Common Python Tools: Using virtualenv, Installing with Pip, and Managing

Authored by: ***ASPHostServer Administrator*** *[asphostserver@gmail.com]*
Saved From: *http://faq.asphosthelpdesk.com/article.php?id=267*

When it comes to working with Python, especially in the domain of application development, there are certain tools that you will see being mentioned often in various places or open source code. Despite being extremely commonly used, unfortunately sometimes it is hard to get a hold of a good manual to walk you through each step, which is absolutely vital when it comes to getting familiar with such important and needed tools.

In this DigitalOcean article, we aim to fill you in on not only the basics, but also the logic behind popular Python tools and items as we dive into using them in real life scenarios. We will begin with downloading and installing some common libraries, setting and working with virtual environments (using virtualenv), and managing packages for development and production of your own applications.

This article is aimed at beginners as well as those seeking to obtain more in-depth knowledge. If you would like to see and learn more, please feel free to make a suggestion in the comments section below.

Python on CentOS

Please remember that if you are using a CentOS/RHEL system, you should abstain from working with the default Python interpreter that is shipped with the operating system. Instead, you should opt for installing Python yourself.

Likewise, in order to install **pip** and **virtualenv** on CentOS with a custom Python installation, you can follow the instructions on that article.

Python and Packages

Although Python applications can be made of a **single** file, usually they consist of a series of functions, objects (classes), handy tools and of course, variables spread across **multiple** file(s), placed **inside** *modules*. These modules together make up what is referred as a ***package***.

The traditional way of installing a package involves first spotting it and then downloading. It sounds soft and simple because it actually*is*like many things in Python -*but it is not perfect*.

When the files are ready and unpacked, using the ***disutils*** module, you can install it by calling `setup.py`:

Installation example of a package:

```
# Example: cd [package name]
 cd a_package
```

```
python setup.py install
```

disutils (distribution utilities) is a toolset used for the packaging and distributing of code. It is shipped by default with Python (i.e. included in the *standard library*).

In spite of the simplicity of the procedure explained above, it is no use if the challenge abstracted from installing exists elsewhere in the process: **finding and managing them**. This is where package management via tools comes in - bringing along several benefits such as:

- Uninstalling (e.g.`pip uninstall package_x`),
- Versioning (e.g.`pip install django==x`),
- And automatic dependency management (as packages can depend on others).

Package Management

Packages in Python can be tools, libraries, frameworks and applications. Given the popularity and the beauty of the language, there are tens of thousands of packages available which you can make use of for *your own projects*.

Package Management Tools

The two most common Python package managers are ***pip*** and ***easy_install***. Both of them aim to help users with the tasks of:

- Downloading, Installing and Uninstalling
- Building
- Managing Python packages and much more

Both of them might appear to do the same thing from the outside and their joint dependence on the common library ***setuptools*** increases this notion.

However, in this case, it is what's hidden from the eye that makes the difference — and a lot of it as well.

pip vs easy_install

The first tool created for the task was **easy_install**. Although it was a relief and a pleasure to use compared to doing everything manually, over time it has proven to be problematic in certain aspects. That created the grounds for development of **pip**, another package manager.

pip (as defined by the project itself) is a replacement for **easy_install**, which brings many benefits over its predecessor, including, but not limited to:

- Downloading everything before installing
- Providing useful feedback during the process

- Keeping history of actions being taken
- Providing useful error messages following Python tradition
- Complements ***virtualenv*** and works with it *very* nicely

To learn more about *pip*, consider reading its introduction located at PyPI package index by clicking here.

A Thorough *pip* How-To

In this section, we will talk about getting the necessary dependencies for *pip*, installing its latest built followed by a walk-through of the core functionality offered such as **installing**, **uninstalling**, **freezing** and **managing requirements**.

When would I use *pip*?

As promised at the introduction, we aim to give you examples of real life scenarios.

