pex Documentation

Release 1.3.2.post2

Brian Wickman

Apr 25, 2018

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"

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

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format is and why it could be useful for you, see *What are .pex files?* For the impatient, there is also a (slightly outdated) lightning talk published by Twitter University: WTF is PEX?.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:

CHAPTER 1

What are .pex files?

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*.

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.

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.

CHAPTER 2

Building .pex files

The easiest way to build .pex files is with 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.

CHAPTER 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.
(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.
```

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
>>> ...
```

Requirements can also be specified using the requirements.txt format, using `pex -r`. This can be a handy way to freeze a virtualenv into a PEX file:

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

3.2 Specifying entry points

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

3.2.1 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/
```

3.2.2 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

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 `from package import target; 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 sphinx_rtd_theme -e sphinx:main -- --help
Sphinx v1.2.2
Usage: /var/folders/4d/9tz0cd5n2n7947xs21gspsxc0000gp/T/tmpLr8ibZ [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
...
```

3.2.3 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
```

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.

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
```

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`).

3.5.1 --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`.

3.5.2 --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.

3.5.3 --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.)

3.5.4 --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.

3.5.5 --platform

The platform to build the pex for. Right now it defaults to the current system, but you can specify something like `linux-x86_64` or `macosx-10.6-x86_64`. This will look for bdists for the particular platform.

To build manylinux wheels for specific tags, you can add them to the platform with hyphens like `PLATFORM-PYVER-IMPL-ABI`, where `PLATFORM` is either `manylinux1-x86_64` or `manylinux1-i686`, `PYVER` is a two-digit string representing the python version (e.g., `36`), `IMPL` is the python implementation abbreviation (e.g., `cp`, `pp`, `jp`), and `ABI` is the ABI tag (e.g., `cp36m`, `cp27mu`, `abi3`, `none`). A complete example: `manylinux1_x86_64-36-cp-cp36m`.

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.

CHAPTER 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.

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.

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, ...]
```

CHAPTER 5

Other ways to build PEX files

There are other supported ways to build pex files:

- Using pants. See Pants Python documentation.
- Programmatically via the pex API.

CHAPTER 6

PEX API Reference

6.1 Module contents

6.2 pex.crawler module

Support for webpage parsing and crawling.

class pex.crawler.**Crawler**(*context=None*, *threads=1*)

Bases: object

A multi-threaded crawler that supports local (disk) and remote (web) crawling.

classmethod reset_cache()

Reset the internal crawl cache. This is intended primarily for tests.

class pex.crawler.**PageParser**

Bases: object

A helper class to extract and differentiate ordinary and download links from webpages.

classmethod links(*page*)

return all links on a page, including potentially rel= links.

classmethod rel_links(*page*)

return rel= links that should be scraped, skipping obviously data links.

pex.crawler.**unescape**(*s*)

Unescapes html. Taken from https://wiki.python.org/moin/EscapingHtml

6.3 pex.environment module

6.4 pex.fetcher module

class pex.fetcher.**FetcherBase**

Bases: abc.AbstractClass

A fetcher takes a Requirement and tells us where to crawl to find it.

pex.fetcher.**normalize_name**(*name*)

Normalize package name according to PEP-503

6.5 pex.finders module

The finders we wish we had in setuptools.

As of setuptools 3.3, the only finder for zip-based distributions is for eggs. The path-based finder only searches paths ending in .egg and not in .whl (zipped or unzipped.)

pex.finders augments pkg_resources with additional finders to achieve functional parity between wheels and eggs in terms of findability with find_distributions.

To use:

```
>>> from pex.finders import register_finders
>>> register_finders()
```

class pex.finders.**ChainedFinder**(*finders*)

Bases: object

A utility to chain together multiple pkg_resources finders.

class pex.finders.**FixedEggMetadata**(*importer*)

Bases: pkg_resources.EggMetadata

An EggMetadata provider that has functional parity with the disk-based provider.

class pex.finders.**WheelMetadata**(*importer*)

Bases: pkg_resources.EggMetadata

Metadata provider for zipped wheels.

pex.finders.**get_script_from_egg**(*name*, *dist*)

Returns location, content of script in distribution or (None, None) if not there.

pex.finders.**register_finders**()

Register finders necessary for PEX to function properly.

pex.finders.**unregister_finders**()

Unregister finders necessary for PEX to function properly.

6.6 pex.http module

class pex.http.**CachedRequestsContext**(*cache=None*, ***kw*)

Bases: *pex.http.RequestsContext*

A requests-based Context with CacheControl support.

class pex.http.**Context**

Bases: abc.AbstractClass

Encapsulate the networking necessary to do requirement resolution.

At a minimum, the Context must implement `open(link)` by returning a file-like object. Reference implementations of `read(link)` and `fetch(link)` are provided based upon `open(link)` but may be further specialized by individual implementations.

exception Error

Bases: exceptions.Exception

Error base class for Contexts to wrap application-specific exceptions.

content(*link*)

Return the encoded content associated with the link.

Parameters **link** – The `Link` to read.

fetch(*link*, *into=None*)

Fetch the binary content associated with the link and write to a file.

Parameters

- **link** – The `Link` to fetch.
- **into** – If specified, write into the directory `into`. If `None`, creates a new temporary directory that persists for the duration of the interpreter.

open(*link*)

Return an open file-like object to the link.

Parameters **link** – The `Link` to open.

read(*link*)

Return the binary content associated with the link.

Parameters **link** – The `Link` to read.

classmethod register(*context_impl*)

Register a concrete implementation of a `Context` to be recognized.

resolve(*link*)

Resolves final link throught all the redirections.

Parameters **link** – The `Link` to open.

class pex.http.**RequestsContext**(*session=None*, *verify=True*, *env=<pex.variables.Variables object>*)

Bases: `pex.http.Context`

A requests-based Context.

class pex.http.**StreamFilelike**(*request*, *link*, *chunk_size=16384*)

Bases: object

A file-like object wrapper around requests streams that performs hash validation.

classmethod detect_algorithm(*link*)

Detect the hashing algorithm from the fragment in the link, if any.

class pex.http.**UrllibContext**(**args*, ***kw*)

Bases: `pex.http.Context`

Default Python standard library Context.

6.7 pex.installer module

class pex.installer.**Installer**(*source_dir*, *strict=True*, *interpreter=None*)

Bases: pex.installer.InstallerBase

Install an unpacked distribution with a setup.py.

class pex.installer.**Packager**(*source_dir*, *strict=True*, *interpreter=None*, *install_dir=None*)

Bases: pex.installer.DistributionPackager

Create a source distribution from an unpacked setup.py-based project.

6.8 pex.interpreter module

pex support for interacting with interpreters.

6.9 pex.iterator module

The glue between fetchers, crawlers and requirements.

class pex.iterator.**Iterator**(*fetchers=None*, *crawler=None*, *follow_links=False*, *allow_prereleases=None*)

Bases: pex.iterator.IteratorInterface

A requirement iterator, the glue between fetchers, crawlers and requirements.

6.10 pex.link module

class pex.link.**Link**(*url*)

Bases: object

Wrapper around a URL.

filename

The basename of this url.

fragment

The url fragment following '#' if any.

classmethod from_filename(*filename*)

Return a *Link* wrapping the local filename.

join(*href*)

Given a href relative to this link, return the *Link* of the absolute url.

Parameters href – A string-like path relative to this link.

local

Is the url a local file?

local_path

Returns the local filesystem path (only works for file:// urls).

path

The full path of this url with any hostname and scheme components removed.

remote
Is the url a remote file?

scheme
The URI scheme used by this Link.

url
The url string to which this link points.

classmethod wrap(*url*)
Given a url that is either a string or `Link`, return a `Link`.

Parameters **url** – A string-like or `Link` object to wrap.

Returns A `Link` object wrapping the url.

classmethod wrap_iterable(*url_or_urls*)
Given a string or `Link` or iterable, return an iterable of `Link` objects.

Parameters **url_or_urls** – A string or `Link` object, or iterable of string or `Link` objects.

Returns A list of `Link` objects.

6.11 pex.package module

pex.package.**EGG_NAME**()
match(string[, pos[, endpos]]) –> match object or None. Matches zero or more characters at the beginning of the string

class pex.package.**EggPackage**(*url*, ***kw*)
Bases: `pex.package.Package`

A Package representing a built egg.

class pex.package.**Package**(*url*)
Bases: `pex.link.Link`

Base class for named Python binary packages (e.g. source, egg, wheel).

compatible(*supported_tags*)
Is this link compatible with the given tag set?

Parameters **supported_tags** (*list of 3-tuples*) – A list of tags that is supported by the target interpeter, as generated by `pex.pep425tags.get_supported()`.

classmethod from_href(*href*, ***kw*)
Convert from a url to Package.

Parameters **href** (*string*) – The url to parse

Returns A Package object if a valid concrete implementation exists, otherwise None.

classmethod register(*package_type*)
Register a concrete implementation of a Package to be recognized by pex.

satisfies(*requirement*, *allow_prereleases=None*)
Determine whether this package matches the requirement.

Parameters

- **requirement** (string or `pkg_resources.Requirement`) – The requirement to compare this Package against

- **allow_prereleases** (*Optional[bool]*) – Whether to allow prereleases to satisfy the *requirement*.

Returns True if the package matches the requirement, otherwise False

class pex.package.**SourcePackage**(*url*, ***kw*)

Bases: *pex.package.Package*

A Package representing an uncompiled/unbuilt source distribution.

classmethod split_fragment(*fragment*)

A heuristic used to split a string into version name/fragment:

```
>>> SourcePackage.split_fragment('pysolr-2.1.0-beta')
('pysolr', '2.1.0-beta')
>>> SourcePackage.split_fragment('cElementTree-1.0.5-20051216')
('cElementTree', '1.0.5-20051216')
>>> SourcePackage.split_fragment('pil-1.1.7b1-20090412')
('pil', '1.1.7b1-20090412')
>>> SourcePackage.split_fragment('django-plugin-2-2.3')
('django-plugin-2', '2.3')
```

class pex.package.**WheelPackage**(*url*, ***kw*)

Bases: *pex.package.Package*

A Package representing a built wheel.

pex.package.**distribution_compatible**(*dist*, *supported_tags=None*)

Is this distribution compatible with the given interpreter/platform combination?

Parameters **supported_tags** – A list of tag tuples specifying which tags are supported by the platform in question.

Returns True if the distribution is compatible, False if it is unrecognized or incompatible.

6.12 pex.pep425tags module

Generate and work with PEP 425 Compatibility Tags.

pex.pep425tags.**get_abbr_impl**()

Return abbreviated implementation name.

pex.pep425tags.**get_abi_tag**()

Return the ABI tag based on SOABI (if available) or emulate SOABI (CPython 2, PyPy).

pex.pep425tags.**get_darwin_arches**(*major*, *minor*, *machine*)

Return a list of supported arches (including group arches) for the given major, minor and machine architecture of an macOS machine.

pex.pep425tags.**get_flag**(*var*, *fallback*, *expected=True*, *warn=True*)

Use a fallback method for determining SOABI flags if the needed config var is unset or unavailable.

pex.pep425tags.**get_impl_tag**()

Returns the Tag for this specific implementation.

pex.pep425tags.**get_impl_ver**()

Return implementation version.

pex.pep425tags.**get_impl_version_info**()

Return sys.version_info-like tuple for use in decrementing the minor version.

`pex.pep425tags.get_platform()`
Return our platform name 'win32', 'linux_x86_64'

`pex.pep425tags.get_supported(version=None, noarch=False, platform=None, impl=None, abi=None)`
Return a list of supported tags for each version specified in *versions*.

Parameters

- **version** – string version (e.g., "33", "32") or None. If None, use local system Python version.
- **platform** – specify the exact platform you want valid tags for, or None. If None, use the local system platform.
- **impl** – specify the exact implementation you want valid tags for, or None. If None, use the local interpreter impl.
- **abi** – specify the exact abi you want valid tags for, or None. If None, use the local interpreter abi.

6.13 pex.pex module

class `pex.pex.PEX(pex='/home/docs/checkouts/readthedocs.org/user_builds/manypex/envs/stable/bin/sphinx-build', interpreter=None, env=<pex.variables.Variables object>)`
Bases: `object`

PEX, n. A self-contained python environment.

cmdline(*args=()*)
The commandline to run this environment.

Parameters **args** – Additional arguments to be passed to the application being invoked by the environment.

execute()
Execute the PEX.

This function makes assumptions that it is the last function called by the interpreter.

classmethod minimum_sys(*inherit_path*)
Return the minimum sys necessary to run this interpreter, a la python -S.

Returns (sys.path, sys.path_importer_cache, sys.modules) tuple of a bare python installation.

classmethod minimum_sys_modules(*site_libs*, *modules=None*)
Given a set of site-packages paths, return a "clean" sys.modules.

When importing site, modules within sys.modules have their __path__'s populated with additional paths as defined by *-nspkg.pth in site-packages, or alternately by distribution metadata such as *.dist-info/namespace_packages.txt. This can possibly cause namespace packages to leak into imports despite being scrubbed from sys.path.

NOTE: This method mutates modules' __path__ attributes in sys.module, so this is currently an irreversible operation.

classmethod patch_pkg_resources(**args*, ***kwds*)
Patch pkg_resources given a new working set.

patch_sys(**args*, ***kwds*)
Patch sys with all site scrubbed.

path()
Return the path this PEX was built at.

run(*args=()*, *with_chroot=False*, *blocking=True*, *setsid=False*, ***kwargs*)
Run the PythonEnvironment in an interpreter in a subprocess.

Parameters

- **args** – Additional arguments to be passed to the application being invoked by the environment.
- **with_chroot** – Run with cwd set to the environment's working directory.
- **blocking** – If true, return the return code of the subprocess. If false, return the Popen object of the invoked subprocess.
