virtualenv

Release 20.0.4

unknown

February 14, 2020

CONTENTS

1 Useful links **3**

1.1 Installation 3
1.2 User Guide 4
1.3 CLI interface 8
1.4 Extend functionality 11
1.5 Development 14
1.6 Release History 17

Python Module Index **21**

Index **23**

`virtualenv` is a tool to create isolated Python environments. Since Python `3.3`, a subset of it has been integrated into the standard library under the venv module. The `venv` module does not offer all features of this library, to name just a few more prominent:

- is slower (by not having the `app-data` seed method),
- is not as extendable,
- cannot create virtual environments for arbitrarily installed python versions (and automatically discover these),
- is not upgrade-able via pip,
- does not have as rich programmatic API (describe virtual environments without creating them).

The basic problem being addressed is one of dependencies and versions, and indirectly permissions. Imagine you have an application that needs version `1` of `LibFoo`, but another application requires version `2`. How can you use both these libraries? If you install everything into your host python (e.g. `python3.8`) it's easy to end up in a situation where two packages have conflicting requirements.

Or more generally, what if you want to install an application *and leave it be*? If an application works, any change in its libraries or the versions of those libraries can break the application. Also, what if you can't install packages into the global `site-packages` directory, due to not having permissions to change the host python environment?

In all these cases, `virtualenv` can help you. It creates an environment that has its own installation directories, that doesn't share libraries with other virtualenv environments (and optionally doesn't access the globally installed libraries either).

CHAPTER

ONE

USEFUL LINKS

Related projects, that build abstractions on top of virtualenv

- virtualenvwrapper - a useful set of scripts for creating and deleting virtual environments
- pew - provides a set of commands to manage multiple virtual environments
- tox - a generic virtualenv management and test automation command line tool, driven by a `tox.ini` configuration file
- nox - a tool that automates testing in multiple Python environments, similar to tox, driven by a `noxfile.py` configuration file

Tutorials

- Corey Schafer tutorial on how to use it
- Using virtualenv with mod_wsgi

Presenting how the package works from within

- Bernat Gabor: status quo of virtual environments
- Carl Meyer: Reverse-engineering Ian Bicking's brain: inside pip and virtualenv

1.1 Installation

1.1.1 via pipx

virtualenv is a CLI tool that needs a Python interpreter to run. If you already have a `Python 3.5+` interpreter the best is to use pipx to install virtualenv into an isolated environment. This has the added benefit that later you'll be able to upgrade virtualenv without affecting other parts of the system.

```
pipx install virtualenv
virtualenv --help
```

1.1.2 via pip

Alternatively you can install it within the global Python interpreter itself (perhaps as a user package via the `--user` flag). Be cautious if you are using a python install that is managed by your operating system or another package manager. `pip` might not coordinate with those tools, and may leave your system in an inconsistent state.

```
python -m pip --user install virtualenv
python -m virtualenv --help
```

1.1.3 via zipapp

You can use virtualenv without installing it too. We publish a Python zipapp, you can just download this from https://bootstrap.pypa.io/virtualenv.pyz and invoke this package with a python interpreter:

```
python virtualenv.pyz --help
```

The root level zipapp is always the current latest release. To get the last supported zipapp against a given python minor release use the link `https://bootstrap.pypa.io/virtualenv/x.y/virtualenv.pyz`, e.g. for the last virtualenv supporting Python 2.7 use https://bootstrap.pypa.io/virtualenv/2.7/virtualenv.pyz.

1.1.4 Python and OS Compatibility

virtualenv works with the following Python interpreter implementations:

- CPython versions 2.7, 3.4, 3.5, 3.6, 3.7, 3.8
- PyPy 2.7 and 3.4+.

This means virtualenv works on the latest patch version of each of these minor versions. Previous patch versions are supported on a best effort approach. virtualenv works on the following platforms:

- Unix/Linux,
- macOS,
- Windows.

1.2 User Guide

1.2.1 Introduction

Virtualenv has one basic command:

```
virtualenv
```

This will create a python virtual environment of the same version as virtualenv is installed into under path `venv`. The path where to generate the virtual environment can be changed via a positional argument being passed in, see the *`dest`* flag. The command line tool has quite a few of flags that modify the components behaviour, for a full list make sure to check out *CLI flags*.

The tool works in two phases:

- **Phase 1** discovers a python interpreter to create a virtual environment from (by default this is the same python as the one `virtualenv` is running from, however we can change this via the *`p`* option).

- **Phase 2** creates a virtual environment at the specified destination (*dest*), this can be broken down into three further sub-steps:
  - create a python that matches the target python interpreter from phase 1,
  - install (bootstrap) seed packages (one or more of pip, setuptools, wheel) in the created virtual environment,
  - install activation scripts into the binary directory of the virtual environment (these will allow end user to *activate* the virtual environment from various shells).

