
virtualenvwrapper Documentation

Release 4.3.0

Doug Hellmann

Jul 13, 2017

Contents

1	Features	3
2	Introduction	5
3	Details	9
3.1	Installation	9
3.2	Command Reference	13
3.3	Customizing Virtualenvwrapper	22
3.4	Project Management	31
3.5	Tips and Tricks	32
3.6	For Developers	34
3.7	Existing Extensions	36
3.8	Why virtualenvwrapper is (Mostly) Not Written In Python	37
3.9	Release History	39
4	References	51
5	Support	53
5.1	Shell Aliases	53
6	License	55

virtualenvwrapper is a set of extensions to Ian Bicking's [virtualenv](#) tool. The extensions include wrappers for creating and deleting virtual environments and otherwise managing your development workflow, making it easier to work on more than one project at a time without introducing conflicts in their dependencies.

CHAPTER 1

Features

1. Organizes all of your virtual environments in one place.
2. Wrappers for managing your virtual environments (create, delete, copy).
3. Use a single command to switch between environments.
4. Tab completion for commands that take a virtual environment as argument.
5. User-configurable hooks for all operations (see *Per-User Customization*).
6. Plugin system for more creating sharable extensions (see *Extending Virtualenvwrapper*).

The best way to explain the features virtualenvwrapper gives you is to show it in use.

First, some initialization steps. Most of this only needs to be done one time. You will want to add the command to source `/usr/local/bin/virtualenvwrapper.sh` to your shell startup file, changing the path to `virtualenvwrapper.sh` depending on where it was installed by pip.

```
$ pip install virtualenvwrapper
...
$ export WORKON_HOME=~/Envs
$ mkdir -p $WORKON_HOME
$ source /usr/local/bin/virtualenvwrapper.sh
$ mkvirtualenv env1
Installing
distribute.....
.....
.....done.
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env1/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env1/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env1/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env1/bin/postactivate
↳New python executable in env1/bin/python
(env1)$ ls $WORKON_HOME
env1 hook.log
```

Now we can install some software into the environment.

```
(env1)$ pip install django
Downloading/unpacking django
  Downloading Django-1.1.1.tar.gz (5.6Mb): 5.6Mb downloaded
  Running setup.py egg_info for package django
Installing collected packages: django
  Running setup.py install for django
    changing mode of build/scripts-2.6/django-admin.py from 644 to 755
```

```
changing mode of /Users/dhellmann/Envs/env1/bin/django-admin.py to 755
Successfully installed django
```

We can see the new package with `lssitepackages`:

```
(env1)$ lssitepackages
Django-1.1.1-py2.6.egg-info      easy-install.pth
distribute-0.6.10-py2.6.egg    pip-0.6.3-py2.6.egg
django                          setuptools.pth
```

Of course we are not limited to a single virtualenv:

```
(env1)$ ls $WORKON_HOME
env1          hook.log
(env1)$ mkvirtualenv env2
Installing distribute.....
.....
.....done.
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env2/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env2/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env2/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env2/bin/postactivate
↳New python executable in env2/bin/python
(env2)$ ls $WORKON_HOME
env1          env2          hook.log
```

Switch between environments with `workon`:

```
(env2)$ workon env1
(env1)$ echo $VIRTUAL_ENV
/Users/dhellmann/Envs/env1
(env1)$
```

The `workon` command also includes tab completion for the environment names, and invokes customization scripts as an environment is activated or deactivated (see *Per-User Customization*).

```
(env1)$ echo 'cd $VIRTUAL_ENV' >> $WORKON_HOME/postactivate
(env1)$ workon env2
(env2)$ pwd
/Users/dhellmann/Envs/env2
```

`postmkvirtualenv` is run when a new environment is created, letting you automatically install commonly-used tools.

```
(env2)$ echo 'pip install sphinx' >> $WORKON_HOME/postmkvirtualenv
(env3)$ mkvirtualenv env3
New python executable in env3/bin/python
Installing distribute.....
.....
.....done.
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env3/bin/predeactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env3/bin/postdeactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env3/bin/preactivate
virtualenvwrapper.user_scripts Creating /Users/dhellmann/Envs/env3/bin/postactivate
Downloading/unpacking sphinx
  Downloading Sphinx-0.6.5.tar.gz (972Kb): 972Kb downloaded
  Running setup.py egg_info for package sphinx
```

```
no previously-included directories found matching 'doc/_build'  
Downloading/unpacking Pygments>=0.8 (from sphinx)  
  Downloading Pygments-1.3.1.tar.gz (1.1Mb): 1.1Mb downloaded  
  Running setup.py egg_info for package Pygments  
Downloading/unpacking Jinja2>=2.1 (from sphinx)  
  Downloading Jinja2-2.4.tar.gz (688Kb): 688Kb downloaded  
  Running setup.py egg_info for package Jinja2  
    warning: no previously-included files matching '*' found under directory 'docs/_  
↳build/doctrees'  
Downloading/unpacking docutils>=0.4 (from sphinx)  
  Downloading docutils-0.6.tar.gz (1.4Mb): 1.4Mb downloaded  
  Running setup.py egg_info for package docutils  
Installing collected packages: docutils, Jinja2, Pygments, sphinx  
  Running setup.py install for docutils  
  Running setup.py install for Jinja2  
  Running setup.py install for Pygments  
  Running setup.py install for sphinx  
    no previously-included directories found matching 'doc/_build'  
    Installing sphinx-build script to /Users/dhellmann/Envs/env3/bin  
    Installing sphinx-quickstart script to /Users/dhellmann/Envs/env3/bin  
    Installing sphinx-autogen script to /Users/dhellmann/Envs/env3/bin  
Successfully installed docutils Jinja2 Pygments sphinx (env3)$  
(venv3)$ which sphinx-build  
/Users/dhellmann/Envs/env3/bin/sphinx-build
```

Through a combination of the existing functions defined by the core package (see *Command Reference*), third-party plugins (see *Extending Virtualenvwrapper*), and user-defined scripts (see *Per-User Customization*) virtualenvwrapper gives you a wide variety of opportunities to automate repetitive operations.

Installation

Supported Shells

virtualenvwrapper is a set of shell *functions* defined in Bourne shell compatible syntax. Its automated tests run under these shells on OS X and Linux:

- bash
- ksh
- zsh

It may work with other shells, so if you find that it does work with a shell not listed here please let me know. If you can modify it to work with another shell without completely rewriting it, then send a pull request through the [bitbucket project page](#). If you write a clone to work with an incompatible shell, let me know and I will link to it from this page.

Windows Command Prompt

David Marble has ported virtualenvwrapper to Windows batch scripts, which can be run under Microsoft Windows Command Prompt. This is also a separately distributed re-implementation. You can download [virtualenvwrapper-win](#) from PyPI.

MSYS

It is possible to use virtualenv wrapper under [MSYS](#) with a native Windows Python installation. In order to make it work, you need to define an extra environment variable named `MSYS_HOME` containing the root path to the MSYS installation.

```
export WORKON_HOME=$HOME/.virtualenvs
export MSYS_HOME=/c/msys/1.0
source /usr/local/bin/virtualenvwrapper.sh
```

or:

```
export WORKON_HOME=$HOME/.virtualenvs
export MSYS_HOME=C:\msys\1.0
source /usr/local/bin/virtualenvwrapper.sh
```

Depending on your MSYS setup, you may need to install the [MSYS mktemp binary](#) in the `MSYS_HOME/bin` folder.

PowerShell

Guillermo López-Anglada has ported virtualenvwrapper to run under Microsoft's PowerShell. We have agreed that since it is not compatible with the rest of the extensions, and is largely a re-implementation (rather than an adaptation), it should be distributed separately. You can download [virtualenvwrapper-powershell](#) from PyPI.

Python Versions

virtualenvwrapper is tested under Python 2.6-3.3.

Basic Installation

virtualenvwrapper should be installed into the same global site-packages area where virtualenv is installed. You may need administrative privileges to do that. The easiest way to install it is using `pip`:

```
$ pip install virtualenvwrapper
```

or:

```
$ sudo pip install virtualenvwrapper
```

Warning: virtualenv lets you create many different Python environments. You should only ever install virtualenv and virtualenvwrapper on your base Python installation (i.e. NOT while a virtualenv is active) so that the same release is shared by all Python environments that depend on it.

An alternative to installing it into the global site-packages is to add it to [your user local directory](#) (usually `~/.local`).

```
$ pip install --install-option="--user" virtualenvwrapper
```

Shell Startup File

Add three lines to your shell startup file (`.bashrc`, `.profile`, etc.) to set the location where the virtual environments should live, the location of your development project directories, and the location of the script installed with this package:

```
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/Devel
source /usr/local/bin/virtualenvwrapper.sh
```

After editing it, reload the startup file (e.g., run `source ~/.bashrc`).

Lazy Loading

An alternative initialization script is provided for loading virtualenvwrapper lazily. Instead of sourcing `virtualenvwrapper.sh` directly, use `virtualenvwrapper_lazy.sh`. If `virtualenvwrapper.sh` is not on your `$PATH`, set `VIRTUALENVWRAPPER_SCRIPT` to point to it.

```
export WORKON_HOME=$HOME/.virtualenvs
export PROJECT_HOME=$HOME/Devel
export VIRTUALENVWRAPPER_SCRIPT=/usr/local/bin/virtualenvwrapper.sh
source /usr/local/bin/virtualenvwrapper_lazy.sh
```

Warning: When the lazy-loading version of the startup script is used, tab-completion of arguments to virtualenvwrapper commands (such as environment names) is not enabled until after the first command has been run. For example, tab completion of environments does not work for the first instance of *workon*.

