pipenv Documentation

Versión 2018.05.18

Kenneth Reitz

10 de julio de 2018

Índice general

Pipenv es una herramienta que apunta a traer todo lo mejor del mundo de empaquetado (bundler, composer, npm, cargo, yarn, etc.) al mundo de Python. *Windows es un ciudadano primera-clase en nuestro mundo*

Automáticamente crea y maneja un entorno virtual para tus proyectos, también como agregar/remover paquetes desde tu `Pipfile` como instalar/desisntalar paquetes. También genera el más importante `Pipfile.lock`, que es usado para producir determinado build.

Pipenv está destinado principalmente a proporcionar a usuarios y desarrolladores de aplicaciones con un metodo sencillo para configurar un entorno de trabajo. Para la distinción entre librerias y aplicaciones y el uso de `setup.py` vs `Pipfile` para definir dependencias, mira *Pipfile vs setup.py*.

Los problemas que Pipenv busca resolver son multifacéticos

- No necesitas usar más `pip` y `virtualenv` separados. Trabajan juntos.
- Manejar un archivo `requirements.txt` puede ser problemático, por eso Pipenv usa en su lugar `Pipfile` y `Pipfile.lock`, que son superiores para usos básicos
- Los Hashes se usan en todas partes, siempre. Seguridad. Automáticamente expone vulnerabilidades de seguridad.
- Te da una vista de tu árbol de dependecias (e.g. `$ pipenv graph`).
- Coordina el flujo de desarrollo cargando archivos `.env`.

CAPÍTULO 1

¡Instala hoy Pipenv!

Si estas en MacOS, puedes instalar Pipenv fácilmente con Homebrew:

```
$ brew install pipenv
```

O, si estás usando Ubuntu 17.10:

```
$ sudo apt install software-properties-common python-software-properties
$ sudo add-apt-repository ppa:pypa/ppa
$ sudo apt update
$ sudo apt install pipenv
```

De lo contrario, solo usa pip:

```
$ pip install pipenv
```

1.1 Pipenv & Entorno Virtuales

Este tutorial te guiará por la instalación y el uso de paquetes de Python.

Te mostrará como instalar y usar las herramientas necesarias y hacer fuertes recomendaciones en las buenas prácticas. Ten en cuenta que Python es usado para un gran cantidad de propósitos diferentes, y precisamente como manejes tus dependecias puede cambiar basado en como decidas publicar tu sofware. La guía presentada aquí es aplicable directamente al desarrollo y despliegue de servicios en red(incluyendo aplicaciones web), pero también es adecuado para el manejo de entornos de desarrollo y pruebas para cualquier tipo de proyecto.

Nota: Esta guía es escrita para Python 3, sin embargo, estas instrucciones deberían funcionar bien en Python 2.7 - Si lo sigues usando, por alguna razón.

1.1.1 Asegurate de tener Python & pip

Antes de continuar, asegurate de que tienes Python y que esta disponible en tu línea de comandos. Puedes verificar esto ejecutando:

```
$ python --version
```

Deberias tener un output como `3.6.2`. Si no tienes Python, por favor instala la última versión 3.x desde python.org o mira la sección Installing Python de *The Hitchhiker's Guide to Python*.

Nota: Si eres nuevo y obtienes un error como este:

```
>>> python
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'python' is not defined
```

Es porque este comando tiene la intención de correr en un *shell* (tambien llamado *termial* o *consola*). Mira Python for Beginners tutorial para empezar para una introducción de el shell de tu sistema operativo e interactuar con Python.

Adicionalmente, necesitas asegurarte que tienes pip disponible. Puedes verificar esto ejecutando:

```
$ pip --version
pip 9.0.1
```

Si tienes instalado Python desde su fuente, con un instalador de python.org, o via **'Homebrew'_** deberías ya tener pip. Si estas en Linux e instalaste a través de tu manejador de paquetes, tal vez necesites instalar pip por separado.

Si tu plan es instalar pipenv usando Hombrew puedes saltarte este paso. El instalador de Homebrew se encarga de pip por ti.

1.1.2 Instalando Pipenv

Pipenv es un manejador de dependencias para los proyectos de Python. Si estas familiarizado con Node.js" npm o Ruby bundler, es similar en espíritu a estas herramientas. Mientras pip puede instalar paquetes de Python, Pipenv es recomendado como herramienta de nivel superior que simplifica el manejo de dependencias para casos comunes.

Use `pip` to install Pipenv:

```
$ pip install --user pipenv
```

Nota: Esto hace una instalación de usuario para prevenir romper cualquier paquete de sistema. Si `pipenv` no esta disponible en tu shell después de la instalación, vas a necesitar agregar la carpeta raíz de binarios del usuario a tu `PATH`.

En Linux y macOS puedes buscar la carpeta raíz de binarios del usuario base ejecutando `python -m site --user-base` y agregando `bin` al final. Por ejemplo, esto normalmente imprimirá `~/.local` (expandiendo con `~` con la ruta absoluta a tu carpeta home) entonces necesitarás agregar `~/.local/bin` a tu `PATH`. Puedes setear tu `PATH` de manera permanente modificando ~/.profile.

En Windows puedes encontrar la carpeta raíz de binarios ejecutando `py -m site --user-site` y reemplazando `site-packages` con `Scripts`. Por ejemplo, esto retornará `C:\Users\Username\AppData\Roaming\Python36\site-packages` entonces vas a necesitar setear tu `PATH` para incluir `C:\Users\Username\AppData\Roaming\Python36\Scripts`. Puedes setear tu `PATH` de manera permanente en el Panel de Control. Puedes necesitar cerrar sesión para que los cambios en `PATH` surtan efecto.

1.1.3 Instalando paquetes para tu proyecto

Pipenv maneja dependencias por proyecto. Para instalar paquetes, cambiate a tu carpeta de proyecto (o solo una carpeta vacía para este tutorial) y ejecuta:

```
$ cd myproject
$ pipenv install requests
```

Pipenv instalará la excelente librería Requests y creará un `Pipfile` para tu carpeta de proyecto. El Pipfile es usado para seguir cual dependencia de tu proyecto necesitas en caso de que quieras reinstalarlas, como cuando comparte el proyecto con otros. Deberías obtener un output parecido a este (aunque la ruta exacta variará):

```
Creating a Pipfile for this project...
Creating a virtualenv for this project...
Using base prefix '/usr/local/Cellar/python3/3.6.2/Frameworks/Python.framework/
↪Versions/3.6'
New python executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python3.6
Also creating executable in ~/.local/share/virtualenvs/tmp-agwWamBd/bin/python
Installing setuptools, pip, wheel...done.

Virtualenv location: ~/.local/share/virtualenvs/tmp-agwWamBd
Installing requests...
Collecting requests
  Using cached requests-2.18.4-py2.py3-none-any.whl
Collecting idna<2.7,>=2.5 (from requests)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting urllib3<1.23,>=1.21.1 (from requests)
  Using cached urllib3-1.22-py2.py3-none-any.whl
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Using cached chardet-3.0.4-py2.py3-none-any.whl
Collecting certifi>=2017.4.17 (from requests)
  Using cached certifi-2017.7.27.1-py2.py3-none-any.whl
Installing collected packages: idna, urllib3, chardet, certifi, requests
Successfully installed certifi-2017.7.27.1 chardet-3.0.4 idna-2.6 requests-2.18.4␣
↪urllib3-1.22

Adding requests to Pipfile's [packages]...
P.S. You have excellent taste!
```

1.1.4 Usando paquetes instalados

Ahora que Requests esta instalado puedes crear un archivo `main.py` para usarlo:

```
import requests

response = requests.get('https://httpbin.org/ip')

print('Your IP is {0}'.format(response.json()['origin']))
```

Entonces puedes ejecutar este script usando `pipenv run`:

```
$ pipenv run python main.py
```

Deberías tener una salida parecida a esta:

```
Your IP is 8.8.8.8
```

Usando `$ pipenv run` se asegura de que tu paquete instalado esta disponible para tu script. También es posible generar un nuevo shell que se asegura de que todos los comandos tienen acceso a tus paquetes instalados con `$ pipenv shell`.

