vpip

eight04

Apr 23, 2022

CONTENTS

`vpip` is a CLI which aims to provide an `npm`-like experience when installing Python packages.

To achieve package isolation, it integrates `pip` with Python 3's `venv` module to install packages to isolated virtual environments.

It also allows you to easily create a virtual environment in the local folder to develop a new package. You can install dependencies to the local venv and `vpip` will update `requirements.txt` and `setup.cfg` automatically.

CHAPTER

ONE

COMMAND REFERENCE

1.1 List of commands

1.1.1 install

```
vpip install [-g | -D] [PACKAGE [PACKAGE ...]]
```

Install packages and save to the dependency.

By default, the package would be installed to the local `.venv` folder, and the package name would be add to the `install_requires` option in `setup.cfg`.

The version range of the development dependency is pinned and the version range of the production dependency is specified in a compatible range.

When `PACKAGE` is not specified. The tool will install all dependencies to the local venv. It executes two commands:

```
pip install -r requirements-lock.txt
pip install -e .
```

`PACKAGE` can also be a URL but it will only work with `-g` flag.

Options:

- `-g, --global` - Install packages to a new venv in `~/vpip/pkg_venvs`. Executables would be linked to the Python Scripts folder so you can still access them from the command line.
- `-D, --save-dev` - Save the package to the development dependency.

1.1.2 uninstall

```
vpip uninstall [-g] PACKAGE [PACKAGE ...]
```

Uninstall packages and remove them from the dependency.

Options:

- `-g, --global` - Uninstall global packages. This would remove the venv from `~/vpip/pkg_venvs` directly, so it actually doesn't use the `pip` command.

1.1.3 update

```
vpip update [-g] [--latest] [PACKAGE [PACKAGE ...]]
```

Update packages and save to the dependency.

By default, this command only chooses a release from the compatible version range. For example: if the current version is `0.4.5`, the compatible range is `>=0.4.5,<0.5`; if the current version is `1.8.7`, the compatible range is `>=1.8.7,<2`.

If no package is specified, the command update all local packages in the dependencies.

Options:

- `-g, --global` - Update global packages.
- `--latest` - Update to the latest version instead of the compatible version.

1.1.4 list

```
vpip list [-g] [--outdated]
```

List packages in the dependencies. Only dependencies are listed so the result is different from `vpip run pip list`.

Options:

- `-g, --global` - List globally installed packages.
- `--outdated` - Check update from pypi.org and list only outdated packages.

1.1.5 outdated

```
vpip outdated [-g]
```

List outdated packages. This command is just a shortcut of `vpip list --outdated`.

1.1.6 run

```
vpip run ...
```

Execute shell command in the local venv. If the command has an option that is conflicted with the CLI, you can insert a `--` to separate the actuall command. For example:

```
# this would disply the help message of pylint instead of vpip
vpip run -- pylint -h
```

1.1.7 link

```
vpip link
```

Link console scripts installed in the local venv to the global Scripts folder so they can be invoked without activating the venv. See *vpip.venv.get_global_script_folders()*

This command checks the `console_scripts` in `setup.cfg` to decide which commands should be linked.

1.1.8 update_venv

```
vpip update_venv [-g [PACKAGE ...]]
```

Update/rebuild the venv folder. It compares the Python version inside the venv with the Python outside of the venv. If they are incompatible then rebuild the folder. Otherwise, this command upgrades `pip`, `wheel`, etc, inside the venv. (See also *PREINSTALLED_PACKAGES*.)

Options:

- `-g, --global` - Update global packages.

1.2 Extend commands

`vpip` allows you to define your own commands. In the `setup.cfg` file, add:

```
[vpip.commands]
# name = command
test = python setup.py test
build = make something
```

After adding these commands, you can invoke them with `vpip test` and `vpip build`. These commands would be run inside the venv. Extra arguments would be appended to the command.

1.3 Command fallback

Another way to extend `vpip` CLI is to define a command fallback. In the `setup.cfg` file, add:

```
[vpip]
command_fallback = python setup.py
```

This is a better solution if you are using a task runner (e.g. pyxcute) and don't want to write down all commands in `setup.cfg`.

CHAPTER

TWO

API REFERENCE

2.1 vpip.cli

This module includes the main CLI entry point.

vpip.cli.**add_arguments**(*parser*, *options*)
Add JSON-formatted argument list to parser.

