Python3 Externals for Max/MSP

A PyJS Project Guide

S. Alireza

2024-07-19

Exploring different ways one can use Python3 in Max/MSP

Table of Contents

List of Figures

List of Tables

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

The Project

1 py-js: python3 objects for max

Simple (and extensible) python3 externals for MaxMSP.

Cross-platform: currently builds 'natively' on macOS (`x86_64` or `arm64`) and Windows (with MSVC).

repo - https://github.com/shakfu/py-js

pdf documentation - Python3-Externals-for-Max-MSP.pdf

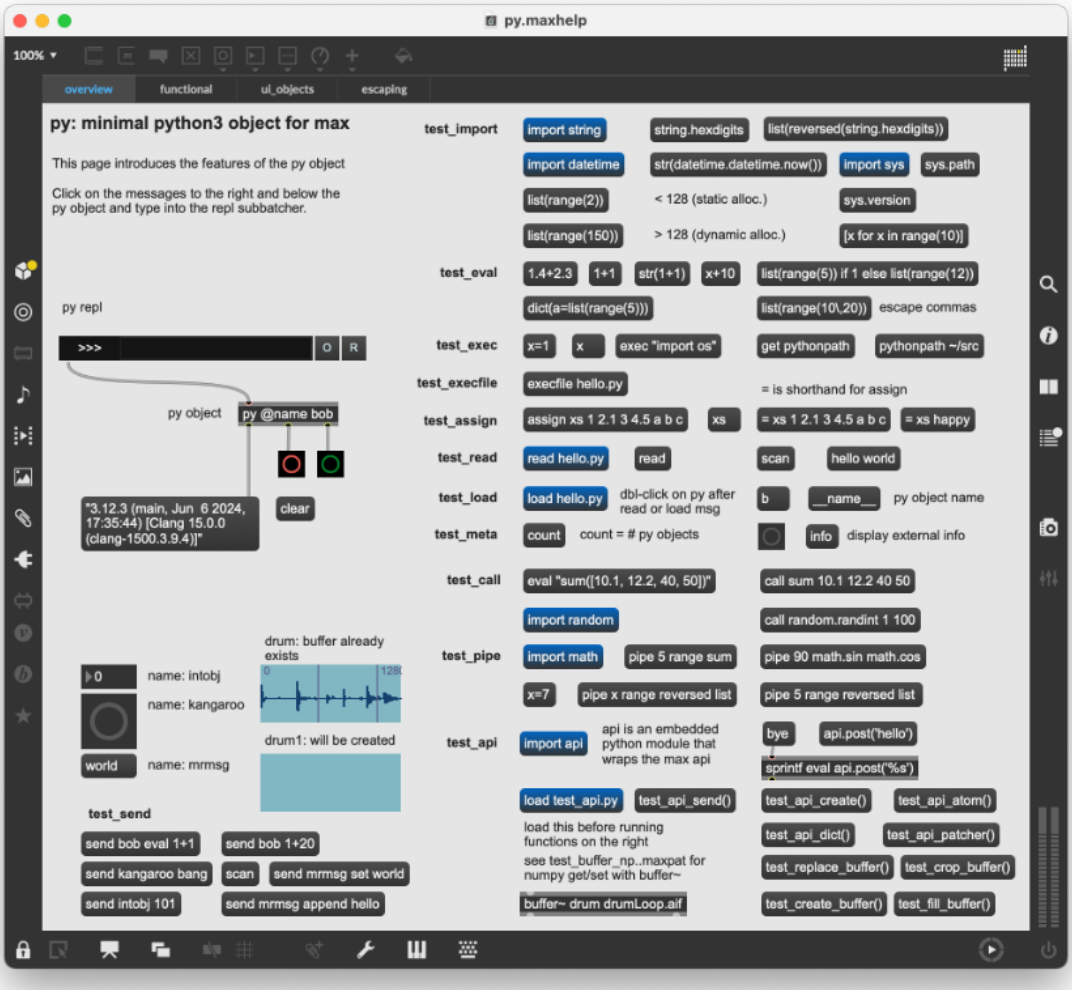


Figure 1.1: py-js py.maxhelp

1.1 Overview

This project started out as an attempt (during a covid-19 lockdown) to develop a basic python3 external for Max/MSP. It then evolved into an umbrella project for exploring different ways of using python3 in Max/MSP.

Along the way, a number of externals have been developed for use in a live Max environment (click on the links to see more detailed documentation):

Python3 Core Externals:

name	sdk	lang	description
py	max-sdk	c	well-featured, many packaging options + cython api
pyjs	max-sdk	c	js/v8-friendly – written as a Max javascript-extension

These two externals have many build options covering almost all deployment scenarios, but if you just want to get up and running quickly use the cmake option `-DBUILD_PYTHON3_CORE_EXTERNALS=ON` or just `make core`.

Python3 Experimental Externals:

name	sdk	lang	description
mamba	max-sdk	c	single-header c library to nest a python3 interpreter in any external
krait	max-sdk	c++	single-header c++ library to nest a python3 interpreter in any external
cobra	max-sdk	c	python3 external providing deferred and clocked function execution
mxpy	max-sdk	c	a translation of pdpython into Max
zedit	max-sdk	c	a web-based python editor using codemirror and the mongoose embedded webserver.
pymx	min-api	c++	concise python3 external, modern, using pybind11 and min-api

With the exception of `pymx`, which has been moved to its own github project because it uses the min-api SDK, the experimental subgroup can be built as a whole with the cmake option, `-DBUILD_PYTHON3_EXPERIMENTAL_EXTERNALS=ON`, or just `make experimentals`.

Alternative Python Implementation Externals:

name	sdk	lang	description
pktpy	max-sdk	c++	uses v.1.4.6 of the pocketpy single-header c++ library
pktpy2	max-sdk	c	uses the newer v2.0.5 pocketpy c11 library
mpy [2]	max-sdk	c	a proof-of-concept embedding micropython

The two pocketpy variants can be built with the cmake option, `-DBUILD_POCKETPY_EXTERNALS=ON` or `make pocketpy`, whereas the external based on micropython, `mpy` is not enabled by default since it is still in early stages and more of a proof-of-concept to embed micropython in an

external. To build it use the `-DBUILD_MICROPYTHON_EXTERNAL=ON` option with cmake or `make mpy`.

Networking Externals:

name	sdk	lang	description
ztp	max-sdk	c	non-blocking communications with zeromq + threads + spawned python
jmx	max-sdk	c	explore how to embed a jupyter client or kernel in an external
zpy	max-sdk	c	uses zeromq for 2way-comms with an external python process

Note: networking (zmq-based) externals are not enabled by default since they require zeromq libraries to be installed. To build them use the `-DBUILD_NETWORKING_EXTERNALS=ON` option with cmake or just `make net`

The common objective in these externals is to help use and distribute python code and libraries in Max applications. Many can be considered experimental, with 80% of development time going to the first two externals (`py` and `pyjs`), and `py` receiving the most recent attention, due to ongoing development of its builtin `api` module which uses cython to wrap a growing subset of the Max c-api.

At the time of this writing, and since the switch to max-sdk-base, the project has the following compatibility profile:

- **MacOS**: both x86_64 and Apple Silicon compatible. Note that the project intentionally only produces 'native' (`x86_64` xor `arm64`) externals with no current plans for 'fat' or universal externals to serve both architectures. You can download codesigned, notarized `x86_64`-based and `arm64`-based python3 externals from the releases section.
- **Windows**: windows support was provided relatively recently, with currently all Python3 externals and also the `pktpy` projects building without issues on Windows. The only caveat is that as of this writing python3 externals are dynamically linked to the local Python3 `.dll` and are therefore not relocatable. One idea to overcome this constraint is to include the external's dependencies in the 'support' folder. This will hopefully be addressed in future iterations. The `pktpy` external, however, is fully portable and self-contained.

The Quickstart section below covers general setup for all of the externals, and will get you up and running with the `py` and `pyjs` externals. More details are provided in the py documentation section: the Building Experimental Externals using Cmake section provides additional info to build the other remaining externals, and the Building self-contained Python3 Externals

for Packages and Standalones section covers more advanced building and deployment scenarios for the `py` and `pyjs` externals with details about their many build variations available via a custom python-based build system which was specifically developed to cater for different scenerios of packaging and deploying the externals in Max packages and standalones.

If you are interested in more detailed project-specific documentation, please refer to the links in the Overview section which link to respective project folders in the `py-js/source/projects` section.

Please feel free to ask questions or make suggestions via the project's github issue tracker.

1.2 Quickstart

This repo has a git submodule dependency with max-sdk-base. This is quite typical for Max externals.

This means you should `git clone` as follows:

```
git clone https://github.com/shakfu/py-js.git
git submodule init
git submodule update
```

or more concisely:

```
git clone --recursive https://github.com/shakfu/py-js.git
```

MacOS

As mentioned earlier, the `py` and `pyjs` objects are the most mature and best documented of the collection. Happily, there is also no need to compile them as they are available for download, fully codesigned and notarized, from the releases section.

If you'd rather build them or any of the other externals yourself then the process is straightforward:

1. You should have a modern `python3` cpython implementation installed on your Mac: preferably either from python.org or from Homebrew. Note that even system python3 provided by Apple will work in a number of cases. Python versions from 3.8 to 3.13 are tested and known to work.
2. Make sure you also have Xcode installed.

3. Git clone the `py-js` repo as per the above method to a path without a space and without possible icloud syncing (i.e don't clone to `$HOME/Documents/Max 8/Packages`) [?] and run the following in the cloned repo:

   ```
   make setup
   ```

   The above will initialize and update the required git submodules and symlink the repo to `$HOME/Documents/Max 8/Packages/py-js` to install it as a Max Package and enable you to test the externals and run the patches.

   [?] It is possible to install `py-js` directly into `$HOME/Documents/Max 8/Packages`, but it requires moving the place of compilation to a location in your filesystem that is not exposed to errors due to icloud syncing or spaces in the path. This split is possible, but it is not recommended for the purposes of this quickstart.

4. Optionally, install cython via `pip3 install cython`, if you want to make changes to the cython-based `api.pyx` module, which wraps the Max c-api.

