pex Documentation

Release 2.1.113

Brian Wickman

Nov 03, 2022

CONTENTS

This project is the home of the .pex file, and the `pex` tool which can create them. `pex` also provides a general purpose Python environment-virtualization solution similar to virtualenv. pex is short for "Python Executable"

CHAPTER

ONE

IN BRIEF

To quickly get started building .pex files, go straight to *Building .pex files*. New to python packaging? Check out packaging.python.org.

CHAPTER

TWO

INTRO & HISTORY

pex contains the Python packaging and distribution libraries originally available through the twitter commons but since split out into a separate project. The most notable components of pex are the .pex (Python EXecutable) format and the associated `pex` tool which provide a general purpose Python environment virtualization solution similar in spirit to virtualenv. PEX files have been used by Twitter to deploy Python applications to production since 2011.

To learn more about what the .pex format is and why it could be useful for you, see *What are .pex files?* For the impatient, there is also a (slightly outdated) lightning talk published by Twitter University: WTF is PEX?. To go straight to building pex files, see *Building .pex files*.

Guide:

2.1 What are .pex files?

2.1.1 tl;dr

PEX files are self-contained executable Python virtual environments. More specifically, they are carefully constructed zip files with a `#!/usr/bin/env python` and special `__main__.py` that allows you to interact with the PEX runtime. For more information about zip applications, see PEP 441.

To get started building your first pex files, go straight to *Building .pex files*.

2.1.2 Why .pex files?

Files with the .pex extension – "PEX files" or ".pex files" – are self-contained executable Python virtual environments. PEX files make it easy to deploy Python applications: the deployment process becomes simply `scp`.

Single PEX files can support multiple platforms and python interpreters, making them an attractive option to distribute applications such as command line tools. For example, this feature allows the convenient use of the same PEX file on both OS X laptops and production Linux AMIs.

2.1.3 How do .pex files work?

PEX files rely on a feature in the Python importer that considers the presence of a `__main__.py` within the module as a signal to treat that module as an executable. For example, `python -m my_module` or `python my_module` will execute `my_module/__main__.py` if it exists.

Because of the flexibility of the Python import subsystem, `python -m my_module` works regardless if `my_module` is on disk or within a zip file. Adding `#!/usr/bin/env python` to the top of a .zip file containing a `__main__.py` and marking it executable will turn it into an executable Python program. pex takes advantage of this feature in order to build executable .pex files. This is described more thoroughly in PEP 441.

2.2 Building .pex files

You can build .pex files using the `pex` utility, which is made available when you `pip install pex`. Do this within a virtualenv, then you can use pex to bootstrap itself:

```
$ pex pex requests -c pex -o ~/bin/pex
```

This command creates a pex file containing pex and requests, using the console script named "pex", saving it in ~/bin/pex. At this point, assuming ~/bin is on your $PATH, then you can use pex in or outside of any virtualenv.

The second easiest way to build .pex files is using the `bdist_pex` setuptools command which is available if you `pip install pex`. For example, to clone and build pip from source:

```
$ git clone https://github.com/pypa/pip && cd pip
$ python setup.py bdist_pex
running bdist_pex
Writing pip to dist/pip-7.2.0.dev0.pex
```

Both are described in more detail below.

2.3 Invoking the `pex` utility

The `pex` utility has no required arguments and by default will construct an empty environment and invoke it. When no entry point is specified, "invocation" means starting an interpreter:

```
$ pex
Python 3.6.2 (default, Jul 20 2017, 03:52:27)
[GCC 7.1.1 20170630] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

This creates an ephemeral environment that only exists for the duration of the `pex` command invocation and is garbage collected immediately on exit.

You can tailor which interpreter is used by specifying `--python=PATH`. PATH can be either the absolute path of a Python binary or the name of a Python interpreter within the environment, e.g.:

```
$ pex
Python 3.6.2 (default, Jul 20 2017, 03:52:27)
[GCC 7.1.1 20170630] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

(continues on next page)

(continued from previous page)

```
(InteractiveConsole)
>>> print "This won't work!"
  File "<console>", line 1
    print "This won't work!"
                          ^
SyntaxError: Missing parentheses in call to 'print'
>>>
$ pex --python=python2.7
Python 2.7.13 (default, Jul 21 2017, 03:24:34)
[GCC 7.1.1 20170630] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> print "This works."
This works.
```

