HOWTO Install & Load External Objects in Pd

by Alexandre Torres Porres

Index

Part 1) Introduction: Defining External Objects & Libraries — page 02

1.1 - What are: Vanilla Objects, Internals & Externals? — page 02

1.2 - What are the Types of External Objects? — page 03

1.2.1 Compiled objects: — page 04

1.2.2 Abstractions: — page 04

1.3 - What are External Libraries? — page 04

1.4 - What are the types of External Libraries? — page 05

Part 2) Installing External Objects and Libraries — page 05

2.1 - Where to include the externals? — page 06

2.2.1 - Where are the Standard Paths? — page 06

2.2 - How to Download Externals from Pd Vanilla? — page 08

Part 3) Loading Externals — page 09

3.1 - Slash declaration — page 09

3.2 - Loading with Path and Startup — page 10

3.2.1 - Path — page 10

3.2.2 - Startup — page 11

3.3 - Using the [declare] object — page 13

Part 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.

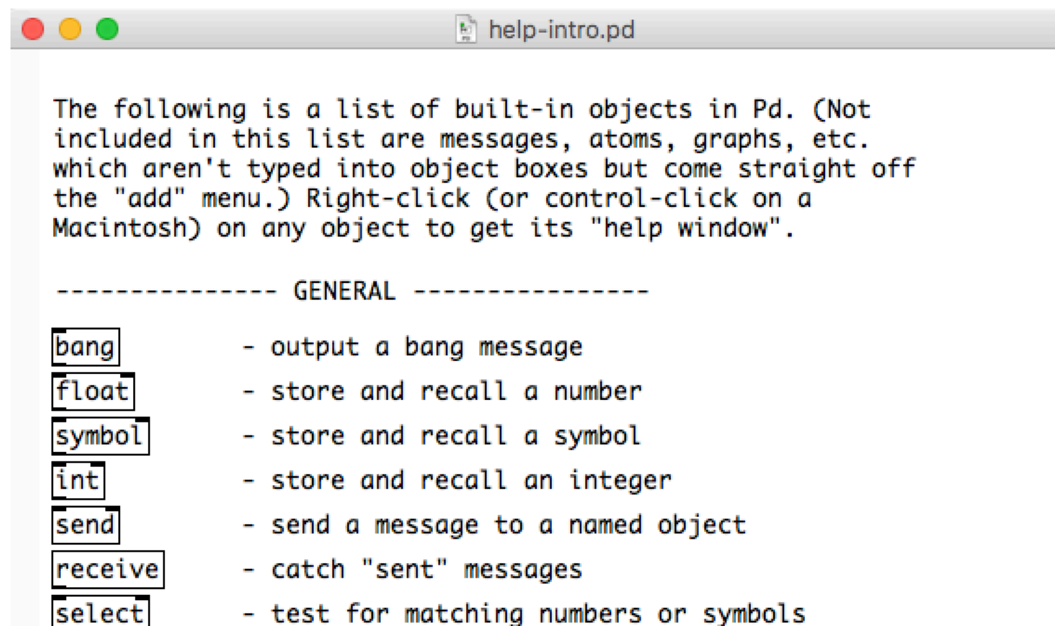

Figure 1 - Excerpt from list of Vanilla objects (help-intro.pd file)

---

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

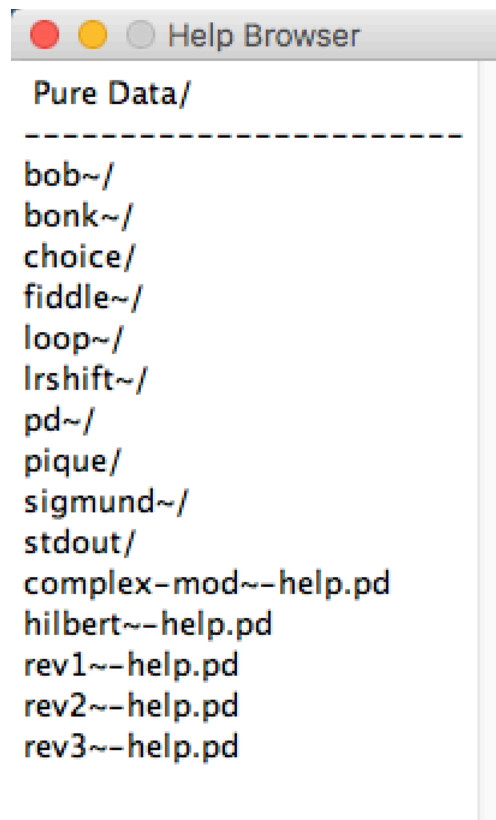

Figure 2 - Extra objects from Pd Vanilla

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 Figure 2 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; it can be **a set of separate binaries**, a **single binary pack**, a collection of abstractions (like “list-abs”), or 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:** Abstractions and Compiled binaries (compiled as a pack or separately).

