virtualenvwrapper Documentation

Publicación 4.2.13.g041c411

Doug Hellmann

27 de September de 2017

Índice general

virtualenvwrapper es un conjunto de extensiones de la herramienta de Ian Bicking virtualenv. Las extensiones incluyen funciones para la creación y eliminación de entornos virtuales y por otro lado administración de tu rutina de desarrollo, haciendo fácil trabajar en más de un proyecto al mismo tiempo sin introducir conflictos entre sus dependencias.

CAPÍTULO 1

Características

1. Organiza todos tus entornos virtuales en un sólo lugar.
2. Funciones para administrar tus entornos virtuales (crear, eliminar, copiar).
3. Usa un sólo comando para cambiar entre los entornos.
4. Completa con Tab los comandos que toman un entorno virtual como argumento.
5. Ganchos configurables para todas las operaciones (ver *Personalizaciones por usuario*).
6. Sistema de plugins para la creación de extensiones compartibles (ver *Extender Virtualenvwrapper*).

CAPÍTULO 2

Introducción

La mejor forma de explicar las características que virtualenvwrapper brinda es mostrarlo en acción.

Primero, algunos pasos de inicialización. La mayoría de esto sólo necesita ser hecho una sola vez. Vas a querer agregar el comando `source /usr/local/bin/virtualenvwrapper.sh` al archivo de inicio de shell, cambiando el path hacia virtualenvwrapper.sh dependiendo en dónde haya sido instalado por 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 execu
(env1)$ ls $WORKON_HOME
env1 hook.log
```

Ahora podemos instalar algún software dentro del entorno.

```
(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
```

Podemos ver el nuevo paquete instalado con `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
```

Por supuesto que no estamos limitados a un sólo 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 execu
(env2)$ ls $WORKON_HOME
env1            env2            hook.log
```

Cambiar entre entornos con `workon`:

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

El comando `workon` también incluye la opción de completar con Tab los nombres de los entornos, e invoca a los scripts personalizados cuando un entorno es activado o desactivado (ver *Personalizaciones por usuario*).

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

postmkvirtualenv es ejecutado cuando un nuevo entorno es creado, dejándote instalar automáticamente herramientas comúnmente utilizadas.

```
(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
```

A través de una combinación de funciones existentes definidas por el *core* del paquete (ver *Referencia de comandos*), plugins de terceros (ver *Extender Virtualenvwrapper*), y scripts definidos por el usuario (ver *Personalizaciones por usuario*) virtualenvwrapper brinda una amplia variedad de oportunidades para automatizar tareas repetitivas.

CAPÍTULO 3

Detalles

Instalación

Shells soportados

virtualenvwrapper is un conjunto de *funciones* de shell definidas en una sintaxis compativle con shells Bourne. Sus test automatizados corren bajo cualquier de estos shells en OS X y Linux:

- `bash`
- `ksh`
- `zsh`

Quizás funcione con otros shells, si encuentras otro shell en dónde funcione que no está listado aquí, por favor házmelo saber. Si puedes modificarlo para que funcione en otro shell sin reescribirlo completamente, envíame una solicitud de pull a través de la página del proyecto en bitbucket. Si escribes un clon para trabajar con un shell incompatible, házmelo saber y voy a incluír el link desde ésta página.

Command Prompt de Windows

David Marble ha portado virtualenvwrapper a scripts batch de Windows, que pueden ser ejecutado en Microsoft Windows Command Prompt. Esto es a su vez una distribución separada de una re-implementación. Puedes descargar virtualenvwrapper-win desde PyPI.

MSYS

Es posible usar virtualenvwrapper en MSYS con una instalación nativa de Python para Windows. Para que funcione, debés definir una variable de entorno extra llamada `MSYS_HOME` que contenga la ruta hacia la instalación de MSYS

```
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
```

Dependiendo de tu configuración de MSYS, quizás necesites instalar el binario MSYS mktemp en la carpeta `MSYS_HOME/bin`.

PowerShell

Guillermo López-Anglada ha portado virtualenvwrapper para que corra en Microsoft PowerShell. Hemos aceptado que debido a que no es compatible con el resto de las extensiones, y que es en su mayoría una re-implementación (en vez de una adaptación), que debería ser distribuído separadamente. Puedes descargar virtualenvwrapper-powershell desde PyPI.

Versiones de Python

virtualenvwrapper está testeado bajo Python 2.6-3.3.

Instalación básica

virtualenvwrapper debe ser instalado en el mismo site-packages global dónde virtualenv está instalado. Quizás necesites privilegios de administrador para hacer eso. La forma más fácil de instalarlo es usando pip:

```
$ pip install virtualenvwrapper
```

or:

```
$ sudo pip install virtualenvwrapper
```

Advertencia: virtualenv te permite crear muchos entornos de Python diferentes. Deberías instalar virtualenv y virtualenvwrapper sólo en la instalación básica de Python de tu sistema (NO mientras un virtualenv esté activo) de modo que la misma versión sea compartida por todos los entornos de Python.

Una alternativa para instalarlo dentro del site-packages global es agregarlo al directorio local de tu usuario (normalmente *~/.local*)

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

Archivo de inicio del shell

Agrega estas tres líneas a tu archivo de inicio del shell (`.bashrc`, `.profile`, etc.) para configurar la ubicación dónde se van a guardar los entornos virtuales, el directorio dónde se van a guardar los proyectos y los scripts instalados con este paquete:

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

Después de editar este, recarga el archivo de inicio (por ejemplo, ejecuta: `source ~/.bashrc`).

Carga por demanda

Un script de inicialización alternativa es provisto para cargar virtualenvwrapper por demanda. En vez de incluir `virtualenvwrapper.sh` directamente, usa `virtualenvwrapper_lazy.sh`. Si

`virtualenvwrapper.sh` no está en tu `$PATH`, configura `VIRTUALENVWRAPPER_SCRIPT` para que apunte a él.