2.3.1 Specifying requirements

Requirements are specified using the same form as expected by `pip` and `setuptools`, e.g. `flask`, `setuptools==2.1.2`, `Django>=1.4,<1.6`. These are specified as arguments to pex and any number (including 0) may be specified. For example, to start an environment with `flask` and `psutil>1`:

```
$ pex flask 'psutil>1'
Python 3.6.2 (default, Jul 20 2017, 03:52:27)
[GCC 7.1.1 20170630] on linux
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

You can then import and manipulate modules like you would otherwise:

```
>>> import flask
>>> import psutil
>>> ...
```

Conveniently, the output of `pip freeze` (a list of pinned dependencies) can be passed directly to `pex`. This provides a handy way to freeze a virtualenv into a PEX file.

```
$ pex $(pip freeze) -o my_application.pex
```

A `requirements.txt` file may also be used, just as with `pip`.

```
$ pex -r requirements.txt -o my_application.pex
```

2.3.2 Specifying entry points

Entry points define how the environment is executed and may be specified in one of three ways.

pex <options> – script.py

As mentioned above, if no entry points are specified, the default behavior is to emulate an interpreter. First we create a simple flask application:

```
$ cat <<EOF > flask_hello_world.py
> from flask import Flask
> app = Flask(__name__)
>
> @app.route('/')
> def hello_world():
>   return 'hello world!'
>
> app.run()
> EOF
```

Then, like an interpreter, if a source file is specified as a parameter to pex, it is invoked:

```
$ pex flask -- ./flask_hello_world.py
* Running on http://127.0.0.1:5000/
```

pex -m

Your code may be within the PEX file or it may be some predetermined entry point within the standard library. `pex -m` behaves very similarly to `python -m`. Consider `python -m pydoc`:

```
$ python -m pydoc
pydoc - the Python documentation tool

pydoc.py <name> ...
    Show text documentation on something.  <name> may be the name of a
    Python keyword, topic, function, module, or package, or a dotted
    reference to a class or function within a module or module in a
    ...
```

This can be emulated using the `pex` tool using `-m pydoc`:

```
$ pex -m pydoc
pydoc - the Python documentation tool

tmpInGItD <name> ...
    Show text documentation on something.  <name> may be the name of a
    Python keyword, topic, function, module, or package, or a dotted
    reference to a class or function within a module or module in a
    ...
```

Arguments will be passed unescaped following `--` on the command line. So in order to get pydoc help on the `flask.app` package in Flask:

```
$ pex flask -m pydoc -- flask.app

Help on module flask.app in flask:

NAME
    flask.app

```

(continues on next page)

(continued from previous page)

```
FILE
    /private/var/folders/rd/_tjz8zts3g14md1kmf38z6w80000gn/T/tmp3PCy5a/.deps/Flask-0.
↪10.1-py2-none-any.whl/flask/app.py

DESCRIPTION
    flask.app
    ~~~~~~~~~
```

and so forth.

Entry points can also take the form `package:target`, such as `sphinx:main` or `fabric.main:main` for Sphinx and Fabric respectively. This is roughly equivalent to running a script that does `import sys, from package import target; sys.exit(target())`.

This can be a powerful way to invoke Python applications without ever having to `pip install` anything, for example a one-off invocation of Sphinx with the readthedocs theme available:

```
$ pex sphinx==1.2.2 sphinx_rtd_theme -e sphinx:main -- --help
Sphinx v1.2.2
Usage: /tmp/tmpydcp6kox [options] sourcedir outdir [filenames...]

General options
^^^^^^^^^^^^^^^
-b <builder>  builder to use; default is html
-a            write all files; default is to only write new and changed files
-E            don't use a saved environment, always read all files
...
```

Although sys.exit is applied blindly to the return value of the target function, this probably does what you want due to very flexible `sys.exit` semantics. Consult your target function and sys.exit documentation to be sure.

Almost certainly better and more stable, you can alternatively specify a console script exported by the app as explained below.

pex -c

If you don't know the `package:target` for the console scripts of your favorite python packages, pex allows you to use `-c` to specify a console script as defined by the distribution. For example, Fabric provides the `fab` tool when pip installed:

```
$ pex Fabric -c fab -- --help
Fatal error: Couldn't find any fabfiles!