- **setsid** – If true, run the PEX in a separate operating system session.

Remaining keyword arguments are passed directly to subprocess.Popen.

6.14 pex.pex_builder module

class pex.pex_builder.**PEXBuilder**(*path=None*, *interpreter=None*, *chroot=None*, *pex_info=None*, *preamble=None*, *copy=False*)
Bases: `object`

Helper for building PEX environments.

add_dist_location(*dist*, *name=None*)
Add a distribution by its location on disk.

Parameters

- **dist** – The path to the distribution to add.
- **name** – (optional) The name of the distribution, should the dist directory alone be ambiguous. Packages contained within site-packages directories may require specifying `name`.

Raises **PEXBuilder.InvalidDistribution** – When the path does not contain a matching distribution.

PEX supports packed and unpacked .whl and .egg distributions, as well as any distribution supported by setuptools/pkg_resources.

add_distribution(*dist*, *dist_name=None*)
Add a `pkg_resources.Distribution` from its handle.

Parameters

- **dist** (`pkg_resources.Distribution`) – The distribution to add to this environment.
- **dist_name** – (optional) The name of the distribution e.g. 'Flask-0.10.0'. By default this will be inferred from the distribution itself should it be formatted in a standard way.

add_egg(*egg*)
Alias for add_dist_location.

add_interpreter_constraint(*ic*)
Add an interpreter constraint to the PEX environment.

Parameters **ic** – A version constraint on the interpreter used to build and run this PEX environment.

add_requirement(*req*)
Add a requirement to the PEX environment.

Parameters **req** – A requirement that should be resolved in this environment.

Changed in version 0.8: Removed `dynamic` and `repo` keyword arguments as they were unused.

add_resource(*filename*, *env_filename*)
Add a resource to the PEX environment.

Parameters

- **filename** – The source filename to add to the PEX.
- **env_filename** – The destination filename in the PEX. This path must be a relative path.

add_source(*filename*, *env_filename*)
Add a source to the PEX environment.

Parameters

- **filename** – The source filename to add to the PEX.
- **env_filename** – The destination filename in the PEX. This path must be a relative path.

build(*filename*, *bytecode_compile=True*)
Package the PEX into a zipfile.

Parameters

- **filename** – The filename where the PEX should be stored.
- **bytecode_compile** – If True, precompile .py files into .pyc files.

If the PEXBuilder is not yet frozen, it will be frozen by `build`. This renders the PEXBuilder immutable.

clone(*into=None*)
Clone this PEX environment into a new PEXBuilder.

Parameters **into** – (optional) An optional destination directory to clone this PEXBuilder into. If not specified, a temporary directory will be created.

Clones PEXBuilder into a new location. This is useful if the PEXBuilder has been frozen and rendered immutable.

Changed in version 0.8: The temporary directory created when `into` is not specified is now garbage collected on interpreter exit.

freeze(*bytecode_compile=True*)
Freeze the PEX.

Parameters **bytecode_compile** – If True, precompile .py files into .pyc files when freezing code.

Freezing the PEX writes all the necessary metadata and environment bootstrapping code. It may only be called once and renders the PEXBuilder immutable.

set_entry_point(*entry_point*)
Set the entry point of this PEX environment.

Parameters **entry_point** (*string or None*) – The entry point of the PEX in the form of `module` or `module:symbol`, or `None`.

By default the entry point is None. The behavior of a `None` entry point is dropping into an interpreter. If `module`, it will be executed via `runpy.run_module`. If `module:symbol`, it is equivalent to `from module import symbol; symbol()`.

The entry point may also be specified via `PEXBuilder.set_executable`.

set_executable (*filename*, *env_filename=None*)
Set the executable for this environment.

Parameters

- **filename** – The file that should be executed within the PEX environment when the PEX is invoked.
- **env_filename** – (optional) The name that the executable file should be stored as within the PEX. By default this will be the base name of the given filename.

The entry point of the PEX may also be specified via `PEXBuilder.set_entry_point`.

set_script (*script*)
Set the entry point of this PEX environment based upon a distribution script.

Parameters **script** – The script name as defined either by a console script or ordinary script within the setup.py of one of the distributions added to the PEX.