5. To build only the `py` and `pyjs` externals, type the following in the root directory of the `py-js` project (other installation options are detailed below):

   ```
   make
   ```

   Note that typing `make` here is the same as typing `make default` or `make all`. This will create two externals `py.mxo` and `pyjs.mxo` in your `externals` folder. These are relatively small in size and are linked to your system python3 installation. This has the immediate benefit that you have access to your curated collection of existing python packages.

   The `make` or `make default` command bypasses an intermediate buildsystem and builds the two core externals via Xcode

   Another build option for core externals is to use `cmake` as an intermediate build system to drive Xcode builds:

   ```
   make core
   ```

The `make`, or `make core` methods of building core the externals are generally very fast and produce externals which have access to python libraries of the local python system referemced during compilation. The tradeoff is that since the external are dynamically linked with local dependencies, they are therefore not usable in standalones and relocatable Max packages.

No worries, if you need portable relocatable python3 externals for your package or standalone and more granular build options then make sure to read the Building self-contained Python3 Externals for Packages and Standalones section.

In any case, open up any of the patch files in the `patchers` directory of the repo or the generated Max package, and look at the `.maxhelp` patchers to understand how the `py` and the `pyjs` objects work.

Windows

Since Windows support still is relatively new, no releases have been made pending further testing.

Currently, the externals which are enabled by default in this project can be built with only a few requirements:

1. Install Visual Studio Community Edition or use the commercial versions as you like.
2. Install Python3 for Windows from python.org
3. (Optional) since Visual Studio has its captive cmake, you can use that, but it is preferable to install cmake independently.

After installation of the above you can build the externals inside your `Documents/Max 9/Packages` folder as follows:

```
git clone --recursive https://github.com/shakfu/py-js
cd py-js
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build . --config Release
```

Open one of the `.maxhelp` files or any of the files in the `patchers` folders to see how things work.

Building Externals using Cmake

You can also use `cmake` to build **all** externals using similar methods to the `max-sdk`.

First make sure you have completed the Quickstart section above. Next you will install `cmake` if necessary and a couple of additional dependencies for some of the subprojects. Of course, skip what is already installed:

```
brew install cmake
```

Now you can build almost all the externals (including `py` and `pyjs`) in one shot using cmake:

```
make projects
```

This builds all of the externals *except* for the networking variants, since they have some additional dependencies related to the lightweight networking zeromq library.

If you would like to build them first install:

```
brew install zmq czmq
```

Then

```
make net
```

Look at the `Makefile` and the root `CMakeLists.txt` for further specialized build options.

After doing the above, the recommended iterative development workflow is to make changes to the source code in the respective project and then `cd py-js/build` and `cmake --build ..` This will cause cmake to only build modified projects efficiently.

Note that for some of the less developed externals and more experimental features please don't be surprised if Max occassionaly segfaults (especially if you start experimenting with the the more experimental parts of the cython wrapped `api` module which operates on the c-level of the Max SDK).

Also note that for `py` and `pyjs` externals the `cmake` build method described does not yet create self-contained python externals which can be used in Max Packages and Standalones. The Building self-contained Python3 Externals for Packages and Standalones section addresses this requirement.

Visit the project-specific links above for more detailed documentation.

1.3 Related projects

- relocatable-python: A tool for building standalone relocatable Python.framework bundles. (used in this project)
- python-build-standalone: Produce redistributable builds of Python. (Interesting but not used in this project)
- Python Apple Support: A meta-package for building a version of Python that can be embedded into a macOS, iOS, tvOS or watchOS project. (directly inspired static linking approach)
- python-cmake-buildsystem: A cmake buildsystem for compiling Python. Not currently used in this project, but may be used in the future.
- py2max : using python3 with Max in an offline capacity to generate max patches.

- maxutils : scripts and utilities to help with codesigning and notarization of Max standalones and externals.
- pocketpy: C++17 header-only Python interpreter for game engines.
- micropython: a lean and efficient Python implementation for microcontrollers and constrained systems

1.4 Prior Art and Thanks

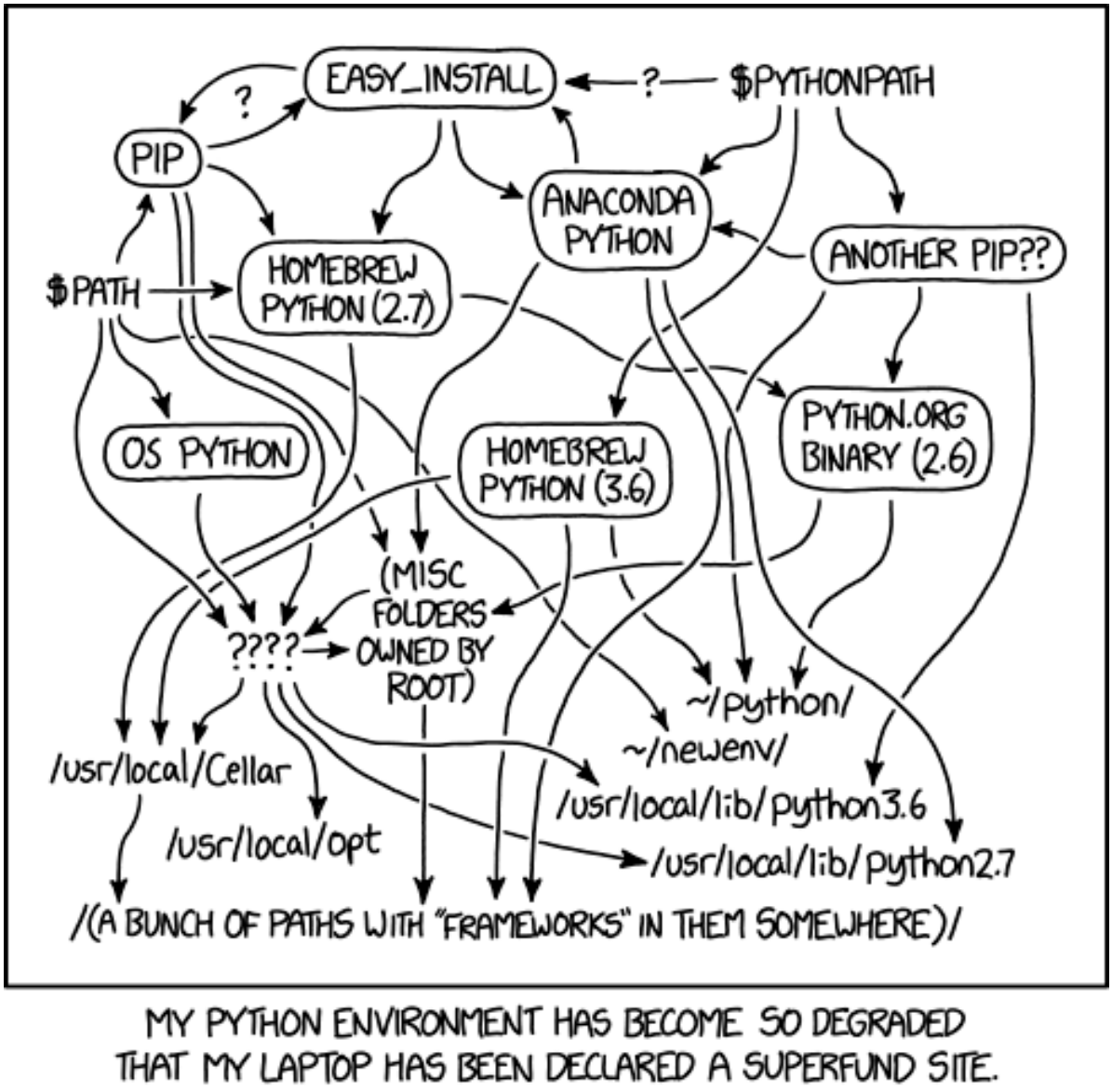


Figure 1.2: py-js testing

I was motivated to start this project because I found myself recurrently wanting to use some python libraries or functions in Max.

Looking around for a python max external I found the following:

- Thomas Grill's py/pyext – Python scripting objects for Pure Data and Max is the most mature Max/Python implementation and when I was starting this project, it seemed very promising but then I read that the 'available Max port is not actively maintained.' I also noted that it was written in C++ and that it needed an additional c++ flext layer to compile. I was further dissuaded from diving in as it supported, at the time, only python 2 which seemed difficult to swallow considering Python2 is basically not developed anymore. Ironically, this project has become more active recently, and I finally was persuaded to go back and try to compile it and finally got it running. I found it to be extremely technically impressive work, but it had probably suffered from the burden of having to maintain several moving dependencies (puredata, max, python, flext, c++). The complexity probably put off some possible contributors which would have made the maintenance of the project easier for Thomas. In any case, it's an awesome project and it would be great if this project could somehow help py/ext in some way or the other.
- max-py – Embedding Python 2 / 3 in MaxMSP with `pybind11`. This looks like a reasonable effort, but only 9 commits and no further commits for 2 years as of this writing.
- nt.python_for_max – Basic implementation of python in max using a fork of Graham Wakefield's old c++ interface. Hasn't really been touched in 3 years.
- net.loadbang.jython – A jython implementation for Max which uses the MXJ java interface. It's looks like a solid effort using Jython 2.7 but the last commit was in 2015.

Around the time of the beginning of my first covid-19 lockdown, I stumbled upon Iain Duncan's Scheme for Max project, and I was quite inspired by his efforts and approach to embed a scheme implementation into a Max external.

So it was decided, during a period with less distractions than usual, to try to make a minimal python3 external, learn the max sdk, the python c-api, and also how to write more than a few lines of c that didn't crash.

It's been an education and I have come to understand precisely a quote I remember somewhere about the c language: that it's "like a scalpel". I now understand this to mean that in skilled hands it can do wonders, otherwise you almost always end up killing the patient.

Thanks to Luigi Castelli for his help with Max/MSP questions, to Stefan Behnel for his help with Cython questions, and to Iain Duncan for providing the initial inspiration and for saving me time with some great implementation ideas.

Thanks to Greg Neagle for zeroing in on the relocatability problem and sharing his elegant solution for Python frameworks via his relocatable-python project on Github.

2 FAQ

2.1 Compatibility

Does this project work with Windows?

Yes, although Windows support came relatively recently, all python externals and also the `pktpy` external can be compiled using the default MSVC buildsystem.

Currently only building via cmake with dynamic linking of externals is supported. Refer to the README quickstart for compilation instructions.

