which means the binaries have different extensions according to each platform. They are:

- Mac OS: *.pd_darwin* binary extension
- Windows: *.dll* binary extension
- Linux: *.pd_linux* binary extension

1.2.2 Abstractions:

You can save pd patches and make them behave like objects by loading them into your patches, these are called “Abstractions”. Note that some of the “extra” objects from Vanilla that you can see in *Figure 2* are Pd files/patches (see how they end with “*.pd*”). Abstractions may contain any kind of objects (internals, compiled externals and even other abstractions).

1.3 - What are External Libraries?

An external library is a collection of external objects of any kind (abstractions or compiled objects). But when it comes to compiled objects, a library can provide them as a **single binary pack** (containing two or more external objects) or as a **set of separate binaries** (where each compiled binary contains only one external object).

1.4 - What are the types of External Libraries?

Libraries can come in all sorts of ways; as only a collection of abstractions (like “list-abs”), only compiled objects, or both. An external library can also provide compiled externals both as **a set of separate binaries** and a **single binary pack**. Basically, any combination of set of external objects.

A nice example that combines all external options is *cyclone 0.3*, which provides most of its objects as a **set of separate binaries**, but also includes a small collection of 12 objects as a **single binary pack** plus a few abstractions.

Wrapping up Part 1)

- Internal objects: Objects that are part of Pd Vanilla’s binary.

- External objects: Objects that are NOT part of Pd Vanilla’s binary.

- Vanilla objects: Built-in objects in the Pd Vanilla distribution (including internals and a small collection of externals - the “extra” objects).

- **Types of external objects:** Compiled binaries or Abstractions.

- External Library: Collection of external objects in any form, be it a single binary pack containing several objects, a set of separate binaries / abstractions or any combination of them.

Chapter 2) Installing External Objects and Libraries

Installing externals in Pd is quite simple, all you need to do is download your externals from somewhere, such as from Pd Vanilla directly, and include them in a proper folder.

2.1 - Where to include the externals?

You can have externals basically anywhere in your computer, but a best practice is to have them where Pd automatically searches for them. These can be the same folder that your patch is saved on (the Relative Path) or the Standard Paths - the fixed folders that Pd looks for externals, which are:

A) Application-specific: The “extra” folder inside a particular Pure Data application.

B) User-specific: A system folder for a specific user in the machine.

C) Global: A system folder for all users on the machine.

The Global folder affects all Pure Data Applications for all users. The User-specific folder affects all Pure Data Applications for that user. And since you can have different versions of Pd installed in your system, the Application-specific folder affects only that particular Pd Application. This can be not only an older and a newer version of Pd, but also both 32-bit and 64-bit versions available for Mac OS and even Pd Extended!

2.2.1 - Where are the Standard Paths?

These depend on your operating system, as listed below:

A) Mac OS:

- Application-specific: ***/Contents/Resources/extra*** - this is inside the Pd Application. Go to your Pd Application - such as *Pd-0.47-1-64bit (usually in ~/Applications) -* right click and choose “Show Package Contents”, then navigate to “extra”.

- User-specific: ***~/Library/Pd***

- Global: ***/Library/Pd*** (needs to be created)

B) <u>Windows:</u>

- <u>Application-specific:</u> ***%ProgramFiles(x86)%\Pd\extra*** (for 64-bit systems) or ***%ProgramFiles(x86)%\Pd\extra*** (for 32-bit systems); this is inside the Pd Application, usually in *C:\Program Files (x86)* for 64-bits. This folder needs to be set to writeable.

- <u>User-specific:</u> ***%AppData%\Pd*** *(needs to be created)* - this is usually in *C:\Users\user_name\AppData\Roaming\Pd*.

- <u>Global:</u> ***%CommonProgramFiles%\Pd*** (needs to be created) - this is usually in *C:\Program Files\Common Files\Pd.*

C) <u>GNU/Linux:</u>

- <u>Application-specific:</u> ***/usr/lib/pd/extra*** if installed via a package manager (apt-get) or ***/usr/local/lib/pd/extra*** if compiled by yourself.