The python in your new virtualenv is effectively isolated from the python that was used to create it.

1.2.2 Python discovery

The first thing we need to be able to create a virtual environment is a python interpreter. This will describe to the tool what type of virtual environment you would like to create, think of it as: version, architecture, implementation.

`virtualenv` being a python application has always at least one such available, the one `virtualenv` itself is using it, and as such this is the default discovered element. This means that if you install `virtualenv` under python `3.8`, virtualenv will by default create virtual environments that are also of version `3.8`.

Created python virtual environments are usually not self-contained. A complete python packaging is usually made up of thousand of files, so it's not efficient to install the entire python again into a new folder. Instead virtual environments are mere shells, that contain little within itself, and borrow most from the system python (this is what you installed, when you installed python itself). This does mean that if you upgrade your system python your virtual environments *might* break, so watch out. The upside of this referring to the system python is that creating virtual environments can be fast.

Here we'll describe the builtin mechanism (note this can be extended though by plugins). The CLI flag *p* or *python* allows you to specify a python specifier for what type of virtual environment you would like, the format is either:

- a relative/absolute path to a Python interpreter,
- a specifier identifying the Python implementation, version, architecture in the following format:

  ```
  {python implementation name}{version}{architecture}
  ```

  We have the following restrictions:
  - the python implementation is all alphabetic characters (`python` means any implementation, and if is missing it defaults to `python`),
  - the version is a dot separated version number,
  - the architecture is either `-64` or `-32` (missing means `any`).

  For example:
  - `python3.8.1` means any python implementation having the version `3.8.1`,
  - `3` means any python implementation having the major version `3`,
  - `cpython3` means a `CPython` implementation havin the version `3`,
  - `pypy2` means a python interpreter with the `PyPy` implementation and major version `2`.

  Given the specifier `virtualenv` will apply the following strategy to discover/find the system executable:
  - If we're on Windows look into the Windows registry, and check if we see any registered Python implementations that match the specification. This is in line with expectation laid out inside PEP-514

  - Try to discover a matching python executable within the folders enumerated on the `PATH` environment variable. In this case we'll try to find an executable that has a name roughly similar to the specification (for exact logic, please see the implementation code).

Warning: As detailed above virtual environments usually just borrow things from the system Python, they don't actually contain all the data from the system Python. The version of the python executable is hardcoded within the python exe itself. Therefore if you upgrade your system Python, your virtual environment will still report the version before the upgrade, even though now other than the executable all additional content (standard library, binary libs, etc) are of the new version.

Baring any major incompatibilities (rarely the case) the virtual environment will continue working, but other than the content embedded within the python executable it will behave like the upgraded version. If a such virtual environment python is specified as the target python interpreter, we will create virtual environments that match the new system Python version, not the version reported by the virtual environment.

1.2.3 Creators

These are what actually setup the virtual environment, usually as a reference against the system python. virtualenv at the moment has two types of virtual environments:

- `venv` - this delegates the creation process towards the `venv` module, as described in PEP 404. This is only available on Python interpreters having version `3.4` or later, and also has the downside that virtualenv **must** create a process to invoke that module (unless virtualenv is installed in the system python), which can be an expensive operation (especially true on Windows).
- `builtin` - this means `virtualenv` is able to do the creation operation itself (by knowing exactly what files to create and what system files needs to be referenced). The creator with name `builtin` is an alias on the first creator that's of this type (we provide creators for various target environments, that all differ in actual create operations, such as CPython 2 on Windows, PyPy2 on Windows, CPython3 on Posix, PyPy3 on Posix, and so on; for a full list see *`creator`*).

1.2.4 Seeders

These will install for you some seed packages (one or more of the: pip, setuptools, wheel) that enables you to install additional python packages into the created virtual environment (by invoking pip). There are two main seed mechanism available:

- `pip` - this method uses the bundled pip with virtualenv to install the seed packages (note, a new child process needs to be created to do this).
- `app-data` - this method uses the user application data directory to create install images. These images are needed to be created only once, and subsequent virtual environments can just link/copy those images into their pure python library path (the `site-packages` folder). This allows all but the first virtual environment creation to be blazing fast (a `pip` mechanism takes usually 98% of the virtualenv creation time, so by creating this install image that we can just link into the virtual environments install directory we can achieve speedups of shaving the initial 1 minutes 10 seconds down to just 8 seconds in case of copy, or `0.8` seconds in case symlinks are available - this is on Windows, Linux/macOS with symlinks this can be as low as `100ms` from 3+ seconds). To override the filesystem location of the seed cache, one can use the `VIRTUALENV_OVERRIDE_APP_DATA` environment variable.