Does this project work with macOS?

Yes, macOS support is the most mature for both x86_64 (Intel) and arm64 (Apple silicon). Note that externals are only built natively (no univeral binaries to keep the size of externals under control).

Python3 installed from python.org works as expected, and even Apple installed system python3 has been tested to work on the default build and some but not all of the other build variants.

Basically, there is no intrinsic reason why it shouldn't work with python3 on your system if the python3 version >= 3.7

Refer to the README quickstart for compilation instructions.

Can I use Homebrew python on macOS?

Definitely, if python is installed using Homebrew you can create externals using `make`, `make projects`, `make homebrew-ext` and `homebrew-pkg` depending on your requirement. See `make help` for some guidance on which target to build.

2.2 Implementation

Is the python interpreter embedded in the external or does the external basically use the host's python interpreter?

The `make default` build creates a lightweight external dynamically linked to your local python3 interpreter; other build variants such as `framework-pkg` embeds python3 into an external that is dynamically linked to a python3 interpreter which is part of the containing Max package; and another such as `framework-ext` embeds python into the external itself without any dependencies. There are other ways as well. The section, Building self-contained Python3 Externals for Packages and Standalones, gives an overview of the different approaches.

Recent work on the `mamba` single-header c library project makes it possible build relocatable python3 externals along the lines of what is possible with `py` and `pyjs`

2.3 Installation

Can I use two different python3 externals in the same patch?

Python3 external types which are not 'isolated' (see the build variations section) cannot be loaded at the same time. So for example, if you build a `framework-ext` variation, `py.mxo` and `pyjs.mxo` will not work together in the same patch, but if you built a `framework-pkg` variation of the same two externals, they should work fine without issues.

Of course, it would be considered redundant to install two different python3 external types in your project at the same time.

2.4 Logging

Every time I open a patch there is a some debug information in the console. Should I be concerned?

It looks like someone left (**debug?**)=on in this patch and it further may have cached some paths to related on the build system in the patch. You should be able to switch it off by setting (**debug?**)=off. If they still remain, open the `.maxpat` in question in an editor (it's a JSON file) and remove the cached paths.

You can also go to the external c file itself and hardcode DEBUG=0 if you want to switch logging off completely.

2.5 Extensibility

How to extend python with your own scripts and 3rd party libraries?

The easiest solution is to use an external that's linked to your system python3 installation (and not a self-contained external). This is what gets built if you run `make` or `make projects` in the root of the py-js project. If you do it this way, you automatically get access to all of your python libraries, but you give up portability.

This release contains relocatable python3 externals which are useful for distribution in packages and standalones so it's a little bit more involved.

First note that there several ways to add code to the external:

1. The external's site-packages: `py-js/externals/py.mxo/Contents/Resources/lib/python3.X/site-packages`
2. The package `examples` folder: such as `py-js/examples/scripts`
3. Whichever path you set the patcher PYTHONPATH property to (during object creation).

For (1), if you make changes to the external by adding modules or compiled extensions then you will have to re-codesign the external. This is straightforward in `py-js`, just make sure the externals are in the `externals` folder and `make sign`.

For (2), this is just a location that's searched automatically with `load`, `read`, and `execfile` messages so it can contain dependent files.

For (3), this is just a setting that is done at the patch level so it should be straightforward. As mentioned, the extra `pythonpath` is currently only set at object creation. It should be updated when it's changed but this is something on the todo list.

2.6 Specific Python package Compatibility

How to get numpy to work in py-js

The easiest way is to just create an adhoc python external linked to your system python3 setup. If you have `numpy` installed there, then you should be good to go with the following caveat: the type translation system does not currently automatically cover native `numpy dtypes` so they would have to be converted to normal lists before they become translated to to Max `lists`. This is not a hard constraint, just not implemented yet.

You can also add your system `site-packages` to the externals pythonpath attribute.

If you need numpy embedded in a portable variation of py-js, then you have a couple of options. A py-js package build which has 'thin' externals referencing a python distribution in the `support` folder of the package is the way to go and is provided by the `homebrew-pkg` build option for example.

It is also possible to package numpy in a fully relocatable external. It used to be quite involved, and can currently only be done with non-statically built relocatable externals, but a make target has been added to do this. The make command to do this is `make install-numpy`.

Part II

Core Externals

3 py: a general purpose python3 max external

3.1 Overview

This overview will cover the following the py external implementation:

The py external provides a more featureful two-way interface between max and python in a way that feels natural to both languages.

The external has access to builtin python modules and the whole universe of 3rd party modules, and further has the option of importing a builtin `api` module which uses cython to wrap selective portions of the max c-api. This allows regular python code to directly access the max-c-api and script Max objects.

A general purpose Max external that embeds a python3 interpreter and is made up of three integrated parts which make it quite straightforward to extend:

1. The `py` Max external which is written in `c` using both the `Max c-api` and the `Python3 c-api`.
2. A pure python module, `py_prelude.py` which is converted to `py_prelude.h` and compiled with `py` and then pre-loaded into the `globals()` namespace of every `py` instance.
3. A powerful builtin `api` module which is derived from a cython-based wrapper of a subset of the `Max c-api`.

The following cheatsheat provides a brief view of key attributes and methods:

```
globals
    obj_count                   : number of active py objects
    registry                    : global registry to lookup object names

patchers
    subpatchers
        py_repl                 : a basic single line repl for py
        py_repl_plus            : embeds a py object in a py_repl

py max external
    attributes
```

```
        name                        : unique object name
        file                        : file to load into editor
        autoload                    : load file at start
        pythonpath                  : add path to python sys.path
        debug                       : switch debug logging on/off

    methods (messages)
        core
            import <module>         : python import to object namespace
            eval <expression>       : python 'eval' semantics
            exec <statement>        : python 'exec' semantics
            execfile <path>         : python 'execfile' semantics

        extra
            assign <var> [arg]      : max-friendly msg assignments to py object namespace
            call <pyfunc> [arg]     : max-friendly python function calling
            pipe <arg> [pyfunc]     : process py/max value(s) via a pipe of py funcs
            fold <f> <n> [arg]      : applies a two-arg function cumulatively to a sequence
            code <expr|stmt>        : alternative way to eval or exec py code
            anything <expr|stmt>    : anything version of the code method

        time-based
            sched <t> <fn> [arg]    : defer a python function call by t millisecs

        code editor
            read <path>             : read text file into editor
            load <path>             : combo of read <path> -> execfile <path>
            run                     : run the current code in the editor

        interobject
            scan                    : scan patcher and store names of child objects
            send <msg>              : send an arbitrary message to a named object

        meta
            count                   : give a int count of current live py objects

    inlets
        single inlet                : primary input (anything)

    outlets
        left outlet                 : primary output (anything)
        middle outlet               : bang on failure
```

```
        right outlet              : bang on success
```

Key Features

The py external has the following c-level methods:

category	method	param(s)	in/out	can change ns
core	import	module	in	yes
core	eval	expression	out	no
core	exec	statement	in	yes
core	execfile	file	in	yes
extra	assign	var, data	in	yes
extra	call	var(s), data	out	no
extra	code	expr or stmt	out?	yes
extra	anything	expr or stmt	out?	yes
extra	pipe	var, funcs	out	no
extra	fold	f, n, args	out	no
time	sched	ms, fun, args	out	no
editor	read	file	n/a	no
editor	load	file	n/a	no
interobj	scan		n/a	no
interobj	send	name, msg, ..	n/a	no
meta	count		n/a	no

Note that he code method allows for import/exec/eval of python code, which can be said to make those 'fit-for-purpose' methods redundant. However, it has been retained because it provides additional strictness and provides a helpful prefix in messages which indicates message intent.

Core

py/js's *core* features have a one-to-one correspondance to python's *very high layer* as specified here. In the following, when we refer to an *object*, we refer to instances of the py external.

- **Per-object namespaces**. Each object has a unique name (which is provided automatically or can be set by the user), and responds to an `import <module>` message which loads the specified python module in its namespace (essentially a `globals` dictionary). Notably, namespaces can be different for each instance.

- **Eval Messages**. Responds to an `eval <expression>` message in the left inlet which is evaluated in the context of the namespace. `py` objects output results to the left outlet, send a bang from the right outlet upon success or a bang from the middle outlet upon failure.
- **Exec Messages**. Responds to an `exec <statement>` message and an `execfile <filepath>` message which executes the statement or the file's code in the object's namespace. For `py` objects, this produces no output from the left outlet, sends a bang from the right outlet upon success or a bang from the middle outlet upon failure.

Extra

The *extra* category of methods makes the `py` object play nice with the max/msp ecosystem:

- **Assign Messages**. Responds to an `assign <varname> [x1, x2, ..., xN]` which is equivalent to `<varname> = [x1, x2, ..., xN]` in the python namespace. This is a way of creating variables in the object's python namespace using max message syntax. This produces no output from the left outlet, a bang from the right outlet upon success, or a bang from the middle outlet upon failure.
- **Call Messages**. Responds to a `call <func> arg1 arg2 ... argN` kind of message where `func` is a python callable in the py object's namespace. This corresponds to the python `callable(*args)` syntax. This makes it easier to call python functions in a max-friendly way. If the callable does not have variable arguments, it will alternatively try to apply the arguments as a list i.e. `call func(args)`. Future work will try make `call` correspond to a python generic function call: `<callable> [arg1 arg2 ... arg_n] [key1=val1 key2=val2 ... keyN=valN]`. This outputs results to the left outlet, a bang from the right outlet upon success, or a bang from the middle outlet upon failure.
- **Pipe message**. Like a `call` in reverse, responds to a `pipe <arg> <f1> <f2> ... <fN>` message. In this sense, a value is *piped* through a chain of python functions in the objects namespace and returns the output to the left outlet, a bang from the right outlet upon success, or a bang from the middle outlet upon failure.
- **Code or Anything Messages**. Responds to a `code <expression || statement>` or (anything) `<expression || statement>` message. Arbitrary python code (expression or statement) can be used here, because the whole message body is converted to a string, the complexity of the code is only limited by Max's parsing and excaping rules. (This is classified as EXPERIMENTAL and evolving).

Interobject Communication

- **Scan Message**. Responds to a `scan` message with arguments. This scans the parent patcher of the object and stores scripting names in the global registry.
- **Send Message**. Responds to a `send <object-name> <msg> <msg-body>` message. Used to send *typed* messages to any named object. Evokes a `scan` for the patcher's objects if a `registry` of names is empty.

