Python3 Externals for Max/MSP

A PyJs Project Guide

S. Alireza

2023-01-07

Exploring different ways one can use Python3 in Max/MSP

Table of contents

Preface

This is a Quarto book.

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

1 python3 objects for max

Simple (and extensible) python3 externals for MaxMSP.

Currently builds 'natively' on macOS `x86_64`and `arm64`.

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

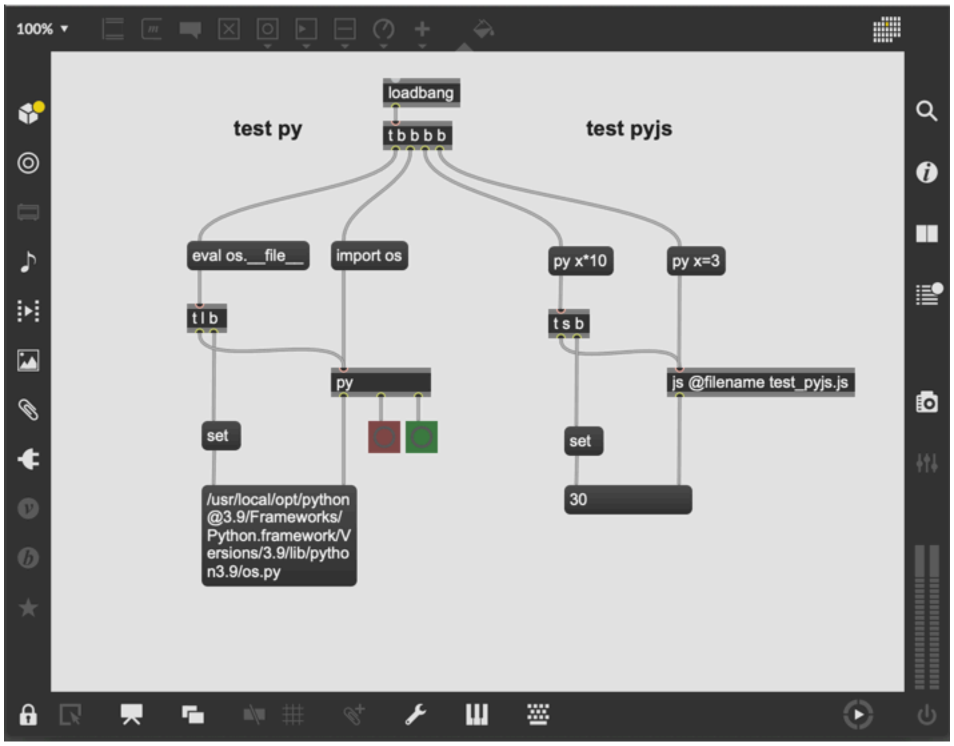

100%
test py
test pyjs
loadbang
t b b b b
eval os.__file__
import os
py x*10
py x=3
t l b
t s b
py
js @filename test_pyjs.js
set
set
/usr/local/opt/python@3.9/Frameworks/Python.framework/Versions/3.9/lib/python3.9/os.py
30

Figure 1.1: py-js test

1.1 Preface

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

Along the way, a number of python3 or python-related externals have been developed for use in a live Max environment:

name	sdk	lang	description
py	max-sdk	c	well-featured, many packaging options + cython api
pyjs	max-sdk	c	js-friendly – written as a Max javascript-extension
mxpy	max-sdk	c	a translation of pdpython into Max
pymx [1]	min-devkit	c++	concise, modern, using pybind11
zpy	max-sdk	c	uses zeromq for 2way-comms with an external python process
cobra	max-sdk	c	python3 external providing deferred and clocked function execution
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
pktpy	max-sdk	c++	uses the pocketpy single-header c++ library
zedit	max-sdk	c	a web-based python editor using codemirror and the mongoose embedded webserver.
mpy	max-sdk	c	a proof-of-concept embedding micropython

[1] pymx has been moved to its own github project because it uses the min-devkit sdk.