Quick-Start

1. Run: `workon`
2. A list of environments, empty, is printed.
3. Run: `mkvirtualenv temp`
4. A new environment, `temp` is created and activated.
5. Run: `workon`
6. This time, the `temp` environment is included.

Configuration

virtualenvwrapper can be customized by changing environment variables. Set the variables in your shell startup file *before* loading `virtualenvwrapper.sh`.

Location of Environments

The variable `WORKON_HOME` tells virtualenvwrapper where to place your virtual environments. The default is `$HOME/.virtualenvs`. If the directory does not exist when virtualenvwrapper is loaded, it will be created automatically.

Location of Project Directories

The variable `PROJECT_HOME` tells virtualenvwrapper where to place your project working directories. The variable must be set and the directory created before *mkproject* is used.

See also:

- *Project Management*

Project Linkage Filename

The variable `VIRTUALENVWRAPPER_PROJECT_FILENAME` tells `virtualenvwrapper` how to name the file linking a `virtualenv` to a project working directory. The default is `.project`.

See also:

- *Project Management*

Location of Hook Scripts

The variable `VIRTUALENVWRAPPER_HOOK_DIR` tells `virtualenvwrapper` where the *user-defined hooks* should be placed. The default is `$WORKON_HOME`.

See also:

- *Per-User Customization*

Location of Hook Logs

The variable `VIRTUALENVWRAPPER_LOG_FILE` tells `virtualenvwrapper` where the logs for the hook loader should be written. The default is to not log from the hooks.

Python Interpreter, `virtualenv`, and `$PATH`

During startup, `virtualenvwrapper.sh` finds the first `python` and `virtualenv` programs on the `$PATH` and remembers them to use later. This eliminates any conflict as the `$PATH` changes, enabling interpreters inside virtual environments where `virtualenvwrapper` is not installed or where different versions of `virtualenv` are installed. Because of this behavior, it is important for the `$PATH` to be set **before** sourcing `virtualenvwrapper.sh`. For example:

```
export PATH=/usr/local/bin:$PATH
source /usr/local/bin/virtualenvwrapper.sh
```

To override the `$PATH` search, set the variable `VIRTUALENVWRAPPER_PYTHON` to the full path of the interpreter to use and `VIRTUALENVWRAPPER_VIRTUALENV` to the full path of the `virtualenv` binary to use. Both variables *must* be set before sourcing `virtualenvwrapper.sh`. For example:

```
export VIRTUALENVWRAPPER_PYTHON=/usr/local/bin/python
export VIRTUALENVWRAPPER_VIRTUALENV=/usr/local/bin/virtualenv
source /usr/local/bin/virtualenvwrapper.sh
```

Default Arguments for `virtualenv`

If the application identified by `VIRTUALENVWRAPPER_VIRTUALENV` needs arguments, they can be set in `VIRTUALENVWRAPPER_VIRTUALENV_ARGS`. The same variable can be used to set default arguments to be passed to `virtualenv`. For example, set the value to `--no-site-packages` to ensure that all new environments are isolated from the system `site-packages` directory.

```
export VIRTUALENVWRAPPER_VIRTUALENV_ARGS='--no-site-packages'
```


Temporary Files

virtualenvwrapper creates temporary files in `$TMPDIR`. If the variable is not set, it uses `/tmp`. To change the location of temporary files just for virtualenvwrapper, set `VIRTUALENVWRAPPER_TMPDIR`.

Site-wide Configuration

Most UNIX systems include the ability to change the configuration for all users. This typically takes one of two forms: editing the *skeleton* files for new accounts or editing the global startup file for a shell.

Editing the skeleton files for new accounts means that each new user will have their private startup files preconfigured to load virtualenvwrapper. They can disable it by commenting out or removing those lines. Refer to the documentation for the shell and operating system to identify the appropriate file to edit.

Modifying the global startup file for a given shell means that all users of that shell will have virtualenvwrapper enabled, and they cannot disable it. Refer to the documentation for the shell to identify the appropriate file to edit.

Upgrading to 2.9

Version 2.9 includes the features previously delivered separately by `virtualenvwrapper.project`. If you have an older version of the project extensions installed, remove them before upgrading.

Upgrading from 1.x

The shell script containing the wrapper functions has been renamed in the 2.x series to reflect the fact that shells other than bash are supported. In your startup file, change `source /usr/local/bin/virtualenvwrapper_bashrc` to `source /usr/local/bin/virtualenvwrapper.sh`.

Command Reference

All of the commands below are to be used on the Terminal command line.

Managing Environments

mkvirtualenv

Create a new environment, in the `WORKON_HOME`.

Syntax:

```
mkvirtualenv [-a project_path] [-i package] [-r requirements_file] [virtualenv_
↳options] ENVNAME
```

All command line options except `-a`, `-i`, `-r`, and `-h` are passed directly to `virtualenv`. The new environment is automatically activated after being initialized.

```
$ workon
$ mkvirtualenv mynewenv
New python executable in mynewenv/bin/python
Installing distribute.....
.....
```

```
.....  
done.  
(mynewenv)$ workon  
mynewenv  
(mynewenv)$
```

The `-a` option can be used to associate an existing project directory with the new environment.

The `-i` option can be used to install one or more packages (by repeating the option) after the environment is created.

The `-r` option can be used to specify a text file listing packages to be installed. The argument value is passed to `pip -r` to be installed.

See also:

- [premkvirtualenv](#)
- [postmkvirtualenv](#)
- requirements file format

mktmpenv

Create a new virtualenv in the `WORKON_HOME` directory.

Syntax:

```
mktmpenv [VIRTUALENV_OPTIONS]
```

A unique virtualenv name is generated.

```
$ mktmpenv  
Using real prefix '/Library/Frameworks/Python.framework/Versions/2.7'  
New python executable in 1e513ac6-616e-4d56-9aa5-9d0a3b305e20/bin/python  
Overwriting 1e513ac6-616e-4d56-9aa5-9d0a3b305e20/lib/python2.7/distutils/__init__.py  
with new content  
Installing distribute.....  
.....done.  
This is a temporary environment. It will be deleted when deactivated.  
(1e513ac6-616e-4d56-9aa5-9d0a3b305e20) $
```

lsvirtualenv

List all of the environments.

Syntax:

```
lsvirtualenv [-b] [-l] [-h]
```

- | | |
|-----------|--|
| -b | Brief mode, disables verbose output. |
| -l | Long mode, enables verbose output. Default. |
| -h | Print the help for <code>lsvirtualenv</code> . |

See also:

- [get_env_details](#)

showvirtualenv

Show the details for a single virtualenv.

Syntax:

```
showvirtualenv [env]
```

See also:

- [get_env_details](#)

rmvirtualenv

Remove an environment, in the WORKON_HOME.

Syntax:

```
rmvirtualenv ENVNAME
```

You must use *deactivate* before removing the current environment.

```
(mynewenv)$ deactivate
$ rmvirtualenv mynewenv
$ workon
$
```

See also:

- [prermvirtualenv](#)
- [postrmvirtualenv](#)

cpvirtualenv

Duplicate an existing virtualenv environment. The source can be an environment managed by virtualenvwrapper or an external environment created elsewhere.

Warning: Copying virtual environments is not well supported. Each virtualenv has path information hard-coded into it, and there may be cases where the copy code does not know to update a particular file. **Use with caution.**

Syntax:

```
cpvirtualenv ENVNAME [TARGETENVNAME]
```

Note: Target environment name is required for WORKON_HOME duplications. However, target environment name can be omitted for importing external environments. If omitted, the new environment is given the same name as the original.

```
$ workon
$ mkvirtualenv source
New python executable in source/bin/python
Installing distribute.....
```

```
.....
done.
(source)$ cpvirtualenv source dest
Making script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/easy_install_
↳relative
Making script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/easy_install-2.6_
↳relative
Making script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/pip relative
Script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/postactivate cannot be_
↳made relative (it's not a normal script that starts with #!/Users/dhellmann/Devel/
↳virtualenvwrapper/tmp/dest/bin/python)
Script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/postdeactivate cannot be_
↳made relative (it's not a normal script that starts with #!/Users/dhellmann/Devel/
↳virtualenvwrapper/tmp/dest/bin/python)
Script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/preactivate cannot be_
↳made relative (it's not a normal script that starts with #!/Users/dhellmann/Devel/
↳virtualenvwrapper/tmp/dest/bin/python)
Script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/predeactivate cannot be_
↳made relative (it's not a normal script that starts with #!/Users/dhellmann/Devel/
↳virtualenvwrapper/tmp/dest/bin/python)
(dest)$ workon
dest
source
(dest)$
```

See also:

- [precpvirtualenv](#)
- [postcpvirtualenv](#)
- [premkvirtualenv](#)
- [postmkvirtualenv](#)

allvirtualenv

Run a command in all virtualenvs under WORKON_HOME.

Syntax:

```
allvirtualenv command with arguments
```

Each virtualenv is activated, bypassing activation hooks, the current working directory is changed to the current virtualenv, and then the command is run. Commands cannot modify the current shell state, but can modify the virtualenv.

```
$ allvirtualenv pip install -U pip
```

Controlling the Active Environment

workon

List or change working virtual environments

Syntax:

```
workon [environment_name]
```

If no `environment_name` is given the list of available environments is printed to stdout.

```
$ workon
$ mkvirtualenv env1
  New python executable in env1/bin/python
Installing distribute.....
.....
done.
(env1)$ mkvirtualenv env2
New python executable in env2/bin/python
Installing distribute.....
.....
done.
(env2)$ workon
env1
env2
(env2)$ workon env1
(env1)$ echo $VIRTUAL_ENV
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env1
(env1)$ workon env2
(env2)$ echo $VIRTUAL_ENV
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env2
(env2)$
```

See also:

- [predeactivate](#)
- [postdeactivate](#)
- [preactivate](#)
- [postactivate](#)

deactivate

Switch from a virtual environment to the system-installed version of Python.