```
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
```

Advertencia: Cuando la versión de carga por demandan es usada, tab-completion de los argumentos y comandos de virtualenvwrapper (como nombres de entornos) no es habilitada hasta después de que el primer comando sea ejecutado. Por ejemplo, tab-completion de entornos no funciona para la primera instancia de *workon*.

Inicio rápido

1. Ejecuta: `workon`
2. Una lista de entornos, vacía, es impresa.
3. Ejecuta: `mkvirtualenv temp`
4. Un nuevo entorno, `temp` es creado y activado.
5. Ejecuta: `workon`
6. Esta vez, el entorno `temp` es incluido.

Configuraciones

virtualenvwrapper puede ser customizado cambiando variables de entorno. Configura las variable en el archivo de inicio *antes* de cargar `virtualenvwrapper.sh`.

Ubicación de los entornos

La variable `WORKON_HOME` le dice a virtualenvwrapper dónde alojar tus entornos virtuales. Por omisión es `$HOME/.virtualenvs`. Si el directorio no existe cuando virtualenvwrapper es cargado, éste será creado automáticamente.

Ubicación del los directorios de proyecto

La variable `PROJECT_HOME` le dice a virtualenvwrapper dónde se van a alojar los directorios de proyecto. La variable debe estar seteada y el directorio creado antes de que *mkproject* sea usado.

Ver también:

- *Administración de proyectos*

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`.

Ver también:

- *Administración de proyectos*

Ubicación de los scripts de gancho

La variable `VIRTUALENVWRAPPER_HOOK_DIR` le dice a virtualenvwrapper dónde van a ser guardados los *user-defined hooks*. El lugar por omisión es `$WORKON_HOME`.

Ver también:

- *Personalizaciones por usuario*

Ubicación de los logs de los ganchos

La variable `VIRTUALENVWRAPPER_LOG_FILE` le indica a virtualenvwrapper dónde deben ser escritos los logs para los scripts de gancho. El lugar por omisión es no logear desde los ganchos.

Intérprete de Python, virtualenv y $PATH

Durante el inicio, `virtualenvwrapper.sh` busca el primer `python` y `virtualenv` en la variable `$PATH` y recuerda éste para su posterior uso. Esto elimina cualquier conflicto con los cambios en `$PATH`, permitiendo intérpretes dentro de entornos en los cuales virtualenvwrapper no está instalado. Debido a este comportamiento, es importante configurar la variable `$PATH` **antes** de hacer la inclusión de `virtualenvwrapper.sh` (mediante `source`). Por ejemplo:

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

Para reemplazar la búsqueda en `$PATH`, se puede configurar la variable `VIRTUALENVWRAPPER_PYTHON` hacia la ruta absoluta del intérprete a usar y `VIRTUALENVWRAPPER_VIRTUALENV` hacia la ruta absoluta del binario de `virtualenv` a usar. Ambos deben ser seteadas **antes** de incluir `virtualenvwrapper.sh`). Por ejemplo:

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

Argumentos por omision para virtualenv

Si la aplicación identificada por `VIRTUALENVWRAPPER_VIRTUALENV` necesita argumentos, ellos pueden ser configurados en `VIRTUALENVWRAPPER_VIRTUALENV_ARGS`. La misma variable puede ser usada para configurar los argumentos que van a ser pasados a `virtualenv`. Por ejemplo, configurar su valor a `--no-site-packages` para asegurarse que los nuevos entornos estarán aislados del directorio `site-packages` del sistema.

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

Archivos temporales

virtualenvwrapper crea archivos temporales en `$TMPDIR`. Si la variable no está configurada, este usa `/tmp`. Para cambiar la ubicación de los archivos temporales sólo para virtualenvwrapper, configura `VIRTUALENVWRAPPER_TMPDIR`.

Configuración global

La mayoría de los sistemas UNIX tienen la habilidad de cambiar las configuraciones para todos los usuarios. Ésto típicamente toma una de dos formas: editar los archivos *skeleton* para nuevas cuentas o editar el archivo globar de startup para el shell.

Editar los archivos skeleton para nuevas cuentas significa que cada nuevo usuario tendrá sus archivos de inicio preconfigurados para cargar virtualenvwrapper. Ellos pueden deshabilitarlo comentando or quitando esas líneas. Vaya a la documentación del shell y el sistema operativo para identificar cuáles son los archivos apropiados para editar.

Modificar los archivos globales de startup para un shell dado significa que todos los usuarios de ese shell tendrán virtualenvwrapper habilitado y no lo podrán deshabilitar. Vaya a la documentación del shell para identificar cuáles son los archivos apropiados para editar.

Actualizar a 2.9

La versión 2.9 incluye las características anteriormente distribuídas de forma separada por `virtualenvwrapper.project`. Si tienes una versión vieja de la extensión project instalada, elimínalas antes de actualizar.

Actualizar desde 1.x

El script de shell que contiene las funciones ha sido renombrado en la serie 2.x para reflejar el hecho de que otros shells, además de bash, son soportados. En tu archivo de inicio del shell, cambia `source /usr/local/bin/virtualenvwrapper_bashrc` por `source /usr/local/bin/virtualenvwrapper.sh`.

Referencia de comandos

Todos los comandos, mostrados a continuación, son para ser utilizados en una Terminal de línea de comandos.

Administrar entornos

mkvirtualenv

Crea un nuevo entorno, dentro de WORKON_HOME.

Sintaxis:

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

Todas las opciones de línea de comandos excepto `-a`, `-i`, `-r`, y `-h` son pasados directamente a `virtualenv`. El nuevo entorno es automáticamente activado luego de su inicialización.

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

La opción `-a` puede ser usada para asociar un directorio de proyecto existente con el nuevo entorno.

La opción `-i` puede ser usada para instalar uno o más paquetes (repitiendo la opción) luego que el entorno sea creado.

La opción `-r` puede ser usada para especificar un archivo de texto con la lista de paquetes a ser instalados. El valor del argumento es pasado a `pip -r` para que sean instalados.

Ver también:

- *premkvirtualenv*
- *postmkvirtualenv*
- requirements file format

mktmpenv

Crea un nuevo virtualenv en el directorio `WORKON_HOME`.

Sintaxis:

```
mktmpenv [VIRTUALENV_OPTIONS]
```

Un nombre único es generado para el virtualenv.

```
$ 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

Lista todos los entornos.

Sintaxis:

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

-b	Modo breve, deshabilita la salida verbosa.
-l	Modo largo, habilita la salida verbosa. Por defecto.
-h	Imprime la ayuda de lsvirtualenv.

Ver también:

- *get_env_details*

showvirtualenv

Muestra los detalles de un virtualenv.

Syntax:

```
showvirtualenv [env]
```

Ver también:

- *get_env_details*

rmvirtualenv

Elimina un entorno, dentro de WORKON_HOME.

Sintaxis:

```
rmvirtualenv ENVNAME
```

Debes usar *deactivate* antes de eliminar el entorno actual.

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

Ver también:

- *prermvirtualenv*
- *postrmvirtualenv*

cpvirtualenv

Duplica un entorno virtualenv existente. El entorno de origen puede ser un entorno manejado con virtualenvwrapper o uno externo creado en otro lugar.

Advertencia: Copiar un entorno virtual no está del todo soportado. Cada entorno virtual tiene información sobre los paths hardcodeado dentro de él, y quizás el código copiado no sepa actualizar un archivo en particular. **Usar con cuidado.**

Sintaxis:

```
cpvirtualenv ENVNAME [TARGETENVNAME]
```

Nota: El nombre de entorno de destino es necesario para duplicar un entorno existente en WORKON_HOME. Sin embargo, éste puede ser omitido para importar entornos externos. Si se omite, el nuevo entorno tendrá el mismo nombre que el 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
Script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/postdeactivate cannot be made relative
Script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/preactivate cannot be made relative (it'
Script /Users/dhellmann/Devel/virtualenvwrapper/tmp/dest/bin/predeactivate cannot be made relative (i
(dest)$ workon
dest
source
(dest)$
```

Ver también:

- *precpvirtualenv*
- *postcpvirtualenv*
- *premkvirtualenv*
- *postmkvirtualenv*

allvirtualenv

Ejecuta un comando dentro de todos los entornos virtuales bajo WORKON_HOME.

Sintaxis:

```
allvirtualenv command with arguments

Cada entorno virtual es activado, ejecutando los ganchos de activación,
el directorio de trabajo actual es cambiado al entorno virtual activado
y el comando es ejecutado. Los comandos no pueden modificar el estado del
shell actual, pero pueden modificar el entorno virtual.