Editing Support

- **Line REPL**. The `py` object has two bpatcher line `repls`: one, `py_repl_plux.maxpat` which embeds a `py` object and another, `py_repl.maxpat` which has an outlet to connect to one. The repls include a convenient menu with all of the `py` object's methods and also feature `coll`-based history via arrow-up/arrow-down recall of entries in a session. A `coll` can made to save all commands if required.
- **Multiedit REPL**. Another bpatcher, `py_multiedit.maxpat`, combines a `textedit` object for writing multiliine python code to be executed in the respective `py` external's namespace, and a simple line repl strictly for evaluating objects in the namespace.
- **External Editor Filewatcher**. `py_extedit.maxpat` is a bpatcher which wraps the `filewatcher` object and opens a *watched* file in an external editor. If the file is saved by the editor, it will be sent out as text via the outlet and can be received, for example, by the `py` object's inlet, to enable a kind of load-on-save workflow.
- **Code Editor**. Double-clicking on the `py` object opens a code-editor. This is populated by a `read` message which reads a file into the editor and saves the filepath to the external's attribute. A `load` message also `read`s the file followed by `execfile`. Saving the text in the editor uses the attribute filepath and execs the saved code to the object's namespace.
- **Experimental Remote Console**. A method (due to Iain Duncan) of sending code to the `py` node via `udp` has been implemented and allows for send-from-editor and send-from-interactive-console capabilities. The clients are still in their infancy, but this method looks promising since you get syntax highlighting, syntax checking, and other features. It assumes you want to treat your `py` nodes as remotely accessible `server/interpreters-in-max`.

```
zedit: [python interpreter / web server] <-> [web-editor / web-console]
```

- **`zedit`: a python3 external with an embedded web server**. `zedit` is a python3-enabled external by virtue of using the `mamba` single-header library and also embeds the mongoose embedded webserver. On the frontend, it uses modern javascript, jquery-terminal

and the widely-used code-mirror web text editor widget to create a web-editor / web-console which can be accessed from a browser and which communicates via the mongoose webserver with the underlying python interpreter.

Scripting Max with Python via the builtin `api` module

- **Exposing Max API to Python** A portion of the Max api in `c74support/max-includes` has been converted to a cython `.pxd` file called `api_max.pxd`. This makes it available for a cython implementation file, `api.pyx` which is converted to c-code during builds and embedded in the external. This code enables a custom python builtin module called `api` which can be imported by python scripts in `py` objects or via `import` messages to the object. This allows the subset of the Max-api which has been wrapped in cython code to be called directly by python scripts or via messages in a patcher.

The `api` module provides a number of functions and cython extension classes which make it relatively easy to call `Max c-api` methods from python. This is without doubt the most powerful feature of the `py` external.

As of this writing the following extension classes which wrap their corresponding Max objects are included in the `api` module: `Atom`, `AtomArray`, `Table`, `Buffer`, `Dictionary`, `Database`, `DatabaseView`, `DatabaseResult`, `Linklist`, `Binbuf`, `Hashtab`, `Patcher`, `MaxObject` and `MaxApp`.

In addition, a cython extension class, `PyExternal`, gives python code access to the c-based `py` external's data and methods.

To give a sense of the level of integration which is possible as a result of this module, the following example demonstrates how `numpy` and `scipy.signal` can be used to read and write to and from a live Max `buffer~` object using the `api` module's `Buffer` extension class:

```
import api

import numpy as np
from scipy import signal

def get_buffer_samples(name: str, sample_file: str) -> np.array:
    buf = api.create_buffer(name, sample_file)
    xs = np.array(buf.get_samples())
    assert len(xs) == buf.n_samples
    api.post(f"get {n_samples} samples from buffer {name}")
    return xs
```

```python
def set_buffer_samples(name: str, duration_ms: int):
    buf = api.create_empty_buffer(name, duration_ms)
    t = np.linspace(0, 1, buf.n_samples, endpoint=False, dtype=np.float64)
    xs = signal.sawtooth(2 * np.pi * 5 * t)
    buf.set_samples(xs)
    api.post(f"set {buf.n_samples} samples to buffer {name}")
```

See the `examples/tests` folder and the `patchers/tests` folder for more examples.

Deployment Scenarios

There are 3 general deployment variations:

1. **Linked to system python**. Linking the externals to your system python (homebrew, built from source, etc.) This has the benefit of re-using your existing python modules and is the default option.
2. **Embedded in package**. Embedding the python interpreter in a Max package: in this variation, a dedicated python distribution (zipped or otherwise) is placed in the `support` folder of the `py/js` package (or any other package) and is linked to the `py` external. This makes it size efficient and usable in standalones.
3. **Embedded in external**. The external itself as a container for the python interpreter: a custom python distribution (zipped or otherwise) is stored inside the external bundle itself, which can make it portable and usable in standalones.

As of this writing all three deployment scenarios are availabe, however it is worth looking more closely into the tradeoffs in each case, and the related build variations which exist.

Deployment Scenario	`py`
Linked to sys python	yes
Embeddded in package	yes
Embeddded in external	yes

3.2 Quickstart

This repo has a git submodule dependency with max-sdk-base. This is quite typical for Max externals.

This means you should `git clone` as follows:

```
git clone https://github.com/shakfu/py-js.git
git submodule init
git submodule update
```

or more concisely:

```
git clone --recursive https://github.com/shakfu/py-js.git
```

Windows

Since Windows support still is relatively new, no releases have been made pending further testing.

Currently, the externals which are enabled by default in this project can be built with only a few requirements:

1. Install Visual Studio Community Edition or use the commercial versions as you like.
2. Install Python3 for Windows from python.org
3. (Optional) since Visual Studio has its captive cmake, you can use that, but it is preferable to install cmake independently.

After installation of the above you can build the externals inside your `Documents/Max 8/Packages` folder as follows:

```
git clone --recursive https://github.com/shakfu/py-js
cd py-js
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
cmake --build . --config Release
```

Open one of the `.maxhelp` files or any of the files in the `patchers` folders to see how things work.

macOS

As mentioned earlier, the py and pyjs objects are the most mature and best documented of the collection. Happily, there is also no need to compile them as they are available for download, fully codesigned and notarized, from the releases section.

If you'd rather build them or any of the other externals yourself then the process is straightforward:

1. You should have a modern python3 cpython implementation installed on your Mac: preferably either from python.org or from Homebrew. Note that even system python3 provided by Apple will work in a number of cases. Python versions from 3.8 to 3.13 are tested and known to work.
2. Make sure you also have Xcode installed.
3. Git clone the py-js repo as per the above method to a path without a space and without possible icloud syncing (i.e don't clone to $HOME/Documents/Max 8/Packages) [?] and run the following in the cloned repo:

   ```
   make setup
   ```

   The above will initialize and update the required git submodules and symlink the repo to $HOME/Documents/Max 8/Packages/py-js to install it as a Max Package and enable you to test the externals and run the patches.

   [?] It is possible to install py-js directly into $HOME/Documents/Max 8/Packages, but it requires moving the place of compilation to a location in your filesystem that is not exposed to errors due to icloud syncing or spaces in the path. This split is possible, but it is not recommended for the purposes of this quickstart.
4. Install cython via pip3 install cython, required for translating the cython-based api.pyx, which wraps the the Max c-api, to c.
5. To build only the py and pyjs externals, type the following in the root directory of the py-js project (other installation options are detailed below):

   ```
   make
   ```

Note that typing make here is the same as typing make default or make all. This will create two externals py.mxo and pyjs.mxo in your externals folder. These are relatively small in size and are linked to your system python3 installation. This has the immediate benefit that you have access to your curated collection of existing python packages. The tradeoff is that these externals are dynamically linked with local dependencies and therefore not usable in standalones and relocatable Max packages.

No worries, if you need portable relocatable python3 externals for your package or standalone then make sure to read the Building self-contained Python3 Externals for Packages and Standalones section

Open up any of the patch files in the `patchers` directory of the repo or the generated Max package, and also look at the `.maxhelp` patchers to understand how the `py` and the `pyjs` objects work.

Building Experimental Externals using Cmake

You can also use `cmake` to build **all** externals using similar methods to the `max-sdk`.

First make sure you have completed the Quickstart section above. Next you will install `cmake` if necessary and a couple of additional dependencies for some of the subprojects. Of course, skip what is already installed:

```
brew install cmake zmq czmq
```

Now you can build all externals (including `py` and `pyjs`) in one shot using cmake:

```
make projects
```

After doing the above, the recommended iterative development workflow is to make changes to the source code in the respective project and then `cd py-js/build` and `cmake --build ..` This will cause cmake to only build modified projects efficiently.

Note that for some of the less developed externals and more experimental features please don't be surprised if Max seg-faults (especially if you start experimenting with the cython wrapped `api` module which operates on the c-level of the Max SDK).

Also note that for `py` and `pyjs` externals the `cmake` build method described does not yet create self-contained python externals which can be used in Max Packages and Standalones.

The following section addresses this requirement.

Building self-contained Python3 Externals for Packages and Standalones

The `py` and `pyjs` externals have a custom python build manager which provides the flexibility to create a number of `build variants` which can vary in size and features, or be selected depending on whether the external is to be packaged in standalones or Max packages.