The common objective in these externals is to help you use and distribute your python code and libraries in your Max applications. Many can be considered experimental, with 80% of development time going to the first two externals (`py` and `pyjs`). Please see below for an overview and feature comparison.

At the time of this writing, and since the switch to the new max-sdk-base, the project **is compatible with Apple Silicon-based machines** but intentionally only produces 'native' (`x86_64` or `arm64`) externals with no 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.

This README will mostly cover the first two mature externals (`py.mxo` and `pyjs.mxo`) and 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 any of the other subprojects, please look into the respective folder in the `py-js/source/projects` section.

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. 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.

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

2 Quickstart

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 3.11 are tested and known to work.

2. Make sure you also have Xcode installed.

3. Git clone the `py-js` repo 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`, which is used for wrapping the max api and worth installing in case you want to play around or extend the max api wrapper (which is written in cython) for the `py` or `pyjs` externals.

5. To build only the `py` or `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 quite 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 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.

2.1 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 `make`. 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.

2.2 Building self-contained Python3 Externals for Packages and Standalones

Currently only the py and pyjs externals are built using these methods, which rely on a python builder manager. 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] These 'tiny' variants are intended to be the smallest possible portable pyjs externals. 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 theoreticla 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, is the pktpy subproject in this repo.

[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.

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
```

2.3 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 variations work on your particular setup.

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

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

2.4 Using Self-contained Python Externals in a 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.

2.5 py.mxo

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`.

2.6 pyjs.mxo

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 `py_test_standalone_only_pyjs.maxpat` patcher whici is included in `py-js/patchers`.

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/Reso` 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!

Part I

The Externals

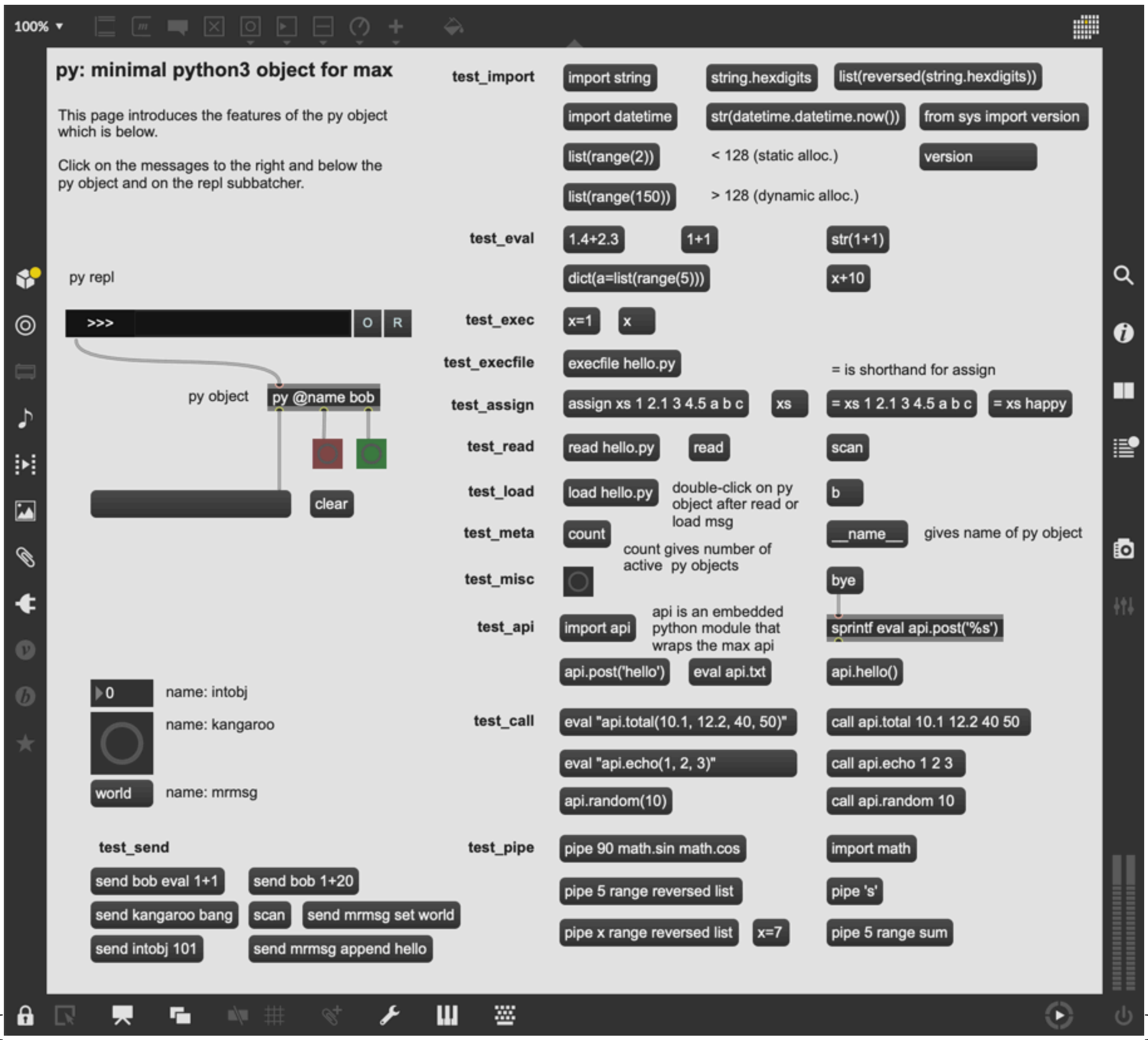
3 py: a general purpose python3 max external

A general purpose Max external which 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 builtin `api` module which is derived from a cython-wrapper of a subset of the Max c-api.

The following 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
```