Raises `PEXBuilder.InvalidExecutableSpecification` if the script is not found in any distribution added to the PEX.

set_shebang (*shebang*)
Set the exact shebang line for the PEX file.

For example, pex_builder.set_shebang('/home/wickman/Local/bin/python3.4'). This is used to override the default behavior which is to have a #!/usr/bin/env line referencing an interpreter compatible with the one used to build the PEX.

Parameters **shebang** (*str*) – The shebang line. If it does not include the leading '#!' it will be added.

6.15 pex.pex_info module

class pex.pex_info.**PexInfo** (*info=None*)
Bases: `object`

PEX metadata.

Build metadata: build_properties: BuildProperties # (key-value information about the build system) code_hash: str # sha1 hash of all names/code in the archive distributions: {dist_name: str} # map from distribution name (i.e. path in

the internal cache) to its cache key (sha1)

requirements: list # list of requirements for this environment

Environment options pex_root: string # root of all pex-related files eg: ~/.pex entry_point: string # entry point into this pex script: string # script to execute in this pex environment

at most one of script/entry_point can be specified

zip_safe: True, default False # is this pex zip safe? inherit_path: false/fallback/prefer # should this pex inherit site-packages + PYTHONPATH? ignore_errors: True, default False # should we ignore inability to resolve dependencies? always_write_cache: False # should we always write the internal cache to disk first?

this is useful if you have very large dependencies that # do not fit in RAM constrained environments

Changed in version 0.8: Removed the `repositories` and `indices` information, as they were never implemented.

build_properties
Information about the system on which this PEX was generated.

Returns A dictionary containing metadata about the environment used to build this PEX.

inherit_path
Whether or not this PEX should be allowed to inherit system dependencies.

By default, PEX environments are scrubbed of all system distributions prior to execution. This means that PEX files cannot rely upon preexisting system libraries.

By default inherit_path is false. This may be overridden at runtime by the $PEX_INHERIT_PATH environment variable.

interpreter_constraints
A list of constraints that determine the interpreter compatibility for this pex, using the Requirement-style format, e.g. `'CPython>=3', or just '>=2.7,<3'` for requirements agnostic to interpreter class.

This property will be used at exec time when bootstrapping a pex to search PEX_PYTHON_PATH for a list of compatible interpreters.

merge_pex_path(*pex_path*)
Merges a new PEX_PATH definition into the existing one (if any). :param string pex_path: The PEX_PATH to merge.

pex_path
A colon separated list of other pex files to merge into the runtime environment.

This pex info property is used to persist the PEX_PATH environment variable into the pex info metadata for reuse within a built pex.

zip_safe
Whether or not this PEX should be treated as zip-safe.

If set to false and the PEX is zipped, the contents of the PEX will be unpacked into a directory within the PEX_ROOT prior to execution. This allows code and frameworks depending upon __file__ existing on disk to operate normally.

By default zip_safe is True. May be overridden at runtime by the $PEX_FORCE_LOCAL environment variable.

class pex.pex_info.**PexPlatform**(*interpreter*, *version*, *strict*)
Bases: `tuple`

interpreter
Alias for field number 0

strict
Alias for field number 2

version
Alias for field number 1

6.16 pex.resolver module

class pex.resolver.**CachingResolver**(*cache*, *cache_ttl*, **args*, ***kw*)
Bases: *pex.resolver.Resolver*

A package resolver implementing a package cache.

class pex.resolver.**Resolver**(*allow_prereleases=None*, *interpreter=None*, *platform=None*, *pkg_blacklist=None*)

Bases: object

Interface for resolving resolvable entities into python packages.

class pex.resolver.**StaticIterator**(*packages*, *allow_prereleases=None*)

Bases: pex.iterator.IteratorInterface

An iterator that iterates over a static list of packages.

pex.resolver.**patched_packing_env**(**args*, ***kwds*)

Monkey patch packaging.markers.default_environment

pex.resolver.**platform_to_tags**(*platform*, *interpreter*)

Splits a "platform" like linux_x86_64-36-cp-cp36m into its components.