Imagine that you are undertaking the development of a small application. You have set yourself a roadmap, and everything is going well. Then you discover a library (or a module) that can be of great help to you if you included in your application. You can download it the traditional way as we have explained. However, once you have not just one but 3, 4 or even 20 packages you need to deal with, this process becomes cumbersome. Include managing them (e.g. updating, uninstalling, replacing, using a different version), you can see the problems you will need to deal with, which are made redundant using *pip*, the package manager.

Installing *pip*

In order to install *pip*, we first need to take care of its dependencies. Do not worry though, it is very easy.

setuptools

As explained above, one of the dependencies of *pip* is the ***setuptools*** library. It builds on the (standard) functionality of Python's *distribution utilities* toolset called ***distutils***. Given that distils is provided by default, all we need left is *setuptools*.

We are going to securely download the setup files for **setuptools** using`curl`. **cURL** is a system library which allows data transfer over various protocols (i.e. a common language for data exchange between applications, such as HTTP). It will verify the *SSL* certificates from the source and pass the data to the *Python interpreter*.

These setup files, which Python interpreter is going to execute, automate the installation process as they set up the latest stable version on our system.

Execute the following command:

```
curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py | python -
```

This installation gives us the ability to use pip globally across the system. However, this is not the preferred

way to install any other package. What is recommended is to always use self-contained Python environments, **virtualenv**. We will talk about it in the next section.

Note: You might need to explicitly gain *super user* privileges in order to continue with the download. In that case, consider using:

```
sudo curl https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py |
python -
```

After having its single dependency installed, we can now continue with downloading and setting up*pip*.

We will be again using`curl`to have the setup file securely downloaded and installed.

Execute the following command:

```
curl https://raw.github.com/pypa/pip/master/contrib/get-pip.py | python -
```

Default installation folder for *pip* is located at:

```
/usr/local/bin
```

In order to use it without stating the full path, it must be appended to **PATH**.

Updating *PATH*:

```
export PATH="/usr/local/bin:$PATH"
```

After completing this step, we are ready to work with *pip*.

Using *pip*

Using *pip* is really fun and can be considered headache free. If you have dealt with extremely unnecessary problems in the past and did not even understand why, *pip* will ensure that they are kept to a minimum for you hereon.

Installing packages using *pip*

---

pip can do many things but it would not be a mistake to state that the most often used function of it is **installing packages**. There are several ways it can handle this job for you.

Finding and installing packages:

```
# Example: pip install [package name]
 # Let's install the excellent *requests* library
 pip install requests
```

Finding and installing a specific version:

```
# Example: pip install [package name]==[version]
 # Let's install version 2.0.0. of requests
 pip install requests==2.0.0
```

Installing from a URL or a VCS repository:

```
# Example: pip install [url]
 # Let's install virtualenv
 pip install https://github.com/pypa/virtualenv/tarball/1.9.X
```

Installing inside a *virtualenv*:

```
# Example: pip install [env name] [package name]
 # This will either install inside an environment, or create one
 # Let's install requests inside a virtualenv called *venv*
 pip install -E venv requests
```

Uninstalling packages with *pip*:

The second most common function of *pip* is probably uninstalling packages.

Uninstalling a package:

```
# Example: pip uninstall [package name]
 # Let's remove requests library
 pip uninstall requests
```

Upgrading packages with *pip*

If you are thinking of uninstalling to install a newer version of an application, you can try *upgrading*.

Upgrading a package:

```
# Example: pip install --upgrade [package name]
 # Let's upgrade requests library
 pip install --upgrade requests
```

Searching for packages with *pip*

Before deciding to remove or upgrade a package, you might feel the need to first *search* for one.

Searching for a package:

```
# Example: pip search [package name]
 # Let's find all django packages
 # This might take a while (there's tonnes of them!)
 pip search django
```

Creating a list of installed packages with *pip*

One of the most truly exceptional and life saving abilities of pip is being able to create - with ease - lists ("freeze") of **packages installed**. This is also often called **requirements**. Depending on your Python environment (whether it be a virtual or a global one), *pip* will create a file listing all the packages installed with one single command.