Parameters

- **parser** (*argparse.ArgumentParser*) – The parser, or anything that implements `add_argument` method.
- **options** (*list*) – List of options. See the source code in vpip.commands for example.

vpip.cli.**cli**(*args=None*)
CLI entry point.

Parameters **args** (*list*) – Argument list. Use `sys.argv[1:]` if None.

You can invoke this function like:

```
from vpip.cli import cli
cli(["install", "-g", "my-package"])
```

vpip.cli.**patch_argparse**()
Patch `argparse`. Make `ArgumentParser.add_subparsers()` accept a new keyword argument `fallback`, which is a function receving a `args` list and returning a new `args` or None.

This function can be called multiple times, but only the first call will patch the module.

2.2 vpip.commands

This subpackage containing all builtin commands.

2.2.1 Submodules

vpip.commands.install

install command.

vpip.commands.install.**get_pkg_from_url**(*url*)

vpip.commands.install.**install_global**(*packages*, *upgrade=False*, *latest=False*)
Install global packages.

Parameters

- **packages** (*list[str]*) – List of package name.
- **upgrade** (*bool*) – Upgrade package. By default, the function skipped already installed packages.
- **latest** (*bool*) – Upgrade to the latest version. By default, only compatible versions are selected.

vpip.commands.install.**install_global_url**(*url*)

vpip.commands.install.**install_local**(*packages*, *dev=False*, ***kwargs*)
Install local packages and save to dependency.

Parameters

- **packages** (*list[str]*) – List of package name.
- **dev** (*bool*) – If true then save to development depedency. Otherwise, save to production dependency.
- **kwargs** (*dict*) – Other arguments are sent to *vpip.pip_api.install()*.

vpip.commands.install.**install_local_first_time**()
Create the venv and install all dependencies.

If the lock file exists, execute `pip install -r requirements-lock.txt`.

Otherwise `pip install -e . && pip install -r requirements.txt`.

vpip.commands.install.**run**(*ns*)

vpip.commands.link

link command.

class vpip.commands.link.**Linker**(*src*)
Bases: object

make(*dest*)

unlink(*dest*)

class vpip.commands.link.**WinLinker**(*src*)
Bases: *vpip.commands.link.Linker*

make(*dest*)

vpip.commands.link.**get_current_pkg**()

vpip.commands.link.**link_console_script**(*pkg*)
Find console scripts of the package and try to link the executable to the global scripts folder.

Parameters **pkg** (*str*) – Package name.

vpip.commands.link.**run**(*ns*)

vpip.commands.list

List installed packages.

class vpip.commands.list.**PackageInfo**(*name*, *version=None*)

Bases: object

Package information formatter.

Usage:

```
print(PackageInfo("my_package", "0.1.0"))
```

Parameters

- **name** (*str*) – Package name.
- **version** (*str*) – Package version. If none then it shows `(not installed)` when formatted.

check_update()

Check update and setup *update_result*.

name

update_result

Update result. This would be set to the return value of *vpip.pypi.check_update()*.

version

vpip.commands.list.**iter_global_packages**()

Iterate through globally installed packages.

Return type Iterator[*PackageInfo*]

vpip.commands.list.**print_global_packages**(*check_outdated=False*)

vpip.commands.list.**print_local_packages**(*check_outdated=False*)

vpip.commands.list.**run**(*ns*)

vpip.commands.outdated

This command is just a shortcut of `vpip list --outdated`.

vpip.commands.outdated.**run**(*ns*)

vpip.commands.run

run command.

vpip.commands.run.**get_shell_executable**()

vpip.commands.run.**run**(*ns*, *extra*)

vpip.commands.uninstall

uninstall command.

vpip.commands.uninstall.**clean_unused**()
Remove unused packages

vpip.commands.uninstall.**get_top_packages**(*packages: List[str]*) → List[str]
Return top-level packages

vpip.commands.uninstall.**run**(*ns*)

vpip.commands.uninstall.**uninstall_global**(*packages*)
Uninstall global packages.

Parameters **packages** (*list[str]*) – Package names.

vpip.commands.uninstall.**uninstall_local**(*packages*)
Uninstall packages and remove from dependencies.

Parameters **packages** (*list[str]*) – Package names.

vpip.commands.update

update command.

vpip.commands.update.**run**(*ns*)

vpip.commands.update.**update_local**(*packages: List[str]*, *latest: bool = False*)
Update local packages.