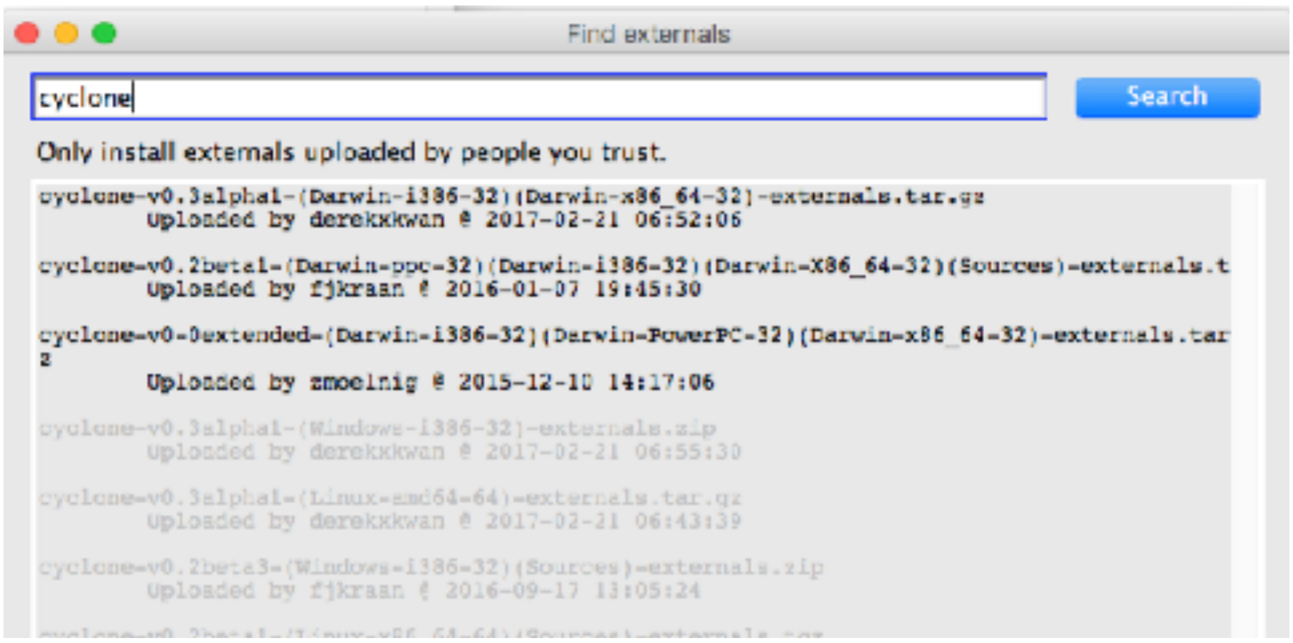
- <u>User-specific:</u> **~/.local/lib/pd/extra** (preferred since version Pd-0.47-1) or ***~/pd-externals*** (deprecated but still usable).

- <u>Global:</u> /***usr/local/lib/pd-externals*** (needs to be created).

<u>2.2 - How to Download Externals from Pd Vanilla?</u>

Since version 0.47-0, Pd Vanilla has its own external manager! This is a built in *.tcl* plug-in named ‘deken’ <https://github.com/pure-data/deken> that has been incorporated into the Pd Vanilla distribution to manage the download of external libraries and externals.