::

  $ allvirtualenv pip install -U pip
```

Controlar los entornos activos

workon

Lista o cambia el entorno de trabajo actual

Sintaxis:

```
workon [environment_name]
```

Si no se especifica el `environment_name`, la lista de entornos disponibles es impresa en la salida estándar.

```
$ 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 $VIRTUAL_ENV
/Users/dhellmann/Devel/virtualenvwrapper/tmp/env2
(env2)$
```

Ver también:

- *predeactivate*
- *postdeactivate*
- *preactivate*
- *postactivate*

deactivate

Cambia de un entorno virtual a la versión instalada de Python en el sistema.

Sintaxis:

```
deactivate
```

Nota: Este comando es actualmente parte de virtualenv, pero es encapsulado para proveer ganchos antes y después, al igual que workon hace para *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

$
```

Ver también:

- *predeactivate*
- *postdeactivate*

Rápida navegación dentro de virtualenv

Existen dos funciones que proveen atajos para navegar dentro del virtualenv actualmente activado.

cdvirtualenv

Cambia el directorio de trabajo actual hacia `$VIRTUAL_ENV`.

Sintaxis:

```
cdvirtualenv [subdir]
```

Al llamar `cdvirtualenv` se cambia el directorio de trabajo actual hacia la sima de virtualenv (`$VIRTUAL_ENV`). Un argumento adicional es agregado a la ruta, permitiendo navegar directamente dentro de un subdirectorio.

```
$ 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

Cambia el directorio de trabajo actual al `site-packages` del `$VIRTUAL_ENV`.

Sintaxis:

```
cdsitepackages [subdir]
```

Debido a que la ruta exacta hacia el directorio site-packages dentro del virtualenv depende de la versión de Python, `cdsitepackages` es provisto como un atajo para `cdvirtualenv lib/python${pyvers}/site-packages`. Un argumento opcional también está permitido, para especificar un directorio heredado dentro del directorio `site-packages` y así ingresar a este.

```
$ 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

`lssitepackages` muestra el contenido del directorio `site-packages` del entorno actualmente activado.

Sintaxis:

```
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
```

Administración de rutas

add2virtualenv

Agrega los directorios especificados al path de Python para el entorno virtual actualmente activo.

Sintaxis:

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

A veces esto es útli para compartir paquetes instalados que no están en el directorio `site-packages` del sistema y no deben ser instalados en cada entorno virtual. Una posible solución es crear enlaces simbólicos (*symlinks*) hacia el código dentro del directorio `site-packages` del entorno, pero también es fácil agregar a la variable PYTHONPATH directorios extras que están incluidos en los archivos `.pth` dentro de `site-packages` usando `add2virtualenv`.

1. Descarga (*check out*) el código de un proyecto grande, como Django.
2. Ejecuta: `add2virtualenv path_to_source`.
3. Ejecuta: `add2virtualenv`.
4. Un mensaje de uso y una lista de las rutas "extras" actuales es impreso.

Los nombres de los directorios son agregados a un archivo llamado `_virtualenv_path_extensions.pth` dentro del directorio site-packages de este entorno.

Basado en una contribución de James Bennett y Jannis Leidel.

toggleglobalsitepackages

Controla si el virtualenv activo tendrá acceso a los paquetes en el directorio `site-packages` global de Python.

Sintaxis:

```
toggleglobalsitepackages [-q]
```

Muestra el nuevo estado del virtualenv. Usa la opción `-q` para apagar la salida por pantalla.

```
$ 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)$
```

Administración de directorios de proyecto

Ver también:

Administración de proyectos

mkproject

Crea un nuevo virtualenv en WORKON_HOME y el directorio del proyecto en PROJECT_HOME.

Sintaxis:

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

-f, --force Crea un entorno virtual incluso si el directorio del proyecto ya existe

La opción template puede repetirse varias veces para utilizar diferentes templates en la creación del proyecto. Los templates son aplicados en el orden mencionados en la línea de comandos. Todas las otras opciones son pasadas a `mkvirtualenv` para crear un virtualenv con el mismo nombre que el proyecto.

```
$ 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 $VIRTUAL_ENV
/Users/dhellmann/Envs/myproj
(myproj)$
```

Ver también:

- *premkproject*
- *postmkproject*

setvirtualenvproject

Asocia un virtualenv existente a un proyecto existente.

Sintaxis:

```
setvirtualenvproject [virtualenv_path project_path]
```

Los argumentos de `setvirtualenvproject` son paths absolutos hacia el virtualenv y el directorio del proyecto. Una asociación es hecha para que cuando `workon` active el virtualenv también active el proyecto.

```
$ 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)
```

Cuando ningún argumento es pasado, se asume el virtualenv y el directorio activo.

Cualquier número de entornos virtuales puede referirse al mismo directorio de proyecto, haciendo fácil cambiar entre versiones de Python o otras dependencias necesarias para testing.

cdproject

Cambia el directorio de trabajo actual al especificado como directorio del proyecto para el virtualenv activo.

Sintaxis:

```
cdproject
```

Manegar paquetes instalados

wipeenv

Elimina todos los paquetes de terceros instalados en el entorno virtual actual.

Syntax:

```
wipeenv
```

Otros comandos

virtualenvwrapper

Imprime una lista de comandos y sus descripciones como una ayuda básica.

Sintaxis:

```
virtualenvwrapper
```

Personalizar Virtualenvwrapper

virtualenvwrapper agrega varios ganchos que puedes usar para cambiar tus configuraciones, el entorno del shell, u otras configuraciones al crear, eliminar o cambiar entre entornos. Estos ganchos son expuestos en dos formas:

Personalizaciones por usuario

Los scripts de personalización de usuarios finales son *incluidos* uno por uno (permitiéndoles modificar su entorno de shell) o *ejecutados* como un programa externo en el momento apropiado.