Remember that -f can be used to specify fabfile path, and use -h for help.

Aborting.
```

Even scripts defined by the "scripts" section of a distribution can be used, e.g. with boto:

```
$ pex boto -c mturk
usage: mturk [-h] [-P] [--nicknames PATH]
             {bal,hit,hits,new,extend,expire,rm,as,approve,reject,unreject,bonus,
↪notify,give-qual,revoke-qual}
             ...
mturk: error: too few arguments
```

Note: If you run `pex -c` and come across an error similar to `pex.pex_builder.InvalidExecutableSpecification: Could not find script 'mainscript.py' in any distribution within PEX!`, double-check your setup.py and ensure that `mainscript.py` is included in your setup's `scripts` array. If you are using `console_scripts` and run into this error, double check your `console_scripts` syntax - further information for both `scripts` and `console_scripts` can be found in the Python packaging documentation.

2.3.3 Saving .pex files

Each of the commands above have been manipulating ephemeral PEX environments – environments that only exist for the duration of the pex command lifetime and immediately garbage collected.

If the `-o PATH` option is specified, a PEX file of the environment is saved to disk at `PATH`. For example we can package a standalone Sphinx as above:

```
$ pex sphinx sphinx_rtd_theme -c sphinx -o sphinx.pex
```

Instead of executing the environment, it is saved to disk:

```
$ ls -l sphinx.pex
-rwxr-xr-x  1 wickman  wheel  4988494 Mar 11 17:48 sphinx.pex
```

This is an executable environment and can be executed as before:

```
$ ./sphinx.pex --help
Sphinx v1.2.2
Usage: ./sphinx.pex [options] sourcedir outdir [filenames...]

General options
^^^^^^^^^^^^^^^
-b <builder>  builder to use; default is html
-a            write all files; default is to only write new and changed files
-E            don't use a saved environment, always read all files
...
```

As before, entry points are not required, and if not specified the PEX will default to just dropping into an interpreter. If an alternate interpreter is specified with `--python`, e.g. pypy, it will be the default hashbang in the PEX file:

```
$ pex --python=pypy flask -o flask-pypy.pex
```

The hashbang of the PEX file specifies PyPy:

```
$ head -1 flask-pypy.pex
#!/usr/bin/env pypy
```

and when invoked uses the environment PyPy:

```
$ ./flask-pypy.pex
Python 2.7.3 (87aa9de10f9c, Nov 24 2013, 20:57:21)
[PyPy 2.2.1 with GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.2.79)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> import flask
```

To specify an explicit Python shebang line (e.g. from a non-standard location or not on $PATH), you can use the `--python-shebang` option:

```
$ dist/pex --python-shebang='/Users/wickman/Python/CPython-3.4.2/bin/python3.4' -o my.
↪pex
$ head -1 my.pex
#!/Users/wickman/Python/CPython-3.4.2/bin/python3.4
```

Furthermore, this can be manipulated at runtime using the `PEX_PYTHON` environment variable.

2.3.4 Tailoring requirement resolution

In general, `pex` honors the same options as pip when it comes to resolving packages. Like pip, by default `pex` fetches artifacts from PyPI. This can be disabled with `--no-index`.

If PyPI fetching is disabled, you will need to specify a search repository via `-f/--find-links`. This may be a directory on disk or a remote simple http server.

For example, you can delegate artifact fetching and resolution to `pip wheel` for whatever reason – perhaps you're running a firewalled mirror – but continue to package with pex:

```
$ pip wheel -w /tmp/wheelhouse sphinx sphinx_rtd_theme
$ pex -f /tmp/wheelhouse --no-index -e sphinx:main -o sphinx.pex sphinx sphinx_rtd_
↪theme
```

2.3.5 Tailoring PEX execution at build time

There are a few options that can tailor how PEX environments are invoked. These can be found by running `pex --help`. Every flag mentioned here has a corresponding environment variable that can be used to override the runtime behavior which can be set directly in your environment, or sourced from a `.pexrc` file (checking for `~/.pexrc` first, then for a relative `.pexrc`).

`--zip-safe/--not-zip-safe`

Whether or not to treat the environment as zip-safe. By default PEX files are listed as zip safe. If `--not-zip-safe` is specified, the source of the PEX will be written to disk prior to invocation rather than imported via the zipimporter. NOTE: Distribution zip-safe bits will still be honored even if the PEX is marked as zip-safe. For example, included .eggs may be marked as zip-safe and invoked without the need to write to disk. Wheels are always marked as not-zip-safe and written to disk prior to PEX invocation. `--not-zip-safe` forces `--always-write-cache`.

`--always-write-cache`

Always write all packaged dependencies within the PEX to disk prior to invocation. This forces the zip-safe bit of any dependency to be ignored.

--inherit-path

By default, PEX environments are completely scrubbed empty of any packages installed on the global site path. Setting `--inherit-path` allows packages within site-packages to be considered as candidate distributions to be included for the execution of this environment. This is strongly discouraged as it circumvents one of the biggest benefits of using .pex files, however there are some cases where it can be advantageous (for example if a package does not package correctly an an egg or wheel.)

--ignore-errors

If not all of the PEX environment's dependencies resolve correctly (e.g. you are overriding the current Python interpreter with `PEX_PYTHON`) this forces the PEX file to execute despite this. Can be useful in certain situations when particular extensions may not be necessary to run a particular command.

--platform

The (abbreviated) platform to build the PEX for. This will look for wheels for the particular platform.

The abbreviated platform is described by a string of the form `PLATFORM-IMPL-PYVER-ABI`, where `PLATFORM` is the platform (e.g. `linux-x86_64`, `macosx-10.4-x86_64`), `IMPL` is the python implementation abbreviation (`cp` or `pp`), `PYVER` is either a two or more digit string representing the python version (e.g., `36` or `310`) or else a component dotted version string (e.g., `3.6` or `3.10.1`) and `ABI` is the ABI tag (e.g., `cp36m`, `cp27mu`, `abi3`, `none`). A complete example: `linux_x86_64-cp-36-cp36m`.

Constraints: when `--platform` is used the environment marker `python_full_version` will not be available if `PYVER` is not given as a three component dotted version since `python_full_version` is meant to have 3 digits (e.g., `3.8.10`). If a `python_full_version` environment marker is encountered during a resolve, an `UndefinedEnvironmentName` exception will be raised. To remedy this, either specify the full version in the platform (e.g, `linux_x86_64-cp-3.8.10-cp38`) or use `--complete-platform` instead.

--complete-platform

The completely specified platform to build the PEX for. This will look for wheels for the particular platform.

The complete platform can be either a path to a file containing JSON data or else a JSON object literal. In either case, the JSON object is expected to have two fields with any other fields ignored. The `marker_environment` field should have an object value with string field values corresponding to PEP-508 marker environment entries. It is OK to only have a subset of valid marker environment fields but it is not valid to present entries not defined in PEP-508. The `compatible_tags` field should have an array of strings value containing the compatible tags in order from most specific first to least specific last as defined in PEP-425. Pex can create complete platform JSON for you by running it on the target platform like so: `pex3 interpreter inspect --markers --tags`. For more options, particularly to select the desired target interpreter see: `pex3 interpreter inspect --help`.

2.3.6 Tailoring PEX execution at runtime

Tailoring of PEX execution can be done at runtime by setting various environment variables. The source of truth for these environment variables can be found in the pex.variables API.

2.4 Using `bdist_pex`

pex provides a convenience command for use in setuptools. `python setup.py bdist_pex` is a simple way to build executables for Python projects that adhere to standard naming conventions.

2.4.1 `bdist_pex`

The default behavior of `bdist_pex` is to build an executable using the console script of the same name as the package. For example, pip has three entry points: `pip`, `pip2` and `pip2.7` if you're using Python 2.7. Since there exists an entry point named `pip` in the `console_scripts` section of the entry points, that entry point is chosen and an executable pex is produced. The pex file will have the version number appended, e.g. `pip-7.2.0.pex`.

If no console scripts are provided, or the only console scripts available do not bear the same name as the package, then an environment pex will be produced. An environment pex is a pex file that drops you into an interpreter with all necessary dependencies but stops short of invoking a specific module or function.

2.4.2 `bdist_pex --bdist-all`

If you would like to build all the console scripts defined in the package instead of just the namesake script, `--bdist-all` will write all defined entry_points but omit version numbers and the `.pex` suffix. This can be useful if you would like to virtually install a Python package somewhere on your `$PATH` without doing something scary like `sudo pip install`:

```
$ git clone https://github.com/sphinx-doc/sphinx && cd sphinx
$ python setup.py bist_pex --bdist-all --bdist-dir=$HOME/bin
running bdist_pex
Writing sphinx-apidoc to /Users/wickman/bin/sphinx-apidoc
Writing sphinx-build to /Users/wickman/bin/sphinx-build
Writing sphinx-quickstart to /Users/wickman/bin/sphinx-quickstart
Writing sphinx-autogen to /Users/wickman/bin/sphinx-autogen
$ sphinx-apidoc --help | head -1
Usage: sphinx-apidoc [options] -o <output_path> <module_path> [exclude_path, ...]
```

2.5 Using Pants

The Pants build system can build pex files. See here for details.

2.6 PEX Recipes and Notes

2.6.1 Uvicorn and other customizable application servers

Often you want to run a third-party application server and have it use your code. You can always do this by writing a shim bit of python code that starts the application server configured to use your code. It may be simpler though to use `--inject-env` and `--inject-args` to seal this configuration into a PEX file without needing to write a shim.

For example, to package up a uvicorn-powered server of your app coroutine in `example.py` that ran on port 8888 by default you could:

```
$ pex "uvicorn[standard]" -c uvicorn --inject-args 'example:app --port 8888' -
↪oexample-app.pex
$ ./example-app.pex
INFO:     Started server process [2014]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8888 (Press CTRL+C to quit)
^CINFO:     Shutting down
INFO:     Finished server process [2014]
```

You could then over-ride the port with:

```
$ ./example-app.pex --port 0
INFO:     Started server process [2248]
INFO:     Waiting for application startup.
INFO:     ASGI 'lifespan' protocol appears unsupported.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:45751 (Press CTRL+C to quit)
```

2.6.2 Long running PEX applications and daemons

If your PEXed application will run a long time, at some point you'll likely need to debug or otherwise inspect it using operating system tools. Unless you built your application as a non-`--venv --layout loose` PEX, its final process information will be inscrutable in `ps` output since all other PEX forms re-execute themselves against an installed version of themselves in the configured `PEX_ROOT`.

You'll see something like this as a result:

```
$ ./my.pex --foo bar &
$ ps -o command | grep pex
/home/jsirois/.pyenv/versions/3.10.2/bin/python3.10 /home/jsirois/.pex/unzipped_pexes/
↪94790b07dc3768a9926dab999b41a87e399e0aa9 --foo bar
```

The original PEX file is not mentioned anywhere in the `ps` output. Worse, if you have many PEX processes it will be unclear which process corresponds to which PEX.

To remedy this, simply add setproctitle as a dependency for your PEX. The PEX runtime will then detect the presence of `setproctitle` and alter the process title so you see both the Python being used to run your PEX and the PEX file being run:

```
$ ./my.pex --foo bar &
$ ps -o command | grep pex
/home/jsirois/.pyenv/versions/3.10.2/bin/python3.10 /home/jsirois/dev/pantsbuild/
↪jsirois-pex/my.pex --foo bar
```

2.6.3 PEX app in a container

If you want to use a PEX application in a container, you can get the smallest container footprint and the lowest latency application start-up by installing it with the `venv` Pex tool. First make sure you build the pex with `--include-tools` (or `--venv`), and then install it in the container like so:

```
FROM python:3.10-slim as deps
COPY /my-app.pex /
RUN PEX_TOOLS=1 /usr/local/bin/python3.10 /my-app.pex venv --scope=deps --compile /my-
↪app

FROM python:3.10-slim as srcs
COPY /my-app.pex /
RUN PEX_TOOLS=1 /usr/local/bin/python3.10 /my-app.pex venv --scope=srcs --compile /my-
↪app