- 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.

Part 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 and common practice is to have them in on the Standard Paths - which are folders Pd automatically looks for externals, they 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?

Besides the Application-specific folder, Pd Vanilla does not create any of the other folders, so you have to create them yourself. Here’s the list of the Standard Paths for all operating systems:

A) Mac OS:

- Application-specific: ***/$PdPath/Contents/Resources/extra*** - this is inside the Pd Application (like *Pd-0.47-1-64bit in ~/Applications);* right click it and choose “Show Package Contents”, then navigate to “extra”.

- User-specific: ***~/Library/Pd*** (*/Users/user_name/Library/Pd*)

- Global: ***/Library/Pd***

B) Windows:

- Application-specific: ***%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.

- User-specific: ***%AppData%\Pd***

(usually in *C:\Users\user_name\AppData\Roaming\Pd*).

- Global: ***%CommonProgramFiles%\Pd***

(usually in *C:\Program Files\Common Files\Pd*).

C) GNU/Linux:

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

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

- Global: ***/usr/local/lib/pd-externals***.

One of the advantages of using a Standard Path to install externals is that they show up in the Help Browser (Help Menu => Browser). See below.

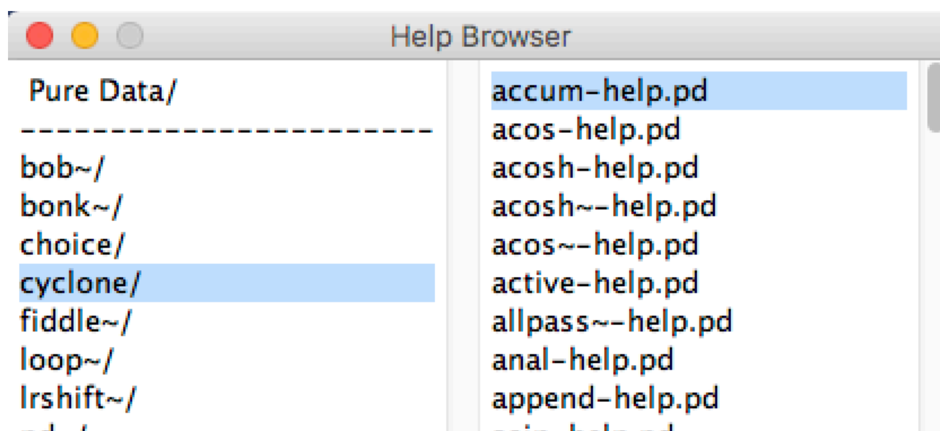

Figure 3 - Help Browser showing the cyclone library, which was installed in one of the Standard Paths

2.2 - How to Download Externals from Pd Vanilla?

Since version 0.47, Pd has its own external downloader! This is a built in *.tcl* plug-in named ‘deken’ <https://github.com/pure-data/deken>. Many externals are still only found elsewhere on the internet, though anyone can upload externals to be available from Pd, but you should only install the ones uploaded by people you trust.

To download from Pd, just go to the “Help” menu and select “Find Externals”. Then you can just type the library’s name you want and hit enter or click “search”. if an external list has also been uploaded, you can look for an external name and the library that contains it is shown.