1.2.5 Activators

These are activation scripts that will mangle with your shells settings to ensure that commands from within the python virtual environment take priority over your system paths. For example if invoking `pip` from your shell returned the system pythons pip before activation, once you do the activation this should refer to the virtual environments `pip`. Note, though that all we do is change priority; so if your virtual environments `bin`/`Scripts` folder does not contain some executable, this will still resolve to the same executable it would have resolved before the activation.

For a list of shells we provide activators see *activators*. The location of these is right alongside the python executables (usually `Scripts` folder on Windows, `bin` on POSIX), and are named as `activate` (and some extension that's specific per activator; no extension is bash). You can invoke them, usually by source-ing (the source command might vary by shell - e.g. bash is `.`):

```
source bin/activate
```

This is all it does; it's purely a convenience of prepending the virtual environments binary folder onto the `PATH` environment variable. Note you don't have to activate a virtual environment to use it. In this case though you would need to type out the path to the executables, rather than relying on your shell to resolve them to your virtual environment.

The `activate` script will also modify your shell prompt to indicate which environment is currently active. The script also provisions a `decativate` command that will allow you to undo the operation:

```
deactivate
```

Note: If using Powershell, the `activate` script is subject to the execution policies on the system. By default Windows 7 and later, the system's execution policy is set to `Restricted`, meaning no scripts like the `activate` script are allowed to be executed.

However, that can't stop us from changing that slightly to allow it to be executed. You may relax the system execution policy to allow running of local scripts without verifying the code signature using the following:

```
Set-ExecutionPolicy RemoteSigned
```

Since the `activate.ps1` script is generated locally for each virtualenv, it is not considered a remote script and can then be executed.

A longer explanation of this can be found within Allison Kaptur's 2013 blog post: There's no magic: virtualenv edition explains how virtualenv uses bash and Python and `PATH` and `PYTHONHOME` to isolate virtual environments' paths.

1.2.6 Programmatic API

At the moment `virtualenv` offers only CLI level interface. If you want to trigger invocation of Python environments from within Python you should be using the `virtualenv.cli_run` method; this takes an `args` argument where you can pass the options the same way you would from the command line. The run will return a session object containing data about the created virtual environment.

```
from virtualenv import cli_run

cli_run(["venv"])
```

`virtualenv.`**`cli_run`**(*args*, *options=None*)

Create a virtual environment given some command line interface arguments

Parameters

- **args** – the command line arguments
- **options** – passing in a `argparse.Namespace` object allows return of the parsed options

Returns the session object of the creation (its structure for now is experimental and might change on short notice)

class `virtualenv.session.`**Session**(*verbosity*, *interpreter*, *creator*, *seeder*, *activators*)

Represents a virtual environment creation session

property verbosity

The verbosity of the run

property interpreter

Create a virtual environment based on this reference interpreter

property creator

The creator used to build the virtual environment (must be compatible with the interpreter)

property seeder

The mechanism used to provide the seed packages (pip, setuptools, wheel)

property activators

Activators used to generate activations scripts

1.3 CLI interface

1.3.1 CLI flags

`virtualenv` is primarily a command line application.

It modifies the environment variables in a shell to create an isolated Python environment, so you'll need to have a shell to run it. You can type in `virtualenv` (name of the application) followed by flags that control its behaviour. All options have sensible defaults, and there's one required argument: then name/path of the virtual environment to create. The default values for the command line options can be overridden via the *Configuration file* or *Environment Variables*. Environment variables takes priority over the configuration file values (`--help` will show if a default comes from the environment variable as the help message will end in this case with environment variables or the configuration file).

The options that can be passed to virtualenv, along with their default values and a short description are listed below.

virtualenv [OPTIONS]

Named Arguments		
--version		display the version of the virtualenv package and it's location, then exit
--with-traceback	`False`	on failure also display the stacktrace internals of virtualenv

verbosity verbosity = verbose - quiet, default INFO, mapping => CRITICAL=0, ERROR=1, WARNING=2, INFO=3, DEBUG=4, NOTSET=5		
-v, *--verbose*	2	increase verbosity
-q, *--quiet*	0	decrease verbosity

discovery

core options shared across all discovery		
--discovery	builtin	interpreter discovery method; choice of: builtin
-p, *--python*	the python executable virtualenv is installed into	target interpreter for which to create a virtual (either absolute path or identifier string)

creator

core options shared across all creator		
--creator	builtin if exist, else venv	create environment via; choice of: cpython2-posix, cpython2-win, cpython3-posix, cpython3-win, pypy2-posix, pypy2-win, pypy3-posix, pypy3-win, venv
dest		directory to create virtualenv at
--clear	False	remove the destination directory if exist before starting (will overwrite files otherwise)
--system-site-packages	False	give the virtual environment access to the system site-packages dir
--symlinks	True	try to use symlinks rather than copies, when symlinks are not the default for the platform
--copies, *--always-copy*	False	try to use copies rather than symlinks, even when symlinks are the default for the platform

