fades Documentation

Release 4

Facundo Batista, Nicolás Demarchi

Jun 05, 2020

Contents

CHAPTER 1

Contents:

1.1 What is fades?

fades is a system that automatically handles the virtualenvs in the cases normally found when writing scripts and simple programs, and even helps to administer big projects.

resources/logo256.png

fades will automagically create a new virtualenv (or reuse a previous created one), installing the necessary dependencies, and execute your script inside that virtualenv, with the only requirement of executing the script with *fades* and also marking the required dependencies.

(If you don't have a clue why this is necessary or useful, I'd recommend you to read this small text about Python and the Management of Dependencies *.)*

The first non-option parameter (if any) would be then the child program to execute, and any other parameters after that are passed as is to that child script.

fades can also be executed without passing a child script to execute: in this mode it will open a Python interactive interpreter inside the created/reused virtualenv (taking dependencies from `--dependency` or `--requirement` options).

Contents

1.2 How to use it?

Click in the following image to see a video/screencast that shows most of fades features in just 5'...

...or inspect these several small GIFs that show each a particular *fades* functionality, but please keep also reading for more detailed information...

1.2.1 Yes, please, I want to read

When you write an script, you have to take two special measures:

- need to execute it with *fades* (not *python*)
- need to mark those dependencies

At the moment you execute the script, fades will search a virtualenv with the marked dependencies, if it doesn't exists fades will create it, and execute the script in that environment.

1.2.2 How to execute the script with fades?

You can always call your script directly with fades:

```
fades myscript.py
```

However, for you to not forget about fades and to not execute it directly with python, it's better if you put at the beggining of the script the indication for the operating system that it should be executed with fades...

```
#!/usr/bin/fades
```

...and also set the executable bit in the script:

```
chmod +x yourscript.py
```

You can also execute scripts directly from the web, passing directly the URL of the pastebin where the script is pasted (most common pastebines are supported, pastebin.com, gist, linkode.org, but also it's supported if the URL points to the script directly):

```
fades http://myserver.com/myscript.py
```

1.2.3 How to mark the dependencies to be installed?

The procedure to mark a module imported by the script as a *dependency to be installed by fades* is by using a comment.

This comment will normally be in the same line of the import (recommended, less confusing and less error prone in the future), but it also can be in the previous one.

The simplest comment is like:

```
import somemodule    # fades
from somepackage import othermodule    # fades
```

The `fades` is mandatory, in this examples the repository is PyPI, see *About different repositories* below for other examples.

With that comment, *fades* will install automatically in the virtualenv the `somemodule` or `somepackage` from PyPI.

Also, you can indicate a particular version condition, examples:

```
import somemodule    # fades == 3
import somemodule    # fades >= 2.1
import somemodule    # fades >=2.1,<2.8,!=2.6.5
```

Sometimes, the project itself doesn't match the name of the module; in these cases you can specify the project name (optionally, before the version):

```
import bs4    # fades beautifulsoup4
import bs4    # fades beautifulsoup4 == 4.2
```

1.2.4 What if no script is given to execute?

If no script or program is passed to execute, *fades* will provide a virtualenv with all the indicated dependencies, and then open an interactive interpreter in the context of that virtualenv.

Here is where it comes very handy the `-i`/`--ipython` option, if that REPL is preferred over the standard one.

In the case of using an interactive interpreter, it's also very useful to make *fades* to automatically import all the indicated dependencies, passing the `--autoimport` parameter.

1.2.5 Other ways to specify dependencies

Apart of marking the imports in the source file, there are other ways to tell *fades* which dependencies to install in the virtualenv.

One way is through command line, passing the `--dependency` parameter. This option can be specified multiple times (once per dependency), and each time the format is `repository::dependency`. The dependency may have versions specifications, and the repository is optional (defaults to 'pypi').

Another way is to specify the dependencies in a text file, one dependency per line, with each line having the format previously described for the `--dependency` parameter. This file is then indicated to fades through the `--requirement` parameter. This option can be specified multiple times.

In case of multiple definitions of the same dependency, command line overrides everything else, and requirements file overrides what is specified in the source code.

Finally, you can include package names in the script docstring, after a line where "fades" is written, until the end of the docstring; for example:

```
"""Script to do stuff.

It's a very important script.

We need some dependencies to run ok, installed by fades:
    request
    otherpackage
"""
```

1.2.6 About different repositories

fades supports installing the required dependencies from multiples repositories: besides PyPI, you can specify URLs that can point to projects from GitHub, Launchpad, etc. (basically, everything that is supported by `pip` itself).