Syntax:

```
deactivate
```

Note: This command is actually part of `virtualenv`, but is wrapped to provide before and after hooks, just as `workon` does for `activate`.

```
$ workon
$ echo $VIRTUAL_ENV

$ mkvirtualenv env1
New python executable in env1/bin/python
Installing distribute.....
.....
```

```
.....
done.
(env1)$ echo $VIRTUAL_ENV
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env1
(env1)$ deactivate
$ echo $VIRTUAL_ENV
$
```

See also:

- *predeactivate*
- *postdeactivate*

Quickly Navigating to a virtualenv

There are two functions to provide shortcuts to navigate into the currently-active virtualenv.

cdvirtualenv

Change the current working directory to \$VIRTUAL_ENV.

Syntax:

```
cdvirtualenv [subdir]
```

Calling `cdvirtualenv` changes the current working directory to the top of the virtualenv (\$VIRTUAL_ENV). An optional argument is appended to the path, allowing navigation directly into a subdirectory.

```
$ mkvirtualenv env1
New python executable in env1/bin/python
Installing distribute.....
.....
done.
(env1)$ echo $VIRTUAL_ENV
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env1
(env1)$ cdvirtualenv
(env1)$ pwd
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env1
(env1)$ cdvirtualenv bin
(env1)$ pwd
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env1/bin
```

cdsitepackages

Change the current working directory to the `site-packages` for \$VIRTUAL_ENV.

Syntax:

```
cdsitepackages [subdir]
```

Because the exact path to the `site-packages` directory in the `virtualenv` depends on the version of `Python`, `cdsitepackages` is provided as a shortcut for `cdvirtualenv lib/python${pyvers}/site-packages`. An optional argument is also allowed, to specify a directory hierarchy within the `site-packages` directory to change into.

```
$ mkvirtualenv env1
New python executable in env1/bin/python
Installing distribute.....
.....
done.
(env1)$ echo $VIRTUAL_ENV
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env1
(env1)$ cdsitepackages PyMOTW/bisect/
(env1)$ pwd
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env1/lib/python2.6/site-packages/PyMOTW/
↪bisect
```

lssitepackages

Calling `lssitepackages` shows the content of the `site-packages` directory of the currently-active `virtualenv`.

Syntax:

```
lssitepackages
```

```
$ mkvirtualenv env1
New python executable in env1/bin/python
Installing distribute.....
.....
done.
(env1)$ $ workon env1
(env1)$ lssitepackages
distribute-0.6.10-py2.6.egg      pip-0.6.3-py2.6.egg
easy-install.pth               setuptools.pth
```

Path Management

add2virtualenv

Adds the specified directories to the `Python` path for the currently-active `virtualenv`.

Syntax:

```
add2virtualenv directory1 directory2 ...
```

Sometimes it is desirable to share installed packages that are not in the system `site-packages` directory and which should not be installed in each `virtualenv`. One possible solution is to symlink the source into the environment `site-packages` directory, but it is also easy to add extra directories to the `PYTHONPATH` by including them in a `.pth` file inside `site-packages` using `add2virtualenv`.

1. Check out the source for a big project, such as Django.
2. Run: `add2virtualenv path_to_source`.

3. Run: `add2virtualenv`.
4. A usage message and list of current “extra” paths is printed.

The directory names are added to a path file named `_virtualenv_path_extensions.pth` inside the site-packages directory for the environment.

Based on a contribution from James Bennett and Jannis Leidel.

toggleglobalsitepackages

Controls whether the active virtualenv will access the packages in the global Python `site-packages` directory.

Syntax:

```
toggleglobalsitepackages [-q]
```

Outputs the new state of the virtualenv. Use the `-q` switch to turn off all output.

```
$ mkvirtualenv env1
New python executable in env1/bin/python
Installing distribute.....
.....
done.
(env1)$ toggleglobalsitepackages
Disabled global site-packages
(env1)$ toggleglobalsitepackages
Enabled global site-packages
(env1)$ toggleglobalsitepackages -q
(env1)$
```

Project Directory Management

See also:

[Project Management](#)

mkproject

Create a new virtualenv in the `WORKON_HOME` and project directory in `PROJECT_HOME`.

Syntax:

```
mkproject [-f|--force] [-t template] [virtualenv_options] ENVNAME
```

-f, --force Create the virtualenv even if the project directory already exists

The `template` option may be repeated to have several templates used to create a new project. The templates are applied in the order named on the command line. All other options are passed to `mkvirtualenv` to create a virtual environment with the same name as the project.

```
$ mkproject myproj
New python executable in myproj/bin/python
Installing distribute.....
.....
.....
```



```
done.
Creating /Users/dhellmann/Devel/myproj
(myproj)$ pwd
/Users/dhellmann/Devel/myproj
(myproj)$ echo $VIRTUAL_ENV
/Users/dhellmann/Envs/myproj
(myproj)$
```

See also:

- [premkproject](#)
- [postmkproject](#)

setvirtualenvproject

Bind an existing virtualenv to an existing project.

Syntax:

```
setvirtualenvproject [virtualenv_path project_path]
```

The arguments to `setvirtualenvproject` are the full paths to the virtualenv and project directory. An association is made so that when `workon` activates the virtualenv the project is also activated.

```
$ mkproject myproj
New python executable in myproj/bin/python
Installing distribute.....
.....
done.
Creating /Users/dhellmann/Devel/myproj
(myproj)$ mkvirtualenv myproj_new_libs
New python executable in myproj/bin/python
Installing distribute.....
.....
done.
Creating /Users/dhellmann/Devel/myproj
(myproj_new_libs)$ setvirtualenvproject $VIRTUAL_ENV $(pwd)
```

When no arguments are given, the current virtualenv and current directory are assumed.

Any number of virtualenvs can refer to the same project directory, making it easy to switch between versions of Python or other dependencies for testing.

cdproject

Change the current working directory to the one specified as the project directory for the active virtualenv.

Syntax:

```
cdproject
```

Managing Installed Packages

wipeenv

Remove all of the installed third-party packages in the current virtualenv.

Syntax:

```
wipeenv
```

Other Commands

virtualenvwrapper

Print a list of commands and their descriptions as basic help output.

Syntax:

```
virtualenvwrapper
```

Customizing Virtualenvwrapper

virtualenvwrapper adds several hook points you can use to change your settings, shell environment, or other configuration values when creating, deleting, or moving between environments. These hooks are exposed in two ways:

Per-User Customization

The end-user customization scripts are either *sourced* (allowing them to modify your shell environment) or *run* as an external program at the appropriate trigger time.

The global scripts applied to all environments should be placed in the directory named by `VIRTUALENVWRAPPER_HOOK_DIR`. The local scripts should be placed in the `bin` directory of the virtualenv.

get_env_details

Global/Local both

Argument(s) env name

Sourced/Run run

`$VIRTUALENVWRAPPER_HOOK_DIR/get_env_details` is run when `workon` is run with no arguments and a list of the virtual environments is printed. The hook is run once for each environment, after the name is printed, and can print additional information about that environment.

initialize

Global/Local global

Argument(s) None

Sourced/Run sourced

`$VIRTUALENVWRAPPER_HOOK_DIR/initialize` is sourced when `virtualenvwrapper.sh` is loaded into your environment. Use it to adjust global settings when `virtualenvwrapper` is enabled.

premkvirtualenv

Global/Local global

Argument(s) name of new environment

Sourced/Run run

`$VIRTUALENVWRAPPER_HOOK_DIR/premkvirtualenv` is run as an external program after the virtual environment is created but before the current environment is switched to point to the new env. The current working directory for the script is `$WORKON_HOME` and the name of the new environment is passed as an argument to the script.

postmkvirtualenv

Global/Local global

Argument(s) none

Sourced/Run sourced

`$VIRTUALENVWRAPPER_HOOK_DIR/postmkvirtualenv` is sourced after the new environment is created and activated. If the `-a <project_path>` flag was used, the link to the project directory is set up before this script is sourced.

precpvirtualenv

Global/Local global

Argument(s) name of original environment, name of new environment

Sourced/Run run

`$VIRTUALENVWRAPPER_HOOK_DIR/precpvirtualenv` is run as an external program after the source environment is duplicated and made relocatable, but before the `premkvirtualenv` hook is run or the current environment is switched to point to the new env. The current working directory for the script is `$WORKON_HOME` and the names of the source and new environments are passed as arguments to the script.

postcpvirtualenv

Global/Local global

Argument(s) none

Sourced/Run sourced

`$VIRTUALENVWRAPPER_HOOK_DIR/postcpvirtualenv` is sourced after the new environment is created and activated.

preactivate

Global/Local global, local

Argument(s) environment name

Sourced/Run run

The global `$VIRTUAL_ENVWRAPPER_HOOK_DIR/preactivate` script is run before the new environment is enabled. The environment name is passed as the first argument.

The local `$VIRTUAL_ENV/bin/preactivate` hook is run before the new environment is enabled. The environment name is passed as the first argument.

postactivate

Global/Local global, local

Argument(s) none

Sourced/Run sourced

The global `$VIRTUAL_ENVWRAPPER_HOOK_DIR/postactivate` script is sourced after the new environment is enabled. `$VIRTUAL_ENV` refers to the new environment at the time the script runs.