seeder

core options shared across all seeder		
--seeder	app-data	seed packages install method; choice of: app-data, pip
--no-seed, *--without-pip*	False	do not install seed packages
--download	False	pass to enable download of the latest pip/setuptools/wheel from PyPI
--no-download, *--never-download*	True	pass to disable download of the latest pip/setuptools/wheel from PyPI
--extra-search-dir	[]	a path containing wheels the seeder may also use beside bundled (can be set 1+ times)
--pip	latest	pip version to install, bundle for bundled
--setuptools	latest	setuptools version to install, bundle for bundled
--wheel	latest	wheel version to install, bundle for bundled
--no-pip	False	do not install pip
--no-setuptools	False	do not install setuptools
--no-wheel	False	do not install wheel

app-data options specific to seeder app-data		
--clear-app-data	False	clear the app data folder of seed images
--symlink-app-data	False	symlink the python packages from the app-data folder (requires seed pip>=19.3)

activators

core options shared across all activators		
--activators	comma separated list of activators supported	activators to generate - default is all supported; choice of: bash, batch, cshell, fish, powershell, python, xonsh
--prompt		provides an alternative prompt prefix for this environment

1.3.2 Defaults

Configuration file

virtualenv looks for a standard ini configuration file. The exact location depends on the operating system you're using, as determined by appdirs application data definition. The configuration file location is printed as at the end of the output when `--help` is passed.

The keys of the settings are derived from the long command line option. For example, *--python* would be specified as:

```
[virtualenv]
python = /opt/python-3.3/bin/python
```

Options that take multiple values, like *extra-search-dir* can be specified as:

```
[virtualenv]
extra-search-dir =
    /path/to/dists
    /path/to/other/dists
```

Environment Variables

Each command line option has a corresponding environment variables with the name format `VIRTUALENV_<UPPER_NAME>`. The `UPPER_NAME` is the name of the command line options capitalized and dashes (`'-'`) replaced with underscores (`'_'`).

For example, to use a custom Python binary, instead of the one virtualenv is run with, you can set the environment variable `VIRTUALENV_PYTHON` like:

```
env VIRTUALENV_PYTHON=/opt/python-3.8/bin/python virtualenv
```

This also works for appending command line options, like *extra-search-dir*, where a literal newline is used to separate the values:

```
env VIRTUALENV_EXTRA_SEARCH_DIR="/path/to/dists\n/path/to/other/dists" virtualenv
```

The equivalent CLI-flags based invocation, for the above example, would be:

```
virtualenv --extra-search-dir=/path/to/dists --extra-search-dir=/path/to/other/dists
```

1.4 Extend functionality

`virtualenv` allows one to extend the builtin functionality via a plugin system. To add a plugin you need to:

- write a python file containing the plugin code which follows our expected interface,
- package is a a python library,
- install it alongside the virtual environment.

Warning: The public API of some of these components is still to be finalized, consider the current interface a beta one until we get some feedback on how well we planned ahead. We expect to do this by end of Q3 2020. Consider the class interface explained below as initial draft proposal. We reserve the right to change the API until then, however such changes will be communicated in a timely fashion, and you'll have time to migrate. Thank you for your understanding.

1.4.1 Python discovery

The python discovery mechanism is a component that needs to answer the following answer: based on some type of user input give me a Python interpreter on the machine that matches that. The builtin interpreter achieves tries to discover an installed Python interpreter (based on PEP-515 and `PATH` discovery) on the users machine where the user input is a python specification. An alternative such discovery mechanism for example would be to use the popular pyenv project to discover, and if not present install the requested Python interpreter. Python discovery mechanisms must be registered under key `virtualenv.discovery`, and the plugin must implement *virtualenv.discovery.discover.Discover*:

```
virtualenv.discovery =
     pyenv = virtualenv_pyenv.discovery:PyEnvDiscovery
```

class virtualenv.discovery.discover.**Discover**(*options*)

Discover and provide the requested Python interpreter

Create a new discovery mechanism.

Parameters **options** – the parsed options as defined within *add_parser_arguments()*

classmethod **add_parser_arguments**(*parser*)

Add CLI arguments for this discovery mechanisms.

Parameters **parser** – the CLI parser

abstract **run**()

Discovers an interpreter.