So you can just open Pd, click on the “Help” menu and select **Help > Find externals**. Then you can just type the library’s name you want and hit enter or click “search”. You can also look for an external name and the library that contains it might be shown. All available versions of the library will be shown to you, but the versions specific to your system are highlighted. See image below:



Only install externals uploaded by people you trust.

```
cyclone-v0.3alpha1-(Darwin-i386-32)(Darwin-x86_64-32)-externals.tar.gz
        Uploaded by derekxkwan @ 2017-02-21 06:52:06

cyclone-v0.2beta1-(Darwin-ppc-32)(Darwin-i386-32)(Darwin-X86_64-32)(Sources)-externals.t
        Uploaded by fjkraan @ 2016-01-07 19:45:30

cyclone-v0-0extended-(Darwin-i386-32)(Darwin-PowerPC-32)(Darwin-x86_64-32)-externals.tar
z
        Uploaded by zmoelnig @ 2015-12-10 14:17:06

cyclone-v0.3alpha1-(Windows-i386-32)-externals.zip
        Uploaded by derekxkwan @ 2017-02-21 06:55:30

cyclone-v0.3alpha1-(Linux-amd64-64)-externals.tar.gz
        Uploaded by derekxkwan @ 2017-02-21 06:43:39

cyclone-v0.2beta3-(Windows-i386-32)(Sources)-externals.zip
        Uploaded by fjkraan @ 2016-09-17 13:05:24
```

Figure 3 - Searching for External Libraries from Pd Vanilla

When you click on the version you want, Pd asks you if you want to download it to your User-specific folder, but you can select another download target if you want. As soon as the download is finished, the compressed file is automatically decompressed into a folder containing the library.

Chapter 3) Loading Externals

3.1 - Using namespaces (slash declaration):

This works only for external objects that are either abstractions or compiled as a **separate binary**!

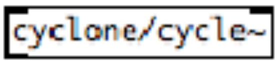
As mentioned, Pd automatically searches for externals in default folders. These folders are:

- The **Relative Path** folder (where the patch is saved on).
- The **Standard Paths** (Application-specific or “extra” folder; User-specific & Global).

Therefore, if you have an abstraction or objects compiled as a **separate binary** in any of these folders, the object will be loaded when created without the need of anything else. But this uncommon, it usually only happens when a patch is provided with abstractions in the same folder as the main patch. And in some cases the abstractions are given in a subfolder, in which case you need to specify it.

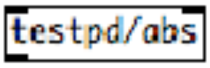
Also, it's a common practice that externals are provided organized into libraries and provided in folders that have the same name as the library. Therefore, you need to specify which folder the object is in too.

One way to do it is simply by inserting the folder/library name where the object is found. For example, loading the [cycle~] external from the *cyclone* library can be done like this:



```
cyclone/cycle~
```

Another example below loads an abstraction called [abs] that is found inside a subfolder mamed “*testpd*”, relative to where the patch loading the object is.