The `Makefile` in the project root provides a simplified interface to this builder. See the Current Status of Builders section for further information.

idx	command	type	format	py size	pyjs size
1	`make static-ext`	static	external	9.0	8.8
2	`make static-tiny-ext`	static	external	6.7	6.2 [2]
3	`make shared-ext`	shared	external	16.4	15.8
4	`make shared-tiny-ext`	shared	external	6.7	6.2 [2]
5	`make framework-pkg`	framework	package	22.8	22.8 [3]

[2] In this table, size figures are for python 3.10.x but for python 3.11.4 they increase to 8.5 MB and 8.1 respectively. Generally, external size increases with each new python version as features are added, but this is also somewhat mitigated by the removal of deprecated builtin packages and extensions. If you want to achieve the theoretical minimal size for the `py` and `pyjs` externals, use python 3.8.x and/or a tiny variant (with a more recent version). Another option, if you need circa 1 MB size for a self-contained external, look at the `pktpy` subproject in this repo. Note the size of externals in Python 3.12.4 (although some of extra size is attributed improved ssl integration):

[3] Size, in this case, is not the individual external but the uncompressed size of the package which includes patches, help files and **both** externals. This can also vary by python version used to compile the external.

idx	command	type	format	py size	pyjs size
1	`make static-ext`	static	external	15.0	13.3
2	`make static-tiny-ext`	static	external	11.4	9.8
3	`make shared-ext`	shared	external	20.4	18.7
4	`make shared-tiny-ext`	shared	external	11.4	9.6
5	`make framework-ext`	shared	external	22.5	20.8

for Python 3.13.0, which implemented a number of deprecations, external sizes have come down a little:

idx	command	type	format	py size	pyjs size
1	`make static-ext`	static	external	14.4	12.6
2	`make static-tiny-ext`	static	external	10.2	8.5
3	`make shared-ext`	shared	external	19.1	17.3
4	`make shared-tiny-ext`	shared	external	10.6	8.8
5	`make framework-ext`	shared	external	21.2	20.2

This section assumes that you have completed the Quickstart above and have a recent python3 installation (python.org, homebrew or otherwise).

Again, if you'd rather not compile anything there are self-contained python3 externals which can be included in standalones in the releases section.

If you don't mind compiling (and have xcode installed) then pick one of the following options:

1. To build statically-compiled self-contained python3 externals:

   ```
   make static-ext
   ```

   You may also prefer the tiny variant:

   ```
   make static-tiny-ext
   ```

2. To build self-contained python3 exernals which include a dynamically linked libpythonX.Y.dylib:

   ```
   make shared-ext
   ```

   or for the corresponding tiny variant:

   ```
   make shared-tiny-ext
   ```

3. To build python3 externals in a package, linked to a python installation in its `support` folder

   ```
   make framework-pkg
   ```

With all of the above options, a python3 source distribution (matching your own python3 version) is automatically downloaded from python.org with dependencies, and then compiled into a static or shared version of python3 which is then used to compile the externals.

At the end of this process you should find two externals in the `py-js/externals` folder: `py.mxo` and `pyjs.mxo`.

Although the above options deliver somewhat different products (see below for details), with options (1) and (2) the external 'bundle' contains an embedded python3 interpreter with a zipped standard library in the `Resources` folder and also has a `site-packages` directory for your own code; with option (3), the externals are linked to, and have been compiled against, a relocatable python3 installation in the `support` folder.

Depending on your choice above, the python interpreter in each external is either statically compiled or dynamically linked, and in all three cases we have a self-contained and relocatable structure (external or package) without any non-system dependencies. This makes it appropriate for use in Max Packages and Standalones.

There are other build variations which are discussed in more detail below. You can always see which ones are available via typing `make help` in the `py-js` project folder:

```
$ make help

>>> general
make projects             : build all subprojects using standard cmake process

>>> pyjs targets
make                      : non-portable pyjs externals linked to your system
make homebrew-pkg         : portable package w/ pyjs (requires homebrew python)
make homebrew-ext         : portable pyjs externals (requires homebrew python)
make shared-pkg           : portable package with pyjs externals (shared)
make shared-ext           : portable pyjs externals (shared)
make shared-tiny-ext      : tiny portable pyjs externals (shared)
make static-ext           : portable pyjs externals (static)
make static-tiny-ext      : tiny portable pyjs externals (static)
make framework-pkg        : portable package with pyjs externals (framework)
make framework-ext        : portable pyjs externals (framework)
make relocatable-pkg      : portable package w/ more custom options (framework)

>>> python targets
make python-shared        : minimal shared python build
make python-shared-ext    : minimal shared python build for externals
make python-shared-pkg    : minimal shared python build for packages
make python-static        : minimal statically-linked python build
make python-framework     : minimal framework python build
make python-framework-ext : minimal framework python build for externals
make python-framework-pkg : minimal framework python build for packages
make python-relocatable   : custom relocatable python framework build
```

Automated Test of Build Variations

If you would like to see which build variations are compatible with your current setup, there's an automated test which attempts to compile all build variations in sequence and will log all results to a `logs` directory:

```
make test
```

This can take a long time, but it is worth doing to understand which variants work on your particular setup.

If you want to test or retest one individual variant, just prefix `test-` to the name of variant as follows:

```
make test-shared-pkg
```

Using Self-contained Python Externals in a macOS Standalone

If you have downloaded any pre-built externals from releases or if you have built self-contained python externals as per the methods above, then you should be ready to use these in a standalone.

To release externals in a standalone they must be codesigned and notarized. To this end, there are scripts in `py-js/source/projects/py/scripts` to make this a little easier.

py external

If you included `py.mxo` as an external in your standalone, then you should have no issue as Max will install it automatically during its build-as-standalone process.

You can test if it works without issues by building either of these two example patcher documents, included in `py-js/patchers`, as a max standalone:

1. `py_test_standalone_info_py.maxpat`
2. `py_test_standalone_only_py.maxpat`

Open the resulting standalone and test that the `py` object works as expected.

To demonstrate the above, a pre-built standalone that was built using exactly the same steps as above is in the releases section: `py_test_standalone_demo.zip`.

pyjs external

If you opted to include `pyjs.mxo` as an external in your standalone, then it may be a litte more involved:

You can first test if it works without issues by building 'a max standalone' from the `test_standalone_pyjs.maxpat` patcher which is included in `py-js/patchers/tests/test_standalone`.

Open the resulting standalone and test that the `pyjs` object works as expected. If it doesn't then try the following workaround:

To fix a sometimes recurrent issue where the standalone build algorithm doesn't pick up `pyjs.mxo`: if you look inside the built standalone bundle, `py_test_standalone_only_pyjs.app/Contents/Resou` you may not find `pyjs.mxo`. This is likely a bug in Max 8 but easily resolved. Fix it by manually copying the `pyjs.mxo` external into this folder and then copy the `javascript` and `jsextensions` folders from the root of the `py-js` project and place them into the

`pyjs_test_standalone.app/Contents/Resources/C74` folder. Now re-run the standalone app again and now the `pyjs` external should work. A script is provided in `py-js/source/projects/py/scripts/fix-pyjs-standalone.sh` to do the above in an automated way.

Please read on for further details about what the py-js externals can do.

Have fun!

3.3 Packaging

As mentioned previously, the py-js `builder` subproject can be used to build fit-for-purpose python variants for python3 externals. In addition, it can also package, sign, notarize and deploy the same externals for distribution.

These features are implemented in `py-js/source/project/py/builder/packaging.py` and are exposed via two interfaces:

The argparse-based interface of builder

```
$ python3 -m builder package --help
usage: builder package [-h] [-v VARIANT] [-d] [-k KEYCHAIN_PROFILE]
                       [-i DEV_ID]
                       ...

options:
  -h, --help            show this help message and exit
  -v VARIANT, --variant VARIANT
                        build variant name
  -d, --dry-run         run without actual changes.
  -k KEYCHAIN_PROFILE, --keychain-profile KEYCHAIN_PROFILE
                        Keychain Profile
  -i DEV_ID, --dev-id DEV_ID
                        Developer ID

package subcommands:
  package, sign and release external

                        additional help
    collect_dmg         collect dmg
    dist                create project distribution folder
```

```
    dmg                 package distribution folder as .dmg
    notarize_dmg        notarize dmg
    sign                sign all required folders recursively
    sign_dmg            sign dmg
    staple_dmg          staple dmg
```

The Project's Makefile frontend

Since the `Makefile` frontend basically just calls the `builder` interface in a simplified way, we will use it to explain the basic sequential packaging steps.

1. Recursively sign all externals in the `external folder` and/or binaries in the `support` folder

   ```
   make sign
   ```

2. Gather all project resources into a distribution folder and then convert it into a `.dmg`

   ```
   make dmg
   ```

3. Sign the DMG

   ```
   make sign-dmg
   ```

4. Notarize the DMG (send it to Apple for validation and notarization)

   ```
   make notarize-dmg
   ```

5. Staple a valid notarization ticket to the DMG

   ```
   make staple-dmg
   ```

6. Zip the DMG and collect into in the `$HOME/Downloads/PY-JS` folder

   ```
   make collect-dmg
   ```

To do all of the above in one step:

```
make release
```

Note that it is important to sign externals (this is done by Xcode automatically) if you want to to distribute to others (or in the case of Apple Silicon, even use yourself). If the externals are signed, then you can proceed to the notarization step if you have an Apple Developer License (100 USD/year) or, alternatively, you can ask users to remove the product's quarantine state or let Max do this automatically on opening the external.

Notarization Requirements

To complete the notarization process, an Apple Developer Account and an app-specific password are required.

1. Create local credentials based on your apple developer id and app-specific password

```
xcrun notarytool store-credentials "<keychain-profile-name>" --apple-id "<apple-id>" --tea
```

2. Export `DEV_ID` and `KEYCHAIN_PROFILE` environment variables:

```
export DEV_ID="<first> <lastname>"
export KEYCHAIN_PROFILE="<name-of-credentials>"
```

3. Run the whole process (i.e. steps 1-6) with one command:

```
make release
```

Github Actions

There are a number of Github actions in the project which basically automate the testing, packaging, and possibly the notariztion steps described above.

Caveats

- The externals in this project have been mostly developed on MacOS and have not yet been extensively tested on Windows.
- Despite their relative maturity, the `py` and `pyjs` objects are still only v0.2.x and still need further unit/functional/integration/field testing!
- As of this writing, the `api` module, does not (like apparently all 3rd party python c-extensions) unload properly between patches and requires a restart of Max to work after you close the first patch which uses it. Unfortunately, this is a known bug in python which is being worked on and may be fixed in future versions (python 3.13 perhaps?).
- `Numpy`, the popular python numerical analysis package, falls in the above category. As of python 3.9.x, it thankfully doesn't crash but gives the following error:

```
[py __main__] import numpy: SystemError('Objects/structseq.c:401: bad argument to internal func
```

This just means that the user opened a patch with a `py-js` external that imports `numpy`, then closed the patch and (in the same Max session) re-opened it, or created a new patch importing `numpy` again.

To fix it, just restart Max and use it normally in your patch. Treat each patch as a session and restart Max after each session. It's a pain, but unfortunately a limitation of current python c-extensions.