Returns the interpreter ready to use for virtual environment creation

property **interpreter**

Returns the interpreter as returned by *run()*, cached

1.4.2 Creators

Creators are what actually perform the creation of a virtual environment. The builtin virtual environment creators all achieve this by referencing a global install; but would be just as valid for a creator to install a brand new entire python under the target path; or one could add additional creators that can create virtual environments for other python implementations, such as IronPython. They must be registered under and entry point with key `virtualenv.discovery` , and the class must implement *virtualenv.create.creator.Creator*:

```
virtualenv.create =
     cpython3-posix = virtualenv.create.via_global_ref.builtin.cpython.
↪cpython3:CPython3Posix
```

class virtualenv.create.creator.**Creator**(*options*, *interpreter*)
A class that given a python Interpreter creates a virtual environment

Construct a new virtual environment creator.

Parameters

- **options** – the CLI option as parsed from *add_parser_arguments()*
- **interpreter** – the interpreter to create virtual environment from

classmethod can_create(*interpreter*)
Determine if we can create a virtual environment.

Parameters **interpreter** – the interpreter in question

Returns `None` if we can't create, any other object otherwise that will be forwarded to *add_parser_arguments()*

classmethod add_parser_arguments(*parser*, *interpreter*, *meta*)
Add CLI arguments for the creator.

Parameters

- **parser** – the CLI parser
- **interpreter** – the interpreter we're asked to create virtual environment for
- **meta** – value as returned by *can_create()*

abstract create()
Perform the virtual environment creation.

1.4.3 Seed mechanism

Seeders are what given a virtual environment will install somehow some seed packages into it. They must be registered under and entry point with key `virtualenv.seed` , and the class must implement *virtualenv.seed.seeder.Seeder*:

```
virtualenv.seed =
     db = virtualenv.seed.fromDb:InstallFromDb
```

class virtualenv.seed.seeder.**Seeder**(*options*, *enabled*)
A seeder will install some seed packages into a virtual environment.

Parameters

- **options** – the parsed options as defined within *add_parser_arguments()*
- **enabled** – a flag weather the seeder is enabled or not

classmethod add_parser_arguments(*parser*, *interpreter*)
Add CLI arguments for this seed mechanisms.

Parameters

- **parser** – the CLI parser
- **interpreter** – the interpreter this virtual environment is based of

abstract run(*creator*)
Perform the seed operation.

Parameters **creator** – the creator (based of *virtualenv.create.creator.Creator*) we used to create this virtual environment

1.4.4 Activation scripts

If you want add an activator for a new shell you can do this by implementing a new activator. They must be registered under and entry point with key `virtualenv.activate` , and the class must implement *virtualenv.activation.activator.Activator*:

```
virtualenv.activate =
     bash = virtualenv.activation.bash:BashActivator
```

class virtualenv.activation.activator.**Activator**(*options*)
Generates an activate script for the virtual environment

Create a new activator generator.

Parameters **options** – the parsed options as defined within *add_parser_arguments()*

classmethod supports(*interpreter*)
Check if the activation script is supported in the given interpreter.

Parameters **interpreter** – the interpreter we need to support

Returns `True` if supported, `False` otherwise

classmethod add_parser_arguments(*parser*, *interpreter*)
Add CLI arguments for this activation script.

Parameters

- **parser** – the CLI parser
- **interpreter** – the interpreter this virtual environment is based of

abstract generate(*creator*)
Generate the activate script for the given creator.

Parameters **creator** – the creator (based of *virtualenv.create.creator.Creator*) we used to create this virtual environment

1.5 Development

1.5.1 Getting started

`virtualenv` is a volunteer maintained open source project and we welcome contributions of all forms. The sections below will help you get started with development, testing, and documentation. We're pleased that you are interested in working on virtualenv. This document is meant to get you setup to work on virtualenv and to act as a guide and reference to the development setup. If you face any issues during this process, please open an issue about it on the issue tracker.

Setup

virtualenv is a command line application written in Python. To work on it, you'll need:

- **Source code: available on GitHub. You can use `git` to clone the** repository:

```
git clone https://github.com/pypa/virtualenv
cd virtualenv
```

- **Python interpreter**: We recommend using `CPython`. You can use this guide to set it up.
- tox: to automatically get the projects development dependencies and run the test suite. We recommend installing it using pipx.

Running from source tree

The easiest way to do this is to generate the development tox environment, and then invoke virtualenv from under the `.tox/dev` folder

```
tox -e dev
.tox/dev/bin/virtualenv  # on Linux
.tox/dev/Scripts/virtualenv  # on Windows
```

Running tests

virtualenv's tests are written using the pytest test framework. tox is used to automate the setup and execution of virtualenv's tests.

To run tests locally execute:

```
tox -e py
```

This will run the test suite for the same Python version as under which `tox` is installed. Alternatively you can specify a specific version of python by using the `pyNN` format, such as: `py38`, `pypy3`, etc.