If a simple platform without hyphens is specified, we will fall back to using the current interpreter's tags.

pex.resolver.**resolve**(*requirements*, *fetchers=None*, *interpreter=None*, *platform=None*, *context=None*, *precedence=None*, *cache=None*, *cache_ttl=None*, *allow_prereleases=None*, *pkg_blacklist=None*)

Produce all distributions needed to (recursively) meet *requirements*

Parameters

- **requirements** – An iterator of Requirement-like things, either pkg_resources.Requirement objects or requirement strings.
- **fetchers** – (optional) A list of Fetcher objects for locating packages. If unspecified, the default is to look for packages on PyPI.
- **interpreter** – (optional) A PythonInterpreter object to use for building distributions and for testing distribution compatibility.
- **versions** – (optional) a list of string versions, of the form ["33", "32"], or None. The first version will be assumed to support our ABI.
- **platform** – (optional) specify the exact platform you want valid tags for, or None. If None, use the local system platform.
- **impl** – (optional) specify the exact implementation you want valid tags for, or None. If None, use the local interpreter impl.
- **abi** – (optional) specify the exact abi you want valid tags for, or None. If None, use the local interpreter abi.
- **context** – (optional) A Context object to use for network access. If unspecified, the resolver will attempt to use the best available network context.
- **precedence** – (optional) An ordered list of allowable Package classes to be used for producing distributions. For example, if precedence is supplied as (WheelPackage, SourcePackage), wheels will be preferred over building from source, and eggs will not be used at all. If (WheelPackage, EggPackage) is suppplied, both wheels and eggs will be used, but the resolver will not resort to building anything from source.
- **cache** – (optional) A directory to use to cache distributions locally.
- **cache_ttl** – (optional integer in seconds) If specified, consider non-exact matches when resolving requirements. For example, if setuptools==2.2 is specified and setuptools 2.2 is available in the cache, it will always be used. However, if a non-exact requirement such as setuptools>=2,<3 is specified and there exists a setuptools distribution newer than cache_ttl seconds that satisfies the requirement, then it will be used. If the distribution

is older than cache_ttl seconds, it will be ignored. If `cache_ttl` is not specified, resolving inexact requirements will always result in making network calls through the `context`.

- **allow_prereleases** – (optional) Include pre-release and development versions. If unspecified only stable versions will be resolved, unless explicitly included.
- **pkg_blacklist** – (optional) A blacklist dict (str->str) that maps package name to an interpreter constraint. If a package name is in the blacklist and its interpreter constraint matches the target interpreter, skip the requirement. This is needed to ensure that universal requirement resolves for a target interpreter version do not error out on interpreter specific requirements such as backport libs like *functools32*. For example, a valid blacklist is {'functools32': 'CPython>3'}. NOTE: this keyword is a temporary fix and will be reverted in favor of a long term solution tracked by: https://github.com/pantsbuild/pex/issues/456

Returns List of `pkg_resources.Distribution` instances meeting `requirements`.

Raises

- **Unsatisfiable** – If `requirements` is not transitively satisfiable.
- **Untranslateable** – If no compatible distributions could be acquired for a particular requirement.

This method improves upon the setuptools dependency resolution algorithm by maintaining sets of all compatible distributions encountered for each requirement rather than the single best distribution encountered for each requirement. This prevents situations where `tornado` and `tornado==2.0` could be treated as incompatible with each other because the "best distribution" when encountering `tornado` was tornado 3.0. Instead, `resolve` maintains the set of compatible distributions for each requirement as it is encountered, and iteratively filters the set. If the set of distributions ever becomes empty, then `Unsatisfiable` is raised.

Changed in version 0.8: A number of keywords were added to make requirement resolution slightly easier to configure. The optional `obtainer` keyword was replaced by `fetchers`, `translator`, `context`, `threads`, `precedence`, `cache` and `cache_ttl`, also all optional keywords.