1.1.5 Próximos pasos

¡Felicitaciones, ahora sabes cómo instalar y usar paquetes de Python!

1.2 Instalación de Pipenv con Homebrew

Homebrew es un manejador de paquetes de sistema popular y open-source para macOS

Instalando Pipenv via Homebrew va a mantener Pipenv y todas sus dependencias en un entorno virtual aislado para que no interfiera con el resto de tus instalaciones de Python.

Una vez tengas instalado Homebrew solo ejecuta:

```
$ brew install pipenv
```

Para actualizar pipenv en cualquier momento:

```
$ brew upgrade pipenv
```

1.3 Instalación pragmática de Pipenv

Si tienes una instalación funcional de pip, mantiene cierto «toolchain» escribe los módulos de Python como utilidades globales en tu entorno de usuario, pip instalación de usuario permite instalaciones en tu carpeta home. Nota que debido a interacciones entre dependencias, deberías limitar las herramientas instaladas de esta manera para un flujo de trabajo con Python como virtualenv, pipenv, tox y software similares.

Para instalar:

```
$ pip install --user pipenv
```

Para más información mira la documentación de instalaciones de usuario, pero para agregar herramientas cli desde una instalación de usuario con pip a tu path, agrega el output de:

```
$ python -c "import site; import os; print(os.path.join(site.USER_BASE, 'bin'))"
```

Para actualizar pipenv en cualquier momento:

```
$ pip install --user --upgrade pipenv
```

1.4 Crude Installation of Pipenv

Si ni siquiera tienes instalado pip, puedes usar un método de instalación en bruto, el cual arrancará en todo tu sistema:

```
$ curl https://raw.githubusercontent.com/kennethreitz/pipenv/master/get-pipenv.py |␣
↪python
```

¡Felicidades, ahora tienes pip y Pipenv instalados!

CAPÍTULO 2

Testimonios de Usuarios

Jannis Leidel, former pip maintainer— *Pipenv is the porcelain I always wanted to build for pip. It fits my brain and mostly replaces virtualenvwrapper and manual pip calls for me. Use it.*

David Gang— *This package manager is really awesome. For the first time I know exactly what my dependencies are which I installed and what the transitive dependencies are. Combined with the fact that installs are deterministic, makes this package manager first class, like cargo.*

Justin Myles Holmes— *Pipenv is finally an abstraction meant to engage the mind instead of merely the filesystem.*

CAPÍTULO 3

Características de Pipenv

- Habilita verdaderos *builds deterministas*, mientras fácilmente especificas *solo lo que quieres*.
- Genera y verifica hashes en los archivos para bloquear dependencias.
- Automáticamente instala la versión de Python, si `pyenv` esta disponible
- Automáticamente busca tu proyecto home, recursivamente, buscando por un `Pipfile`
- Automáticamente genera un `Pipfile`, si no existe
- Automáticamente crea un entorno virtual en una locación estándar
- Automáticamente agrega/remueve paquetes a un `Pipfile` cuando se instala/desinstala
- Automáticamente carga archivos `.env`, si estos existen.

Los comandos principales son `install`, `uninstall` and `lock`, el cual genera un `Pipfile.lock`. Estos tienen la intención de reemplazar el uso de `$ pip install`, así como manejar manualmente un entorno virtual (para activar uno, corre `$ pipenv shell`).

3.1 Conceptos Básicos

- Un entorno virtual se creará automáticamente, cuando no exista.
- Cuando no se pasen parámetros a `install`, todos los paquetes `[packages]` especificados se instalarán.
- Para iniciar un entorno virtual con Python 3, corre `$ pipenv --three`.
- Para iniciar un entorno virtual con Python 2, corre `$ pipenv --two`.
- De lo contrario, cualquier entorno virtual será por defecto.

3.2 Otros Comandos

- `graph` va a imprimir un bonito árbol de todas tus dependencias instaladas.

- `shell` generará un shell con el entorno virtual activado.
- `run` va a correr el comando dado desde el entorno virtual, con algún argumento adelante (e.g. `$ pipenv run python` o `$ pipenv run pip freeze`)
- `check` asegura que los requerimientos en PEP 508 se están cumpliendo en el entorno actual.

CAPÍTULO 4

Más guías de documentación

4.1 Uso Básico de Pipenv

Este documento cubre algunos de las características más básicas de Pipenv.

4.1.1 Ejemplo de Pipfile & Pipfile.lock

Este es un ejemplo sencillo de un `Pipfile` y el resultado de `Pipfile.lock`.

Ejemplo de Pipfile

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[packages]
requests = "*"


[dev-packages]
pytest = "*"
```

Ejemplo de Pipfile.lock

```
{
    "_meta": {
        "hash": {
            "sha256":
↪"8d14434df45e0ef884d6c3f6e8048ba72335637a8631cc44792f52fd20b6f97a"
        },
        "host-environment-markers": {
            "implementation_name": "cpython",
            "implementation_version": "3.6.1",
            "os_name": "posix",
            "platform_machine": "x86_64",
            "platform_python_implementation": "CPython",
            "platform_release": "16.7.0",
            "platform_system": "Darwin",
            "platform_version": "Darwin Kernel Version 16.7.0: Thu Jun 15 17:36:27␣
↪PDT 2017; root:xnu-3789.70.16~2/RELEASE_X86_64",
            "python_full_version": "3.6.1",
            "python_version": "3.6",
            "sys_platform": "darwin"
        },
        "pipfile-spec": 5,
        "requires": {},
        "sources": [
            {
                "name": "pypi",
                "url": "https://pypi.python.org/simple",
                "verify_ssl": true
            }
        ]
    },
    "default": {
        "certifi": {
            "hashes": [

↪"sha256:54a07c09c586b0e4c619f02a5e94e36619da8e2b053e20f594348c0611803704",
```

```
↪"sha256:40523d2efb60523e113b44602298f0960e900388cf3bb6043f645cf57ea9e3f5"
            ],
            "version": "==2017.7.27.1"
        },
        "chardet": {
            "hashes": [

↪"sha256:fc323ffcaeaed0e0a02bf4d117757b98aed530d9ed4531e3e15460124c106691",

↪"sha256:84ab92ed1c4d4f16916e05906b6b75a6c0fb5db821cc65e70cbd64a3e2a5eaae"
            ],
            "version": "==3.0.4"
        },
        "idna": {
            "hashes": [