`tox` has been configured to forward any additional arguments it is given to `pytest`. This enables the use of pytest's rich CLI. As an example, you can select tests using the various ways that pytest provides:

```
# Using markers
tox -e py -- -m "not slow"
# Using keywords
tox -e py -- -k "test_extra"
```

Some tests require additional dependencies to be run, such is the various shell activators (`bash`, `fish`, `powershell`, etc). These tests will automatically be skipped if these are not present, note however that in CI all tests are run; so even if all tests succeed locally for you, they may still fail in the CI.

Running linters

virtualenv uses pre-commit for managing linting of the codebase. `pre-commit` performs various checks on all files in virtualenv and uses tools that help follow a consistent code style within the codebase. To use linters locally, run:

```
tox -e fix_lint
```

Note: Avoid using `# noqa` comments to suppress linter warnings - wherever possible, warnings should be fixed instead. `# noqa` comments are reserved for rare cases where the recommended style causes severe readability problems.

Building documentation

virtualenv's documentation is built using Sphinx. The documentation is written in reStructuredText. To build it locally, run:

```
tox -e docs
```

The built documentation can be found in the `.tox/docs_out` folder and may be viewed by opening `index.html` within that folder.

Release

virtualenv's release schedule is tied to `pip`, `setuptools` and `wheel`. We bundle the latest version of these libraries so each time there's a new version of any of these, there will be a new virtualenv release shortly afterwards (we usually wait just a few days to avoid pulling in any broken releases).

1.5.2 Contributing

Submitting pull requests

Submit pull requests against the `master` branch, providing a good description of what you're doing and why. You must have legal permission to distribute any code you contribute to virtualenv and it must be available under the MIT License. Provide tests that cover your changes and run the tests locally first. virtualenv *supports* multiple Python versions and operating systems. Any pull request must consider and work on all these platforms.

Pull Requests should be small to facilitate review. Keep them self-contained, and limited in scope. Studies have shown that review quality falls off as patch size grows. Sometimes this will result in many small PRs to land a single large feature. In particular, pull requests must not be treated as "feature branches", with ongoing development work happening within the PR. Instead, the feature should be broken up into smaller, independent parts which can be reviewed and merged individually.

Additionally, avoid including "cosmetic" changes to code that is unrelated to your change, as these make reviewing the PR more difficult. Examples include re-flowing text in comments or documentation, or addition or removal of blank lines or whitespace within lines. Such changes can be made separately, as a "formatting cleanup" PR, if needed.

Automated testing

All pull requests and merges to 'master' branch are tested using Azure Pipelines (configured by `azure-pipelines.yml` file at the root of the repository). You can find the status and results to the CI runs for your PR on GitHub's Web UI for the pull request. You can also find links to the CI services' pages for the specific builds in the form of "Details" links, in case the CI run fails and you wish to view the output.

To trigger CI to run again for a pull request, you can close and open the pull request or submit another change to the pull request. If needed, project maintainers can manually trigger a restart of a job/build.

NEWS entries

The `changelog.rst` file is managed using towncrier and all non trivial changes must be accompanied by a news entry. To add an entry to the news file, first you need to have created an issue describing the change you want to make. A Pull Request itself *may* function as such, but it is preferred to have a dedicated issue (for example, in case the PR ends up rejected due to code quality reasons).

Once you have an issue or pull request, you take the number and you create a file inside of the `docs/changelog` directory named after that issue number with an extension of:

- `feature.rst`,
- `bugfix.rst`,
- `doc.rst`,
- `removal.rst`,
- `misc.rst`.

Thus if your issue or PR number is `1234` and this change is fixing a bug, then you would create a file `docs/changelog/1234.bugfix.rst`. PRs can span multiple categories by creating multiple files (for instance, if you added a feature and deprecated/removed the old feature at the same time, you would create `docs/changelog/1234.bugfix.rst` and `docs/changelog/1234.remove.rst`). Likewise if a PR touches multiple issues/PRs you may create a file for each of them with the same contents and towncrier will deduplicate them.

Contents of a NEWS entry

The contents of this file are reStructuredText formatted text that will be used as the content of the news file entry. You do not need to reference the issue or PR numbers here as towncrier will automatically add a reference to all of the affected issues when rendering the news file.

In order to maintain a consistent style in the `changelog.rst` file, it is preferred to keep the news entry to the point, in sentence case, shorter than 120 characters and in an imperative tone – an entry should complete the sentence `This change will ...`. In rare cases, where one line is not enough, use a summary line in an imperative tone followed by a blank line separating it from a description of the feature/change in one or more paragraphs, each wrapped at 120 characters. Remember that a news entry is meant for end users and should only contain details relevant to an end user.

Choosing the type of NEWS entry

A trivial change is anything that does not warrant an entry in the news file. Some examples are: code refactors that don't change anything as far as the public is concerned, typo fixes, white space modification, etc. To mark a PR as trivial a contributor simply needs to add a randomly named, empty file to the `news/` directory with the extension of `.trivial`.