When a dependency is specified, *fades* deduces the proper repository. For example, in the following examples *fades* will install requests from the latest revision from PyPI in the first case, and in the second case the latest revision from the project itself from GitHub:

```
-d requests
-d git+https://github.com/kennethreitz/requests.git#egg=requests
```

If you prefer, you can be explicit about which kind of repository *fades* should use, prefixing the dependency with the special token double colon (`::`):

```
-d pypi::requests
-d vcs::git+https://github.com/kennethreitz/requests.git#egg=requests
```

There are two basic repositories: `pypi` which will make *fades* to install the desired dependency from PyPI, and `vcs`, which will make *fades* to treat the dependency as a URL for a version control system site. In the first case, for PyPI, a full range of version comparators can be specified, as usual. For `vcs` repositories, though, the comparison is always exact: if the very same dependency is specified, a *virtualenv* is reused, otherwise a new one will be created and populated.

In both cases (specifying the repository explicitly or implicitly) there is no difference if the dependency is specified in the command line, in a `requirements.txt` file, in the script's docstring, etc. In the case of marking the `import` directly in the script, it slightly different.

When marking the `import` it normally happens that the package itself to be installed has the name of the imported module, and because of that it can only be found in PyPI. So, in the following cases the `pypi` repository is not only deduced, but unavoidable:

```
import requests  # fades
from foo import bar  # fades
import requests  # fades <= 3
```

But if the package is specified (normally needed because it's different than the module name), or if a version control system URL is specified, the same possibilities stated above are available: let *fades* to deduce the proper repository or mark it explicitly:

```
import bs4  # fades beautifulsoup
import bs4  # fades pypi::beautifulsoup
import requests  # fades git+https://github.com/kennethreitz/requests.git#egg=requests
import requests  # fades vcs::git+https://github.com/kennethreitz/requests.git
↪#egg=requests
```

One last detail about the `vcs` repository: the format to write the URLs is the same (as it's passed without modifications) than what `pip` itself supports (see pip docs for more details).

Furthermore, you can install from local projects. It's just fine to use a dependency that starts with `file:`. E.g. (please note the triple slash, because we're mixing the protocol indication with the path):

```
fades -d file:///home/crazyuser/myproject/allstars/
```

1.2.7 How to control the virtualenv creation and usage?

You can influence several details of all the virtualenv related process.

The most important detail is which version of Python will be used in the virtualenv. Of course, the corresponding version of Python needs to be installed in your system, but you can control exactly which one to use.

No matter which way you're executing the script (see above), you can pass a `-p` or `--python` argument, indicating the Python version to be used just with the number (`2.7`), the whole name (`python2.7`) or the whole path (`/usr/bin/python2.7`).

Other detail is the verbosity of *fades* when telling what is doing. By default, *fades* only will use stderr to tell if a virtualenv is being created, and to let the user know that is doing an operation that requires an active network connection (e.g. installing a new dependency).

If you call *fades* with `-v` or `--verbose`, it will send all internal debugging lines to stderr, which may be very useful if any problem arises. On the other hand if you pass the `-q` or `--quiet` parameter, *fades* will not show anything (unless it has a real problem), so the original script stderr is not polluted at all.

If you want to use IPython shell you need to call *fades* with `-i` or `--ipython` option. This option will add IPython as a dependency to *fades* and it will launch this shell instead of the python one.

You can also use `--system-site-packages` to create a venv with access to the system libs.

Finally, no matter how the virtualenv was created, you can always get the base directory of the virtualenv in your system using the `--get-venv-dir` option.

1.2.8 Running programs in the context of the virtualenv

The `-x/--exec` parameter allows you to execute any program (not just a Python one) in the context of the virtualenv.

By default the mandatory given argument is considered the executable name, relative to the virtualenv's `bin` directory, so this is specially useful to execute installed scripts/program by the declared dependencies. E.g.:

```
fades -d flake8 -x flake8 my_script_to_be_verified_by_flake8.py
```

Take in consideration that you can pass an absolute path and it will be respected (but not a relative path, as it will depend of the virtualenv location).