This example script adds a space between the virtual environment name and your old PS1 by making use of `_OLD_VIRTUAL_PS1`.

```
PS1="( `basename \"$VIRTUAL_ENV\"` ) $_OLD_VIRTUAL_PS1"
```

The local `$VIRTUAL_ENV/bin/postactivate` script is sourced after the new environment is enabled. `$VIRTUAL_ENV` refers to the new environment at the time the script runs.

This example script for the PyMOTW environment changes the current working directory and the `PATH` variable to refer to the source tree containing the PyMOTW source.

```
pymotw_root=/Users/dhellmann/Documents/PyMOTW
cd $pymotw_root
PATH=$pymotw_root/bin:$PATH
```

predeactivate

Global/Local local, global

Argument(s) none

Sourced/Run sourced

The local `$VIRTUAL_ENV/bin/predeactivate` script is sourced before the current environment is deactivated, and can be used to disable or clear settings in your environment. `$VIRTUAL_ENV` refers to the old environment at the time the script runs.

The global `$VIRTUAL_ENVWRAPPER_HOOK_DIR/predeactivate` script is sourced before the current environment is deactivated. `$VIRTUAL_ENV` refers to the old environment at the time the script runs.

postdeactivate

Global/Local local, global

Argument(s) none

Sourced/Run sourced

The `$VIRTUAL_ENV/bin/postdeactivate` script is sourced after the current environment is deactivated, and can be used to disable or clear settings in your environment. The path to the environment just deactivated is available in `$VIRTUAL_ENVWRAPPER_LAST_VIRTUAL_ENV`.

prermvirtualenv

Global/Local global

Argument(s) environment name

Sourced/Run run

The `$VIRTUAL_ENVWRAPPER_HOOK_DIR/prermvirtualenv` script is run as an external program before the environment is removed. The full path to the environment directory is passed as an argument to the script.

postrmvirtualenv

Global/Local global

Argument(s) environment name

Sourced/Run run

The `$VIRTUAL_ENVWRAPPER_HOOK_DIR/postrmvirtualenv` script is run as an external program after the environment is removed. The full path to the environment directory is passed as an argument to the script.

premkproject

Global/Local global

Argument(s) name of new project

Sourced/Run run

`$WORKON_HOME/premkproject` is run as an external program after the virtual environment is created and after the current environment is switched to point to the new env, but before the new project directory is created. The current working directory for the script is `$PROJECT_HOME` and the name of the new project is passed as an argument to the script.

postmkproject

Global/Local global

Argument(s) none

Sourced/Run sourced

`$WORKON_HOME/postmkproject` is sourced after the new environment and project directories are created and the virtualenv is activated. The current working directory is the project directory.

Extending Virtualenvwrapper

Long experience with home-grown solutions for customizing a development environment has proven how valuable it can be to have the ability to automate common tasks and eliminate persistent annoyances. Carpenters build jigs, software developers write shell scripts. `virtualenvwrapper` continues the tradition of encouraging a craftsman to modify their tools to work the way they want, rather than the other way around.

Use the hooks provided to eliminate repetitive manual operations and streamline your development workflow. For example, set up the `pre_activate` and `post_activate` hooks to trigger an IDE to load a project file to reload files from the last editing session, manage time-tracking records, or start and stop development versions of an application server. Use the `initialize` hook to add entirely new commands and hooks to virtualenvwrapper. And the `pre_mkdirvirtualenv` and `post_mkdirvirtualenv` hooks give you an opportunity to install basic requirements into each new development environment, initialize a source code control repository, or otherwise set up a new project.

There are two ways to attach your code so that virtualenvwrapper will run it: End-users can use shell scripts or other programs for personal customization (see *Per-User Customization*). Extensions can also be implemented in Python by using *Distribute entry points*, making it possible to share common behaviors between systems and developers.

Defining an Extension

Note: Virtualenvwrapper is delivered with a plugin for creating and running the user customization scripts (*user_scripts*). The examples below are taken from the implementation of that plugin.

Code Organization

The Python package for `virtualenvwrapper` is a *namespace package*. That means multiple libraries can install code into the package, even if they are not distributed together or installed into the same directory. Extensions can (optionally) use the `virtualenvwrapper` namespace by setting up their source tree like:

- `virtualenvwrapper/`
 - `__init__.py`
 - `user_scripts.py`

And placing the following code in `__init__.py`:

```
"""virtualenvwrapper module
"""

__import__('pkg_resources').declare_namespace(__name__)
```

Note: Extensions can be loaded from any package, so using the `virtualenvwrapper` namespace is not required.

Extension API

After the package is established, the next step is to create a module to hold the extension code. For example, `virtualenvwrapper/user_scripts.py`. The module should contain the actual extension entry points. Supporting code can be included, or imported from elsewhere using standard Python code organization techniques.

The API is the same for every extension point. Each uses a Python function that takes a single argument, a list of strings passed to the hook loader on the command line.

```
def function_name(args):
    # args is a list of strings passed to the hook loader
```

The contents of the argument list are defined for each extension point below (see *Extension Points*).

Extension Invocation

Direct Action

Plugins can attach to each hook in two different ways. The default is to have a function run and do some work directly. For example, the `initialize()` function for the user scripts plugin creates default user scripts when `virtualenvwrapper.sh` is loaded.

```
def initialize(args):
    for filename, comment in GLOBAL_HOOKS:
        make_hook(os.path.join('$WORKON_HOME', filename), comment)
    return
```

Modifying the User Environment

There are cases where the extension needs to update the user's environment (e.g., changing the current working directory or setting environment variables). Modifications to the user environment must be made within the user's current shell, and cannot be run in a separate process. To have code run in the user's shell process, extensions can define hook functions to return the text of the shell statements to be executed. These *source* hooks are run after the regular hooks with the same name, and should not do any work of their own.

The `initialize_source()` hook for the user scripts plugin looks for a global initialize script and causes it to be run in the current shell process.

```
def initialize_source(args):
    return """
#
# Run user-provided scripts
#
[ -f "$WORKON_HOME/initialize" ] && source "$WORKON_HOME/initialize"
"""
```

Warning: Because the extension is modifying the user's working shell, care must be taken not to corrupt the environment by overwriting existing variable values unexpectedly. Avoid creating temporary variables where possible, and use unique names where variables cannot be avoided. Prefixing variables with the extension name is a good way to manage the namespace. For example, instead of `temp_file` use `user_scripts_temp_file`. Use `unset` to release temporary variable names when they are no longer needed.

Warning: `virtualenvwrapper` works under several shells with slightly different syntax (`bash`, `sh`, `zsh`, `ksh`). Take this portability into account when defining source hooks. Sticking to the simplest possible syntax usually avoids problems, but there may be cases where examining the `SHELL` environment variable to generate different syntax for each case is the only way to achieve the desired result.

Registering Entry Points

The functions defined in the plugin need to be registered as *entry points* in order for `virtualenvwrapper`'s hook loader to find them. `Distribute` entry points are configured in the `setup.py` for your package by mapping the entry point name to the function in the package that implements it.

This partial copy of `virtualenvwrapper's setup.py` illustrates how the `initialize()` and `initialize_source()` entry points are configured.

```
# Bootstrap installation of Distribute
import distribute_setup
distribute_setup.use_setuptools()

from setuptools import setup

setup(
    name = 'virtualenvwrapper',
    version = '2.0',

    description = 'Enhancements to virtualenv',

    # ... details omitted ...

    namespace_packages = [ 'virtualenvwrapper' ],

    entry_points = {
        'virtualenvwrapper.initialize': [
            'user_scripts = virtualenvwrapper.user_scripts:initialize',
        ],
        'virtualenvwrapper.initialize_source': [
            'user_scripts = virtualenvwrapper.user_scripts:initialize_source',
        ],

        # ... details omitted ...
    },
)
```

The `entry_points` argument to `setup()` is a dictionary mapping the entry point *group names* to lists of entry point specifiers. A different group name is defined by `virtualenvwrapper` for each extension point (see [Extension Points](#)).

The entry point specifiers are strings with the syntax `name = package.module:function`. By convention, the *name* of each entry point is the plugin name, but that is not required (the names are not used).

See also:

- [namespace packages](#)
- [Extensible Applications and Frameworks](#)

The Hook Loader

Extensions are run through a command line application implemented in `virtualenvwrapper.hook_loader`. Because `virtualenvwrapper.sh` is the primary caller and users do not typically need to run the app directly, no separate script is installed. Instead, to run the application, use the `-m` option to the interpreter:

```
$ python -m virtualenvwrapper.hook_loader -h
Usage: virtualenvwrapper.hook_loader [options] <hook> [<arguments>]

Manage hooks for virtualenvwrapper

Options:
  -h, --help            show this help message and exit
  -s, --source          Print the shell commands to be run in the current
```



```

shell
-l, --list          Print a list of the plugins available for the given
                    hook
-v, --verbose      Show more information on the console
-q, --quiet        Show less information on the console
-n NAMES, --name=NAMES
                    Only run the hook from the named plugin

```

To run the extensions for the initialize hook:

```
$ python -m virtualenvwrapper.hook_loader -v initialize
```

To get the shell commands for the initialize hook:

```
$ python -m virtualenvwrapper.hook_loader --source initialize
```

In practice, rather than invoking the hook loader directly it is more convenient to use the shell function, `virtualenvwrapper_run_hook` to run the hooks in both modes.:

```
$ virtualenvwrapper_run_hook initialize
```

All of the arguments given to shell function are passed directly to the hook loader.