Becoming a maintainer

If you want to become an official maintainer, start by helping out. As a first step, we welcome you to triage issues on virtualenv's issue tracker. virtualenv maintainers provide triage abilities to contributors once they have been around for some time and contributed positively to the project. This is optional and highly recommended for becoming a virtualenv maintainer. Later, when you think you're ready, get in touch with one of the maintainers and they will initiate a vote among the existing maintainers.

Note: Upon becoming a maintainer, a person should be given access to various virtualenv-related tooling across multiple platforms. These are noted here for future reference by the maintainers:

- GitHub Push Access
- PyPI Publishing Access
- CI Administration capabilities
- ReadTheDocs Administration capabilities

1.6 Release History

1.6.1 v20.0.4 (2020-02-14)

Features - 20.0.4

- When aliasing interpreters, use relative symlinks - by @asottile. (#1596)

Bugfixes - 20.0.4

- Allow the use of `/` as pathname component separator on Windows - by `vphilippon` (#1582)
- Lower minimal version of six required to 1.9 - by `ssbarnea` (#1606)

1.6.2 v20.0.3 (2020-02-12)

Bugfixes - 20.0.3

- On Python 2 with Apple Framework builds the global site package is no longer added when the *`system-site-packages`* is not specified - by @gaborbernat. (#1561)
- Fix system python discovery mechanism when prefixes contain relative parts (e.g. `..`) by resolving paths within the python information query - by @gaborbernat. (#1583)
- Expose a programmatic API as `from virtualenv import cli_run` - by @gaborbernat. (#1585)

- Fix `app-data` *`seeder`* injects a extra `.dist-info.virtualenv` path that breaks `importlib.metadata`, now we inject an extra `.virtualenv` - by @gaborbernat. (#1589)

Improved Documentation - 20.0.3

- Document a programmatic API as `from virtualenv import cli_run` under *Programmatic API* - by @gaborbernat. (#1585)

1.6.3 v20.0.2 (2020-02-11)

Features - 20.0.2

- Print out a one line message about the created virtual environment when no *`verbose`* is set, this can now be silenced to get back the original behaviour via the *`quiet`* flag - by @pradyunsg. (#1557)
- Allow virtualenv's app data cache to be overridden by `VIRTUALENV_OVERRIDE_APP_DATA` - by @asottile. (#1559)
- Passing in the virtual environment name/path is now required (no longer defaults to `venv`) - by @gaborbernat. (#1568)
- Add a CLI flag *`with-traceback`* that allows displaying the stacktrace of the virtualenv when a failure occurs - by @gaborbernat. (#1572)

Bugfixes - 20.0.2

- Support long path names for generated virtual environment console entry points (such as `pip`) when using the `app-data` *`seeder`* - by @gaborbernat. (#997)
- Improve python discovery mechanism:
  - do not fail if there are executables that fail to query (e.g. for not having execute access to it) on the `PATH`,
  - beside the prefix folder also try with the platform dependent binary folder within that,

  by @gaborbernat. (#1545)
- When copying (either files or trees) do not copy the permission bits, last access time, last modification time, and flags as access to these might be forbidden (for example in case of the macOs Framework Python) and these are not needed for the user to use the virtual environment - by @gaborbernat. (#1561)
- While discovering a python executables interpreters that cannot be queried are now displayed with info level rather than warning, so now they're no longer shown by default (these can be just executables to which we don't have access or that are broken, don't warn if it's not the target Python we want) - by @gaborbernat. (#1574)
- The `app-data` *`seeder`* no longer symlinks the packages on UNIX and copies on Windows. Instead by default always copies, however now has the *`symlink-app-data`* flag allowing users to request this less robust but faster method - by @gaborbernat. (#1575)

Improved Documentation - 20.0.2

- Add link to the legacy documentation for the changelog by @jezdez. (#1547)
- Fine tune the documentation layout: default width of theme, allow tables to wrap around, soft corners for code snippets - by @pradyunsg. (#1548)

1.6.4 v20.0.1 (2020-02-10)

Features - 20.0.1

- upgrade embedded setuptools to `45.2.0` from `45.1.0` for Python `3.4+` - by @gaborbernat. (#1554)

Bugfixes - 20.0.1

- Virtual environments created via relative path on Windows creates bad console executables - by @gaborbernat. (#1552)
- Seems sometimes venvs created set their base executable to themselves; we accept these without question, so we handle virtual environments as system pythons causing issues - by @gaborbernat. (#1553)

1.6.5 v20.0.0. (2020-02-10)

Improved Documentation - 20.0.0.