For example, if you want to run a shell script that in turn runs a Python program that needs to be executed in the context of the virtualenv, you can do the following:

```
fades -r requirements.txt --exec /var/lib/foobar/special.sh
```

Finally, if the intended code to run is prepared to be executed as a module (what you would normally run as *python3 -m some_module*), you can use the same parameter with *fades* to run that module inside the virtualenv:

```
fades -r requirements.txt -m some_module
```

1.2.9 How to deal with packages that are upgraded in PyPI

When you tell *fades* to create a virtualenv using one dependency and don't specify a version, it will install the latest one from PyPI.

For example, you do `fades -d foobar` and it installs foobar in version 7. At some point, there is a new version of foobar in PyPI, version 8, but if do `fades -d foobar` it will just reuse previously created virtualenv, with version 7, not downloading the new version and creating a new virtualenv with it!

You can tell fades to do otherwise, just do:

```
fades -d foobar --check-updates
```

... and *fades* will search updates for the package on PyPI, and as it will found version 8, will create a new virtualenv using the latest version.

From this moment on, if you request `fades -d foobar` it will bring the virtualenv with the new version. If you want to get a virtualenv with not-the-latest version for any dependency, just specify the proper versions.

You can even use the `--check-updates` parameter when specifying the package version. Say you call `fades -d foobar==7`, *fades* will install version 7 no matter which one is the latest. But if you do:

```
fades -d foobar==7 --check-updates
```

... it will still use version 7, but will inform you that a new version is available!

1.2.10 What about pinning dependencies?

One nice benefit of *fades* is that every time dependencies change in your project, you actually get to use a new virtualenv automatically.

If you don't pin the dependencies in your requirements file, this has another nice side effect: everytime you use them in a new environment (or if you have *–check-updates* set) you will get latest versions, effectively avoiding the trap of sticking in old versions forever.

However, this has a bad side. If it happens that a dependency of your project released a revision between the moment you run the tests and the moment your project is deployed to the server, it may happen that you actually put in production an untested combination. Furthermore, it may happen that even if you do pin your dependencies, the dependencies of those dependencies may not be pinned, and you get into the same situation.

For example, you may have the `requests == 2.19.1` dependency, but `requests` declares its own dependencies, for example `chardet >= 3.0.2`, and when running tests locally you may get `chardet` in version `3.0.3`, but nothing guarantees you that when deploying your project to a server (effectively building everything from scratch) you will not get a newer version of `chardet`, which may be totally fine but in fact it's something that you did NOT test locally.

Here is where *fades* comes to the rescue with the `--freeze` option. If this parameter is given, *fades* will operate exactly as it normally would, but also will dump the result of `pip freeze` into the specified file.

So to continue with the example above, you could run your tests like:

```
fades -d "requests == 2.19.1" --freeze=reqs-frozen.txt -x python3 -m unittest
```

... which will leave you `reqs-frozen.txt` with a content similar to:

```
certifi==2018.4.16
chardet==3.0.4
pip==18.0
requests==2.19.1
...
```

And then you could use *that file* for deployment, which has *all packages* pinned, so you will get exactly what you was expecting.

1.2.11 Under the hood options

For particular use cases you can send specifics arguments to `virtualenv`, `pip` and `python`. using the `--virtuaenv-options`, `--pip-options` and `--python-options` respectively. You have to use that argument for each argument sent.

Examples:

```
fades -d requests --virtualenv-options="--always-copy" --virtualenv-options="--extra-search
tmp"
```

```
fades -d requests --pip-options="--index-url='http://example.com'"
```

```
fades --python-options=-B foo.py
```

1.2.12 Setting options using config files

You can also configure fades using *.ini* config files. fades will search config files in */etc/fades/fades.ini*, the path indicated by *xdg* for your system (for example *~/config/fades/fades.ini*) and *.fades.ini*.

So you can have different settings at system, user and project level.

With fades installed you can get your config dir running:

```
python -c "from fades.helpers import get_confdir; print(get_confdir())"
```

The config files are in *.ini* format. (configparser) and fades will search for a *[fades]* section.

You have to use the same configurations that in the CLI. The only difference is with the config options with a dash, it has to be replaced with a underscore.:

```
[fades]
ipython=true
verbose=true
python=python3
check_updates=true
dependency=requests;django>=1.8  # separated by semicolon
```

There is a little difference in how fades handle these settings: "dependency", "pip-options" and "virtualenv-options". In these cases you have to use a semicolon separated list.