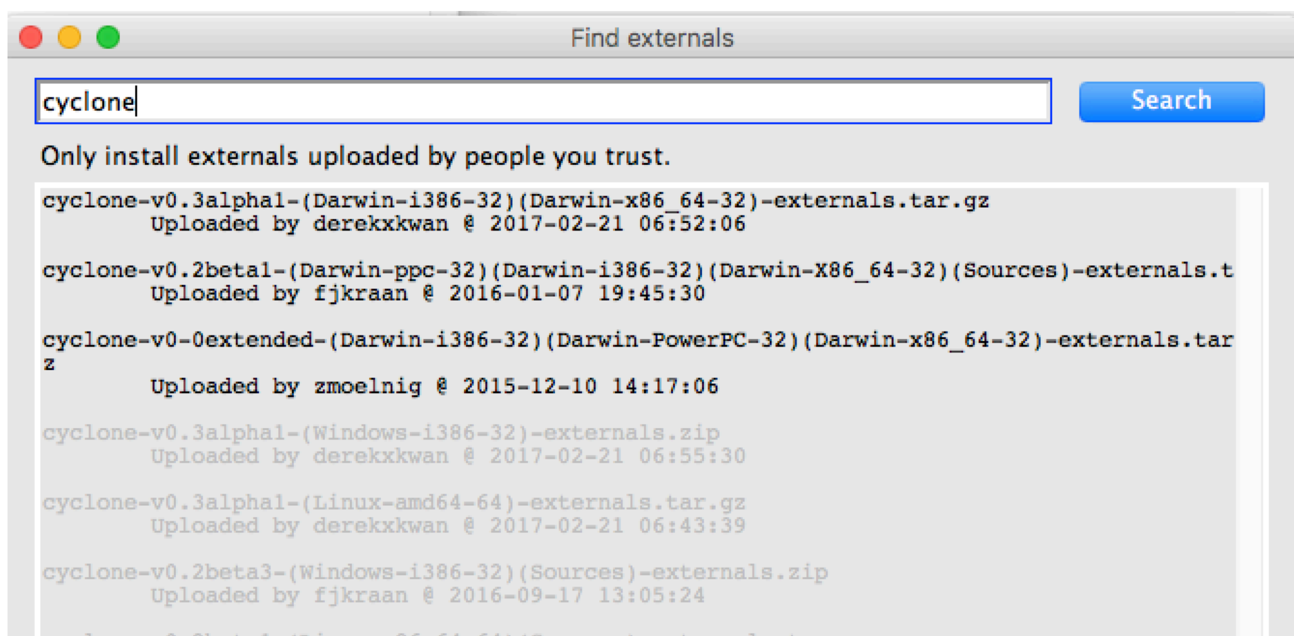

Figure 4 - Searching for External Libraries from Pd Vanilla

All available versions of the searched library will be shown to you, but the versions specific to your system are highlighted. Click on the version you want to download it. See figure above.

By default, Pd will search for Standard Paths to download to. The User-Specific folder has a priority and shows up first if it's been created. If neither the Global or User-Specific Standard Paths have been created, then the Application-Specific folder is chosen, but if you do not have writing privileges to this folder, Pd will prompt you to download to the User-Specific Standard Path, and create it if you click "yes".

You can also specify a different folder of your choice to download to and Pd will remember that choice in the future. As soon as the download is finished, the compressed file is automatically decompressed into the chosen folder.

Part 3) Loading Externals

3.1 - Slash declarations:

Note that "Slash Declarations" only work for external objects that are either abstractions or compiled as a **separate binary**!

Pd automatically searches for externals in default folders. They are:

- The **Relative Path** folder (where the patch is saved on).
- The **Standard Paths** (Application-specific, User-specific & Global).

Therefore, if you have an abstraction or an object compiled as a **separate binary** in any of these folders, the object will be loaded without the need of anything else.

As mentioned, externals are usually installed in one of the Standard Paths. Having externals in a Relative Path is uncommon, this usually happens only 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 this doesn't work unless you specify the folder.

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 (that is, the "cyclone" library comes as a folder named "cyclone"). Therefore, you need to specify which folder the object is as well.

One way to specify a folder is with "Slash Declarations", which simply involves inserting the folder/library name where the object is found. For example, given that the "cyclone" library is in a Standard Path, loading the [cycle~] external from it can be done like this:

cyclone/cycle~

The example below loads an abstraction called [abs] that is found inside a subfolder named "testpd", located in the Relative Path:

testpd/abs

3.2 - Loading with Path and Startup:

3.2.1 - Path:

This is also in respect to abstractions or objects compiled as a **separate binary.** If you add a library folder to Pd's Path (Pd menu => Preferences => Path), you don't need to use slash declarations! Any path containing externals may be included, but this is normally one of **Standard Paths**, as described before. See figure below, where a folder in the User-specific folder is added.