- `core` features relying on pure python code are supposed to be the most stable, and *should* not crash under most circumstances, `extra` features are less stable since they are more experimental, etc..
- The `api` module is the most experimental and evolving part of this project, and is completely optional. If you don't want to use it, don't import it or don't use an external which provides it.

Current Status of Builders

As mentioned earlier, as of this writing this project uses a combination of a `Makefile` in the project root, a basic `cmake` build option and a custom python build system, `builder`, which resides in the `py-js/source/py/builder` package. The `Makefile` is a kind of 'frontend' to the more complex python build system. The latter can be used directly of course. A view into its many options can be obtained by typing the following:

```
cd py-js/source/py
python3 -m builder --help
```

`builder` was developed to handle the more complex case of downloading the source code of python (from python.org) and also its dependencies from their respective sites and then building custom python binaries with which to reliably compile python3 externals which are portable, relocatable, self-contained, small-in-size, and usable in Max Packages and Standalones.

Build Variations

One of the objectives of this project is to cater to a number of build variations. As of this writing, the following table gives an overview of the different builds and their differences:

There is generally tradeoff of size vs. portability:

build command	format	size_mb	deploy_as	pip	portable	numpy
make	framework	0.3	external	yes [1]	no	yes
make homebrew-ext	hybrid [3]	13.6	external	no	yes	yes
make homebrew-pkg	hybrid [3]	13.9	package	yes	yes	yes
make static-ext	static	9.0	external	no	yes	no [2]
make shared-ext	shared	15.7	external	no	yes	yes
make shared-pkg	shared	18.7	package	yes	no [4]	yes
make framework-ext	framework	16.8	external	no	yes	yes
make framework-pkg	framework	16.8	package	yes	yes	yes

[1] has automatic access to your system python's site-packages

[2] current static external implementation does not work with numpy due to symbol access issues.

[3] *hybrid* means that the source system was a `framework` and the destination system is `shared`.

[4] the shared-pkg variant does not build a compliant 'Framework-type' bundle and hence cannot be notarized.

- *pip*: the build allows or provides for pip installation
- *portable*: the externals can be deployed as portable packages or standalones
- *numpy*: numpy compatibility

Packages vs Self-contained Externals

The Max package format is a great way to move a bunch of related patches and externals around. This format also makes a lot of sense for `py-js`, giving a number of advantages over other alternatives:

1. Portable: Relocatable, you can move it around and it still works.
2. Extendable: Can include a full fit-for-purpse python3 installation in the `support` directory with its own site-packages. Packages can be `pip` installed and all of the `site-packages` is automatically made available to the thin 'client' python3 externals in the package's `externals` folder.

3. Size-efficient, since you don't need to duplicate functionality in each external
4. Standalone installable: Recent changes in Max have allowed for this to work in standalones. Just create your standalone application from a patcher which which includes the `py` and `pyjs` objects. Once it is built into a `<STANDALONE>` then copy the whole aforementioned `py` package to `<STANDALONE>/Contents/Resources/C74/packages` and delete the redundant `py.mxo` in `<STANDALONE>/Contents/Resources/C74/externals` since it already exists in the just-copied package.
5. Better for codesigning / notarizing scenarios since Packages are not sealed bundles like externals.

On the other hand, sometimes you just want an external which embeds a python distribution and custom extensions and code:

1. Portable: Relocatable, you can move it around and it still works.
2. Extendable: Can include new pure python code and be provided with new additionas to `sys.path`
3. Size-efficient and fit-for-purpose
4. Standalone installable. Easiest to install in standalones
5. Can be codesigned and notarized relatively easily. [1]

[1] If you want to codesign and notarize it for use in your standalone or package, the codesigning / notarization script and related entitlements file can be found in the source/py/scripts folder.

The relocatable-python variation

relocatable-python is Greg Neagle's excellent tool for building standalone relocatable Python.framework bundles.

It works so well, that its been included in the `builder` application as an external (embedded dependency).

It can be seen in the `relocatable-pkg` make option which will download a nice default `Python.framework` to the `support` directory used for compiled both `py` and `pyjs` externals:

```
make relocatable-pkg
```

More options are available if you use the `builder` package directly:

```
$ python3 -m builder pyjs relocatable_pkg --help
usage: __main__.py pyjs relocatable_pkg [-h] [--destination DESTINATION]
                                        [--baseurl BASEURL]
                                        [--os-version OS_VERSION]
                                        [--python-version PYTHON_VERSION]
                                        [--pip-requirements PIP_REQUIREMENTS]
                                        [--pip-modules PIP_MODULES]
                                        [--no-unsign] [--upgrade-pip]
                                        [--without-pip] [--release] [-b] [-i]
                                        [--dump]

optional arguments:
  -h, --help            show this help message and exit
  --destination DESTINATION
                        Directory destination for the Python.framework
  --baseurl BASEURL     Override the base URL used to download the framework.
  --os-version OS_VERSION
                        Override the macOS version of the downloaded pkg.
                        Current supported versions are "10.6", "10.9", and
                        "11". Not all Python version and macOS version
                        combinations are valid.
  --python-version PYTHON_VERSION
                        Override the version of the Python framework to be
                        downloaded. See available versions at
                        https://www.python.org/downloads/mac-osx/
  --pip-requirements PIP_REQUIREMENTS
                        Path to a pip freeze requirements.txt file that
                        describes extra Python modules to be installed. If not
                        provided, no modules will be installed.
  --pip-modules PIP_MODULES
                        list of extra Python modules to be installed.
  --no-unsign           Do not unsign binaries and libraries after they are
                        relocatablized.
  --upgrade-pip         Upgrade pip prior to installing extra python modules.
  --without-pip         Do not install pip.
  --release             set configuration to release
  -b, --build           build python
  -i, --install         install python to build/lib
  --dump                dump project and product vars
```

Sidenote about building on a Mac

If you are developing the package in `$HOME/Documents/Max 8/Packages/py` and you have your iCloud drive on for Documents, you will find that `make` or `xcodebuild` will reliably fail with 1 error during development, a codesigning error that is due to icloud sync creating detritus in the dev folder. This can be mostly ignored (unless your only focus is codesigning the external).

The solution is to move the external project folder to folder that's not synced-with-icloud (such as `$HOME/Downloads` for example) and then run `xattr -cr .` in the project directory to remove the detritus (which ironically Apple's system is itself creating) and then it should succeed (provided you have your `Info.plist` and `bundle id` correctly specified). Then just symlink the folder to `$HOME/Documents/Max 8/Packages/` to prevent this from recurring.

I've tried this several times and and it works (for "sign to run locally" case and for the "Development" case).

Code Style

The coding style for this project can be applied automatically during the build process with `clang-format`. On OS X, you can easily install this using brew:

```
brew install clang-format
```

The style used in this project is specified in the `.clang-format` file.

3.4 Caveats

- Packaging and deployment of python3 externals has improved considerably but is still a work-in-progress: basically needing further documentation, consolidation and cleanup. For example, there are currently two build systems which overlap: a python3 based build system to handle complex packaging cases and cmake for handling the quick and efficient development builds and general cases.
- As of this writing, the `api` module, does not (like apparently all 3rd party python c-extensions) unload properly between patches and requires a restart of Max to work after you close the first patch which uses it. Unfortunately, this is a known bug in python which is being worked on and may be fixed in future versions (python 3.12 perhaps?).
- `Numpy`, the popular python numerical analysis package, falls in the above category. In newer versions of Python the situation is improving as above, but in python 3.9.x, it thankfully doesn't crash but gives the following error:

```
[py __main__] import numpy: SystemError('Objects/structseq.c:401: bad argument to internal func
```

This just means that the user opened a patch with a `py-js` external that imports `numpy`, then closed the patch and (in the same Max session) re-opened it, or created a new patch importing `numpy` again.

To fix it, just restart Max and use it normally in your patch. Treat each patch as a session and restart Max after each session.

- `core` features relying on pure python code are supposed to be the most stable, and *should* not crash under most circumstances, `extra` features are less stable since they are more experimental, etc..
- The `api` module is the most experimental, powerful and evolving part of this project, and is completely optional. If you don't want to use it, don't import it.

4 pyjs: a python3 jsextension for max

General purpose python3 jsextension, which means that it is a c-based Max external which can be accessed via the javascript `js` object or the `v8` object introduced in Max 9.

```
pyjs max external (jsextension)
    attributes
        name                    : unique object name
        file                    : file to load in object namespace
        pythonpath              : add path to python sys.path
        debug                   : switch debug logging on/off

    methods
        core (messages)
            import <module>     : python import to object namespace
            eval <expression>   : python 'eval' semantics
            exec <stmnt>        : python 'exec' semantics
            execfile <path>     : python 'execfile' semantics

        extra
            code <expr|stmt>    : eval/exec/import python code (see above)


        in-code (non-message)
            eval_to_json <expr> : python 'eval' returns json
```

Note that the source files in this projects are soft-linked from the `py` project. The reason for this is that the `py` and `pyjs` were originally developed together and there is still extensive non-cmake driven build infrastructure which assumes this.

Creating a separate folder for pyjs with its own CMakeLists.txt file means that the pyjs external will be built when `make projects` is called.

Ultimately all externals should have their own folders and documentation, but it will take some iterations to get there.

4.1 Overview

The pyjs max external/jsextension provides a PyJS class and a minimal subset of the py external's features which work well with the Max js object or the v8 object and javascript code (like returning json directly from evaluations of python expressions).

The external has access to builtin python modules and the whole universe of 3rd party modules.

General purpose python3 jsextension, which means that it's a c-based Max external which can only be accessed via the javascript js or v8 interface.

```
pyjs max external (jsextension)
    attributes
        name                    : unique object name
        file                    : file to load in object namespace
        pythonpath              : add path to python sys.path
        debug                   : switch debug logging on/off

    methods
        core (messages)
            import <module>     : python import to object namespace
            eval <expression>   : python 'eval' semantics
            exec <stmnt>        : python 'exec' semantics
            execfile <path>     : python 'execfile' semantics

        extra
            code <expr|stmt>    : eval/exec/import python code (see above)


        in-code (non-message)
            eval_to_json <expr> : python 'eval' returns json
```

Key Features

The pyjs external implements the following c-level methods:

category	method	param(s)	in/out	can change ns
core	import	module	in	yes
core	eval	expression	out	no
core	exec	statement	in	yes

category	method	param(s)	in/out	can change ns
core	execfile	file	in	yes
extra	code	expr or stmt	out?	yes
in-code	eval_to_json	expression	out	no

Note that the `code` method allows for import/exec/eval of python code, which can be said to make those 'fit-for-purpose' methods redundant. However, it has been retained because it provides additional strictness and provides a helpful prefix in messages which indicates message intent.