- Fixes typos, repeated words and inconsistent heading spacing. Rephrase parts of the development documentation and CLI documentation. Expands shorthands like `env var` and `config` to their full forms. Uses descriptions from respective documentation, for projects listed in `related links` - by @pradyunsg. (#1540)

1.6.6 v20.0.0b2 (2020-02-04)

Features - 20.0.0b2

- Improve base executable discovery mechanism:
  - print at debug level why we refuse some candidates,
  - when no candidates match exactly, instead of hard failing fallback to the closest match where the priority of matching attributes is: python implementation, major version, minor version, architecture, patch version, release level and serial (this is to facilitate things to still work when the OS upgrade replace/upgrades the system python with a never version, than what the virtualenv host python was created with),
  - always resolve system_executable information during the interpreter discovery, and the discovered environment is the system interpreter instead of the venv/virtualenv (this happened before lazily the first time we accessed, and caused reporting that the created virtual environment is of type of the virtualenv host python version, instead of the system pythons version - these two can differ if the OS upgraded the system python underneath and the virtualenv host was created via copy),

  by @gaborbernat. (#1515)
- Generate `bash` and `fish` activators on Windows too (as these can be available with git bash, cygwin or mysys2) - by @gaborbernat. (#1527)
- Upgrade the bundled `wheel` package from `0.34.0` to `0.34.2` - by @gaborbernat. (#1531)

Bugfixes - 20.0.0b2

- Bash activation script should have no extensions instead of `.sh` (this fixes the virtualenvwrapper integration) - by @gaborbernat. (#1508)
- Show less information when we run with a single verbosity (`-v`):
  - no longer shows accepted interpreters information (as the last proposed one is always the accepted one),
  - do not display the `str_spec` attribute for `PythonSpec` as these can be deduced from the other attributes,
  - for the `app-data` seeder do not show the type of lock, only the path to the app data directory,

  By @gaborbernat. (#1510)
- Fixed cannot discover a python interpreter that has already been discovered under a different path (such is the case when we have multiple symlinks to the same interpreter) - by @gaborbernat. (#1512)
- Support relative paths for `-p` - by @gaborbernat. (#1514)
- Creating virtual environments in parallel fail with cannot acquire lock within app data - by @gaborbernat. (#1516)
- pth files were not processed under Debian CPython2 interpreters - by @gaborbernat. (#1517)
- Fix prompt not displayed correctly with upcoming fish 3.10 due to us not preserving `$pipestatus` - by @krobelus. (#1530)
- Stable order within `pyenv.cfg` and add `include-system-site-packages` only for creators that reference a global Python - by user:*gaborbernat*. (#1535)

Improved Documentation - 20.0.0b2

- Create the first iteration of the new documentation - by @gaborbernat. (#1465)
- Project readme is now of type MarkDown instead of reStructuredText - by @gaborbernat. (#1531)

1.6.7 v20.0.0b1 (2020-01-28)

- First public release of the rewrite. Everything is brand new and just added.

Warning: The current virtualenv is the second iteration of implementation. From version `0.8` all the way to `16.7.9` we numbered the first iteration. Version `20.0.0b1` is a complete rewrite of the package, and as such this release history starts from there. The old changelog is still available in the legacy branch documentation.

PYTHON MODULE INDEX

V

virtualenv, 7

INDEX

A

Activator (*class in virtualenv.activation.activator*), 13
activators() (*virtualenv.session.Session property*), 8
add_parser_arguments() (*virtualenv.activation.activator.Activator class method*), 13
add_parser_arguments() (*virtualenv.create.creator.Creator class method*), 12
add_parser_arguments() (*virtualenv.discovery.discover.Discover class method*), 11
add_parser_arguments() (*virtualenv.seed.seeder.Seeder class method*), 12

C

can_create() (*virtualenv.create.creator.Creator class method*), 12
cli_run() (*in module virtualenv*), 7
create() (*virtualenv.create.creator.Creator method*), 12
Creator (*class in virtualenv.create.creator*), 12
creator() (*virtualenv.session.Session property*), 8

D

Discover (*class in virtualenv.discovery.discover*), 11

G

generate() (*virtualenv.activation.activator.Activator method*), 13

I

interpreter() (*virtualenv.discovery.discover.Discover property*), 11
interpreter() (*virtualenv.session.Session property*), 8

R

run() (*virtualenv.discovery.discover.Discover method*), 11
run() (*virtualenv.seed.seeder.Seeder method*), 13

S

Seeder (*class in virtualenv.seed.seeder*), 12
seeder() (*virtualenv.session.Session property*), 8
Session (*class in virtualenv.session*), 8
supports() (*virtualenv.activation.activator.Activator class method*), 13

V

verbosity() (*virtualenv.session.Session property*), 8
virtualenv (*module*), 7