Los scripts aplicados globalmente para todos los entornos deben ser ubicados en el directorio llamado *VIRTUALENVWRAPPER_HOOK_DIR*. Los scripts locales deben ser ubicados en el directorio `bin` del virtualenv.

get_env_details

Global/Local ambos

Argumento(s) env name

Incluido/Ejecutado ejecutado

`$VIRTUALENVWRAPPER_HOOK_DIR/get_env_details` es ejecutado cuando `workon` es ejecutado sin argumentos y una lista de entornos virtuales es impresa en pantalla. El gancho es ejecutado una vez por entorno, luego de que el nombre sea impreso, y puede imprimir información adicional sobre ese entorno.

initialize

Global/Local global

Argumento(s) ninguno

Incluido/Ejecutado incluido

`$VIRTUALENVWRAPPER_HOOK_DIR/initialize` es incluido cuando `virtualenvwrapper.sh` es cargado dentro de tu entorno. Usa este para ajustar configuraciones globales cuando virtualenvwrapper es habilitado.

premkvirtualenv

Global/Local global

Argumento(s) nombre de un nuevo virtualenv

Incluido/Ejecutado ejecutado

`$VIRTUALENVWRAPPER_HOOK_DIR/premkvirtualenv` es ejecutado como un programa externo luego que de un entorno virtual es creado pero antes de que el entorno actual sea cambiado para apuntar al nuevo entorno. El directorio de trabajo actual para este script es `$WORKON_HOME` y el nombre del nuevo entorno es pasado como argumento al script.

postmkvirtualenv

Global/Local global

Argumento(s) ninguno

Incluido/Ejecutado incluido

`$VIRTUALENVWRAPPER_HOOK_DIR/postmkvirtualenv` es incluido después de que un nuevo entorno es creado y activado. Si la opción `-a` <ruta_del_proyecto> es usada, el enlace hacia el directorio del proyecto es hecho antes de que el script sea incluido.

precpvirtualenv

Global/Local global

Argumento(s) nombre del entorno original, nombre del nuevo entorno

Incluido/Ejecutado ejecutado

`$VIRTUALENVWRAPPER_HOOK_DIR/precpvirtualenv` es ejecutado como un programa externo luego de que un entorno es duplicado y hecho reubicable, pero antes de que `premkvirtualenv` sea ejecutado o se haya cambiado al nuevo entorno creado. El directorio de trabajo actual para este script es `$WORKON_HOME` y los nombres del entorno original y el nuevo son pasados como argumento al script.

postcpvirtualenv

Global/Local global

Argumento(s) ninguno

Incluido/Ejecutado incluido

`$VIRTUALENVWRAPPER_HOOK_DIR/postcpvirtualenv` es incluido luego de que un nuevo entorno es creado y activado.

preactivate

Global/Local global, local

Argumento(s) nombre de entorno

Incluido/Ejecutado ejecutado

El script global `$VIRTUALENVWRAPPER_HOOK_DIR/preactivate` es ejecutado antes de que el nuevo entorno sea habilitado. El nombre de entorno es pasado como primer argumento.

El gancho `$VIRTUAL_ENV/bin/preactivate` es ejecutado antes de que el nuevo entorno sea habilitado. El nombre del entorno es pasado como primer argumento.

postactivate

Global/Local global, local

Argumento(s) ninguno

Incluido/Ejecutado incluido

El script global `$VIRTUALENVWRAPPER_HOOK_DIR/postactivate` es incluido luego de que el nuevo entorno sea habilitado. `$VIRTUAL_ENV` hace referencia al nuevo entorno al momento en el que se ejecuta el script.

Este script de ejemplo añade un espacio entre el nombre del entorno virtual y la tu variable PS1 haciendo uso de `_OLD_VIRTUAL_PS1`.

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

El script local `$VIRTUAL_ENV/bin/postactivate` es incluido luego de que el nuevo entorno es habilitado. `$VIRTUAL_ENV` hace referencia al nuevo entorno al momento en el que el script es ejecutado.

Este script de ejemplo para el entorno PyMOTW cambia el directorio de trabajo actual y la referencia de la variable PATH hacia el árbol que contiene el código de PyMOTW.

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

predeactivate

Global/Local local, global

Argumento(s) ninguno

Incluido/Ejecutado incluido

El script local `$VIRTUAL_ENV/bin/predeactivate` es incluido antes de que el entorno actual sea desactivado, y puede ser usado para deshabilitar o limpiar configuraciones en tu entorno. `$VIRTUAL_ENV` hace referencia al entorno viejo al momento de ejecutar este script.

El script global `$VIRTUALENVWRAPPER_HOOK_DIR/predeactivate` es incluido antes de que el entorno actual sea desactivado. `$VIRTUAL_ENV` hace referencia al entorno viejo al momento de ejecutar este script.

postdeactivate

Global/Local local, global

Argumento(s) ninguno

Incluido/Ejecutado incluido

El script `$VIRTUAL_ENV/bin/postdeactivate` es incluido luego de que el entorno actual sea desactivado, y puede ser usado para deshabilitar o limpiar configuraciones en tu entorno. El path hacia el entorno que recientemente se ha desactivado está disponible en `$VIRTUALENVWRAPPER_LAST_VIRTUALENV`.

prermvirtualenv

Global/Local global

Argumento(s) nombre de entorno

Incluido/Ejecutado ejecutado

EL script `$VIRTUALENVWRAPPER_HOOK_DIR/prermvirtualenv` es ejecutado como un programa externo antes de que el entorno sea eliminado. El path absoluto hacia el entorno es pasado como argumento al script.

postrmvirtualenv

Global/Local global

Argumento(s) nombre de entorno

Incluido/Ejecutado ejecutado

El script `$VIRTUALENVWRAPPER_HOOK_DIR/postrmvirtualenv` es ejecutado como un programa externo luego de que el entorno sea eliminado. El path absoluto hacia el directorio del entorno es pasado como argumento al script.

premkproject

Global/Local global

Argumento(s) nombre del nuevo proyecto

Incluido/Ejecutado ejecutado

`$WORKON_HOME/premkproject` es ejecutado como un programa externo luego de que el entorno virtual es creado y luego de que el entorno actual es cambiado al nuevo entorno, pero antes de que el nuevo directorio de proyecto sea creado. El directorio de trabajo actual para el script es `$PROJECT_HOME` y el nombre del nuevo proyecto es pasado como argumento al script.

postmkproject

Global/Local global

Argumento(s) ninguno

Incluido/Ejecutado incluido

`$WORKON_HOME/postmkproject` es incluido luego de que el nuevo entorno y directorio de proyecto son creados y el virtualenv es activado. El directorio de trabajo es el directorio del proyecto.

Extender Virtualenvwrapper

Una gran experiencia con soluciones caseras para personalizar un entorno de desarrollo ha demostrado cuán valioso puede ser tener la capacidad de automatizar tareas comunes y eliminar molestias persistentes. Carpinteros construyen plantillas de guía, desarrolladores de software escriben scripts de shell. virtualenvwrapper continúa la tradición de animar a un artesano a modificar sus herramientas para trabajar de la manera que ellos quieran, en vez de al revés.

Usa los ganchos provistos para eliminar operaciones manuales repetitivas y hacer más simple tu flujo de desarrollo. Por ejemplo, configura los ganchos *pre_activate* y *post_activate* para provocar que un IDE cargue un proyecto o recargue los archivos desde la última sesión de edición, administra el logueo de horas, o inicia y detiene versiones de desarrollo de un servidor de aplicaciones. Usa el gancho *initialize* para agregar nuevos comandos y ganchos a virtualenvwrapper. Los ganchos *pre_mkvirtualenv* y *post_mkvirtualenv* te brindan la oportunidad de instalar requerimientos básicos dentro de cada nuevo entorno de desarrollo, inicializar el repositorio de control de versiones para el código, o por otro lado configurar un nuevo proyecto.