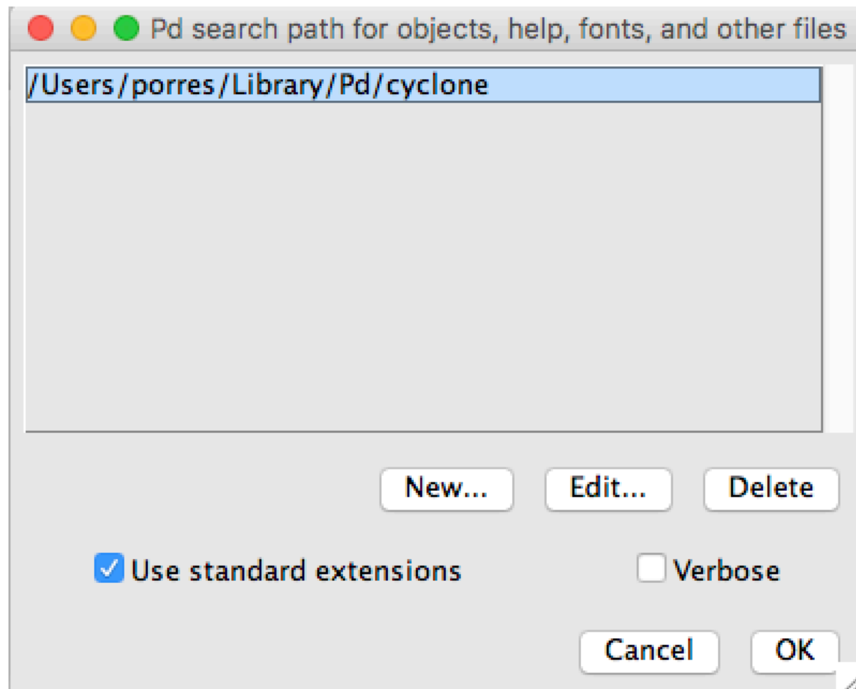

Figure 5 - Loading the cyclone library in the “Path”
in Mac OS’ User-Specific Standard Path

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

cycle~

3.2.2 - Startup

It needs to be clear you can’t define a “path” or use slash declarations 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! Instead, you can load a **single binary pack** containing two or more external objects with the Startup menu.

For that, go to “Pd menu => 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” so the name of the library is loaded and listed.

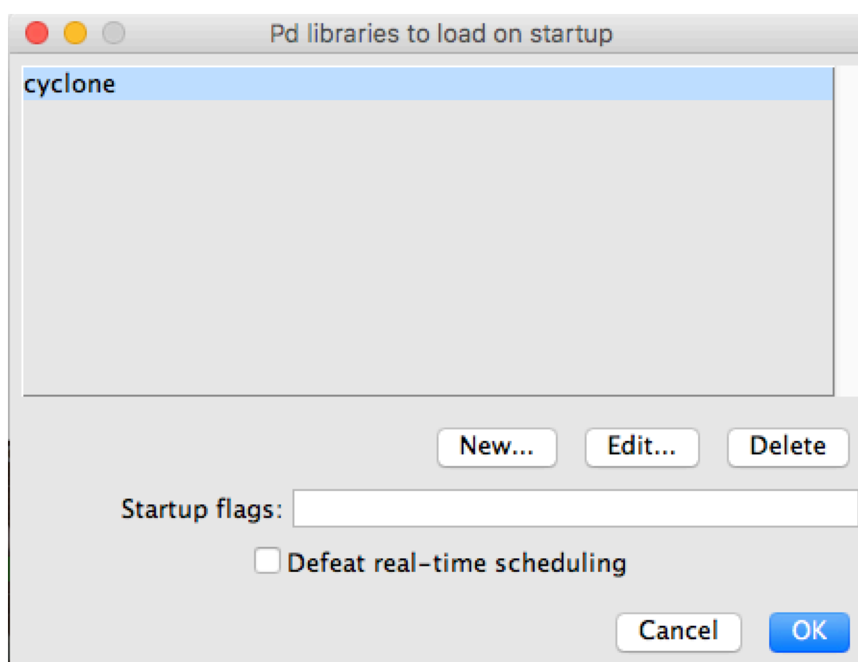

Figure 6 - Loading the cyclone binary/library in “Startup”

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 . 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.

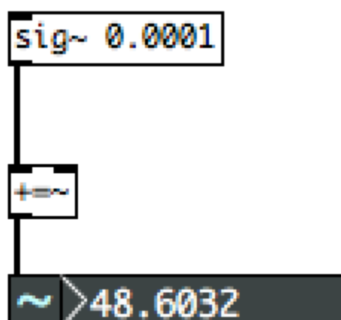

Figure 7 - Loading the [+=~] object from the cyclone binary

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 Standard Paths (folders that Pd automatically searches for externals). In this case, where 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, defining a Path 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 (that is: the "Relative Path"). The *-stdpath* flag defines a path relative to any of the **Standard Paths** that Pd automatically searches for externals. But both can actually take absolute paths.

Similarly, the *-lib* flag loads a library (that is: a “binary pack”) relative to where the patch is saved (“Relative Path”), whereas the *-stdlib* flag loads a library in any of the **Standard Paths**. Find more details in the help file of the [declare] object.

Below, we have an example of using [declare] to load cyclone from a Standard Path. We load the cyclone folder to the path with *-stdpath* so we can access abstractions and objects compiled as separate binaries, and we load cyclone’s **single binary pack** with *-stdlib*.

```
[declare -stdpath cyclone]    [declare -stdlib cyclone]

[cycle~]                      [+=~]
```

Figure 8 - Loading the cyclone path and binary with [declare]