FROM python:3.10-slim
COPY --from=deps /my-app /my-app
COPY --from=srcs /my-app /my-app
ENTRYPOINT ["/my-app/pex"]
```

Here, the first two `FROM` images are illustrative. The only requirement is they need to contain the Python interpreter your app should be run with (`/usr/local/bin/python3.10` in this example).

The Pex `venv` tool will:

1) Install the PEX as a traditional venv at `/my-app` with a script at `/my-app/pex` that runs just like the original PEX.
2) Pre-compile all PEX Python code installed in the venv.

Notably, the PEX venv install is done using a multi-stage build to ensure only the final venv remains on disk and it uses two layers to ensure changes to application code do not lead to re-builds of lower layers. This accommodates the common case of modifying and re-deploying first party code more often than third party dependencies.

2.6.4 PEX-aware application

If your code benefits from knowing whether it is running from within a PEX or not, you can inspect the `PEX` environment variable. If it is set, it will be the absolute path of the PEX your code is running in. Normally this will be a PEX zip file, but it could be a directory path if the PEX was built with a `--layout` of `packed` or `loose`.

2.6.5 Gunicorn and PEX

Normally, to run a wsgi-compatible application with Gunicorn, you'd just point Gunicorn at your application, tell Gunicorn how to run it, and you're ready to go - but if your application is shipping as a PEX file, you'll have to bundle Gunicorn as a dependency and set Gunicorn as your entry point. Gunicorn can't enter a PEX file to retrieve the wsgi instance, but that doesn't prevent the PEX from invoking Gunicorn.

This retains the benefit of zero *pip install*'s to run your service, but it requires a bit more setup as you must ensure Gunicorn is packaged as a dependency. The following snippets assume Flask as the wsgi framework, Django setup should be similar:

```
$ pex flask gunicorn myapp -c gunicorn -o ~/service.pex
```

Once your pex file is created, you need to make sure to pass your wsgi app instance name to the CLI at runtime for Gunicorn to know how to hook into it, configuration can be passed in the same way:

```
$ service.pex myapp:appinstance -c /path/to/gunicorn_config.py
```

And there you have it, a fully portable python web service.

2.6.6 PEX and Proxy settings

While building pex files, you may need to fetch dependencies through a proxy. The easiest way is to use pex cli with the requests extra and environment variables. Following are the steps to do just that:

1) Install pex with requests

```
$ pip install pex[requests]
```

2) Set the environment variables

```
$ # Hopefully your proxy supports https! If not, you can export HTTP_PROXY:
$ # export HTTP_PROXY='http://user:pass@address:port'
$ export HTTPS_PROXY='https://user:pass@address:port'
```

3) Now you can test by running