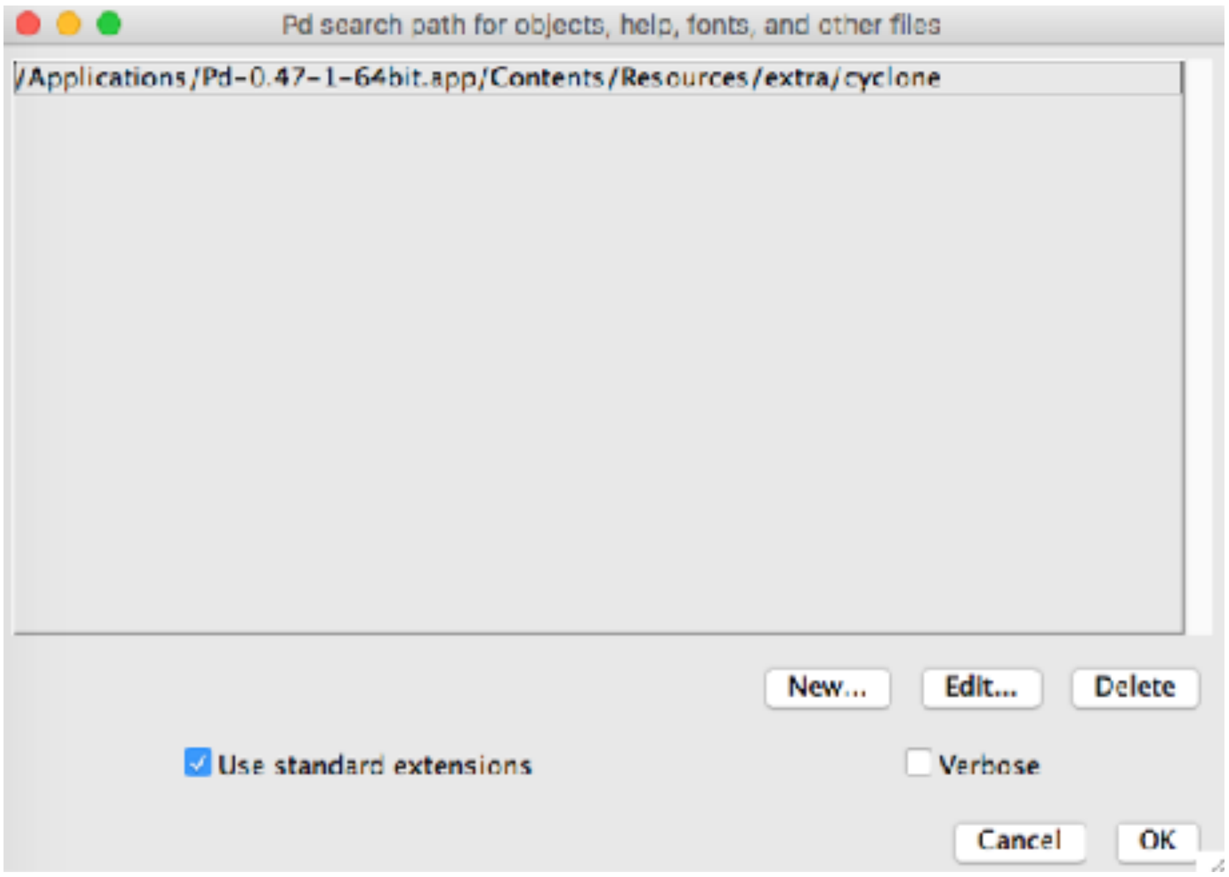


```
testpd/abs
```

3.2 - Loading with Path and Startup:

3.1 - Path:

If you can add a library folder to Pd's Path (**Preferences** => **Path**), you don't need to use namespaces anymore. Any path can be included, but the common practice is to use one of **Standard Paths**, the folders where Pd automatically searches for externals, as in the Application-specific folder in the figure below.


Pd search path for objects, help, fonts, and other files

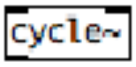
/Applications/Pd-0.47-1-64bit.app/Contents/Resources/extra/cyclone

[✓] Use standard extensions
[ ] Verbose

Figure 4 - Loading the cyclone library in the "Path"

Above, the cyclone folder/library is included in the Path. Now, when restarting Pd, you can call any object from cyclone (a compiled **separate binary** or abstraction) without worrying about the slash declaration.

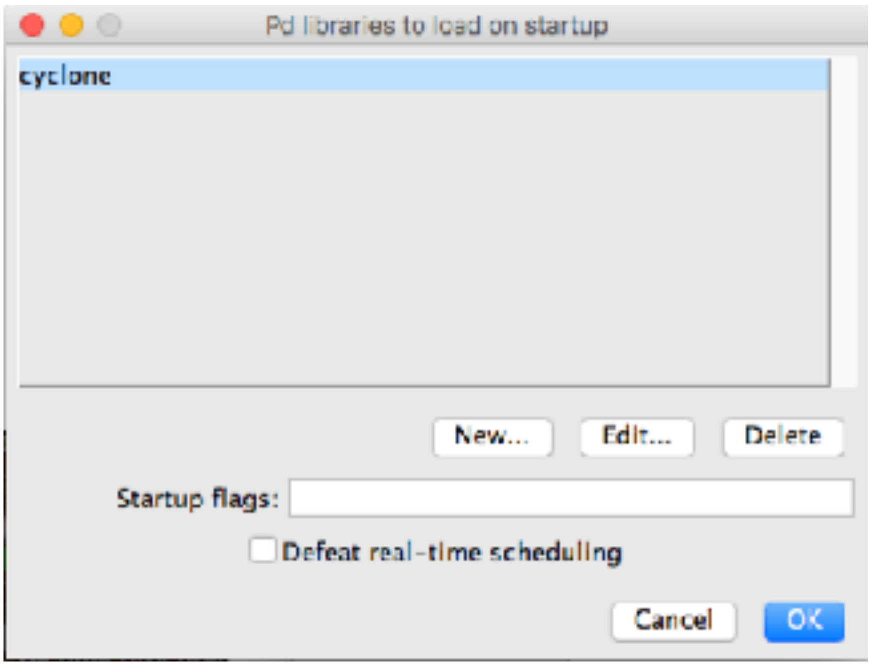
That would be simply like this:



3.2 - Startup

It needs to be clear you can’t define a “path” or use namespaces and slash declaration for objects that are loaded from a **single binary pack**. That is only valid for external objects that are either abstractions or compiled as a separate binary! For a **single binary pack** containing two or more external objects, you can use the Startup menu.

So, in order to load it, go to “**Preferences** => **Startup**”, where you can see the window named “*Pd libraries to load on startup*”. Next, click new and insert the binary/library name (it’s common that the **single binary pack** name is the sane name the library’s) and then click “OK” and the name of the library is listed.


Pd libraries to load on startup

cyclone

Figure 5 - Loading the cyclone binary/library in the “Path”

It's quite usual that a **single binary pack** includes all of the objects from a library; as previously mentioned, cyclone 0.3 is a special case that includes objects as abstractions, as a set of separate binaries but also has a set a single binary pack that loads objects with non alphanumeric names, which need to be loaded as a single binary pack to avoid issues. One such object is [+=~], a signal accumulator. So, after you have cyclone loaded in the startup, you can restart Pd and load such objects as below.

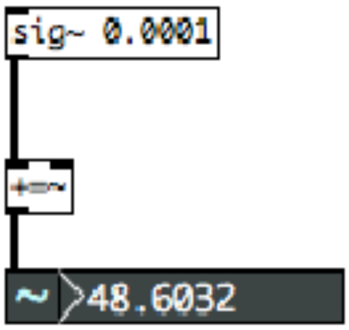


Figure 6 - Loading the [+=~] object from the cyclone binary

It is important that Pd knows where the **single binary pack** is. In the case of the above example, the single binary pack is found in the cyclone folder, which was included in one of the folders that Pd automatically searches for externals. In this case (when the binary has the same name of the folder it is included in and placed in one of the standard paths), you don't need to worry and specify the folder path. Otherwise, it would be necessary.

3.3 - Using the [declare] object:

The [declare] object from Pd Vanilla behaves quite similarly to using "Path" (with the *-path* or *-stdpath* flag) and "Startup" (with the *-lib* or *-stdlib* flag).

The *-path* flag defines a path relative to where the pd patch that loads it is saved. The *-stdpath* flag defines a path relative to any of the **Standard Paths** that Pd automatically searches for externals.

Similarly, the *-lib* flag defines a library relative to where the patch is saved, whereas the *-stdlib* flag defines a path relative to any of the **Standard Paths**. Find more details in the help file of the [declare] object.

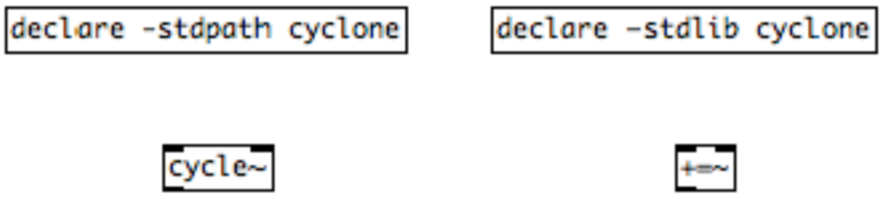

```
declare -stdpath cyclone
declare -stdlib cyclone
cycle~
+=~
```

Figure 7 - Loading the cyclone path and binary with [declare]