[]

py: minimal python3 object for max

This page introduces the features of the py object which is below.

Click on the messages to the right and below the py object and on the repl subbatcher.

py repl

py object — py @name bob

name: intobj
name: kangaroo
name: mrmsg

test_send
```
send bob eval 1+1    send bob 1+20
send kangaroo bang   scan   send mrmsg set world
send intobj 101      send mrmsg append hello
```

test_import
```
import string    string.hexdigits    list(reversed(string.hexdigits))
import datetime  str(datetime.datetime.now())   from sys import version
list(range(2))   < 128 (static alloc.)   version
list(range(150)) > 128 (dynamic alloc.)
```

test_eval
```
1.4+2.3    1+1    str(1+1)
dict(a=list(range(5)))    x+10
```

test_exec
```
x=1    x
```

test_execfile
```
execfile hello.py
```

= is shorthand for assign

test_assign
```
assign xs 1 2.1 3 4.5 a b c    xs    = xs 1 2.1 3 4.5 a b c    = xs happy
```

test_read
```
read hello.py    read    scan
```

test_load
```
load hello.py    b
```
double-click on py object after read or load msg

test_meta
```
count    __name__
```
count gives number of active py objects

gives name of py object

test_misc
```
bye
```

test_api
```
import api    sprintf eval api.post('%s')
api.post('hello')    eval api.txt    api.hello()
```
api is an embedded python module that wraps the max api

test_call
```
eval "api.total(10.1, 12.2, 40, 50)"    call api.total 10.1 12.2 40 50
eval "api.echo(1, 2, 3)"                call api.echo 1 2 3
api.random(10)                          call api.random 10
```

test_pipe
```
pipe 90 math.sin math.cos      import math
pipe 5 range reversed list     pipe 's'
pipe x range reversed list   x=7   pipe 5 range sum
```

3.1 Overview

This overview will cover the following two external implementations:

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.

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 a number of build variations exist.

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

3.1.1 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

category	method	param(s)	in/out	can change ns
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.

3.1.1.1 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.

3.1.1.2 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. (EXPERIMENTAL and evolving).

3.1.1.3 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.