```
$ pex -v pex
```

For more information on the requests module support for proxies via environment variables, see the official documentation here: http://docs.python-requests.org/en/master/user/advanced/#proxies.

2.7 PEX runtime environment variables

PEX_ALWAYS_CACHE

Boolean.

Deprecated: This env var is no longer used; all internally cached distributions in a PEX

are always installed into the local Pex dependency cache.

PEX_COVERAGE

Boolean.

Enable coverage reporting for this PEX file. This requires that the “coverage” module is

available in the PEX environment.

Default: false.

PEX_COVERAGE_FILENAME

Filename.

Write the coverage data to the specified filename. If PEX_COVERAGE_FILENAME is not

specified but PEX_COVERAGE is, coverage information will be printed to stdout and not saved.
PEX_EMIT_WARNINGS
Boolean.

Emit UserWarnings to stderr. When false, warnings will only be logged at PEX_VERBOSE >= 1.
When unset the build-time value of `–emit-warnings` will be used.

Default: unset.
PEX_EXTRA_SYS_PATH
NO DESC
PEX_FORCE_LOCAL
Boolean.

Deprecated: This env var is no longer used since user code is now always unzipped before
execution.
PEX_IGNORE_ERRORS
Boolean.

Ignore any errors resolving dependencies when invoking the PEX file. This can be useful if
you know that a particular failing dependency is not necessary to run the application.

Default: false.
PEX_IGNORE_RCFILES
Boolean.

Explicitly disable the reading/parsing of pexrc files (~/.pexrc).

Default: false.
PEX_INHERIT_PATH
String (false|prefer|fallback)

Allow inheriting packages from site-packages, user site-packages and the PYTHONPATH. By
default, PEX scrubs any non stdlib packages from sys.path prior to invoking the application.
Using ‘prefer’ causes PEX to shift any non-stdlib packages before the pex environment on
sys.path and using ‘fallback’ shifts them after instead.

Using this option is generally not advised, but can help in situations when certain
dependencies do not conform to standard packaging practices and thus cannot be bundled into
PEX files.

See also PEX_EXTRA_SYS_PATH for how to *add* to the sys.path.

Default: false.
PEX_INTERPRETER
Boolean.

Drop into a REPL instead of invoking the predefined entry point of this PEX. This can be
useful for inspecting the PEX environment interactively. It can also be used to treat the PEX
file as an interpreter in order to execute other scripts in the context of the PEX file, e.g.
“PEX_INTERPRETER=1 ./app.pex my_script.py”. Equivalent to setting PEX_MODULE to empty.

Default: false.
PEX_MODULE
String.

Override the entry point into the PEX file. Can either be a module, e.g.
‘SimpleHTTPServer’, or a specific entry point in module:symbol form, e.g. “myapp.bin:main”.
PEX_PATH
A set of one or more PEX files.

Merge the packages from other PEX files into the current environment. This allows you to
do things such as create a PEX file containing the “coverage” module or create PEX files
containing plugin entry points to be consumed by a main application. Paths should be
specified in the same manner as $PATH. For example, on a Unix system
PEX_PATH=/path/to/pex1.pex:/path/to/pex2.pex and so forth.

See also PEX_EXTRA_SYS_PATH for how to add arbitrary entries to the sys.path.
PEX_PROFILE
Boolean.

Enable application profiling. If specified and PEX_PROFILE_FILENAME is not specified, PEX
will print profiling information to stdout.

PEX_PROFILE_FILENAME
Filename.

Profile the application and dump a profile into the specified filename in the standard
"profile" module format.
PEX_PROFILE_SORT
String.

Toggle the profile sorting algorithm used to print out profile columns.

Default: 'cumulative'.
PEX_PYTHON
String.

Override the Python interpreter used to invoke this PEX. Can be either an absolute path to
an interpreter or a base name e.g. "python3.3". If a base name is provided, the $PATH will
be searched for an appropriate match.
PEX_PYTHON_PATH
String.

A {pathsep!r} separated string containing paths of blessed Python interpreters for
overriding the Python interpreter used to invoke this PEX. Can be absolute paths to
interpreters or standard $PATH style directory entries that are searched for child files
that are python binaries.

For example, on a Unix system: "/path/to/python27:/path/to/python36-distribution/bin"
PEX_ROOT
Directory.

The directory location for PEX to cache any dependencies and code. PEX must write not-zip-
safe eggs and all wheels to disk in order to activate them.

Default: ~/.pex
PEX_SCRIPT
String.

The script name within the PEX environment to execute. This must either be an entry point
as defined in a distribution's console_scripts, or a script as defined in a distribution's
scripts section. While Python supports any script including shell scripts, PEX only
supports invocation of Python scripts in this fashion.
PEX_TEARDOWN_VERBOSE
Boolean.

Enable verbosity for when the interpreter shuts down. This is mostly only useful for
debugging PEX itself.

Default: false.
PEX_TOOLS
Boolean.

Run the PEX tools.

Default: false.
PEX_UNZIP
Boolean.

Deprecated: This env var is no longer used since unzipping PEX zip files before execution
is now the default.
PEX_VENV
Boolean.

Force this PEX to create a venv under $PEX_ROOT and re-execute from there. If the pex file
will be run multiple times under a stable $PEX_ROOT the venv creation will only be performed
once and subsequent runs will enjoy lower startup latency.

Default: false.
PEX_VENV_BIN_PATH
String (false|prepend|append).

When running in PEX_VENV mode, optionally add the scripts and console scripts of
distributions in the PEX file to the $PATH.

Default: false.

PEX_VERBOSE

Integer.

Set the verbosity level of PEX debug logging. The higher the number, the more logging, with 0 being disabled. This environment variable can be extremely useful in debugging PEX environment issues.

Default: 0