↪"sha256:8c7309c718f94b3a625cb648ace320157ad16ff131ae0af362c9f21b80ef6ec4",

↪"sha256:2c6a5de3089009e3da7c5dde64a141dbc8551d5b7f6cf4ed7c2568d0cc520a8f"
            ],
            "version": "==2.6"
        },
        "requests": {
            "hashes": [

↪"sha256:6a1b267aa90cac58ac3a765d067950e7dbbf75b1da07e895d1f594193a40a38b",

↪"sha256:9c443e7324ba5b85070c4a818ade28bfabedf16ea10206da1132edaa6dda237e"
            ],
            "version": "==2.18.4"
        },
        "urllib3": {
            "hashes": [

↪"sha256:06330f386d6e4b195fbfc736b297f58c5a892e4440e54d294d7004e3a9bbea1b",

↪"sha256:cc44da8e1145637334317feebd728bd869a35285b93cbb4cca2577da7e62db4f"
            ],
            "version": "==1.22"
        }
    },
    "develop": {
        "py": {
            "hashes": [

↪"sha256:2ccb79b01769d99115aa600d7eed99f524bf752bba8f041dc1c184853514655a",

↪"sha256:0f2d585d22050e90c7d293b6451c83db097df77871974d90efd5a30dc12fcde3"
            ],
            "version": "==1.4.34"
        },
        "pytest": {
            "hashes": [

↪"sha256:b84f554f8ddc23add65c411bf112b2d88e2489fd45f753b1cae5936358bdf314",

↪"sha256:f46e49e0340a532764991c498244a60e3a37d7424a532b3ff1a6a7653f1a403a"
```

```
            ],
            "version": "==3.2.2"
        }
    }
}
```

4.1.2 Recomendaciones generales & Control de versión

- Generalmente, mantén a ambos `Pipfile` y `Pipfile.lock` en tu control de versión.
- No mantengas `Pipfile.lock` en tu control de version si estas usando multiples versiones de Python
- Especifica tu versión de Python en la sección de tu *Pipfile*'s `[requires]` . En resumen, deberias tener solo una versión de Python, como herramienta de desarrollo.
- `pipenv install` es totalmente compatible con la sintaxis de `pip install`, puedes encontrar toda su documentación aqui.

4.1.3 Ejemplo del flujo de trabajo de Pipenv

Clona / crea el repositorio del proyecto:

```
...
$ cd myproject
```

Instala desde Pipfile, si hay uno:

```
$ pipenv install
```

O, agrega un paquete a tu nuevo proyecto:

```
$ pipenv install <package>
```

Esto creara un `Pipfile` si no existe. Si existe, automaticamente se editara con los nuevos paquetes que proporciones.

A continuacion, activa el shell de Pipenv:

```
$ pipenv shell
$ python --version
...
```

4.1.4 Ejemplo de uso del flujo de trabajo.

- Averigua que cambio en upstream: `$ pipenv update --outdated`.
- **Actualizar paquetes, dos opciones:**
    1. ¿Quieres actualizar todo? Solo haz `$ pipenv update`.
    2. ¿Quieres actualizar paquete por paquete? `$ pipev update <pkg>` for cada paquete desactualizado.

4.1.5 Importando desde requirements.txt

Si solo tienes un archivo `requirements.txt` disponible cuando ejecutes `pipenv install`, pipenv automáticamente importara el contenido de este archivo y creara un `Pipfile` por ti.

También puedes especificar `$ pipenv install -r path/to/requirements.txt` para importar un archivo requirements.

Si tu archivo requirements tiene versiones fijas, vas a querer editar el nuevo `Pipfile` para removerlos, y dejar que `pipenv` siga las versiones fijas. Si quieres dejar las versiones fijas en tu `Pipfile.lock` por ahora, ejecuta `pipenv lock --keep-outdated`. Asegurate de *actualizar* pronto!

4.1.6 Especifica la versión de un parquete

Para instalar con pipenv una versión especifica de una librería, el uso es simple:

```
$ pipenv install requests==2.13.0
```

Esto actualizara tu `Pipfile` para reflejar este requisito, automáticamente

4.1.7 Especifica la versión de Python

Para crear un nuevo entorno virtual, usando una versión especifica de Python que tengas instalada (y en tu `PATH`), usa la bandera `--python VERSION`, asi:

Usar Python 3:

```
$ pipenv --python 3
```

Usar Python3.6:

```
$ pipenv --python 3.6
```

Usar Python 2.7.14:

```
$ pipenv --python 2.7.14
```

Cuando des una versión de Python, de esta manera, Pipenv automáticamente escaneara tu sistema en busca de la versión de Python dada.

Si un `Pipfile` no ha sido creado todavía, uno se creara por ti, que se vera como esto:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true

[dev-packages]

[packages]

[requires]
python_version = "3.6"
```

Nota la inclusión de `[requires] python_version = "3.6"`. Esto especifica que tu aplicación requiere esta versión de Python, y la usara automáticamente cuando ejecutes `pipenv install` con este `Pipfile` en el futuro (e.j. en otras maquinas) Si esto no es verdad, siéntete libre de remover esta sección.

Si no especificas una versión de Python en la linea de comandos, tanto el `[requires] python_full_version` o `python_version` sera seleccionado automáticamente, usando cualquier instalación de `python` por defecto, cuando se ejecute

4.1.8 Dependencias editables (e.j. `-e .`)

Le puedes decir a Pipenv para instalar una ruta como editable - a menudo es util para el directorio actual cuando se trabaje en un paquete:

```
$ pipenv install --dev -e .

$ cat Pipfile
...
[dev-packages]
"e1839a8" = {path = ".", editable = true}
...
```

Nota que todas las subdependencias se agregaran al `Pipfile.lock`

Nota: Las Subdependencias **no** son agregadas al `Pipfile.lock` si dejas la opcion `-e` por fuera.

4.1.9 Environment Management with Pipenv

Los tres comandos principales que usaras en el manejo de tu pipenv entorno son `$ pipenv install`, `$ pipenv uninstall`, and `$ pipenv lock`.

$ pipenv install

`$ pipenv install` es usado para la instalación de paquetes en tu entorno virtual con pipenv y actualización de tu Pipfile

Junto con el comando de instalación básico, que toma la forma:

```
$ pipenv install [package names]
```

El usuario puede proporcionar estos parámetros adicionales:

- `--two` — Realiza la instalación en un entorno virtual usando la ruta `python2` del sistema.
- `--three` — Realiza la instalación en un entorno virtual usando la ruta `python3` del sistema.
- `--python` — Realiza la instalación en un entorno virtual usando la versión del interprete de python proporcionada.

Advertencia: Ninguno de los comandos mencionados deberían usarse juntos. También son **destructivos** y borraran tu actual entorno virtual antes de reemplazarlo con una versión apropiada.

Nota: El entorno virtual creado por Pipenv puede ser diferente de lo que esperas. Caracteres peligrosos (e.j. `$` ` ! * @` así como el espacio, siguiente linea, carriage return, y tabulación) son convertidos a guion

bajo(_). Adicionalmente, la ruta completa al directorio actual es codificada en un «valor slug» y se agrega para asegurar que el nombre del entorno virtual es único.