3.1.1.4 Editing Support

- **Line REPL**. The `py`has two bpatcher line `repls`, one of which embeds a `py` object and another 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. Of course, a coll can made to save all commands if required.
- **Experimental Remote Console**. A new 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`.

- **Code Editor**. Double-clicking 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 an attribute. A `load` message also `reads` the file followed by `execfile`. Saving the text in the editor uses the attribute filepath and execs the saved text to the object's namespace.

3.1.1.5 Scripting

- **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.

3.2 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.
- 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 f
```

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 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 only be accessed via the javascript `js` 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
```

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 and javascript code (like returning json directly from evaluations of python expressions).

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

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

4.1.1 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
core	execfile	file	in	yes
extra	code	expr or stmt	out?	yes

category	method	param(s)	in/out	can change ns
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.

4.1.1.1 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.

4.1.1.2 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`.

4.1.1.3 Editing Support

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

4.1.1.4 Scripting

- **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 `pyjs`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.

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.
- 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 f
```

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 and evolving part of this project, and is completely optional. If you don't want to use it, don't import it.

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

```
make projects
```

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 any max external can provide general or specialized python 'services'.

The project is implemented in the header file `py.h`.

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 Building

From the root of the `py-js` project

```
make projects
```

This will build all subprojects, including `mamba`, using the standard cmake build process.

6.2 Tests

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

7 cobra: an ITM-based python evaluator

This project is an experimental attempt to defer the evaluation of a python function via Max's ITM-based sequencing.

7.1 Building

From the root of the `py-js` project

```
make projects
```

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

7.2 Help

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

8 mpy: micropython external

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

Currently only tested on macOS.

Doesn't do anything now.. Still a work-in-progress

9 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.
```

9.1 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

9.2 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

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

This subproject aim to provide a web-based code-editor for the python3 externals in this project.

Note there is also a very nice alternative, light-weight and practical solution to this requirement: the`py_external_editor.maxpat` abstraction in this project's `patchers` folder, which is also implemented as a bpatcher.

11.1 Current Status

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

11.2 Future Direction

- use jquery.terminal

Part II

Builder

12 builder: pyjs python external builder

12.1 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 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.

12.2 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	isolated
make	framework	0.3	external	yes [1]	no	yes	yes
make brew-ext	hybrid [3]	13.6	external	no	yes	yes	no
make brew-pkg	hybrid [3]	13.9	package	yes	yes	yes	yes
make static-ext	static	9.0	external	no	yes	no [2]	yes
make shared-ext	shared	15.7	external	no	yes	yes	no
make shared-pkg	shared	18.7	package	yes	no [4]	yes	yes
make framework-ext	framework	16.8	external	no	yes	yes	no

build command	format	size_mb	deploy_as	pip	portable	numpy	isolated
make framework-pkg	framework	16.8	package	yes	yes	yes	yes

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

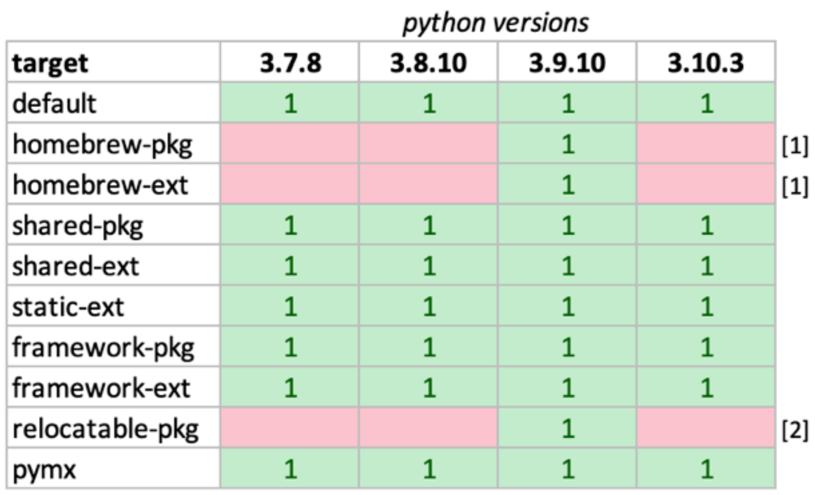
[2] current static external implementation does not work with numpy due to symbol access issues.