Existen dos maneras para adjuntar tu código para que virtualenvwrapper lo ejecute: los usuarios finales pueden usar scripts de shell o otros programas para la personalización personal (ver *Personalizaciones por usuario*). Las extensiones también pueden ser implementadas en Python usando *puntos de entrada* con Distribute ,

Definir una extensión

Nota: Virtualenvwrapper es distribuido con un plugin para la creación y ejecución de los scripts de personalización de los usuarios (*user_scripts*). Los ejemplos siguientes han sido tomados de la implementación de ese plugin.

Organización del código

El paquete Python para `virtualenvwrapper` es un *namespace package*. Eso significa que múltiples librerías pueden instalar código dentro del paquete, incluso si ellas no son distribuidas juntas o instaladas dentro del mismo directorio. Las extensiones pueden (opcionalmente) usar el namespace de `virtualenvwrapper` configurando su estructura de directorios así:

- virtualenvwrapper/
  - __init__.py
  - user_scripts.py

Y agregando el siguiente código dentro de `__init__.py`:

```
"""virtualenvwrapper module
"""

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

Nota: Las extensiones pueden ser cargadas desde cualquier paquete, así que usar el espacio de nombres de `virtualenvwrapper` no es requerido.

Extensión API

Después de que el paquete está establecido, el siguiente paso es crear un módulo para alojar el código de la extensión. Por ejemplo, `virtualenvwrapper/user_scripts.py`. El módulo debe contener la extensión actual a los *entry points*. Soporte de código puede ser incluido, o importado desde algún lugar usando la técnica de organización de código estándar de Python.

FIXME: I don't like the last paragraph

La API es la misma para todos los puntos de extensión. Cada uno usa una función de Python que toma un sólo argumento, una lista de strings pasada al script que carga los ganchos en la línea de comandos.

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

El contenido de la lista de argumentos está definida para cada punto de extensión a continuación (ver *Puntos de extensión*).

Invocación de la extensión

Acción directa Los plugins pueden ser colgados a cada uno de los ganchos de dos formas diferentes. La estándar es tener una función y hacer algún trabajo directamente. Por ejemplo, la función `initialize()` para el plugin de los scripts de usuarios crea scripts de usuarios por default cuando `virtualenvwrapper.sh` es cargada.

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

Modificar el entorno de usuario Hay casos en dónde la extensión necesita actualizar el entorno del usuario (por ejemplo, cambiar el directorio de trabajo actual o configurar variables de entorno). Las modificaciones al entorno del usuario deben ser hechas dentro del shell actual del usuario, y no pueden ser ejecutadas en un proceso separado. Para tener código ejecutado en un proceso shell del usuario, las extensiones pueden definir funciones gancho y retornar el texto de los comandos de shell a ser ejecutados. Estos ganchos *fuente* son ejecutados después de los ganchos comunes con el mismo nombre, y no deben hacer ningún trabajo por ellos mismos.

El gancho `initialize_source()` para el plugin de scripts de usuarios busca un script `initializa` global y causa que este sea ejecutado en el proceso de shell actual.

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

Advertencia: Como las extensiones están modificando el shell de trabajo del usuario, se debe tener mucho cuidado de corromper el entorno sobreescribiendo variables con valores inesperados. Evita crear variables temporales cuando sea posible. Poner prefijos a las variables con el nombre de la extensión es una buena forma de manejar espacios de nombres. Por ejemplo, en vez de `temp_file` usa `user_scripts_temp_file`. Usa `unset` para liberar nombres de variables temporales cuando no sean más necesarias.

Advertencia: virtualenvwrapper funciona en varios shells con una sintaxis ligeramente diferente (bash, sh, zsh, ksh). Ten en cuenta esta portabilidad cuando definas ganchos incluidos (*sourced hooks*). Mantener la sintaxis lo más simple posible evitará problemas comunes, pero quizás haya casos donde examinar la variable de entorno `SHELL` y generar diferente sintaxis para cada caso sea la única manera de alcanzar el resultado deseado.

Registrar puntos de entrada

Las funciones definidas en el plugin necesitan ser registradas como *puntos de entrada* para que el cargador de ganchos de virtualenvwrapper los encuentre. Los puntos de entrada de Distribute se configuran en el `setup.py` de tu paquete coincidiendo el nombre del punto de entrada con la función en el paquete que lo implementa.

Una copia parcial del `setup.py` de virtualenvwrapper ilustra cómo los puntos de entrada `initialize()` y `initialize_source()` son configurados.

```
# 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 ...
        },
    )
```

El argumento `entry_points` de `setup()` es un diccionario que mapea los *grupos de nombre* de puntos de entrada a listas de puntos de entrada específicos. Un nombre de grupo diferente es definido por virtualenvwrapper por cada punto de extensión (ver *Puntos de extensión*).

Los identificadores de puntos de entrada son strings con la sintaxis `name = package.module:function`. Por convención, el *nombre* de cada punto de entrada es el nombre del plugin, pero esto no es requerido (los nombres no son usados).

Ver también:

- namespace packages
- Extensible Applications and Frameworks

El cargador de ganchos

Las extensiones son ejecutadas mediante una aplicación de líneas de comando implementada en `virtualenvwrapper.hook_loader`. Como `virtualenvwrapper.sh` es invocado primero y los usuarios generalmente no necesitan ejecutar la aplicación directamente, ningún otro script es instalado por separado. En vez, para ejecutar la aplicación, usa la opción `-m` del intérprete:

```
$ 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
```

Para ejecutar las extensiones para el gancho *initialize*:

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

Para obtener los comandos de shell para el gancho *initialize*:

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

En la práctica, en vez de invocar al cargador de ganchos directamente es conveniente usar la función de shell, `virtualenvwrapper_run_hook` para ejecutar los ganchos en ambos modos.:

```
$ virtualenvwrapper_run_hook initialize
```

Todos los argumentos pasados a la función de shell son pasados directamente al cargador de ganchos.

Registro (*Logging*)

El cargador de ganchos configura el registro para que los mensajes sean escritos en `$WORKON_HOME/hook.log`. Los mensajes quizás sean escritos en stderr, dependiendo de la flash verbose. Por default los mensajes con un nivel mayor o igual a *info* se escriben en stderr, y los de nivel *debug* o mayor van al archivo de registro. Usar el registro de esta forma provee un mecanismo conveniente para que los usuarios controlen la verbosidad de las extensiones.

Para usar el registro en tu extensión, simplemente instancia un registro y llama a sus métodos `info()`, `debug()` y otros métodos de mensajería.