Changed in version 1.0: The `translator` and `threads` keywords have been removed. The choice of threading policy is now implicit. The choice of translation policy is dictated by `precedence` directly.

Changed in version 1.0: `resolver` is now just a wrapper around the *`Resolver`* and *`CachingResolver`* classes.

`pex.resolver.`**`resolve_multi`**(*requirements*, *fetchers=None*, *interpreters=None*, *platforms=None*, *context=None*, *precedence=None*, *cache=None*, *cache_ttl=None*, *allow_prereleases=None*, *pkg_blacklist=None*)

A generator function that produces all distributions needed to meet *requirements* for multiple interpreters and/or platforms.

Parameters

- **requirements** – An iterator of Requirement-like things, either `pkg_resources.Requirement` objects or requirement strings.
- **fetchers** – (optional) A list of `Fetcher` objects for locating packages. If unspecified, the default is to look for packages on PyPI.
- **interpreters** – (optional) An iterable of `PythonInterpreter` objects to use for building distributions and for testing distribution compatibility.
- **platforms** – (optional) An iterable of PEP425-compatible platform strings to use for filtering compatible distributions. If unspecified, the current platform is used, as determined by *Platform.current()*.
- **context** – (optional) A `Context` object to use for network access. If unspecified, the resolver will attempt to use the best available network context.

- **precedence** – (optional) An ordered list of allowable `Package` classes to be used for producing distributions. For example, if precedence is supplied as `(WheelPackage, SourcePackage)`, wheels will be preferred over building from source, and eggs will not be used at all. If `(WheelPackage, EggPackage)` is suppplied, both wheels and eggs will be used, but the resolver will not resort to building anything from source.
- **cache** – (optional) A directory to use to cache distributions locally.
- **cache_ttl** – (optional integer in seconds) If specified, consider non-exact matches when resolving requirements. For example, if `setuptools==2.2` is specified and setuptools 2.2 is available in the cache, it will always be used. However, if a non-exact requirement such as `setuptools>=2,<3` is specified and there exists a setuptools distribution newer than cache_ttl seconds that satisfies the requirement, then it will be used. If the distribution is older than cache_ttl seconds, it will be ignored. If `cache_ttl` is not specified, resolving inexact requirements will always result in making network calls through the `context`.
- **allow_prereleases** – (optional) Include pre-release and development versions. If unspecified only stable versions will be resolved, unless explicitly included.
- **pkg_blacklist** – (optional) A blacklist dict (str->str) that maps package name to an interpreter constraint. If a package name is in the blacklist and its interpreter constraint matches the target interpreter, skip the requirement. This is needed to ensure that universal requirement resolves for a target interpreter version do not error out on interpreter specific requirements such as backport libs like *functools32*. For example, a valid blacklist is {'functools32': 'CPython>3'}. NOTE: this keyword is a temporary fix and will be reverted in favor of a long term solution tracked by: https://github.com/pantsbuild/pex/issues/456

Yields All `pkg_resources.Distribution` instances meeting `requirements`.

Raises

- **Unsatisfiable** – If `requirements` is not transitively satisfiable.
- **Untranslateable** – If no compatible distributions could be acquired for a particular requirement.

6.17 pex.testing module

class `pex.testing.`**IntegResults**

Bases: `pex.testing.results`

Convenience object to return integration run results.

`pex.testing.`**get_dep_dist_names_from_pex**(*pex_path*, *match_prefix=''*)

Given an on-disk pex, extract all of the unique first-level paths under *.deps*.

`pex.testing.`**run_pex_command**(*args*, *env=None*)

Simulate running pex command for integration testing.

This is different from run_simple_pex in that it calls the pex command rather than running a generated pex. This is useful for testing end to end runs with specific command line arguments or env options.

`pex.testing.`**temporary_content**(**args*, ***kwds*)

Write content to disk where content is map from string => (int, string).