Logging

The hook loader configures logging so that messages are written to `$WORKON_HOME/hook.log`. Messages also may be written to `stderr`, depending on the verbosity flag. The default is for messages at *info* or higher levels to be written to `stderr`, and *debug* or higher to go to the log file. Using logging in this way provides a convenient mechanism for users to control the verbosity of extensions.

To use logging from within your extension, simply instantiate a logger and call its `info()`, `debug()` and other methods with the messages.

```

import logging
log = logging.getLogger(__name__)

def pre_mkvirtualenv(args):
    log.debug('pre_mkvirtualenv %s', str(args))
    # ...

```

See also:

- [Standard library documentation for logging](#)
- [PyMOTW for logging](#)

Extension Points

The extension point names for native plugins follow a naming convention with several parts: `virtualenvwrapper.(pre|post)<event>[_source]`. The `<event>` is the action taken by the user or `virtualenvwrapper` that triggers the extension. `(pre|post)` indicates whether to call the extension before or after the event. The suffix `_source` is added for extensions that return shell code instead of taking action directly (see *Modifying the User Environment*).

get_env_details

The `virtualenvwrapper.get_env_details` hooks are run when `workon` is run with no arguments and a list of the virtual environments is printed. The hook is run once for each environment, after the name is printed, and can be used to show additional information about that environment.

initialize

The `virtualenvwrapper.initialize` hooks are run each time `virtualenvwrapper.sh` is loaded into the user's environment. The initialize hook can be used to install templates for configuration files or otherwise prepare the system for proper plugin operation.

pre_mkvirtualenv

The `virtualenvwrapper.pre_mkvirtualenv` hooks are run after the virtual environment is created, but before the new environment is activated. The current working directory for when the hook is run is `$WORKON_HOME` and the name of the new environment is passed as an argument.

post_mkvirtualenv

The `virtualenvwrapper.post_mkvirtualenv` hooks are run after a new virtual environment is created and activated. `$VIRTUAL_ENV` is set to point to the new environment.

pre_activate

The `virtualenvwrapper.pre_activate` hooks are run just before an environment is enabled. The environment name is passed as the first argument.

post_activate

The `virtualenvwrapper.post_activate` hooks are run just after an environment is enabled. `$VIRTUAL_ENV` is set to point to the current environment.

pre_deactivate

The `virtualenvwrapper.pre_deactivate` hooks are run just before an environment is disabled. `$VIRTUAL_ENV` is set to point to the current environment.

post_deactivate

The `virtualenvwrapper.post_deactivate` hooks are run just after an environment is disabled. The name of the environment just deactivated is passed as the first argument.

pre_rmvirtualenv

The `virtualenvwrapper.pre_rmvirtualenv` hooks are run just before an environment is deleted. The name of the environment being deleted is passed as the first argument.

post_rmvirtualenv

The `virtualenvwrapper.post_rmvirtualenv` hooks are run just after an environment is deleted. The name of the environment being deleted is passed as the first argument.

Adding New Extension Points

Plugins that define new operations can also define new extension points. No setup needs to be done to allow the hook loader to find the extensions; documenting the names and adding calls to `virtualenvwrapper_run_hook` is sufficient to cause them to be invoked.

The hook loader assumes all extension point names start with `virtualenvwrapper.` and new plugins will want to use their own namespace qualifier to append to that. For example, the `project` extension defines new events around creating project directories (pre and post). These are called `virtualenvwrapper.project.pre_mkproject` and `virtualenvwrapper.project.post_mkproject`. These are invoked with:

```
virtualenvwrapper_run_hook project.pre_mkproject $project_name
```

and:

```
virtualenvwrapper_run_hook project.post_mkproject
```

respectively.

Project Management

A project directory is associated with a `virtualenv`, but usually contains the source code under active development rather than the installed components needed to support the development. For example, the project directory may contain the source code checked out from a version control system, temporary artifacts created by testing, experimental files not committed to version control, etc.

A project directory is created and bound to a `virtualenv` when `mkproject` is run instead of `mkvirtualenv`. To bind an existing project directory to a `virtualenv`, use `setvirtualenvproject`.

Using Templates

A new project directory can be created empty, or populated using one or more template plugins. Templates should be specified as arguments to `mkproject`. Multiple values can be provided to apply more than one template. For example, to check out a Mercurial repository from on a project on bitbucket and create a new Django site, combine the `bitbucket` and `django` templates.

```
$ mkproject -t bitbucket -t django my_site
```

See also:

- [Templates](#)

- *Location of Project Directories*
- *Project Linkage Filename*

Tips and Tricks

This is a list of user-contributed tips for making virtualenv and virtualenvwrapper even more useful. If you have tip to share, drop me an email or post a comment on [this blog post](#) and I'll add it here.

zsh Prompt

From Nat:

Using zsh, I added some bits to `$WORKON_HOME/post (de) activate` to show the active virtualenv on the right side of my screen instead.

in `postactivate`:

```
PS1="$ _OLD_VIRTUAL_PS1"  
_OLD_RPROMPT="$RPROMPT"  
RPROMPT="%${fg_bold[white]}% (env: ${fg[green]}% `basename \"$VIRTUAL_ENV\"` ${fg_  
↪bold[white]}%)%${reset_color}% $RPROMPT"
```

and in `postdeactivate`:

```
RPROMPT="$ _OLD_RPROMPT"
```

Adjust colors according to your own personal tastes or environment.

Updating cached `$PATH` entries

From Nat:

I also added the command `'rehash'` to `$WORKON_HOME/postactivate` and `$WORKON_HOME/postdeactivate` as I was having some problems with zsh not picking up the new paths immediately.

Tying to pip's virtualenv support

Via <http://becomingguru.com/>:

Add this to your shell login script to make pip use the same directory for virtualenvs as virtualenvwrapper:

```
export PIP_VIRTUALENV_BASE=$WORKON_HOME
```

and Via Nat:

in addition to what becomingguru said, this line is key:

```
export PIP_RESPECT_VIRTUALENV=true
```

That makes pip detect an active virtualenv and install to it, without having to pass it the `-E` parameter.

Creating Project Work Directories

Via James:

In the `postmkvirtualenv` script I have the following to create a directory based on the project name, add that directory to the python path and then `cd` into it:

```
proj_name=$(basename $VIRTUAL_ENV)
mkdir $HOME/projects/$proj_name
add2virtualenv $HOME/projects/$proj_name
cd $HOME/projects/$proj_name
```

In the `postactivate` script I have it set to automatically change to the project directory when I use the `workon` command:

```
proj_name=$(basename $VIRTUAL_ENV)
cd ~/projects/$proj_name
```

Automatically Run workon When Entering a Directory

Justin Lily [posted](#) about some code he added to his shell environment to look at the directory each time he runs `cd`. If it finds a `.venv` file, it activates the environment named within. On leaving that directory, the current `virtualenv` is automatically deactivated.

Harry Marr wrote a similar function that works with [git repositories](#).

Installing Common Tools Automatically in New Environments

Via rizumu:

I have this `postmkvirtualenv` to install the get a basic setup.

```
$ cat postmkvirtualenv
#!/usr/bin/env bash
curl -O http://python-distribute.org/distribute_setup.p... />python distribute_setup.
↪py
rm distribute_setup.py
easy_install pip==dev
pip install Mercurial
```

Then I have a `pip` requirement file with my dev tools.

```
$ cat developer_requirements.txt
ipdb
ipython
pastescript
nose
http://douglatornell.ca/software/python/Nosy-1.0.tar.gz
coverage
sphinx
grin
pyflakes
pep8
```

Then each project has it's own `pip` requirement file for things like `PIL`, `psycopg2`, `django-apps`, `numpy`, etc.

Changing the Default Behavior of `cd`

Via `mae`:

This is supposed to be executed after `workon`, that is as a `postactivate` hook. It basically overrides `cd` to know about the `VENV` so instead of doing `cd` to go to `~` you will go to the `venv` root, IMO very handy and I can't live without it anymore. if you pass it a proper path then it will do the right thing.

```
cd () {
  if (( $# == 0 ))
  then
    builtin cd $VIRTUAL_ENV
  else
    builtin cd "$@"
  fi
}

cd
```

For Developers

If you would like to contribute to `virtualenvwrapper` directly, these instructions should help you get started. Patches, bug reports, and feature requests are all welcome through the [BitBucket site](#). Contributions in the form of patches or pull requests are easier to integrate and will receive priority attention.

Note: Before contributing new features to `virtualenvwrapper` core, please consider whether they should be implemented as an extension instead.

Building Documentation

The documentation for `virtualenvwrapper` is written in reStructuredText and converted to HTML using Sphinx. The build itself is driven by `make`. You will need the following packages in order to build the docs:

- Sphinx
- docutils
- sphinxcontrib-bitbucket

Once all of the tools are installed into a `virtualenv` using `pip`, run `make html` to generate the HTML version of the documentation:

```
$ make html
rm -rf virtualenvwrapper/docs
(cd docs && make html SPHINXOPTS="-c sphinx/pkg")
sphinx-build -b html -d build/doctrees -c sphinx/pkg source build/html
Running Sphinx v0.6.4
loading pickled environment... done
building [html]: targets for 2 source files that are out of date
updating environment: 0 added, 2 changed, 0 removed
reading sources... [ 50%] command_ref
reading sources... [100%] developers

looking for now-outdated files... none found
```

```

pickling environment... done
checking consistency... done
preparing documents... done
writing output... [ 33%] command_ref
writing output... [ 66%] developers
writing output... [100%] index

writing additional files... search
copying static files... WARNING: static directory '/Users/dhellmann/Devel/
↳virtualenvwrapper/plugins/docs/sphinx/pkg/static' does not exist
done
dumping search index... done
dumping object inventory... done
build succeeded, 1 warning.

Build finished. The HTML pages are in build/html.
cp -r docs/build/html virtualenvwrapper/docs

```

The output version of the documentation ends up in `./virtualenvwrapper/docs` inside your sandbox.