```
import logging
log = logging.getLogger(__name__)

def pre_mkvirtualenv(args):
```

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

Ver también:

- Standard library documentation for logging
- PyMOTW for logging

Puntos de extensión

Los nombres de los puntos de extensión para los plugins nativos siguen una convención con varias partes: `virtualenvwrapper.(pre|post)_<event>[_source]`. *<event>* es la acción tomada por el usuario o virtualenvwrapper que provoca la extensión. `(pre|post)` indica si llama a la extensión antes o después de un evento. El sufijo `_source` es agregado para las extensiones que retornan código shell en vez de tomar una acción directamente (ver *Modificar el entorno de usuario*).

get_env_details

Los ganchos `virtualenvwrapper.get_env_details` son ejecutados cuando `workon` es ejecutado sin argumentos y una lista de entornos virtuales es impresa en pantalla. El gancho es ejecutado una vez para cada entorno, luego de que el nombre sea impreso, y puede ser utilizado para mostrar información adicional sobre ese entorno.

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

Los ganchos `virtualenvwrapper.initialize` son ejecutados cada vez que `virtualenvwrapper.sh` es cargado en el entorno del usuario. El gancho *initialize* puede ser usado para instalar plantillas para configurar archivos o preparar el sistema para una operación correcta del plugin.

pre_mkvirtualenv

Los ganchos `virtualenvwrapper.pre_mkvirtualenv` son ejecutados después de que el entorno es creado, pero antes de que el nuevo entorno sea activado. El directorio de trabajo actual para cuando el gancho es ejecutado es `$WORKON_HOME` y el nombre del nuevo entorno es pasado como un argumento.

post_mkvirtualenv

Los ganchos `virtualenvwrapper.post_mkvirtualenv` son ejecutado después de que un nuevo entorno sea creado y activado. `$VIRTUAL_ENV` es configurado para apuntar al nuevo entorno.

pre_activate

Los ganchos `virtualenvwrapper.pre_activate` son ejecutados justo antes de que un entorno sea activado. El nombre del entorno es pasado como primer argumento.

post_activate

Los ganchos `virtualenvwrapper.post_activate` son ejecutados justo después de que un entorno sea activado. `$VIRTUAL_ENV` apunta al entorno actual.

pre_deactivate

Los ganchos `virtualenvwrapper.pre_deactivate` son ejecutados justo antes de que un entorno sea desactivado. `$VIRTUAL_ENV` apunta al entorno actual.

post_deactivate

Los ganchos `virtualenvwrapper.post_deactivate` son ejecutados justo después de que un entorno sea desactivado. El nombre del entorno recién desactivado es pasado como primer argumento.

pre_rmvirtualenv

Los ganchos `virtualenvwrapper.pre_rmvirtualenv` son ejecutados justo antes de que un entorno sea eliminado. El nombre del entorno eliminado es pasado como primer argumento.

post_rmvirtualenv

Los ganchos `virtualenvwrapper.post_rmvirtualenv` son ejecutados justo después de que un entorno haya sido eliminado. El nombre del entorno eliminado es pasado como primer argumento.

Agregar nuevos puntos de extensión

Los plugins que definen nuevas operaciones pueden también definir nuevos puntos de extensión. No es necesario hacer ninguna configuración para permitir que el cargador de ganchos encuentre las extensiones; documentar los nombres y agregar llamadas a `virtualenvwrapper_run_hook` es suficiente para causar que ellos se invoquen.

El cargador de ganchos asume que todos los nombres de puntos de extensión comienzan con `virtualenvwrapper.` y los nuevos plugins querrán usar su propio espacio de nombres para agregar. Por ejemplo, la extensión project define nuevos eventos para crear directorios del proyecto (pre y post). Esas son llamadas a `virtualenvwrapper.project.pre_mkproject` y `virtualenvwrapper.project.post_mkproject`. Estas son invocadas con:

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

y:

```
virtualenvwrapper_run_hook project.post_mkproject
```

respectivamente.

Administración de proyectos

El *directorio de proyecto* está asociado con un virtualenv, pero generalmente contiene el código fuente que se encuentra bajo desarrollo en vez de los componentes necesarios instalados para desarrollar. Por ejemplo, el directorio de proyecto

puede contener el código fuente obtenido de un sistema de control de versiones, información temporal creada para testing, archivos experimentales aún no commiteados, etc.

Un directorio de proyecto es creado y asociado a un virtualenv cuando es ejecutado *mkproject* en vez de *mkvirtualenv*. Para asociar un directorio de proyecto existente a un virtualenv, usa *setvirtualenvproject*.

Usar templates

Un nuevo directorio de proyecto puede ser creado vacío o llenado usando una o más extensiones *template*. Los templates deben ser especificados como argumentos al comando *mkproject*. Multiples valores pueden ser provistos para aplicar más de un template. Por ejemplo, para obtener un repositorio Mercurial de un proyecto de bitbucket y crear un nuevo sitio Django, se pueden combinar los templates *bitbucket* y *django*

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

Ver también:

- *Templates*
- *Ubicación del los directorios de proyecto*
- *Project Linkage Filename*

Consejos y Trucos

Esta es una lista de contribuciones de usuarios para hacer virtualenv y virtualenvwrapper incluso más útil. Si tienes tips para compartir, envíame un email o deja un comentario en esta entrada de mi blog y lo agregaré aquí.

Prompt zsh

De Nat:

Usando zsh, agregué algunas líneas a `$WORKON_HOME/post(de)activate` para mostrar el virtualenv activo en el lado derecho de la pantalla en vez de a la izquierda.

En `postactivate`:

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

Agrega en `postdeactivate`:

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

Ajusta los colores de acuerdo a tu gusto personal o a tu entorno.

Actualizar las entradas de `$PATH` cacheadas

De Nat:

También agregué el comando 'rehas' a `$WORKON_HOME/postactivate` y `$WORKON_HOME/postdeactivate` porque estaba teniendo algunos problemas con zsh ya que no actualizaba los paths inmediatamente.

Atar el soporte para virtualenv de pip

Vía http://becomingguru.com/:

Agrega esto al script de login de tu shell para indicarle a pip que use el mismo directorio para virtualenv que para virtualenwrapper:

```
export PIP_VIRTUALENV_BASE=$WORKON_HOME
```

y Vía Nat:

además de lo que dijo becomingguru, esta línea es clave:

```
export PIP_RESPECT_VIRTUALENV=true
```

Eso hace que pip detecte un virtualenv activo e instale dentro de este, sin pasar el parámetro -E.

Crear los directorio para trabajar en el proyecto

Vía James:

En el script `postmkvirtualenv` tengo lo siguiente para crear un directorio basado en el nombre del proyecto, agregar ese directorio la path de python y luego ingresar a este:

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

En el script `postactivate` tengo configurado para que automáticamente ingrese a este directorio cuando uso el comando workon:

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

Ejecutar automáticamente workon cuando se ingresa a un directorio

Justin Lily escribió un post sobre algún código que agrego a su entorno de shell para buscar en el directorio cada vez que se ejecuta `cd`. Si este encuentra un archivo llamado `.venv`, activa el entorno nombrado dentro. Una vez que se deja el directorio, el virtualenv actual es automáticamente desactivado.

Harry Marr escribió una función similar que funciona con repositorios git.

Instalar herramientas comunes automáticmante en nuevos entornos

Vía rizumu:

Tengo esto en postmkvirtualenv para instalar una configuración básica.