Parameters

- **packages** – A list of packages that shoud be updated.
- **latest** – Whether to upgrade to the latest version.

vpip.commands.update_venv

update_venv command.

vpip.commands.update_venv.**run**(*ns*)

vpip.commands.update_venv.**update_venv**(*vv*, *global_pkg_name=None*)
Update a venv.

Parameters

- **vv** (vpip.venv.Venv) – A venv instance.
- **global_pkg_name** (*str*) – Decide how to rebuild the venv. If set then run *vpip.commands.install.install_global()* when rebuilding venv. Otherwise, run *vpip.commands.install.install_local_first_time()*

If the Python version is upgraded, this command reinstall the entire venv.

2.2.2 Module contents

vpip.commands.**get_modules**()
Get module instances.

Returns A `command_name -> module` map.

Return type dict[str, module]

vpip.commands.**names = ['uninstall', 'outdated', 'update', 'link', 'install', 'update_venv', 'list', 'run']**
List of builtin command names.

2.3 vpip.dependency

Update dependency files.

class vpip.dependency.**DevUpdater**
Bases: *vpip.dependency.Updater*

Development dependency (requirements.txt) updater.

class vpip.dependency.**ProdUpdater**
Bases: *vpip.dependency.Updater*

Production dependency (setup.cfg) updater.

class vpip.dependency.**Updater**
Bases: object

Dependency updater interface. Extend this class to create a new updater.

get_requirements()
Get requirements string.

Return type str

get_spec(*name*, *version*)
Get version specifier.

Parameters

- **name** (*str*) – Installed package name.
- **version** (*str*) – Installed pacakge version.

Returns Version specifier e.g. `"foo==0.1.0"`

Return type str

write_requirements(*lines*)
Write new requirements to file.

Parameters **line** (*list[str]*) – Lines of requirements.

vpip.dependency.**has_lock**()
Detect if there is a lock file (requirements-lock.txt)

vpip.dependency.**parse_requirements**(*text*) → Iterator[packaging.requirements.Requirement]
Parse requirements text.

FIXME: switch to an external function from pip if possible. https://pip.pypa.io/en/stable/reference/requirements-file-format/#requirements-file-format

vpip.dependency.**update_dependency**(*updater*, *added=None*, *removed=None*)
Update dependency and save.

Parameters

- **updater** (Updater) – An Updater instance.
- **added** (*dict*) – A `pkg_name -> version` map. Added packages.
- **removed** (*list[str]*) – A list of package name. Removed packages.

vpip.dependency.**update_lock**()
Run `pip freeze` and update the lock file

2.4 vpip.execute

A utility to spawn subprocess and capture the output.

vpip.execute.**execute**(*cmd*, *capture=False*)
Execute a command.

Parameters

- **cmd** (*list or str*) – Command. If `cmd` is a `str`, the command would be invoked with shell.
- **capture** (*bool*) – If `True` then enter the capture mode: process output will be captured and the function will return a generator yielding lines of the output.

Return type Iterator[str] or None

2.5 vpip.pip_api

`pip` command API.

vpip.pip_api.**create_ns_from_dict**(*d*)
Create a namespace object from a dict.

Parameters **d** (*dict*) – Dictionary.

Return type argparse.Namespace

vpip.pip_api.**execute_pip**(*cmd*, *capture=False*)
Run pip command.

Parameters

- **cmd** (*str*) – `pip` command. It would be prefixed with `python -m pip`.
- **capture** (*bool*) – Whether to capture output.

vpip.pip_api.**freeze**(*include: Optional[Container[str]] = None*, *exclude: Optional[Container[str]] = None*) → List[str]
List installed packages in `pip freeze` format (`my_pkg==1.2.3`).

Parameters

- **include** – If defined, only returns specified packages.
- **exclude** – If defined, exclude specified packages.

vpip.pip_api.**get_compatible_version**(*version*)
Return the compatible version.

Parameters **version** (*str*) – Version string.

Returns The compatible version which could be used as ~={compatible_version}.

Return type str

Suppose the version string is x.y.z:

- If x is zero then return x.y.z.
- Otherwise return x.y.

vpip.pip_api.**install**(*packages: List[str]*, *install_scripts: Optional[str] = None*, *upgrade: bool = False*, *latest: bool = False*, *deps: bool = True*) → List[str]
Install packages and return a list of collected package names.