- `--dev` — Instala ambos `develop` y `default` paquetes desde `Pipfile.lock`.
- `--system` — Usa el comando pip del sistema `pip` y no el que esta en tu entorno virtual.
- `--ignore-pipfile` — Ignora el `Pipfile` e instala desde el `Pipfile.lock`.
- `--skip-lock` — Ignora el `Pipfile.lock` e instala desde el `Pipfile`. Ademas, no escribe en el `Pipfile.lock` reflejando los cambios del `Pipfile`.

$ pipenv uninstall

`$ pipenv uninstall` soporta todos los parámetros de *pipenv install*, así como dos opciones adicionales, `--all` y `--all-dev`.

- `--all` — Este parámetro limpia todos los archivos de tu entorno virtual, pero deja el Pipfile intacto
- `--all-dev` — Este parámetro eliminara todos los paquetes de desarollo del entorno virtual, y los elimina del Pipfile.

$ pipenv lock

`$ pipenv lock` es usado para crear `Pipfile.lock`, el cual declara **todas** las dependencias (y subdependencias) de tu proyecto, sus ultimas versiones, y el actual hash de los archivos descargados. Esto asegura repetibles, y mas importantes *deterministas* builds.

4.1.10 Configuración sobre el shell

Los Shells son típicamente mal configurados para el uso del subshell, así que `$ pipenv shell --fancy` puede producir resultados inesperados. Si este es el caso, intenta `$ pipenv shell`, el cual usa «modo de compatibilidad», e intentará generar una subshell a pesar de la mala configuración.

Una apropiada configuración de shell solo setea variables de entorno como `PATH` durante el inicio de sesión, no en cada subshell generada (como están típicamente configuradas para hacer). En fish, esto se ve así:

```
if status --is-login
    set -gx PATH /usr/local/bin $PATH
end
```

Deberías hacer esto tambien para tu shell, en tu `~/.profile` o `~/.bashrc` o donde sea apropiado.

Nota: El shell se lanza en modo interactivo. Esto significa que si tu shell lee su configuración desde un archivo especifico para el modo interactivo (e.j. bash por defecto busca por un archivo `~/.bashrc` para la configuración del modo interactivo) entonces necesitaras modificar (o crear) este archivo.

4.1.11 Una nota sobre dependencias en SCV

Pipenv resolverá las subdependencias de las dependencias de SCV, pero solo si estas son instaladas en modo editable:

```
$ pipenv install -e git+https://github.com/requests/requests.git#egg=requests

$ cat Pipfile
[packages]
requests = {git = "https://github.com/requests/requests.git", editable=true}
```

Si editable no es true, las subdependencias no se resolverán.

Para mas información acerca de otras opciones disponibles cuando se especifica dependencias de SCV, por favor revisa los aspectos del Pipfile.

4.1.12 Pipfile.lock características de seguridad

`Pipfile.lock` toma ventaja de algunas buenas mejoras de seguridad en `pip`. Por defecto, el `Pipfile.lock` se generara con un hash sha256 para cada paquete descargado. Esto permitará a `pip` garantizar que estas instalando lo que intentas cuando hay una red comprometida, o descargando dependencias desde un endpoint PyPI poco fiable.

We highly recommend approaching deployments with promoting projects from a development environment into production. You can use `pipenv lock` to compile your dependencies on your development environment and deploy the compiled `Pipfile.lock` to all of your production environments for reproducible builds.

4.2 Uso avanzado de Pipenv

Este documento cubre alguna de las características más avanzadas y magnificas de Pipenv.

4.2.1 Advertencias

- Una rueda de dependencias proporcionadas en un `Pipfile` no serán capturadas por `$ pipenv lock`.
- Hay algunos issues conocidos por usar indices privados, relacionados a hashing. Estamos activamente trabajado en solucionar estos problemas. Sin embargo, puedes tener mucha suerte con estos.
- Las instalaciones tienen la intención de ser lo más deterministas posibles - usa la bandera `--sequential` para incrementar esto, si experimentas algún error

4.2.2 Especificando indice de paquete

Si te gusta un paquete especifico para ser instalado por un determinado indice de paquete, puedes hacer lo siguiente:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[[source]]
url = "http://pypi.home.kennethreitz.org/simple"
verify_ssl = false
name = "home"

[dev-packages]

[packages]
requests = {version="*", index="home"}
maya = {version="*", index="pypi"}
records = "*"
```

Muy elegante.

4.2.3 Usando un PyPI Mirror

Si te gusta sobrescribir las urls por defecto de PyPI con la url de un PyPI mirror, puedes hacer lo siguiente:

```
$ pipenv install --pypi-mirror <mirror_url>

$ pipenv update --pypi-mirror <mirror_url>

$ pipenv sync --pypi-mirror <mirror_url>

$ pipenv lock --pypi-mirror <mirror_url>

$ pipenv uninstall --pypi-mirror <mirror_url>
```

Alternativamente, puedes setear la variable de entorno `PIPENV_PYPI_MIRROR`.

4.2.4 Inyectando credenciales en Pipfiles con variables de entorno

Pipenv leerá las variables de entorno(si están definidas) en tu Pipfile. Muy útil si necesitas autenticarte a un PyPI privado:

```
[[source]]
url = "https://$USERNAME:${PASSWORD}@mypypi.example.com/simple"
verify_ssl = true
name = "pypi"
```

Por suerte - pipenv hasheará tu Pipfile *antes* de leer tus variables de entorno (y, amablemente, sustituirá las variables de entorno de nuevo cuando instales desde un archivo lock - así no hay necesidad de hacer nada secreto! Woo!)

4.2.5 Especificando básicamente cualquier cosa

Si te gusta especificar que un paquete especifico solo sea instalado en ciertos sistemas, puedes usar especificadores PEP 508 para lograr esto.

Aquí tienes un `Pipfile` de ejemplo, el cual solo instalará `pywinusb` en sistemas Windows:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[packages]
requests = "*"
pywinusb = {version = "*", sys_platform = "== 'win32'"}
```

Voilà!

Aquí tienes un ejemplo más complejo:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true

[packages]
unittest2 = {version = ">=1.0,<3.0", markers="python_version < '2.7.9' or (python_
↪version >= '3.0' and python_version < '3.4')"}
```

Magia. Magia pura sin adulterar.

4.2.6 Desplegando Dependencias de Sistema

Puedes decirle a Pipenv para instalar el contenido de un Pipfile en su sistema padre con la bandera `--system`:

```
$ pipenv install --system
```

Esto es útil para contenedores Docker, e infraestructuras de despliegue (e.j. Heroku hace esto).

También útil para despliegue es la bandera `--deploy`:

```
$ pipenv install --system --deploy
```

Esto fallará en construcción si el `Pipfile.lock` esta desactualizado, en su lugar generará uno nuevo.

4.2.7 Pipenv y otras distribuciones

Para usar Pipenv con distribuciones Python de terceros (e.j. Anaconda), puedes proporcionar la ruta al binario de Python:

```
$ pipenv install --python=/path/to/python
```

Anaconda usa Conda para manejar paquetes. Para reusar paquetes instalados con Conda, usa la bandera `--site-packages`:

```
$ pipenv --python=/path/to/python --site-packages
```

4.2.8 Generando un `requirements.txt`

Puedes convertir un `Pipfile` y `Pipfile.lock` en un archivo `requirements.txt` muy fácil, y tener todos los beneficios extras y otras buenas cosas que incluimos.

Echemosle un vistazo a este `Pipfile`:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true

[packages]
requests = {version="*"}
```

Y genera un `requirements.txt` de eso:

```
$ pipenv lock -r
chardet==3.0.4
requests==2.18.4
certifi==2017.7.27.1
idna==2.6
urllib3==1.22
```

Si deseas generar un `requirements.txt` con solo requerimientos de desarrollo puedes hacerlo también! Tomemos el siguiente `Pipfile`:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true

[dev-packages]
pytest = {version="*"}
```

Y generara un `requirements.txt` de eso:

```
$ pipenv lock -r --dev
py==1.4.34
pytest==3.2.3
```

Muy elegante.

4.2.9 Detectando vulnerabilidades de seguridad

Pipenv incluye el paquete safety, y lo usará para escanear tu árbol de dependencias para conocidas vulnerabilidades!

Ejemplo:

```
$ cat Pipfile
[packages]
django = "==1.10.1"

$ pipenv check
Checking PEP 508 requirements...
Passed!
Checking installed package safety...

33075: django >=1.10,<1.10.3 resolved (1.10.1 installed)!
Django before 1.8.x before 1.8.16, 1.9.x before 1.9.11, and 1.10.x before 1.10.3,␣
↪when settings.DEBUG is True, allow remote attackers to conduct DNS rebinding␣
↪attacks by leveraging failure to validate the HTTP Host header against settings.
↪ALLOWED_HOSTS.

33076: django >=1.10,<1.10.3 resolved (1.10.1 installed)!
Django 1.8.x before 1.8.16, 1.9.x before 1.9.11, and 1.10.x before 1.10.3 use a␣
↪hardcoded password for a temporary database user created when running tests with an␣
↪Oracle database, which makes it easier for remote attackers to obtain access to the␣
↪database server by leveraging failure to manually specify a password in the␣
↪database settings TEST dictionary.

33300: django >=1.10,<1.10.7 resolved (1.10.1 installed)!
CVE-2017-7233: Open redirect and possible XSS attack vía user-supplied numeric␣
↪redirect URLs
================================================================================

Django relies on user input in some cases  (e.g.
:func:`django.contrib.auth.views.login` and :doc:`i18n </topics/i18n/index>`)
to redirect the user to an "on success" URL. The security check for these
redirects (namely ``django.utils.http.is_safe_url()``) considered some numeric
URLs (e.g. ``http:999999999``) "safe" when they shouldn't be.

Also, if a developer relies on ``is_safe_url()`` to provide safe redirect
targets and puts such a URL into a link, they could suffer from an XSS attack.

CVE-2017-7234: Open redirect vulnerability in ``django.views.static.serve()``
================================================================================

A maliciously crafted URL to a Django site using the
:func:`~django.views.static.serve` view could redirect to any other domain. The
view no longer does any redirects as they don't provide any known, useful
functionality.

Note, however, that this view has always carried a warning that it is not
hardened for production use and should be used only as a development aid.
```

Nota: In order to enable this functionality while maintaining its permissive copyright license, *pipenv* embeds an API client key for the backend Safety API operated by pyup.io rather than including a full copy of the CC-BY-NC-SA licensed Safety-DB database. This embedded client key is shared across all *pipenv check* users, and hence will be subject to API access throttling based on overall usage rather than individual client usage.

4.2.10 Integraciones de Comunidad

Hay un rango de plugins y extensiones mantenidos por la comunidad disponibles para un numero de editores e IDEs, así como diferentes productos los cuales integraron Pipenv en sus proyectos:

- Heroku (Cloud Hosting)
- Platform.sh (Cloud Hosting)
- PyUp (Security Notification)
- Emacs (Editor Integration)
- Fish Shell (Automatic `$ pipenv shell`!)
- VS Code (Editor Integration)

Trabajo en progreso:

- Sublime Text (Editor Integration)
- PyCharm (Editor Integration)
- Mysterious upcoming Google Cloud product (Cloud Hosting)

4.2.11 Abriendo un módulo en tu editor

Pipenv te permite abrir cualquier módulo de Python que este instalado (incluyendo uno en tu código base), con el comando `$ pipenv open`:

```
$ pipenv install -e git+https://github.com/kennethreitz/background.git#egg=background
Installing -e git+https://github.com/kennethreitz/background.git#egg=background...
...
Updated Pipfile.lock!

$ pipenv open background
Opening '/Users/kennethreitz/.local/share/virtualenvs/hmm-mGOawwm_/src/background/
↪background.py' in your EDITOR.
```

Esto te permite a ti leer el código que estas consumiendo, en lugar de buscarlo en GitHub.

Nota: La variable de entorno estándar `EDITOR` es usada para esto. Si estas usando VS Code, por ejemplo, querrás hacer `export EDITOR=code` (Si estas en macOS vas a querer instalar el comando en tu `PATH` primero).

4.2.12 Instalaciones automáticas de Python

Si tienes pyenv instalado y configurado, Pipenv automáticamente te preguntará si quieres instalar la versión requerida de Python si no la tienes disponible.

Esto es una característica muy elegante, y estamos orgullos de ella:

```
$ cat Pipfile
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true

[dev-packages]
```

```
[packages]
requests = "*"

[requires]
python_version = "3.6"

$ pipenv install
Warning: Python 3.6 was not found on your system...
Would you like us to install latest CPython 3.6 with pyenv? [Y/n]: y
Installing CPython 3.6.2 with pyenv (this may take a few minutes)...
...
Making Python installation global...
Creating a virtualenv for this project...
Using /Users/kennethreitz/.pyenv/shims/python3 to create virtualenv...
...
No package provided, installing all dependencies.
...
Installing dependencies from Pipfile.lock...
    5/5 -- 00:00:03
To activate this project's virtualenv, run the following:
 $ pipenv shell
```

Pipenv automáticamente honra tanto el `python_full_version` y *python_version'* PEP 508 especificadores.

4.2.13 Carga automática de `.env`

Si un archivo `.env` esta presente en tu proyecto, `$ pipenv shell` y `$ pipenv run` automáticamente las cargará para ti

$ cat .env HELLO=WORLD

$ pipenv run python Loading .env environment variables… Python 2.7.13 (default, Jul 18 2017, 09:17:00) [GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin Type «help», «copyright», «credits» or «license» for more information. >>> import os >>> os.environ[“HELLO”] “WORLD”

Esto es muy útil para mantener las credenciales de producción fuera de tu código base. ¡No recomendamos publicar archivos `.env` en el control de versiones de tu código fuente!

Si tu archivo `.env` esta localizado en una ruta diferente o tiene un nombre diferente, puedes setear la variable de entorno `PIPENV_DOTENV_LOCATION`:

```
$ PIPENV_DOTENV_LOCATION=/path/to/.env pipenv shell
```

Para prevenir que pipenv cargue el archivo `.env`, setea la variable de entorno `PIPENV_DONT_LOAD_ENV`:

```
$ PIPENV_DONT_LOAD_ENV=1 pipenv shell
```

4.2.14 Atajos personalizados para Scripts

Pipenv soporta atajos personalizados en la sección `scripts`. `pipenv run` automáticamente los cargará y encontrará el comando correcto para reemplazarlo. Dado el `Pipfile`:

```
[scripts]
printfoo = "python -c \"print('foo')\""
```

Puedes escribir en la terminal para ejecutar:

```
$ pipenv run printfoo
foo
```

4.2.15 Soporte para Variables de Entorno

Pipenv soporta el uso de valores en variables de entorno. Por ejemplo:

```
[[source]]
url = "https://${PYPI_USERNAME}:${PYPI_PASSWORD}@my_private_repo.example.com/simple"
verify_ssl = true
name = "pypi"

[dev-packages]

[packages]
requests = {version="*", index="home"}
maya = {version="*", index="pypi"}
records = "*"
```

Las variables de entorno pueden ser especificadas como `${MY_ENVAR}` o `$MY_ENVAR`. En Windows, `%MY_ENVAR%` también es soportada junto con `${MY_ENVAR}` o `$MY_ENVAR`.

4.2.16 Configuración con Variables de Entorno

Pipenv viene con muchas opciones que pueden ser habilitadas vía variables de entorno en shell. Para activarlas, simplemente crea las variables en tu shell y pipenv las detectará.

- `PIPENV_DEFAULT_PYTHON_VERSION` — Usa esta versión de Python cuando crea un entorno virtual, por defecto (e.j. `3.6`).
- `PIPENV_SHELL_FANCY` — Siempre usa modo elegante cuando invocas `pipenv shell`.
- `PIPENV_VENV_IN_PROJECT` — Si esta seteada, usa `.venv` en tu carpeta de proyecto en lugar del manejador global `pew`.
- `PIPENV_COLORBLIND` — Desactiva los colores en la terminal, por alguna razón.
- `PIPENV_NOSPIN` — Desactiva terminal spinner, para logs más limpios. Automáticamente seteado en un entorno CI.
- `PIPENV_MAX_DEPTH` — Setea un entero para el número máximo de búsqueda recursivas para un Pipfile.
- `PIPENV_TIMEOUT` — Setea un entero para el número máximo de segundos que Pipenv esperará

  para que la creación de un entorno virtual se complete. Por defecto es 120 segundos.
- `PIPENV_INSTALL_TIMEOUT` — Setea un entero para el número máximo de segundos que Pipenv esperará

  para la instalación de un paquete antes que se acabe el tiempo. Por defecto es 900 segundos.
- `PIPENV_IGNORE_VIRTUALENVS` — Seteala para desactivar automáticamente usando un entorno virtual activado sobre el entorno virtual actual del proyecto.
- `PIPENV_PIPFILE` — Cuando ejecutes pipenv desde un $PWD diferente a donde se encuentra el Pipfile, indícale a Pipenv donde encontrar el Pipfile de manera especifica con esta variable de entorno.

- PIPENV_CACHE_DIR — Localización para Pipenv guardar el cache de los paquetes.
- PIPENV_HIDE_EMOJIS — Desactiva los emojis en output.
- PIPENV_DOTENV_LOCATION — Localización para Pipenv para cargar tus .env del proyecto.
- PIPENV_DONT_LOAD_ENV — Le dice a Pipenv no cargar los archivos .env automáticamente.

Si te gusta setear estas variables de entorno por cada proyecto. Recomiendo usar el fantástico proyecto direnv.

También nota que el mismo pip soporta variables de entorno, si necesitas personalización adicional.

Por ejemplo:

```
$ PIP_INSTALL_OPTION="-- -DCMAKE_BUILD_TYPE=Release" pipenv install -e .
```

4.2.17 Localización Personalizada de Entorno Virtual

La dependencia pew de Pipenv automáticamente honrará la variable de entorno WORKON_HOME, si la tienes seteada - le puedes decir a pipenv para guardar tus entoros virtuales en cualquier lugar que quieras, e.g.:

```
export WORKON_HOME=~/.venvs
```

Además, puedes tener a Pipenv para que mantenga un solo entorno virtual en project/.venv configurando la variable de entorno PIPENV_VENV_IN_PROJECT.

4.2.18 Testeando Proyectos.

Pipenv esta siendo usando en proyectos como Requests para declarar dependencias de desarrollo y ejecutar el conjunto de tests

Actualmente hemos testeado despliegues de manera exitosa tanto con Travis-CI y con tox.

Travis CI

Una configuración de ejemplo con Travis CI puede ser encontrada en Requests. El proyecto usa un Makefile para definir las funciones comunes como lo son los comandos init y tests. Este es un ejemplo simplificado de un .travis.yml:

```
language: python
python:
    - "2.6"
    - "2.7"
    - "3.3"
    - "3.4"
    - "3.5"
    - "3.6"
    - "3.7-dev"

# command to install dependencies
install: "make"

# command to run tests
script:
    - make test
```

Y el Makefile correspondiente:

```
init:
    pip install pipenv
    pipenv install --dev

test:
    pipenv run py.test tests
```

Automatización de Proyecto con Tox

Alternativamente, puedes configurar un `tox.ini` como el siguiente tanto para local y testeo externo:

```
[tox]
envlist = flake8-py3, py26, py27, py33, py34, py35, py36, pypy

[testenv]
deps = pipenv
commands=
    pipenv install --dev
    pipenv run py.test tests

[testenv:flake8-py3]
basepython = python3.4
commands=
    pipenv install --dev
    pipenv run flake8 --version
    pipenv run flake8 setup.py docs project test
```

Pipenv automáticamente usara el entorno virtual proporcionado por `tox`. Si `pipenv install --dev` instala `pytest`, entonces el comando `py.test` estará disponible en tu entorno virtual y puedes llamado directamente por `py.test tests` en vez de `pipenv run py.test tests`.

Tal vez quieras agregar `--ignore-pipfile` a `pipenv install`, para no modificar accidentalmente el lock-file cada que corre el test. Esto causa que Pipenv igonore los cambios al `Pipfile` y (más importante) previene de agregar el actual entorno al `Pipfile.lock`. Esto puede ser importante como el entorno actual (e.j. el entorno proporcionado por tox) usualmente contendrá el proyecto actual (lo cual puede ser o no lo deseado) y dependencias adicionales desde la directiva `deps` de `tox`. Lo proporcionado inicialmente puede ser desactivado agregando `skip_install = True` al tox.ini.

Este método requiere que seas explicito acerca de actualizar el lock-file, lo cual puede ser una buena idea en cualquier caso.

Un plugin de 3ros, tox-pipenv es también disponible para usa Pipenv nativamente con tox.

4.2.19 Completado en Shell

Para activar el completado en fish, agrega esto a tu configuración:

```
eval (pipenv --completion)
```

También, con bash o zsh, agrega esto a tu configuración:

```
eval "$(pipenv --completion)"
```

Completado mágico en Shell ahora esta activado!

4.2.20 Trabajando con componentes de Python provistos por la plataforma

Es muy común para enlaces de Python específicos de la plataforma para interfaces del sistema operativo solo están disponibles a través del manejador de paquetes del sistema, y por lo tanto no están disponibles en entornos virtuales con *pip*. En estos casos, el entorno virtual puede ser creado con acceso a la carpeta de *site-packages* del sistema:

```
$ pipenv --three --site-packages
```

Para asegurar que todos los componentes instalables de *pip* realmente sean instalados en el entorno vritual y los paquetes del sistema sean solo usados por las interfaces que no participan en la resolución de dependencias a nivel Python en absoluto, usa la configuración:

```
$ PIP_IGNORE_INSTALLED=1 pipenv install --dev
```

4.2.21 Pipfile vs setup.py

Hay una sutil pero muy importante diferencia para hacer entre **aplicaciones** y **librerías**. Esto es una fuente común de confusión en la comunidad de Python.

Las librerías proporcionan una funcionalidad recusable para otras librerías o aplicaciones (Usemos el término **proyectos**). Estas son requeridas para trabajar con otras librerías, todo con su propio set de subdependencias. Definen **dependencias abstractas**. Para evitar conflictos con versiones en subdependencias de diferentes librerías dentro de un proyecto, las librerías nunca deberían fijar versiones de dependencias. Aunque puede especificar menores o (menos frecuentes) limites superiores, si dependen de alguna característica/arreglo/bug. Las dependencias de librerías son especificadas vía `install_requires` en `setup.py`.

Las librerías últimamente están destinadas a ser usadas en alguna **aplicación**. Las aplicaciones son diferentes en que usualmente no dependen de otros proyectos. Están destinadas a ser desplegadas en un entorno especifico y solo entonces deben usar las versiones exactas de todas sus dependencias y subdependencias. Hacer este proceso más sencillo es el objetivo principal de Pipenv.

Para resumir:

- Para librerías, define **dependencias abstractas** vía `install_requires` en `setup.py`. La decisión de qué versión exacta debe ser instalada y donde obtener esa dependencia no es tuya!
- Para aplicaciones, define **dependencias y donde obtenerlas** en el *Pipfile* y usa este archivo para actualizar un conjunto de **dependencias concretas** en `Pipfile.lock`. Este archivo define un entorno idempotent especifico que se sabe funciona para tu proyecto. El `Pipfile.lock` es tu fuente de confianza. El `Pipfile` es un conveniencia para ti para crear ese lock-file, ya que te permite permanecer algo impreciso acerca de la versión exacta de una dependencia para ser usada. Pipenv esta ahí para ayudarte a definir un conjunto de dependencias especificas de trabajo libres de conflicto, que de otra manera sería una muy tediosa tarea.
- Por supuesto, `Pipfile` y Pipenv siguen siendo útiles para librerías de desarrollo, al poder ser usadas para definir un entorno de desarrollo o prueba.
- Y por supuesto, hay proyectos para los cuales la diferencia entre librería y aplicación no es tan clara. En ese caso, usa `install_requires` junto con Pipenv y `Pipfile`.

También puedes hacer esto:

```
$ pipenv install -e .
```

Esto le dirá a Pipenv para hacer lock a todas tus dependencias declaradas en tu `setup.py`.

4.2.22 Cambiando la locación de Cache de Pipenv

Puedes forzar a Pipenv para usar diferentes locaciones de cache configurando la variable de entorno `PIPENV_CACHE_DIR` para la locación que quieres. Esto es útil en las mismas situaciones donde cambiarías a `PIP_CACHE_DIR` una carpeta diferente.

4.2.23 Cambiando donde Pipenv guarda Entorno Virtuales

Por defecto, Pipenv guarda todos tus entorno virtuales en un solo lugar. Usualmente esto no es un problema, pero si te gustaría cambiarlo para comodidad de desarrollo, o si esta causando issues en servidores de construcción puedes setear `PIPENV_VENV_IN_PROJECT` para crear un entorno virtual dentro de la raíz de tu proyecto.

4.2.24 Cambiando la versión por defecto de Python

Por defecto, Pipenv inicializará un proyecto usando cualquier versión de python que tenga python3. Además de iniciar un proyecto con las banderas `--three` o `--two`, también puedes usar `PIPENV_DEFAULT_PYTHON_VERSION` para especificar cual versión usa cuando se inicie un proyecto y `--three` o `--two` no son usados.

4.3 Problemas frecuentes encontrados con Pipenv

Pipenv esta siendo mejorado constantemente por voluntarios, pero sigue siendo un proyecto muy nuevo con recursos limitados, y tiene algunas peculiaridades que se necesitan tratar. Necesitamos la ayuda de todos(¡incluso la tuya!).

Aquí hay algunas preguntas comunes de gente usando Pipenv. Por favor échale un vistazo y mira si resuelve tu problema.

Nota: Asegúrate primero de estar ejecutando la versión más reciente de Pipenv

---

4.3.1 Tus dependencias no pudieron ser resueltas

Asegúrate de que tus dependencias **realmente** resuelven. Si estas seguro de ello, tal vez necesites limpiar cache. Ejecuta el siguiente comando:

```
pipenv run pipenv-resolver --clear
```

e intenta de nuevo.

Si esto no funciona, intenta borrar toda la carpeta cache manualmente. Suele estar ubicado en una de estas rutas:

- `~/Library/Caches/pipenv` (macOS)
- `%LOCALAPPDATA%\pipenv\pipenv\Cache` (Windows)
- `~/.cache/pipenv` (other operating systems)

Pipenv no instala prelanzamientos (e.j. una versión con el sufijo alpha/beta/etc. como *10b1*) por defecto. Necesitarás pasar la bandera `--pre` en tu comando o setear

```
[pipenv]
allow_prereleases = true
```

en tu Pipfile.

4.3.2 No module named <module name>

Esto generalmente es el resultado de usar Pipenv con otros paquetes de sistema. Nosotros **fuertemente** recomendamos instalar Pipenv en un entorno aislado. Desinstalar todas las instalaciones existentes de Pipenv y mirar *Instalación de Pipenv con Homebrew* para escoger una de las maneras recomendadas de instalar Pipenv

4.3.3 Mi Python instalado con pyenv no es encontrado

Asegúrate de tener `PYENV_ROOT` seteado correctamente. Pipenv solo soporta distribuciones CPython, con nombre de versiones como `3.6.4` o similares.

4.3.4 Pipenv no respeta las versiones locales y globales de pyenv

Pipenv por defecto usa la versión de Python con la que fue instalada para crear el entorno virtual. Puedes setear la opción `--python` o `$PYENV_ROOT/shims/python` para dejar que pyenv consulte cuando escoger el interprete. Mira *Especifica la versión de un parquete* para más información

Si quieres a Pipenv para automáticamente «haga lo correcto», puedes setear la variable de entorno `PIPENV_PYTHON` a `$PYENV_ROOT/shims/python`. Esto hará que Pipenv use la versión activa de Python de pyenv para crear el entorno virtual por defecto.

4.3.5 ValueError: unknown locale: UTF-8

macOS tiene un bug en su detección de localización que nos impide detectar tu codificación de shell correctamente. Esto también puede ser un problema en otros sistemas si la variable local no especifica una codificación.

La solución es setear las siguientes dos variables de entorno en un formato estándar de localización:

- `LC_ALL`
- `LANG`

Para Bash, por ejemplo, puedes agregar lo siguiente en tu `~/.bash_profile`:

```
export LC_ALL='en_US.UTF-8'
export LANG='en_US.UTF-8'
```

Para Zsh, el archivo a editar es `~/.zshrc`.

Nota: Puedes cambiar ambos `en_US` y `UTF-8` al lenguaje/region y codificación que uses.

4.3.6 /bin/pip: No such file or directory

Esto puede estar relacionado a tu configuración local. Mira *ValueError: unknown locale: UTF-8* para una posible solución.

4.3.7 `shell` does not show the virtualenv's name in prompt

Esto es intencional. Puedes hacerlo por ti mismo con cualquier plugins de shell, o una configuración inteligente `PS1`. Si lo quieres de vuelta, usa

```
pipenv shell -c
```

en su lugar (no disponible en Windows).

4.3.8 Pipenv no respeta las dependencias en setup.py

No, no lo hace, intencionalmente. Pipfile y setup.py sirven propósitos diferentes, y no debes considerar uno u otro por defecto. Mira *Pipfile vs setup.py* para más información.

4.3.9 Usando `pipenv run` en el programa Supervisor

Cuando configuras un programa supervisor `command` con `pipenv run ...`, necesitas setear una variable de entorno local adecuada para que funcione.

Agrega esta linea debajo de la sección `[supervisord]` en `/etc/supervisor/supervisord.conf`:

```
[supervisord]
environment=LC_ALL='en_US.UTF-8',LANG='en_US.UTF-8'
```

CAPÍTULO 5

Uso de Pipenv

5.1 pipenv

```
pipenv [OPTIONS] COMMAND [ARGS]...
```

Options

--where
Muestra la ruta del proyecto.

--venv
Muestra la ruta donde esta el entorno virtual.

--py
Muestra la ruta donde esta el interprete de Python.

--envs
Muestra opciones para variables de entorno.

--rm
Remove the virtualenv.

--bare
Salida minima.

--completion
Output completion (to be eval'd).

--man
Display manpage.

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

--site-packages
Activa los paquetes de sitio para el entorno virtual.

--version
Show the version and exit.

5.1.1 check

```
pipenv check [OPTIONS] [ARGS]...
```

Options

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

--system
Use system Python.

--unused <unused>
Dada una ruta de codigo, muestra dependencias sin usar.

Arguments

ARGS
Optional argument(s)

5.1.2 clean

```
pipenv clean [OPTIONS]
```

Options

-v, --verbose
Modo detallado.

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

--dry-run
Solo hace output de las dependencias innecesarias

5.1.3 graph

```
pipenv graph [OPTIONS]
```

Options

--bare
Salida minima.

--json
Muestra JSON.

--json-tree
Muestra un arbol JSON .

--reverse
Muestra un arbol de dependencias en reverso.

5.1.4 install

```
pipenv install [OPTIONS] [PACKAGE_NAME] [MORE_PACKAGES]...
```

Options

-d, --dev
Instala paquetes en [dev-packages].

--three, --two
Usa Ptyon3/2 cuando se crea el entorno virutal.

--python <python>
Especifica la version de Python que deberia usar el entorno virtual.

--pypi-mirror <pypi_mirror>
Especifica un PyPI mirror.

--system
System pip management.

-r, --requirements <requirements>
Importa un archivo requirements.txt.

-c, --code <code>
Importa desde un codigo base.

-v, --verbose
Modo detallado.

--ignore-pipfile
Ignora Pipfile cuando esta instalando, usa el Pipfile.lock.

--sequential
Instala las dependencias una por una, en vez de concurrentemente.

--skip-lock
Ignora el mecanismo de lock cuando esta instalando, en su lugar usa el Pipfile.

--deploy
Cancela si el Pipfile.lock esta desactualizado, o la version de Python es incorrecta.

--pre
Permite prelanzamientos.

--keep-outdated
Mantiene a las dependencias desactualizadas de ser actualizadas en Pipfile.lock.

--selective-upgrade
Actualiza paquetes especificos.

Arguments

PACKAGE_NAME
Optional argument

MORE_PACKAGES
Optional argument(s)

5.1.5 lock

```
pipenv lock [OPTIONS]
```

Options

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

--pypi-mirror <pypi_mirror>
Especifica un PyPI mirror.

-v, --verbose
Modo detallado.

-r, --requirements
Genera un output compatible con requirements.txt

-d, --dev
Genera un output compatible con requirements.txt para dependencias de desarrollo.

--clear
Limpia el cache de dependencias.

--pre
Permite prelanzamientos

--keep-outdated
Mantiene las dependencias desactualizadas de ser actualizadas en el Pipfile.lock.

5.1.6 open

```
pipenv open [OPTIONS] MODULE
```

Options

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

Arguments

MODULE
Required argument

5.1.7 run

```
pipenv run [OPTIONS] COMMAND [ARGS]...
```

Options

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

Arguments

COMMAND
Required argument

ARGS
Optional argument(s)

5.1.8 shell

```
pipenv shell [OPTIONS] [SHELL_ARGS]...
```

Options

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

--fancy
Ejecuta un shell en fancy mode (para shells configuradas de manera elegantes).

--anyway
Siempre genera una subshell, incluso con una ya generada.

Arguments

SHELL_ARGS
Optional argument(s)

5.1.9 sync

```
pipenv sync [OPTIONS]
```

Options

-v, --verbose
Modo detallado.

-d, --dev
Adicionalmente instala paquete(s) en [dev-packages].

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

--pypi-mirror <pypi_mirror>
Especifica un PyPI mirror.

--bare
Salida minima.

--clear
Limpia el cache de dependencias.

--sequential
Instala dependencias una por una, en vez de concurrentemente.

5.1.10 uninstall

```
pipenv uninstall [OPTIONS] [PACKAGE_NAME] [MORE_PACKAGES]...
```

Options

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica que version de Python deberia usar el e entorno virtual.

--system
System pip management.

-v, --verbose
Modo detallado.

--lock
Lock afterwards.

--all-dev
Desistala paquetes de [dev-packages].

--all
Elimina todos los paquetes del entorno virtual. No edita el Pipfile.

--keep-outdated
Mantiene las dependencias desactualizadas de ser actualizadas en el Pipfile.lock.

--pypi-mirror <pypi_mirror>
Especifica un PyPI mirror.

Arguments

PACKAGE_NAME
Optional argument

MORE_PACKAGES
Optional argument(s)

5.1.11 update

```
pipenv update [OPTIONS] [MORE_PACKAGES]... [PACKAGE]
```

Options

--three, --two
Usa Python 3/2 cuando crea un entorno virtual.

--python <python>
Especifica cual version de Python deberia usar el entorno virtual.

--pypi-mirror <pypi_mirror>
Especifica un PyPI mirror.

-v, --verbose
Modo detallado.

-d, --dev
Instala paquete(s) en [dev-packages].

--clear
Limpia el cache de dependencias.

--bare
Salida minima.

--pre
Permite prelanzamientos

--keep-outdated
Mantiene las dependencias desactualizadas de ser actualizadas en el Pipfile.lock.

--sequential
Instala dependencias una por una, en vez de concurrentemente.

--outdated
Lista dependencias desactualizadas

--dry-run
Lista dependencias desactualizadas

Arguments

MORE_PACKAGES
Optional argument(s)

PACKAGE
Optional argument

CAPÍTULO 6

Indices y tablas

- genindex
- modindex
- search

Índice

Symbols

A

C

M

P

S