Running Tests

The test suite for virtualenvwrapper uses `shunit2` and `tox`. The `shunit2` source is included in the `tests` directory, but `tox` must be installed separately (`pip install tox`).

To run the tests under `bash`, `zsh`, and `ksh` for Python 2.4 through 2.7, run `tox` from the top level directory of the hg repository.

To run individual test scripts, use a command like:

```
$ tox tests/test_cd.sh
```

To run tests under a single version of Python, specify the appropriate environment when running `tox`:

```
$ tox -e py27
```

Combine the two modes to run specific tests with a single version of Python:

```
$ tox -e py27 tests/test_cd.sh
```

Add new tests by modifying an existing file or creating new script in the `tests` directory.

Creating a New Template

`virtualenvwrapper.project` templates work like `virtualenvwrapper.plugins`. The *entry point* group name is `virtualenvwrapper.project.template`. Configure your entry point to refer to a function that will **run** (source hooks are not supported for templates).

The argument to the template function is the name of the project being created. The current working directory is the directory created to hold the project files (`$PROJECT_HOME/$envname`).

Help Text

One difference between project templates and other virtualenvwrapper extensions is that only the templates specified by the user are run. The `mkproject` command has a help option to give the user a list of the available templates.

The names are taken from the registered entry point names, and the descriptions are taken from the docstrings for the template functions.

Existing Extensions

Below is a list of some of the extensions available for use with virtualenvwrapper.

emacs-desktop

Emacs `desktop-mode` lets you save the state of emacs (open buffers, kill rings, buffer positions, etc.) between sessions. It can also be used as a project file similar to other IDEs. The `emacs-desktop` plugin adds a trigger to save the current desktop file and load a new one when activating a new virtualenv using `workon`.

user_scripts

The `user_scripts` extension is delivered with virtualenvwrapper and enabled by default. It implements the user customization script features described in *Per-User Customization*.

vim-virtualenv

`vim-virtualenv` is Jeremy Cantrell's plugin for controlling virtualenvs from within vim. When used together with virtualenvwrapper, `vim-virtualenv` identifies the virtualenv to activate based on the name of the file being edited.

Templates

Below is a list of some of the templates available for use with `mkproject`.

bitbucket

The `bitbucket` extension automatically clones a mercurial repository from the specified bitbucket project.

django

The `django` extension automatically creates a new Django project.

SublimeText

Song Jin has created a template plugin for automatically generating the project files used by SublimeText. See the `sublime_projectfile_maker` page for details.

See also:

- *Creating a New Template*

Why virtualenvwrapper is (Mostly) Not Written In Python

If you look at the source code for virtualenvwrapper you will see that most of the interesting parts are implemented as shell functions in `virtualenvwrapper.sh`. The hook loader is a Python app, but doesn't do much to manage the virtualenvs. Some of the most frequently asked questions about virtualenvwrapper are “Why didn't you write this as a set of Python programs?” or “Have you thought about rewriting it in Python?” For a long time these questions baffled me, because it was always obvious to me that it had to be implemented as it is. But they come up frequently enough that I feel the need to explain.

tl;dr: POSIX Made Me Do It

The choice of implementation language for virtualenvwrapper was made for pragmatic, rather than philosophical, reasons. The wrapper commands need to modify the state and environment of the user's *current shell process*, and the only way to do that is to have the commands run *inside that shell*. That resulted in me writing virtualenvwrapper as a set of shell functions, rather than separate shell scripts or even Python programs.

Where Do POSIX Processes Come From?

New POSIX processes are created when an existing process invokes the `fork()` system call. The invoking process becomes the “parent” of the new “child” process, and the child is a full clone of the parent. The *semantic* result of `fork()` is that an entire new copy of the parent process is created. In practice, optimizations are normally made to avoid copying more memory than is absolutely necessary (frequently via a copy-on-write system). But for the purposes of this explanation it is sufficient to think of the child as a full replica of the parent.

The important parts of the parent process that are copied include dynamic memory (the stack and heap), static stuff (the program code), resources like open file descriptors, and the *environment variables* exported from the parent process. Inheriting environment variables is a fundamental aspect of the way POSIX programs pass state and configuration information to one another. A parent can establish a series of `name=value` pairs, which are then given to the child process. The child can access them through functions like `getenv()`, `setenv()` (and in Python through `os.environ`).

The choice of the term *inherit* to describe the way the variables and their contents are passed from parent to child is significant. Although a child can change its own environment, it cannot directly change the environment settings of its parent because there is no system call to modify the parental environment settings.

How the Shell Runs a Program

When a shell receives a command to be executed, either interactively or by parsing a script file, and determines that the command is implemented in a separate program file, it uses `fork()` to create a new process and then inside that process it uses one of the `exec` functions to start the specified program. The language that program is written in doesn't make any difference in the decision about whether or not to `fork()`, so even if the “program” is a shell script written in the language understood by the current shell, a new process is created.

On the other hand, if the shell decides that the command is a *function*, then it looks at the definition and invokes it directly. Shell functions are made up of other commands, some of which may result in child processes being created, but the function itself runs in the original shell process and can therefore modify its state, for example by changing the working directory or the values of variables.

It is possible to force the shell to run a script directly, and not in a child process, by *sourcing* it. The `source` command causes the shell to read the file and interpret it in the current process. Again, as with functions, the contents of the file may cause child processes to be spawned, but there is not a second shell process interpreting the series of commands.

What Does This Mean for virtualenvwrapper?

The original and most important features of virtualenvwrapper are automatically activating a virtualenv when it is created by `mkvirtualenv` and using `workon` to deactivate one environment and activate another. Making these features work drove the implementation decisions for the other parts of virtualenvwrapper, too.

Environments are activated interactively by sourcing `bin/activate` inside the virtualenv. The `activate` script does a few things, but the important parts are setting the `VIRTUAL_ENV` variable and modifying the shell's search path through the `PATH` variable to put the `bin` directory for the environment on the front of the path. Changing the path means that the programs installed in the environment, especially the python interpreter there, are found before other programs with the same name.

Simply running `bin/activate`, without using `source` doesn't work because it sets up the environment of the *child* process, without affecting the parent. In order to source the activate script in the interactive shell, both `mkvirtualenv` and `workon` also need to be run in that shell process.

Why Choose One When You Can Have Both?

The hook loader is one part of virtualenvwrapper that *is* written in Python. Why? Again, because it was easier. Hooks are discovered using `setuptools` entry points, because after an entry point is installed the user doesn't have to take any other action to allow the loader to discover and use it. It's easy to imagine writing a hook to create new files on the filesystem (by installing a package, instantiating a template, etc.).

How, then, do hooks running in a separate process (the Python interpreter) modify the shell environment to set variables or change the working directory? They cheat, of course.

Each hook point defined by virtualenvwrapper actually represents two hooks. First, the hooks meant to be run in Python are executed. Then the "source" hooks are run, and they *print out* a series of shell commands. All of those commands are collected, saved to a temporary file, and then the shell is told to source the file.

Starting up the hook loader turns out to be way more expensive than most of the other actions virtualenvwrapper takes, though, so I am considering making its use optional. Most users customize the hooks by using shell scripts (either globally or in the virtualenv). Finding and running those can be handled by the shell quite easily.

Implications for Cross-Shell Compatibility

Other than requests for a full-Python implementation, the other most common request is to support additional shells. `fish` comes up a lot, as do various Windows-only shells. The officially *Supported Shells* all have a common enough syntax that the same implementation works for each. Supporting other shells would require rewriting much, if not all, of the logic using an alternate syntax – those other shells are basically different programming languages. So far I have dealt with the ports by encouraging other developers to handle them, and then trying to link to and otherwise promote the results.

Not As Bad As It Seems

Although there are some special challenges created by the the requirement that the commands run in a user's interactive shell (see the many bugs reported by users who alias common commands like `rm` and `cd`), using the shell as a programming language holds up quite well. The shells are designed to make finding and executing other programs easy, and especially to make it easy to combine a series of smaller programs to perform more complicated operations. As that's what virtualenvwrapper is doing, it's a natural fit.

See also:

- [Advanced Programming in the UNIX Environment](#) by W. Richard Stevens & Stephen A. Rago

- [Fork \(operating system\) on Wikipedia](#)
- [Environment variable on Wikipedia](#)
- [Linux implementation of fork\(\)](#)

Release History

dev

- Add *virtualenvwrapper* to print basic help and a list of commands.

4.2

- Add `tmp-` prefix to temporary environment names created by *mktmpenv*.
- Fix some uses of `cd` that did not account for possible aliasing. Contributed by Ismail Badawi ([ibadawi](#)).
- Fix documentation for *allvirtualenv*, contributed by Andy Dirnberger ([dirn](#)).
- Add `--force` option to *mkproject*, contributed by Clay McClure ([claymcclure](#)).
- Fix handling for project directory argument `-a` to *mkvirtualenv*, based on work by Xupeng Yun.
- Dropped python 3.2 testing.
- Updated test configuration so they work properly under Linux.
- Resolve relative paths before storing the project directory reference in *setvirtualenvproject*. ([issue 207](#))
- Do not create hooks for `rmproject`, since there is no such command. ([issue 203](#))
- Update the tests to use a valid template for creating temporary directories under Linux.
- Fix the use of `which` in `virtualenvwrapper_lazy.sh` in case it is aliased.
- Fix an issue with recursion in completion expansion crashing `zsh`, contributed by [blueyed](#).