Core

py/js's *core* features have a one-to-one correspondance to python's *very high layer* as specified here. In the following, when we refer to *object*, we refer to instances of the `pyjs` external.

- **Per-object namespaces**. Each object has a unique name (which is provided automatically or can be set by the user), and responds to an `import <module>` message which loads the specified python module in its namespace (essentially a `globals` dictionary). Notably, namespaces can be different for each instance.
- **Eval Messages**. Responds to an `eval <expression>` message in the left inlet which is evaluated in the context of the namespace.`pyjs` objects just return an `atomarray` of the results.
- **Exec Messages**. Responds to an `exec <statement>` message and an `execfile <filepath>` message which executes the statement or the file's code in the object's namespace. For `pyjs` objects no output is given.

Extra

The *extra* category of methods makes the `pyjs` object play nice with the max/msp ecosystem:

- **Evaluate to JSON**. Can be used in javascript code only to automatically serialize the results of a python expression as a json string as follows: `evaluate_to_json <expression> -> JSON`.

Editing Support

For `pyjs` objects, code editing is already provided by the js Max object.

Deployment Scenarios

There are 3 general deployment variations:

1. **Linked to system python**. Linking the externals to your system python (homebrew, built from source, etc.) This has the benefit of re-using your existing python modules and is the default option.
2. **Embedded in package**. Embedding the python interpreter in a Max package: in this variation, a dedicated python distribution (zipped or otherwise) is placed in the `support` folder of the `py/js` package (or any other package) and is linked to the `pyjs` extension (or both). This makes it size efficient and usable in standalones.
3. **Embedded in external**. The external itself as a container for the python interpreter: a custom python distribution (zipped or otherwise) is stored inside the external bundle itself, which can make it portable and usable in standalones.

As of this writing all three deployment scenarios are availabe, however it is worth looking more closely into the tradeoffs in each case, and a number of build variations exist.

Deployment Scenario	`pyjs`
Linked to sys python	yes
Embeddded in package	yes
Embeddded in external	yes

Please review the py documentation for more information about building in the case of (2) and (3). Since the `py` and `pyjs` are built together simultaneously, the information provided there is quite relevent for `pyjs`.

4.2 Caveats

- Packaging and deployment of python3 externals has improved considerably but is still a work-in-progress: basically needing further documentation, consolidation and cleanup. For example, there are currently two build systems which overlap: a python3 based build system to handle complex packaging cases and cmake for handling the quick and efficient development builds and general cases.
- `core` features relying on pure python code are supposed to be the most stable, and *should* not crash under most circumstances, `extra` features are less stable since they are more experimental, etc..

Part III

Experimental Externals

5 krait: a single-header c++ python3 library for max externals

Provides a single-header cpp-centric `py_interpreter.h` library with a python3 interpreter class.

Note that `krait.cpp` is just a demonstration of an external using `py_interpreter.h`.

I called it `krait`, in honour of the 'Krait Lightspeeder' in the original Elite.

5.1 Building

From the root of the `py-js` project, there are several options to build the external:

- For a fast non-relocatable build which references your existing python installation

```
make krait
```

- For a relocatable dynamically-linked build

```
make krait-shared
```

- For a relocatable statically-linked build

```
make krait-static
```

- For a relocatable dynamically-linked framework build

```
make krait-framework
```

- For a dynamically-linked framework build for relocatable Max packages

```
make krait-framework-pkg
```

Finally, open the `help/krait.maxhelp` help file to test the external. This will build all subprojects, including `krait`, using the standard cmake build process.

5.2 Tests

See `krait.maxhelp` in `py-js/help`

6 mamba: a single-header c-based python3 library for max externals

This project is a single-header python3 library for Max externals. It is the result of an attempt to modularize the python interpreter for Max and make it re-usable so that it can be easily nested inside another external.

The idea is that by including a single header file, `py.h` any max external can provide general or specialized python 'services'.

The name of this header is likely to change to differentiate it from the `py` object and its header. Other names could be `mpy.h` or `mamba.h`

6.1 Build System

Recent work on mamba's build system has made it now possible, using the `source/scripts/buildpy.py` script from the buildpy project and `cmake`, to build relocatable python3 externals along the lines of what the `builder` module provides for the `py` and `pyjs` externals.

The key benefit of this combination is that it provides a lightweight way to embed Python in Max externals without requiring a full Python installation. The single-header approach makes it easy to integrate into Max projects while the build system ensures the Python environment is properly configured and relocatable. This allows Max developers to leverage Python's extensive ecosystem of libraries and tools while maintaining a small footprint and avoiding dependency issues.

6.2 Usage

There are several options to build the external:

- For a fast non-relocatable build which references your existing python installation

```
make mamba
```

- For a relocatable dynamically-linked build

```
make mamba-shared
```

- For a relocatable statically-linked build

```
make mamba-static
```

- For a relocatable dynamically-linked framework build

```
make mamba-framework
```

- For a dynamically-linked framework build for relocatable Max packages

```
make mamba-framework-pkg
```

Finally, open the `help/mamba.maxhelp` help file to test the external.

6.3 Comparison of Build Variants

Variant	Build Command	Size (MB)	Notes
Local	`make mamba`	0.19	non-relocatable
Shared	`make mamba-shared`	8.6	
Static	`make mamba-static`	11.2	
Framework	`make mamba-framework`	11.9	includes executable
Package	`make mamba-framework-pkg`	11.9	includes executable

Note that in the the package variant above, the external is relocatable as long as it is part of the containing package, as it refers to a `Python.framework` in the `support` folder. The other variants, with the exception of the `local` build, are self-contained externals.

7 cobra: an ITM-based python evaluator

This project provide a proof-of-concept to defer the evaluation of a python function via Max's ITM-based sequencing.

Note that it has a dependency on another subproject: it includes mamba's single header c library, `py.h`, to reduce boilerplate and provide python interpreter 'services'.

7.1 Current Status

Crashes on Python3.13 (this is under investigation)

7.2 Building

From the root of the `py-js` project

```
make projects
```

This will build all subprojects, including `cobra`, using the standard cmake buildsystem.

7.3 Help

See `cobra.maxhelp` in `py-js/help` folder.

8 mxpy: pdpython for max

This is an ongoing attempt to translate pdpython to maxmsp from https://github.com/garthz/pdpython and my fork https://github.com/shakfu/pdpython

```
// pdpython.c : Pd external to bridge data in and out of Python
// Copyright (c) 2014, Garth Zeglin.  All rights reserved.  Provided under the
// terms of the BSD 3-clause license.
```

8.1 TODO

- ☐ make ints, floats, and basic symbols work
- ☐ make `mxpy_eval` to handle bang (see `py_send`), ints, floats, list, etc..
- ☐ Fix the restriction that typed methods will fail, unless we `A_CANT` or `A_GIMMEBACK`?

See a simple `A_GIMMEBACK` example `simplejs.c` example in the max-sdk

see:

- https://cycling74.com/forums/a_gimmeback-routine-appears-in-quickref-menu
- https://cycling74.com/forums/obex-how-to-get-return-values-from-object_method

8.2 Flow

1. `ext_main`: sets up one 'anything' method:

   ```
   class_addmethod(c, (method)mxpy_eval, "anything", A_GIMME, 0)
   ```

2. `mxpy_new`: where object `[mxpy module Class arg1 arg2 arg3 .. argN]`

   ```
   PyObject* args = t_atom_list_to_PyObject_list(argc - 2, argv + 2)
   x->py_object = PyObject_CallObject(func, args)
   ```

   note: if `Class` is actually a function which returns a non-callable value then `x->py_object` contains could be return with a bang

3. `mxpy_eval`: responds to max message and dispatches to the python-Class
4. `emit_outlet_message`: output returned values to outlet, tuples are output in sequence

Part IV

Alternative Externals

9 mpy: micropython external

A proof-of-concept which embeds micropython in a max external

Currently only tested on macOS.

Can be built using `-DBUILD_MICROPYTHON_EXTERNAL` which will default to local micropython code. If the option `-DFETCH_MICROPYTHON` is used cmake will try to build from git clone.

Doesn't do anything now useful except prove that embedding micropytho in an external is possible.. Still a work-in-progress and will be developed further as one familiarizes oneself with the micropython c-api.

10 pktpy: a pocketpy max external

This external embeds pocketpy, a nifty C++17 header-only Python interpreter for game engines, in a Max external.

10.1 Notes

pocketpy is a very cool project to create an embeddable self-contained python implementation for game engines.

As of version 1.0.4, quite a lot of language compatibility has been implemented including custom modules and the following builtin modules:

- bisect
- c (custom module for c-level access)
- collections
- datatime
- easing (a set of easing functions)
- gc
- heapq
- json
- linalg (linear algebra)
- math
- os
- pickle
- random
- re
- requests
- sys
- time
- traceback

This max external project embeds the pocketpy interpreter, provides for easy wrapping of max-api functions in c++, and produces a small sized external (~ 1.0Mb) without dependencies making it completely self-contained, portable and ideal for standalones and packages.

10.2 Structure of Implementation

```
pktpy.cpp -includes-> pktpy.h -includes-> pocketpy.h
```

- `pocketpy.h`: the pocketpy header.
- `pktp.h`: a general middle layer providing a cpp class, `PktpythonInterpreter`, a subclass of `pkpy::VM`, with helpers and round-trip translation methods between pocketpy and the max c-api. The user should ideally not need to change anything in this file.
- `pktpy.cpp`: In this file, the max-api methods are implemented by using the functionality in the middle layer. This is were customization should ocurr (e.g custom bultin methods).
- `user_config.h`: A configuration header to allow for tweaking of the pocketpy VM. Currently the only adjustment has been to set `PK_ENABLE_THREAD 1` which adds additionals locks for multi-threaded applications.