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

[4] the shared-pkg variant does 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
- *isolated*: if yes, then different external types can run concurrently without issue

12.2.1 Python Version Compatibility



	python versions				
target	**3.7.8**	**3.8.10**	**3.9.10**	**3.10.3**	
default	1	1	1	1	
homebrew-pkg			1		[1]
homebrew-ext			1		[1]
shared-pkg	1	1	1	1	
shared-ext	1	1	1	1	
static-ext	1	1	1	1	
framework-pkg	1	1	1	1	
framework-ext	1	1	1	1	
relocatable-pkg			1		[2]
pymx	1	1	1	1	

Figure 12.1: py-js testing

[1] Homebrew only tested on current relase (3.9.10), other versions are expected to work without issues.

[2] Relocatable python can select its own version of python (Only tested with python 3.9.10, other versions should work without issues)

12.2.2 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.

12.3 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
```

12.4 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).

12.5 Packaging

This project has a builtin features to package, sign, notarize and deploy python3 externals for Max/MSP.

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

12.5.1 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
```

12.5.2 The Project's Makefile frontend:

Since the Makefile frontend basically just calles on `builder` interface in a simplified way, we will use it to explain the basic steps which occur sequentially. Note that while it is possible to automate thie process considerable, it is separated here into discrete steps for purposes of illustration and to facilitate debugging:

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
```

Note that the it is important to sign externals (this is done by Xcode automatically) and to distribute to others, you can either ask users to remove the products quarantine state or notarize the product as above, which requires an Apple Developer License.

12.5.3 Github Actions

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

The only caveat is that currently Github only provide `x86_64` runners so one has to build for `arm64` on a dedicated machine.

A FAQ

A.1 Compatibility

A.1.1 Is this macOS only?

This project is currently only macOS x86_64 (Intel) or arm64 (Apple silicon) compatible.

A.1.2 What about compatibility with Windows?

There's no particular reason why this project shouldn't work in windows except that I don't develop in windows any longer. Feel free to send pull requests to help make this happen.

A.1.3 Does it only work with Homebrew python?

It used to work, by default, with Homebrew installed python but current versions don't require Homebrew python. 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

A.2 Implementation

A.2.1 Does it embed python into the external or is the external connecting to the local python installation?

The default build creates a lightweight external linked to your local python3; another variant embeds python3 into an external linked to python3 which resides in a Max package; and another embeds python into the external itself without an dependencies. There are other ways as well. The project README gives an overview of differences between approaches.

A.3 Installation

A.3.1 Can I install two different python3 externals the same project?

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`, `py.mxo` and `pyjs.mxo` will not work together in the same patch, but if you built a `framework-pkg` variations of the same two externals they should work fine without issues.

Of course, it would normally be considered redundant to install two different python3 externals in your project.

A.4 Logging

A.4.1 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.

A.5 Extensibility

A.5.1 How to extend python with your own scripts and 3rd party libraries?

The easiest solution is not to use a self-contained external and use an external that's linked to your system python3 installation. This is what gets built if you run ./build.sh 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/python3.9/site-packages`

2. The package script folder: `py-js/examples/scripts`
3. Whichever path you set the patcher PYTHONPATH property to (during object creation).

For (1), I have tested pure python scripts which should work without re-codesigning the externals, but if you add compiled extensions, then I think you have to re-codesign the external. Check out my maxutils project for help with that.

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 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 changed but this is something on the todo list.

A.6 Specific Python package Compatibility

A.6.1 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 `bin-homebrew-pkg` build option for example.

It is also possible to package numpy in a full relocatable external, it's quite involved, and cannot currently only be done with non-statically built relocatable externals. The releases section has an example of this.

References