4.1.1

- Fix packaging issue with 4.1.

4.1

- Ensure that all `$ ()` style commands that produce paths are quoted. Addresses [issue 164](#).
- Add *wipeenv* command for removing all packages installed in the virtualenv.
- Allow users of `virtualenvwrapper_lazy.sh` to extend the list of API commands that trigger the lazy-loader by extending `_VIRTUALENVWRAPPER_API`. Patch contributed by John Purnell, see [issue 188](#).
- Fix detection of `--python` option to *mkvirtualenv*. Resolves [issue 190](#).
- Add *allvirtualenv* command to run a command across all virtualenvs. Suggested by Dave Coutts in [issue 186](#).
- Fix *lsvirtualenv* when there are spaces in `WORKON_HOME`. Resolves [issue 194](#).
- Switch to `pbr` for packaging.

4.0

Warning: This release includes some potentially incompatible changes for extensions. The python modules for extensions are now *always* run with `PWD=$WORKON_HOME` (previously the value of `PWD` varied depending on the hook). The *shell* portion of any hook (anything sourced by the user's shell when the hook is run) is still run in the same place as before.

- All tests pass under Python 2.6, 2.7, 3.2 and 3.3.
- Fix the name of the script in an error message produced by `virtualenvwrapper_lazy.sh`. (Contributed by [scottstvn](#).)

3.7.1

- Rename functions for generating help so they do not pollute the global namespace, and especially so they do not interfere with tab completion. Contributed by [dauidszotten](#).
- Fix an issue with listing project templates if none are installed. ([issue 179](#))
- Fix an issue with the `--python` option to `mkvirtualenv` becoming *sticky* for future calls that do not explicitly specify the option. ([issue 178](#))

3.7

- Improve tab-completion support for users of the lazy-loading mode. ([upsuper](#))
- Add `--help` option to `mkproject`.
- Add `--help` option to `workon`.
- Turn off logging from the hook loader by default, and replace `VIRTUALENVWRAPPER_LOG_DIR` environment variable with `VIRTUALENVWRAPPER_LOG_FILE`. The rotating log behavior remains the same. The motivation for this change is the race condition caused by that rotating behavior, especially when the wrappers are being used by users with different permissions and umasks. ([issue 152](#))
- Use `flake8` for style checking.

3.6.1

- Replace `realpath` with a more portable way of converting a relative path to an absolute path, used with the `--python` option to `mkvirtualenv` (contributed by Radu Voicilas, [rvoicilas](#)).
- Posted release to PyPI, resolving download redirect issue. ([issue 171](#) and [issue 172](#))

3.6

- Switch to `stevedore` for plugin management
- `mkvirtualenv_help` should use `$VIRTUALENVWRAPPER_PYTHON` instead of calling `virtualenv` directly ([issue 148](#)).
- Fix issue with lazy-loader code under `zsh` ([issue 144](#)).
- Fix issue with `noclobber` option under `zsh` ([issue 137](#)). Fix based on patch from [rob_b](#).
- Fix documentation for `add2virtualenv` to show the correct name for the file containing the new path entry. (contributed by [rvoicilas](#))

- Fix problem with `virtualenvwrapper_show_workon_options` under `zsh` with `chpwd` functions that produce output. (issue 153)

3.5

- Rewrite `cpvirtualenv` to use `virtualenv-clone` instead of making the new environment relocatable. Contributed by Justin Barber ([barberj](#)). This also resolves a problem with `cpvirtualenv` not honoring the `--no-site-packages` flag (issue 102).
- Update docs with link to `virtualenvwrapper-win` port by David Marble.
- Use `command` to avoid functions named the same as common utilities. (issue 119)

3.4

- Add *Lazy Loading* option.

3.3

- Clean up file permissions and remove shebangs from scripts not intended to be executed on the command line. (contributed by [ralphbean](#))
- Worked on some brittle tests.
- Received updates to Japanese translation of the documentation from [t2y](#).
- Fix the test script and runner so the user's `$WORKON_HOME` is not erased if they do not have some test shells installed. (big thanks to [agriffis](#)).
- If the hook loader is told to list plugins but is not given a hook name, it prints the list of core hooks.
- Merge several fixes for path and variable handling for MSYS users from [bwanamarko](#). Includes a fix for [issue 138](#).
- Change `mkvirtualenv` so it catches both `-h` and `--help`.
- Fix some issues with the way temporary files are used for hook scripts. (contributed by [agriffis](#))
- Allow relative path to requirements file with `mkvirtualenv` and `-r` option. ([barberj](#))
- Make whitespace consistent. ([agriffis](#))

3.2

- Make `project_dir` a local variable so that `cdproject` does not interfere with other variables the user might have set. (contributed by [slackorama](#))
- Fix typo in documentation reported by Nick Martin.
- Change trove classifier for license “MIT” to reflect the license text presented in the documentation. *This does not indicate a change in the license, just a correction to the expression of that intent. See `:ref: 'license'`* (contributed by [ralphbean](#) as fix for [issue 134](#))
- Extend `rmvirtualenv` to allow removing more than one environment at a time. (contributed by [ciberglo](#))

- Change the definition of `virtualenvwrapper_get_site_packages_dir` to ask `distutils` for the `site-packages` directory instead of trying to build the path ourselves in the shell script. This should resolve [issue 112](#) and improve support for Python interpreters other than C Python. Thanks to Carl Meyer and Dario Bertini for their contributions toward the fix.

3.1

- Fix a problem with activation hooks when associating a new virtualenv with an existing project directory. ([issue 122](#))
- Fix a problem with `command-add2virtualenv` and paths containing “special” characters such as `&`. ([issue 132](#))

3.0.1

- Fix some packaging issues that made it more difficult to run the tests directly from the `sdist` package. ([issue 126](#))

3.0

- Add Python 3 support, thanks in large part to the efforts of Daniel Kraus ([dakra](#)). Tested under Python 2.6, 2.7, and 3.2.

2.11.1

- Remove the initialization shortcut because it breaks tab completion in sub-shell environments like `screen` and `tmux`. ([issue 121](#))

2.11

- Add `-a` option to `mkvirtualenv` to associate a new virtualenv with an existing project directory. Contributed by Mike Fogel ([mfogel](#)).
- Drops support for Python 2.4 and 2.5. The tools may still work, but I no longer have a development environment set up for testing them, so I do not officially support them.
- Shortcut initialization if it has run before.
- Set hook log file permissions to be group-writable. ([issue 62](#) reported by [hedgeddown](#))
- Add `VIRTUALENVWRAPPER_PROJECT_FILENAME` variable so the `.project` file used to link a virtualenv to a project can be renamed to avoid conflicts with other tools. ([issue 120](#) reported by [arthuralvim](#))

2.10.1

- Changed arguments to `mktmpenv` so it always creates an environment name for you. ([issue 114](#) reported by [alex_gaynor](#))

2.10

- Incorporated patch to add `-d` option to `command-add2virtualenv`, contributed by [miracle2k](#).
- Add `-i` option to `mkvirtualenv`.
- Add `mktmpenv` command for creating temporary environments that are automatically removed when they are deactivated.
- Fixed a problem with `hook_loader` that prevented it from working under Python 2.5 and 2.4.
- Fix a problem with the way template names were processed under `zsh`. ([issue 111](#))

2.9

- Change the shell function shell definition syntax so that `ksh` will treat `typeset`-declared variables as local. No kidding.
- Merge the “project directory” features of the `virtualenvwrapper.project` plugin into the main project, adding `mkproject`, `cdproject`, and `setvirtualenvproject` commands.
- Add `-r` option to `mkvirtualenv` to install dependencies using a `pip` requirements file.

2.8

- Use `VIRTUALENVWRAPPER_VIRTUALENV` in `cpvirtualenv` ([issue 104](#)).
- Add support for `MSYS` environment under Windows. Contributed by Axel H. ([noirbizarre](#)).

2.7.2

- Move setup code for tab completion later in the startup code so all of the needed variables are configured. ([issue 97](#))
- Expand tab completion for `zsh` to work for all commands.

2.7.1

- When testing for `WORKON_HOME` during startup, dereference any symlink to make sure it is a directory.
- Set `VIRTUALENVWRAPPER_HOOK_DIR` and `VIRTUALENV_WRAPPER_LOG_DIR` in `virtualenvwrapper_initialize` after `WORKON_HOME` is set ([issue 94](#)).
- Update the *Basic Installation* instructions to be more explicit about needing to install `virtualenvwrapper` globally (or at least outside of a `virtualenv`).

2.7

- Fix problem with space in `WORKON_HOME` path ([issue 79](#)).
- Fix problem with argument processing in `lsvirtualenv` under `zsh` ([issue 86](#)). Thanks to Nat Williams ([natw](#)) for the bug report and patch.
- If `WORKON_HOME` does not exist, create it. Patch from Carl Karsten ([CarlFK](#)). Test updates based on patches from Matt Austin ([maafy6](#)) and Hugo Lopes Tavares ([hltbra](#)).

- Merge in contributions from Paul McLanahan ([pmclanahan](#)) to fix the test harness to ensure that the test scripts are actually running under the expected shell.
- Merge in new shell command *toggleglobalsitepackages* from Paul McLanahan ([pmclanahan](#)). The new command changes the configuration of the active virtualenv to enable or disable the global `site-packages` directory.
- Fixed some tests that were failing under ksh on Ubuntu 10.10.
- Document the `VIRTUALENVWRAPPER_VIRTUALENV` variable.
- Implement suggestion by Van Lindberg to have `VIRTUALENVWRAPPER_HOOK_DIR` and `VIRTUALENVWRAPPER_LOG_DIR` variables to control the locations of hooks and logs.
- Enabled tab completion for *showvirtualenv* (issue 78).
- Fixed a problem with running *rmvirtualenv* from within the environment being removed (issue 83).
- Removed use of `-e` option in calls to `grep` for better portability (issue 85).