Parameters

- **packages** – A list of package name, which may include the version specifier. It can also be a URL.
- **install_scripts** – Install scripts to a different folder. It uses the --install-option="--install-scripts=..." pip option.
- **upgrade** – Upgrade package.
- **latest** – Whether upgrade to the latest version. Otherwise upgrade to the compatible version. This option has no effect if package includes specifiers.
- **deps** – Whether to install dependencies.

vpip.pip_api.**install_editable**()
Install the current cwd as editable package.

vpip.pip_api.**install_requirements**(*file='requirements.txt'*)
Install requirements.txt file.

vpip.pip_api.**list_**(*not_required=False*, *format='json'*)
List installed packages.

Return type list[argparse.Namespace]

vpip.pip_api.**show**(*packages*, *verbose=False*)
Get package information.

Parameters

- **packages** (*list[str]*) – A list of package name.
- **verbose** (*bool*) – Whether to return verbose info.

Returns A list of namespace objects holding the package information.

Return type list[Namespace]

This function uses pip show under the hood. Property name is generated by case_conversion.snakecase().

vpip.pip_api.**uninstall**(*packages*)
Uninstall packages.

Parameters **package** (*list[str]*) – Package name.

2.6 vpip.pypi

Check update from pypi.org.

class vpip.pypi.**UpdateResult**(*compatible*, *latest*)
Bases: tuple

Create new instance of UpdateResult(compatible, latest)

property compatible
Alias for field number 0

property latest
Alias for field number 1

vpip.pypi.**check_update**(*pkg*, *curr_version*)
Check update from pypi and return the result if there is an update available.

Parameters

- **pkg** (*str*) – Package name.
- **curr_version** (*str*) – Installed version of the package.

Return type *UpdateResult* or None

UpdateResult has two properies, `compatible` and `latest`, which are two different version string.

If `result.compatible` is not None, it must be compatible with `curr_version` and must lager than `curr_version`.

If `result.latest` is not None, it must larger than `result.compatible` and `curr_version`.

vpip.pypi.**get_session**()
Return a static `requests.Session` object used by *check_update()*, so they can share a persistent connection.

vpip.pypi.**is_compatible**(*version: packaging.version.Version*, *new_version: packaging.version.Version*) → bool
Check if two versions are compatible. `new_version` may be smaller than `version`.

2.7 vpip.venv

Building venvs.

2.7.1 Constants

vpip.venv.**GLOBAL_FOLDER: str**
Absolute path to the global package venv folder `~/.vpip/pkg_venvs`

vpip.venv.**PREINSTALLED_PACKAGES: List[str] = ['pip', 'wheel']**
These packages are pre-installed by vpip. They are excluded from the lock file. You can update them via `update_venv` command.

2.7.2 Functions

vpip.venv.**get_global_folder**(*pkg_name*)
Get global venv folder for a package.

Parameters **pkg_name** (*str*) – Package name.

Return type str

vpip.venv.**iter_global_packages**()
Iterate through all venv folders for globally installed packages.

Return type Iterator[str]

vpip.venv.**get_current_venv**()
Get the *Venv* instance pointing to `./.venv`.

Return type *Venv*

vpip.venv.**get_global_pkg_venv**(*pkg*)
Get the *Venv* instance pointing the venv folder of the global installed package.

Parameters **pkg** (*str*) – Package name. It would also be used as the folder name.

Return type *Venv*

vpip.venv.**get_global_script_folders**() → Iterable[Path]
Return a list of folders. Which are used to write global scripts.

Following paths are checked against env variable `PATH`:

```
'~/.local/bin'
'~/bin'
sysconfig.get_path('scripts')
```

They are only returned if included in `PATH`.

2.7.3 Classes

class vpip.venv.**Builder**(*system_site_packages=False*, *clear=False*, *symlinks=False*, *upgrade=False*, *with_pip=False*, *prompt=None*)
Bases: `venv.EnvBuilder`

An environment builder that could be used inside a venv.

It also upgrades pip to the latest version after installed.

class vpip.venv.**Venv**(*env_dir*)
A helper class that is associated to a venv folder. It allows you to easily activate/deactivate the venv.

Parameters **env_dir** (*str*) – The target venv folder.

activate(*auto_create=False*)
Activate the venv. Update PATH and VIRTUAL_ENV environment variables.

Parameters **auto_create** (*bool*) – If True then automatically create the venv when the folder doesn't exist.

This function can be used as a context manager that will *deactivate()* when exited.

create()
Create the venv.

deactivate()
Deactivate the venv.

destroy()
Destroy the venv. Remove the venv folder.

exists()
Check if the folder exists.

Return type bool

PYTHON MODULE INDEX

V

INDEX

N

P

R

S

U

V

W