10.3 Current Status

- `exec`, `eval`, `anything`, `execfile`, methods to enable the execution, evaluation and importation of pocketpy python code with support for basic types (int, float, strings) and lists.
- no separate method for `import`, this is provided as part of `anything` and it works as expected.
- `List` to atom conversion support
- examples of wrapped functions and builtins (local, max api, etc.).
- see `pktpy.maxhelp` for a demo

11 pktpy2: a pocketpy v2.0.x max external

This external embeds version `2.0.5` of pocketpy, a Python interpreter for game engines, in a Max external. It is called `pktpy2` to differentiate it from the `pktpy` external which is based on the version `1.4.6` of `pocketpy`.

The major difference / benefit between v2 and v1 is that the lua-like c-api used in the v2 implementation makes the external a good deal smaller (572KB for v2 vs 1.3MB for v1 so far).

Part V

Networking Externals

12 zedit: a web-based code-editor embedded in an max external

This subproject provides an example of a python3 external whith the following features:

- Embeds a python interpreter via the mamba single header python3 c library
- Embeds the c-based mongoose webserver
- Provides a web-based code-editor
- Provides a web-based interactive terminal

This ui in this project is very powerfull using typescript/javascript-based web technologies. This is probably overkill for a python3 external, but illustratrive nonetheless of the potential achieved by embedding an small webserver in an external.

For simpler exmaples of user interfaces which interact with python3 externals see the `patchers/bpatchers_ui` and the `patchers/bpatcher_py` folders.

12.1 Usage

You have to setup the project before being able to use it:

```
cd py-js/source/projects/zedit/web
make
```

then return to the root of the project

```
cd py-js
make dev
```

It is possible to build the external and embed everything into a bundle by building the project in Release mode.

This will build and deploy the javascript / typescript code to the `zedit/web/public` folder. There is folder called `zedit/webroot` which is a symlink to the above public folder.

Note that the `n4m` folder contains an alternative impplementation which can be ignored for the time being as it is a work-in-pgress

12.2 Current Status

The current implementation uses codemirror, xtermjs and the mongoose embedded webserver library to interact with the user and the max external.

Current features are:

- Embedded webserver running in a separate thread
- Python3 web-editor with dark theme, syntax highlighting, auto-complete, and search, ..
- Basic web terminal
- Commands
  - Help: open a list of keboard commands
  - Open: open a file from the client side
  - Save: dump contents to external
  - Run: dump and run contents

See `zedit.maxhelp` for a demo of the external launching the embedded webserver and running the code-mirror web-editor.

There is also a node-for-max variation on (1) using expressjs as the webserver.

12.3 Future Direction

- use jquery.terminal

13 zpy: python3 via zmq in max

A max external which uses `czmq` to connect to a separate python process

The objective is to replicate what the `py` external does but using czmq.

13.1 Requires

```
brew install czmq
```

13.2 Status

- ☐ proof-of-concept

13.3 TODO

- how to launch python server automatically and close it with the patch

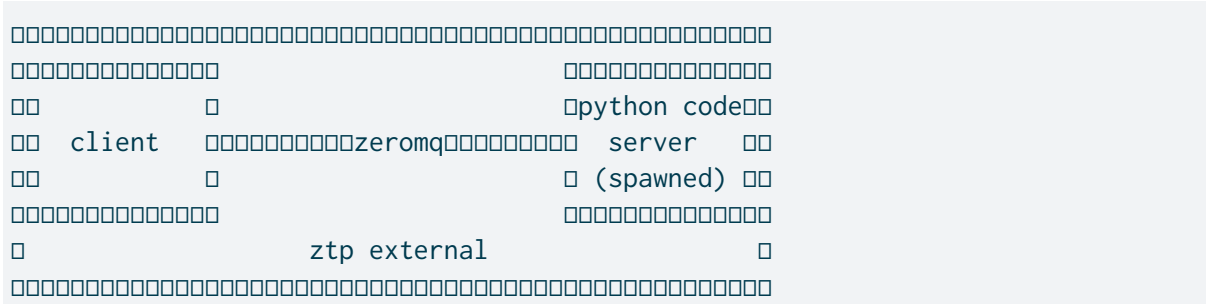
13.4 Alternatives

- ☐ run as subprocess and read and write from stdin and stdout via a pipe

13.5 Research

- https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html
- https://github.com/sheredom/subprocess.h

14 ztp: the zeromq + threads + python Max external



The ztp external (the name refers to python, threads and zeromq) uses zmq with threading for non-blocking communication with a spawned python code interpretation server which evaluates and executes python code and sends back the result.

The combination of threads, zeromq and remote process mgmt of the spawned server make this much more usable than zpy, an earlier effort which lacked threads and which suffered from blocking during communication with the server.

14.1 Requires

```
brew install zmq
```

14.2 Learnings

- This works: one thread per zmq socket as per the zmq rules.
-

15 jmx: jupyter client/kernel for Max/MSP

This is a proof-of-concept subproject to embed a jupyter client or jupyter kernel in a Max/MSP external.

15.1 Requires

```
brew install zmq
```

15.2 Implementation Notes

There's an insidious linker error that emerges at attempts to use `jsonwriter` and the below post sugests having to link the `commonsys.c` in the sdk.

see: https://cycling74.com/forums/linker-error

Also according to the above forum post, one must include `common_symbols_init();` in the `ext_main()` method.

15.3 Jupyter Message Protocol

see: https://jupyter-client.readthedocs.io/en/latest/messaging.html

Example

```
{
    "header": {
        "msg_id": str,  # typically UUID, must be unique per message
        "session": str,  # typically UUID, should be unique per session
        "username": str,
```

```
        # ISO 8601 timestamp for when the message is created
        "date": str,
        # All recognized message type strings are listed below.
        "msg_type": str,
        # the message protocol version
        "version": "5.0",
        # Optional subshell_id
        "subshell_id": str | None,
    },
    "parent_header": {
        # parent_header is a copy of the request's header
        # When a message is the "result" of another message, such as a
        # side-effect (output or status) or direct reply, the parent_header is a
        # copy of the header of the message that "caused" the current message.
        # _reply messages MUST have a parent_header, and side-effects
        # typically have a parent. If there is no parent, an empty dict should be used.
        # This parent is used by clients to route message handling
        # to the right place, such as outputs to a cell.
    },
    "metadata": {
        # The metadata dict contains information about the message that is not part
        # of the content. This is not often used, but can be an extra location to
        # store information about requests and replies, such as extensions adding
        # information about request or execution context.
    },
    "content": {
        # The content dict is the body of the message. Its structure is dictated
        # by the msg_type field in the header.

        # Source code to be executed by the kernel, one or more lines.
        'code' : str,

        # A boolean flag which, if True, signals the kernel to execute
        # this code as quietly as possible.
        # silent=True forces store_history to be False,
        # and will *not*:
        #   - broadcast output on the IOPUB channel
        #   - have an execute_result
        # The default is False.
        'silent' : bool,

        # A boolean flag which, if True, signals the kernel to populate history
```

```
        # The default is True if silent is False.  If silent is True, store_history
        # is forced to be False.
        'store_history' : bool,

        # A dict mapping names to expressions to be evaluated in the
        # user's dict. The rich display-data representation of each will be evaluated after exe
        # See the display_data content for the structure of the representation data.
        'user_expressions' : dict,

        # Some frontends do not support stdin requests.
        # If this is true, code running in the kernel can prompt the user for input
        # with an input_request message (see below). If it is false, the kernel
        # should not send these messages.
        'allow_stdin' : True,

        # A boolean flag, which, if True, aborts the execution queue if an exception is encount
        # If False, queued execute_requests will execute even if this request generates an exce
        'stop_on_error' : True,

    },
    "buffers": [
        # Finally, a list of additional binary buffers can be associated with a
        # message. While this is part of the protocol, no official messages make
        # use of these buffers. They are used by extension messages, such as
        # IPython Parallel's apply and some of ipywidgets' comm messages.
    ],
}
```

Details

```
The session ID in a message header identifies a unique entity with state, such as a kernel proc

A client session ID, in message headers from a client, should be unique among all clients conne

A kernel session ID, in message headers from a kernel, should identify a particular kernel proc

The session ID in a message header can be used to identify the sending entity. For example, if

The subshell_id is only used in shell messages of kernels that support subshells (Kernel subshe

Parent header. When a message is the "result" of another message, such as a side-effect (outpu
```

```
Metadata. The metadata dict contains information about the message that is not part of the cont

Content. The content dict is the body of the message. Its structure is dictated by the msg_type

Buffers. Finally, a list of additional binary buffers can be associated with a message. While t

Changed in version 5.0: version key added to the header.

Changed in version 5.1: date in the header was accidentally omitted from the spec prior to 5.1

Changed in version 5.5: subshell_id added to the header.
```

15.4 Message Types for the Content Dict

Request-Reply

msg types: `execute_reply` sent from the kernel given a `execute_request` msg

```
{
   'status' : 'error',
   'ename' : str,   # Exception name, as a string
   'evalue' : str,  # Exception value, as a string
   'traceback' : list(str), # traceback frames as strings
}
```

Execute

msg type: `execute_request`

```
content = {
    # Source code to be executed by the kernel, one or more lines.
    'code' : str,

    # A boolean flag which, if True, signals the kernel to execute
    # this code as quietly as possible.
    # silent=True forces store_history to be False,
    # and will *not*:
    #   - broadcast output on the IOPUB channel
    #   - have an execute_result
```

```
    # The default is False.
    'silent' : bool,

    # A boolean flag which, if True, signals the kernel to populate history
    # The default is True if silent is False.  If silent is True, store_history
    # is forced to be False.
    'store_history' : bool,

    # A dict mapping names to expressions to be evaluated in the
    # user's dict. The rich display-data representation of each will be evaluated after executi
    # See the display_data content for the structure of the representation data.
    'user_expressions' : dict,

    # Some frontends do not support stdin requests.
    # If this is true, code running in the kernel can prompt the user for input
    # with an input_request message (see below). If it is false, the kernel
    # should not send these messages.
    'allow_stdin' : True,

    # A boolean flag, which, if True, aborts the execution queue if an exception is encountered
    # If False, queued execute_requests will execute even if this request generates an exceptio
    'stop_on_error' : True,
}
```

A Guide to Developers

A.1 Code Style

The coding style for this project can be applied automatically during the build process with `clang-format`. On OS X, you can easily install this using brew:

```
brew install clang-format
```

The style used in this project is specified in the `.clang-format` file.