Creating a fresh list ("freeze"):

Note: This command will output a file in the current working directory.

```
# Example: pip freeze > [file name.ext]
 # Let's list all the packages currently installed
 pip freeze > package_list.txt
```

Creating a list ("freeze") on top of a template:

Note: This command will output a file in the current working directory.

```
# Example: pip freeze -r [existing file.ext] > [filename.ext]
 # Let's append new packages installed after the last freeze
 pip freeze package_list.txt > package_list_new.txt
```

Installing all the packages from a list with *pip

When you are working on an application - preferably inside a virtual environment - you will have all of its dependencies (***required packages***) installed there. After having extracted a list of them using **freeze**, you

can get them installed again with **install**.

Installing all packages from a list ("freeze"):

```
# Example: pip install -r [file name.ext]
 # Let's install back all the packages from the previous example
 pip install -r package_list_new.txt
```

A Thorough *virtualenv* How-To

Let's begin with defining what exactly **virtualenv** is and the situation where it comes in handy.

virtualenv:

In the world of Python, an ***environment*** is a folder (directory) which contains everything that a Python project (application) needs in order to run in an organised, isolated fashion. When it is initiated, it automatically comes with its own Python interpreter - a copy of the one used to create it - alongside its very own *pip*.

There are a number of problems that **virtualenv** solves:

- Creating a fresh, isolated environment for a Python project
- Being able to download packages without requiring admin/sudo privileges
- Easily packaging an application
- Creating a list of dependencies which belongs to a single project created with *pip*
- Easily recovering the dependencies using a requirements file created with *pip*
- Giving way to portability across systems

Using *virtualenv* is **the** recommended way for working with Python projects, regardless of how many you might be busy with. It is very easy to use and an excellent tool to have at your disposable. It truly does wonders when coupled with *pip*.

We will start with installing **virtualenv** the system.

Installing virtualenv

In order to install *virtualenv*, we are going to call in *pip* for help. We will install it as a globally available package for the Python interpreter to run.

There are two ways to obtain the application. The version you will be able to get depends on which one you choose.

The simplest method is using *pip* to search, download and install. This might **not** provide you the latest stable version.

Downloading *virtualenv* using *pip*:

```
# Example: [sudo] pip install virtualenv
 sudo pip install virtualenv
```

Downloading the latest available one using *curl*:

The latest release of *virtualenv* is`1.11.X`.

```
# Example: [sudo] pip install [github repo]/[version]
 sudo pip install https://github.com/pypa/virtualenv/tarball/1.1.X
```

Using*virtualenv*

Using this tool consists of getting it to create a folder, containing the Python interpreter and a copy of *pip*. Afterwards, in order to work with it, we need to either specify the location of that interpreter or `activate` it.

All the applications you install using the interpreter inside the virtual environment will be places within that location.

When you use *pip* to create a list of them, only the ones inside the folder will be compiled into a file.

Remember: When you are done working with one environment, before switching to another - or working with the globally installed one - you need to make sure to `deactivate` it.

Creating / Initiating a *virtual environment* (virtualenv)

Creating an environment using the same interpreter used to run it:

```
# Example: virtualenv [folder (env.) name]
 # Let's create an environment called *my_app*
 virtualenv my_app
```

Creating an environment with a custom Python interpreter:

```
# Example: virtualenv --python=[loc/to/python/] [env. name]
 virtualenv --python=/opt/python-3.3/bin/python my_app
```

Activating a*virtual environment*

```
# Example: source [env. name]/bin/activate
 # Let's activate the Python environment we just created
```

```
source my_app/bin/activate
```

Working with a *virtual environment* without activating

For various reasons, you might choose not to activate the environment before using it. This brings more flexibility to commands you run, however, you need to make sure to target the correct interpreter each and every time.

```
# Example: [env. name]/bin/python [arguments]
 my_app/bin/python python_script.py
```

Using the*pip*installation inside the environment without activation

```
# Example: [env. name]/bin/pip [command] [arguements]
 # Let's install requests library without activating the env.
 my_app/bin/pip install requests
```

Deactivating a *virtual environment*:

```
# Example: deactivate
 # Let's deactivate the environment from earlier
 deactivate
```