```
$ 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
```

Además, tengo un archivo de requerimiento de pip para instalar mis herramientas de desarrollo.

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

Entonces, cada proyecto tiene su propio archivo de requerimientos para cosas como PIL, psycopg2, django-apps, numpy, etc.

Cambiar el comportamiento por default de `cd`

Vía mae:

Esto se supone que es ejecutado después de workon, es como un gancho `postactivate`. Basicamente sobreescribe `cd` para saber sobre VENV entonces en vez de hacer `cd` para ir a ~ irá al root del venv, creo que es muy práctico y no puedo vivir más sin esto. Si le pasas un path apropiado entonces hará lo correcto.

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

cd
```

Para desarrolladores

Si quieres contribuir con virtualenvwrapper directamente, estas instrucciones deberían ayudarte a empezar. Parches, reporte de bugs, y propuestas de características son todas bienvenidas a través del sitio de BitBucket. Contribuciones en la forma de parches o solicitud de *pull* son fáciles de integrar y recibirán prioridad en la atención.

Nota: Antes de contribuir con nuevas características al *core* de virtualenvwrapper, por favor considera, en vez, si no debe ser implementada como una extensión.

Construir la documentación

La documentación para virtualenvwrapper está escrita en reStructuredText y convertida a HTML usando Sphinx. La propia construcción es impulsada por make. Necesitas los siguientes paquetes para construir la documentación:

- Sphinx
- docutils
- sphinxcontrib-bitbucket

Una vez que todas las herramientas están instaladas dentro de un virtualenv usando pip, ejecuta `make html` para generar la versión de HTML de la documentación:

```
$ 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/d
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
```

La versión de publicación de la documentación termina dentro de `./virtualenvwrapper/docs`

Ejecutar tests

La suite de test de virtualenvwrapper usa shunit2 y tox. El código de shunit2 está incluído en el directorio `tests`, pero tox debe ser instalado aparte (`pip install tox`).

Para ejecutar los test en bash, zsh, ksh para Python 2.4 a 2.7, ejecuta `tox` en la cima directorio del repositorio hg.

Para ejecutar un test de un script individual, usa un comando como:

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

Para ejecutar los tests bajo un sola versión de Python, especifica el entorno apropiado cuando ejecutes tox:

```
$ tox -e py27
```

Combina los dos modos para ejecutar los tests específicos con una sóla versión de Python:

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

Agrega nuevos tests modificando un archivo existente o creando un nuevo script en el directorio `tests`.

Crear un Nuevo Template

El template virtualenvwrapper.project funciona como un plugin de virtualenvwrapper. El nombre del grupo del *punto de entrada* es `virtualenvwrapper.project.template`. Configura tu punto de entrada para apuntar a la función que se **ejecutará** (ganchos incluídos no están soportados para los templates).

El argumento para la función del template es el nombre del proyecto que está siendo creado. El directorio de trabajo actual es el directorio creado para hostear los archivos del proyecto (`$PROJECT_HOME/$envname`).

Texto de ayuda

Una diferencia entre los templates de proyectos y otras extensiones de virtualenvwrapper es que sólo los templates especificados por el usuario son ejecutados. El comando `mkproject` tiene una opción de ayuda para darle al usuario una lista de templates disponibles. Los nombres son tomados desde los nombres de puntos de entrada registrados y las descriptiones de los docstrings de las funciones del template.

Extensiones existentes

Debajo se listan algunas de las extensiones disponibles para usar con virtualenvwrapper.

emacs-desktop

Emacs desktop-mode te permite guardar el estado de emacs (buffers abiertos, posiciones de buffers, etc.) entre sesiones. También puede ser usado como un archivo de proyecto similar a otros IDEs. El plugin emacs-desktop agrega un disparador para guardar el archivo de proyecto actual y cargar uno nuevo cuando se active un nuevo entorno usando `workon`.

user_scripts

La extensión `user_scripts` es distribuida con virtualenvwrapper y está habilitada por default. Implementa la característica de script de personalización de usuarios descrita en *Personalizaciones por usuario*.

vim-virtualenv

vim-virtualenv es una extensión de Jeremey Cantrell para controlar los virtualenvs desde adentro de vim. Cuando es usado en conjunto con virtualenvwrapper, vim-virtualenv identifica el virtualenv a activar basándose en el nombre del archivo que está siendo editado.

Templates

Debajo hay una lista de algunos de los templates disponibles para ser usados *mkproject*.

bitbucket

La extensión de bitbucket clona automáticamente un repositorio mercurial desde el proyecto bitbucket especificado.

django

La extensión django crea automáticamente un nuevo proyecto Django.

Ver también:

- *Crear un Nuevo Template*

¿Porqué virtualenvwrapper no está (mayormente) escrito en Python?

Si miras el código fuente de virtualenvwrapper vas a ver que las partes más interesante están implementadas como funciones de shell en `virtualenvwrapper.sh`. El gancho de carga es una aplicación Python, pero no hace mucho para manejar los entornos virtuales. Algunas de las preguntas más frecuentes sobre virtualenvwrapper son "¿Porqué no escribiste esto como un conjunto de programas Python" y "¿"as pensado en re-escribirlo en Python?". Durante mucho tiempo esas preguntas me han desconcertado, pero fue siempre obvio para mí que tenía que implementarlo de la forma que está. Pero ellos preguntaban lo suficientemente frecuente que siento la necesidad de explicarlo.

tl;dr: POSIX hizo que lo haga

La elección del lenguaje de implementación de virtualenvwrapper fue hecha por razones pragmáticas en vez de filosóficas. Los comandos wrapper necesitan modificar el estado y entorno de los *procesos actuales de shell* del usuario, y la única forma de hacer eso es teniendo los comandos ejecutándose *dentro del shell*. Eso resulta en mi escribiendo virtualenvwrapper como un conjunto de funciones de shell, en vez de scripts de shell o incluso programas Python.

¿De dónde vienen los procesos POSIX?

Los nuevos procesos POSIX son creados cuando un proceso existente invoca la llamada al sistema `fork()`. El proceso invocador se convierte en "padre" del nuevo proceso "hijo" y el hijo es un clon del padre. El resultado *semántico* de `fork()` es que una nueva copia entera del proceso padre es creado. En la práctica, las optimizaciones son normalmente hechas para evitar copiar más memoria que la absolutamente necesaria (frecuentemente a través de sistemas copy-on-write). Pero para el propósito de esta explicación es suficiente pernsar en el hijo como una replica del padre.

Las partes importantes del proceso padre que son copiadas incluyen memoria dinámica (la 'stack' y 'heap'), cosas estáticas (el código del programa), recursos como descriptores de archivos, y el *entorno de variables* exportado por el proceso padre. Heredar variables de entorno es un aspecto fundamental en la manera en que los programas POSIX pasan estado e información de configuraciones de uno a otro. Un padre puede establecer una serie de pares `name=value`, los que son luego son pasados a el proceso hijo. El hijo puede acceder a ellas a través de funciones como `getenv()`, `setenv()` (y en Python a través de `os.environ`).