If target is int, write int random bytes. Otherwise write contents of string.

`pex.testing.`**temporary_filename**(**args*, ***kwds*)

Creates a temporary filename.

This is useful when you need to pass a filename to an API. Windows requires all handles to a file be closed before deleting/renaming it, so this makes it a bit simpler.

`pex.testing.write_simple_pex(td, exe_contents, dists=None, sources=None, coverage=False)`
Write a pex file that contains an executable entry point

Parameters

- **td** – temporary directory path
- **exe_contents** (*string*) – entry point python file
- **dists** – distributions to include, typically sdists or bdists
- **sources** – sources to include, as a list of pairs (env_filename, contents)
- **coverage** – include coverage header

6.18 pex.tracer module

class `pex.tracer.TraceLogger(predicate=None, output=<open file '<stderr>', mode 'w'>, clock=<module 'time' (built-in)>, prefix='')`
Bases: `object`

A multi-threaded tracer.

6.19 pex.translator module

class `pex.translator.ChainedTranslator(*translators)`
Bases: `pex.translator.TranslatorBase`

Glue a sequence of Translators together in priority order. The first Translator to resolve a requirement wins.

class `pex.translator.TranslatorBase`
Bases: `abc.AbstractClass`

Translate a link into a distribution.

6.20 pex.util module

class `pex.util.Memoizer`
Bases: `object`

A thread safe class for memoizing the results of a computation.

`pex.util.iter_pth_paths(filename)`
Given a .pth file, extract and yield all inner paths without honoring imports. This shadows python's site.py behavior, which is invoked at interpreter startup.

`pex.util.merge_split(*paths)`
Merge paths into a single path delimited by colons and split on colons to return a list of paths.

Parameters **paths** – a variable length list of path strings

Returns a list of paths from the merged path list split by colons

`pex.util.named_temporary_file(*args, **kwds)`

Due to a bug in python (https://bugs.python.org/issue14243), we need this to be able to use the temporary file without deleting it.

6.21 pex.variables module

class `pex.variables.Variables(environ=None, rc=None, use_defaults=True)`

Bases: `object`

Environment variables supported by the PEX runtime.

PEX_ALWAYS_CACHE

Boolean

Always write PEX dependencies to disk prior to invoking regardless whether or not the dependencies are zip-safe. For certain dependencies that are very large such as numpy, this can reduce the RAM necessary to launch the PEX. The data will be written into $PEX_ROOT, which by default is $HOME/.pex. Default: false.

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_FORCE_LOCAL

Boolean

Force this PEX to be not-zip-safe. This forces all code and dependencies to be written into $PEX_ROOT prior to invocation. This is an option for applications with static assets that refer to paths relative to __file__ instead of using pkgutil/pkg_resources. Default: false.

PEX_HTTP_RETRIES

Integer

The number of HTTP retries when performing dependency resolution when building a PEX file. Default: 5.

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

Boolean

Allow inheriting packages from site-packages. By default, PEX scrubs any packages and namespace packages from sys.path prior to invoking the application. This is generally not advised, but can be used in

situations when certain dependencies do not conform to standard packaging practices and thus cannot be bundled into PEX files. 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, e.g. PEX_PATH=/path/to/pex1.pex:/path/to/pex2.pex and so forth.

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 colon-separated string containing paths of blessed Python interpreters for overriding the Python interpreter used to invoke this PEX. Must be absolute paths to the interpreter.

Ex: "/path/to/python27:/path/to/python36"

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_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

classmethod from_rc(*rc=None*)
Read pex runtime configuration variables from a pexrc file.

Parameters **rc** – an absolute path to a pexrc file.

Returns A dict of key value pairs found in processed pexrc files.

Return type dict

patch(**args*, ***kwds*)
Update the environment for the duration of a context.

strip_defaults()
Returns a copy of these variables but with defaults stripped.

Any variables not explicitly set in the environment will have a value of *None*.

Python Module Index

p

Index

L

M

N

O

P

R

S

T

U

V

W

Z