2.6.3

- Hard-code the version information in the `setup.py` and `conf.py` scripts. This still doesn't work for <http://readthedocs.org>, since the doc build needs the `sphinxcontrib.bitbucket` extension, but will make it easier to transition the docs to another site later.

2.6.2

- Attempted to make the doc build work with <http://readthedocs.org>.
- Merged in [Japanese translation of the documentation](#) from Tetsuya Morimoto.
- Incorporate a suggestion from Ales Zoulek to let the user specify the virtualenv binary through an environment variable (`VIRTUALENVWRAPPER_VIRTUALENV`).

2.6.1

- Fixed `virtualenvwrapper_get_python_version` (issue 73).

2.6

- Fixed a problem with hook script line endings under Cygwin (issue 68).
- Updated documentation to include a list of the compatible shells (*Supported Shells*) and Python versions (*Python Versions*) (issue 70).
- Fixed installation dependency on virtualenv (issue 60).
- Fixed the method for determining the Python version so it works under Python 2.4 (issue 61).
- Converted the test infrastructure to use `tox` instead of home-grown scripts in the Makefile.

2.5.3

- Point release uploaded to PyPI during outage on doughellmann.com.

2.5.2

- Apply patch from Zach Voase to fix *lsvirtualenv* under zsh. Resolves [issue 64](#).

2.5.1

- Make *workon* list brief environment details when run without argument, instead of full details.

2.5

- Add *showvirtualenv* command. Modify *lsvirtualenv* to make verbose output the default.

2.4

- Add *lsvirtualenv* command with `-l` option to run *get_env_details* hook instead of always running it when *workon* has no arguments.

2.3

- Added `get_env_details` hook.

2.2.2

- Integrate Fred Palmer's patch to escape more shell commands to avoid aliases. Resolves [issue 57](#).
- Fix a problem with `egrep` argument escaping ([issue 55](#)).
- Fix a problem with running `mkvirtualenv` without arguments ([issue 56](#)).

2.2.1

- Escape `which` calls to avoid aliases. Resolves [issue 46](#).
- Integrate Manuel Kaufmann's patch to unset `GREP_OPTIONS` before calling `grep`. Resolves [issue 51](#).
- Escape `$` in regex to resolve [issue 53](#).
- Escape `rm` to avoid issues with aliases and resolve [issue 50](#).

2.2

- Switched hook loader execution to a form that works with Python 2.4 to resolve [issue 43](#).
- Tested under Python 2.7b1. See [issue 44](#).
- Incorporated performance improvements from David Wolever. See [issue 38](#).
- Added some debug instrumentation for [issue 35](#).

2.1.1

- Added Spanish translation for the documentation via Manuel Kaufmann's fork at <http://bitbucket.org/humitos/virtualenvwrapper-es-translation/>
- Fixed improper use of python from \$PATH instead of the location where the wrappers are installed. See [issue 41](#).
- Quiet spurious error/warning messages when deactivating a virtualenv under zsh. See [issue 42](#).

2.1

- Add support for ksh. Thanks to Doug Latornell for doing the research on what needed to be changed.
- Test import of virtualenvwrapper.hook_loader on startup and report the error in a way that should help the user figure out how to fix it ([issue 33](#)).
- Update *mkvirtualenv* documentation to include the fact that a new environment is activated immediately after it is created ([issue 30](#)).
- Added hooks around *cpvirtualenv*.
- Made deactivation more robust, especially under ksh.
- Use Python's `tempfile` module for creating temporary filenames safely and portably.
- Fix a problem with `virtualenvwrapper_show_workon_options` that caused it to show * as the name of a virtualenv when no environments had yet been created.
- Change the hook loader so it can be told to run only a set of named hooks.
- Add support for listing the available hooks, to be used in help output of commands like `virtualenvwrapper.project's mkproject`.
- Fix `mkvirtualenv -h` option behavior.
- Change logging so the `$WORKON_HOME/hook.log` file rotates after 10KiB.

2.0.2

- Fixed [issue 32](#), making `virtualenvwrapper.user_scripts` compatible with Python 2.5 again.

2.0.1

- Fixed [issue 29](#), to use a default value for `TMPDIR` if it is not set in the user's shell environment.

2.0

- Rewrote hook management using `Distribute` entry points to make it easier to share extensions.

1.27

- Added `cpvirtualenv` command [Thomas Desvenain]

1.26

- Fix a problem with error messages showing up during init for users with the wrappers installed site-wide but who are not actually using them. See [issue 26](#).
- Split up the tests into multiple files.
- Run all tests with all supported shells.

1.25

- Merged in changes to cdsitepackages from William McVey. It now takes an argument and supports tab-completion for directories within site-packages.

1.24.2

- Add user provided *Tips and Tricks* section.
- Add link to Rich Leland's screencast to *References* section.

1.24.1

- Add license text to the header of the script.

1.24

- Resolve a bug with the preactivate hook not being run properly. Refer to [issue 21](#) for complete details.

1.23

- Resolve a bug with the postmkvirtualenv hook not being run properly. Refer to [issue 19](#) and [issue 20](#) for complete details.

1.22

- Automatically create any missing hook scripts as stubs with comments to expose the feature in case users are not aware of it.

1.21

- Better protection of \$WORKON_HOME does not exist when the wrapper script is sourced.

1.20

- Incorporate lssitepackages feature from Sander Smits.
- Refactor some of the functions that were using copy-and-paste code to build path names.
- Add a few tests.

1.19

- Fix problem with `add2virtualenv` and relative paths. Thanks to Doug Latornell for the bug report James Bennett for the suggested fix.

1.18.1

- Incorporate patch from Sascha Brossmann to fix a [issue 15](#). Directory normalization was causing `WORKON_HOME` to appear to be a missing directory if there were control characters in the output of `pwd`.

1.18

- Remove warning during installation if `sphinxcontrib.paverutils` is not installed. ([issue 10](#))
- Added some basic developer information to the documentation.
- Added documentation for `deactivate` command.

1.17

- Added documentation updates provided by Steve Steiner.

1.16

- Merged in changes to `cdvirtualenv` from wam and added tests and docs.
- Merged in changes to make error messages go to `stderr`, also provided by wam.

1.15

- Better error handling in `mkvirtualenv`.
- Remove bogus `VIRTUALENV_WRAPPER_BIN` variable.

1.14

- Wrap the `virtualenv` version of `deactivate()` with one that lets us invoke the `predeactivate` hooks.
- Fix `virtualenvwrapper_show_workon_options` for colorized versions of `ls` and write myself a note so I don't break it again later.
- Convert `test.sh` to use true tests with `shunit2`

1.13

- Fix [issue 5](#) by correctly handling symlinks and limiting the list of envs to things that look like they can be activated.

1.12

- Check return value of `virtualenvwrapper_verify_workon_home` everywhere, thanks to Jeff Forcier for pointing out the errors.
- Fix instructions at top of README, pointed out by Matthew Scott.
- Add `cdvirtualenv` and `cdsitepackages`, contributed by James Bennett.
- Enhance `test.sh`.

1.11

- Optimize `virtualenvwrapper_show_workon_options`.
- Add global `postactivate` hook.

1.10

- Pull in fix for colored `ls` from Jeff Forcier ([changeset b42a25f7b74a](#)).

1.9

- Add more hooks for operations to run before and after creating or deleting environments based on changes from Chris Hasenpflug.

1.8.1

- Corrected a problem with change to `mkvirtualenv` that lead to release 1.8 by using an alternate fix proposed by James in comments on release 1.4.

1.8

- Fix for processing the argument list in `mkvirtualenv` from jorgevargas ([issue 1](#))

1.7

- Move to `bitbucket.org` for hosting
- clean up TODO list and svn keywords
- add license section below

1.6.1

- More zsh support (fixes to `rmvirtualenv`) from Byron Clark.

1.6

- Add completion support for zsh, courtesy of Ted Leung.

1.5

- Fix some issues with spaces in directory or env names. They still don't really work with virtualenv, though.
- Added documentation for the postactivate and predeactivate scripts.

1.4

- Includes a new .pth management function based on work contributed by James Bennett and Jannis Leidel.

1.3.x

- Includes a fix for a nasty bug in rmvirtualenv identified by John Shimek.

CHAPTER 4

References

`virtualenv`, from Ian Bicking, is a pre-requisite to using these extensions.

For more details, refer to the column I wrote for the May 2008 issue of Python Magazine: [virtualenvwrapper | And Now For Something Completely Different](#).

Rich Leland has created a short [screencast](#) showing off the features of `virtualenvwrapper`.

Manuel Kaufmann has [translated this documentation into Spanish](#).

Tetsuya Morimoto has [translated this documentation into Japanese](#).

Join the [virtualenvwrapper Google Group](#) to discuss issues and features.

Report bugs via the [bug tracker on BitBucket](#).

Shell Aliases

Since `virtualenvwrapper` is largely a shell script, it uses shell commands for a lot of its actions. If your environment makes heavy use of shell aliases or other customizations, you may encounter issues. Before reporting bugs in the bug tracker, please test *without* your aliases enabled. If you can identify the alias causing the problem, that will help make `virtualenvwrapper` more robust.

CHAPTER 6

License

Copyright Doug Hellmann, All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Doug Hellmann not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

DOUG HELLMANN DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL DOUG HELLMANN BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.