La elección de el término *heredar* para describir la forma en que las variables y sus contenidos son pasados del padre al hijo es significante. Aunque, un hijo puede cambiar su propio entorno, éste no puede directamente cambiar las configuraciones de entorno de su padre porque no hay una llamada al sistem para modificar configuraciones de entorno de los padres.

Como el shell ejecuta un programa

Cuando un shell recive un comando para ser ejecutado, interactivamente o pasando un archivo de script, y determina que el comando está implementado en un archivo de programa separado, usa `fork()` para crear un nuevo proceso y luego dentro de ese proceso usa una de las funciones `exec` para empezar el programa especificado. El lenguaje en el cual el programa está escrito no hace ninguna diferencia en la decisión sobre el `fork()`, entonces aunque el "programa" sea un script escrito en el lenguaje entendido por el shell actual, un nuevo proceso es creado.

Por otro lado, si el shell decide que el comando es una *función*, entonces se fija en la definición y la invoca diréctamente. Las funciones de shell están hechas de otros comandos, algunos de los cuales quizás resulten en la creación de procesos hijos, pero la función en sí misma se ejecuta en el proceso de shell original y puede por lo tanto modificar su estado, por ejemplo cambiando el directorio de trabajo o los valores de las variables.

Es posible fozar al shell a ejecutar un script directamente, y no en un proceso hijo, *incluyéndolo*. El comando `source` hace que el shell lea el archivo e interprete éste en el proceso actual. De nuevo, como con las funciones, el contenido del archivo puede causar que procesos hijos sean creados, pero no hay un segundo shell interpretando la serie de comandos.

¿Qué significa ésto para virtualenvwrapper?

Lo original y más importante característica de virtualenvwrapper son la activación automática de un entorno virtual cuando éste es creado por el comando `mkvirtualenv` y usando `workon` para desactivar un entorno y activar otro. Hacer que esas características funcionen llevó a las decisiones de implementación de las otras partes de virtualenvwrapper, también.

Los entornos son activados interactivamente incluyendo `bin/source` dentro del entorno. El script `activate` hace algunas cosas, pero las partes importantes son setear la variable `VIRTUAL_ENV` y modificar la ruta de búsqueda del shell a través de la variable `PATH` para poner el directorio `bin` del entorno en frente del path. Cambiar el path significa que los programas instalados dentro del entorno, especialmente el intérprete de python de ahí, son encontrados antes que otros programas con el mismo nombre.

Simplemente ejecutando `bin/activate`, sin usar `source` no funciona porque éste configura el entorno de los procesos *hijos*, sin afectar al padre. Para incluir el script de activación en el shell interactivo, ambos `mkvirtualenv` y `workon` necesitan ser ejecutados en ese proceso de shell.

¿Porqué elegir uno cuando tienes ambos?

El cargador de ganchos es una parte de virtualenvwrapper que *está* escrita en Python. ¿Porqué? De nuevo, porque es más fácil. Los ganchos son descubiertos usando puntos de entrada de setuptools, porque después de que un punto de entrada es instalado el usuario no tiene que tomar ninguna otra acción para permitir al cargador descubrirlo y usarlo. Es fácil imaginar escribir un gancho para crear nuevos archivos en el sistema de archivos (instalando un paquete, instanciando un template, etc.).

Como, entonces, hacen los ganchos corriendo en un proceso separado (el intérprete de Python) para modificar el entorno del shell y setear variables o cambiar el directorio de trabajo? Hacen trampa, por supuesto.

Cada gancho definido por virtualenvwrapper actualmente representa dos ganchos. Primero, los ganchos para Python son ejecutados. Luego los ganchos "source" son ejecutados, y ellos *imprimen* una serie de comandos shell. Todos esos comandos son recolectados, guardados en un archivo temporal, y luego se le dice al shell que lo incluya.

Desde sus comienzos el cargador de ganchos fue mucho más costoso que la mayoría de las otras acciones que virtualenvwrapper hace, por eso, estoy considerando hacer que su uso sea opcional. La mayoría de los usuarios personalizan los ganchos haciendo uso de scripts de shell (ya sea globalmente o dentro del entorno virtual). Encontra y ejecutando aquellos que pueden ser manejados por el shell fácilmente.

Implicancia para compatibilidad en diferentes shells

Además de las peticiones por una implementación completa en Python, la otra petición más común es soportar shells adicionales. fish sale a menudo, debido a varios usuarios de Windows únicamente. Los *Shells soportados* todos tienen en común suficiente sintaxis que hace que la misma implementación funcione para todos. Soportar otros shells requiriría re-escribir mucho, si no todo, de la lógica usando syntaxis alternativa – esos otros shells son básicamente diferentes

lenguajes de programación. Hasta cierto punto he tratado con los ports alentando a otros desarolladores a manejarlos y luego intentantdo linkearlos y promocionar los resultados.

No tan malo como parece

Aunque hay algunos desafíos especiales creados por el requerimiento de que los comandos corran dentro del shell interactivo del usuario (ver los muchos bugs reportados por usuarios quienes tienen algias en comandos comunes como `rm` y `cd`), usar el shell como un lenguaje de programación se sostiene bastante bien. Los shells están diseñados para buscar y ejecutar otros programas fácilmente, y específicamente para hacer fácil combinar una serie de programas pequeños para realizar operaciones mucho más complicadas. Como es lo que virtualenvwrapper está haciendo, es un encaje natura.

Ver también:

- 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 *command-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 scottstvnsn.)

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 davidszotten.
- 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 *Carga por demanda* 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 *Instalación básica* 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 (*Shells soportados*) and Python versions (*Versiones de Python*) (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 spurrious 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 *Consejos y Trucos* section.
- Add link to Rich Leland's screencast to *Referencias* 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 colorized 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.

CAPÍTULO 4

Referencias

virtualenv, de Ian Bicking, es un pre-requisito para usar estas extensiones.

Para más detalles, referirse a la columna que escribí para la revista de python (Python Magazine) en Mayo de 2008: virtualenvwrapper | And Now For Something Completely Different.

Rich Leland ha grabado un pequeño screencast mostrando las características de virtualenvwrapper.

Manuel Kaufmann ha traducido esta documentación al Español.

Tetsuya Morimoto ha traducido esta documentación al Japonés.

Manuel Kaufmann has translated this documentation into Spanish.

Tetsuya Morimoto has translated this documentation into Japanese.

CAPÍTULO 5

Support

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.

CAPÍTULO 6

Soporte

Únete al Grupo de Google de virtualenvwrapper para discutir problemas y característas.

Reporta problemas en el bug tracker de BitBucket.

Alias de shell

Debido a que virtualenvwrapper es en su mayoría un script de shell, éste usa comandos de shell para muchas de sus acciones. Si tu entorno hace uso extendido de alias o otro tipo de customizaciones, quizás encuentres algunos problemas. Antes de reportar bugs en el bug tracker, por favor prueba *sin* tus alias habilitados. Si puedes identificar el alias que causa el problema, ayudará a hacer a virtualenv más robusto.

CAPÍTULO 7

Licencia

Nota: Esta traducción fue realizada por Manuel Kaufmann.

Ver también:

- La traducción al español
- The original English version of the documentation.