The most important thing is that these options will be merged. So if you configure in */etc/fades/fades.ini* "dependency=requests" you will have requests in all the virtualenvs created by fades.

1.2.13 How to clean up old virtualenvs?

When using *fades* virtual environments are something you should not have to think about. *fades* will do the right thing and create a new virtualenv that matches the required dependencies. There are cases however when you'll want to do some clean up to remove unnecessary virtual environments from disk.

By running *fades* with the `--rm` argument, *fades* will remove the virtualenv matching the provided UUID if such a virtualenv exists (one easy way to find out the virtualenv's UUID is calling *fades* with the `--get-venv-dir` option.

Another way to clean up the cache is to remove all venvs that haven't been used for some time. In order to do this you need to call *fades* with `--clean-unused-venvs`. When fades it's called with this option, it runs in mantain mode, this means that fades will exit after finished this task. All virtualenvs that haven't been used for more days than the value indicated in param will be removed.

It is recommended to have some automatically way of run this option; ie, add a cron task that perform this command:

```
fades --clean-unused-venvs=42
```

1.2.14 Some command line examples

Execute `foo.py` under *fades*, passing the `--bar` parameter to the child program, in a virtualenv with the dependencies indicated in the source code:

```
fades foo.py --bar
```

Execute `foo.py` under *fades*, showing all the *fades* messages (verbose mode):

```
fades -v foo.py
```

Execute `foo.py` under *fades* (passing the `--bar` parameter to it), in a virtualenv with the dependencies indicated in the source code and also `dependency1` and `dependency2` (any version > 3.2):

```
fades -d dependency1 -d "dependency2>3.2" foo.py --bar
```

Execute the Python interactive interpreter in a virtualenv with `dependency1` installed:

```
fades -d dependency1
```

Execute the Python interactive interpreter in a virtualenv after installing there all dependencies taken from the `requirements.txt` file:

```
fades -r requirements.txt
```

Execute the Python interactive interpreter in a virtualenv after installing there all dependencies taken from files `requirements.txt` and `requirements_devel.txt`:

```
fades -r requirements.txt -r requirements_devel.txt
```

Use the `django-admin.py` script to start a new project named `foo`, without having to have django previously installed:

```
fades -d django -x django-admin.py startproject foo
```

Remove a virtualenv matching the given uuid from disk and cache index:

```
fades --rm 89a2bf83-c280-4918-a78d-c35506efd69d
```

Download the script from the given pastebin and executes it (previously building a virtualenv for the dependencies indicated in that pastebin, of course):

```
fades http://linkode.org/#4QI4TrPlGf1gK2V7jPBC47
```

Run all the tests in a project (running `pytest` directly as a module, for better behaviour) and at the same time freeze dependencies for later deployment:

```
fades -r requirements.txt --freeze -m pytest -v
```

1.2.15 Some examples using fades in project scripts

Including *fades* in project helper scripts makes it easy to stop worrying about the virtualenv activation/deactivation when working in that project, and also solves the problem of needing to update/change/fix an already created virtualenv if the dependencies change.

This is an example of how a script to run your project may look like:

```
#!/bin/sh
if (command -v fades > /dev/null)
then
    # fades FTW!
    fades -r requirements.txt bin/start
else
    echo 2
    # hope you are in the correct virtualenv
    python3 bin/start
fi
```

To run the tests, it's super handy to have a script that also takes care of the development dependencies:

```
#!/bin/sh
fades -r requirements.txt -r reqs-dev.txt -x python -m pytest -s "$@"
```

1.2.16 What if Python is updated in my system?

The virtualenvs created by fades depend on the Python version used to create them, considering its major and minor version.

This means that if run fades with a Python version and then run it again with a different Python version, it may need to create a new virtualenv.

Let's see some examples. Let's say you run fades with `python`, which is a symlink in your `/usr/bin/` to `python3.4` (running it directly by hand or because fades is installed to use that Python version).

If you have Python 3.4.2 installed in your system, and it's upgraded to Python 3.4.3, fades will keep reusing the already created virtualenvs, as only the micro version changed, not minor or major.

But if Python 3.5 is installed in your system, and the default `python` is pointed to this new one, fades will start creating all the virtualenvs again, with this new version.

This is a good thing, because you want that the dependencies installed with one specific Python in the virtualenv are kept being used by the same Python version.

However, if you want to avoid this behaviour, be sure to always call fades with the specific Python version (`/usr/bin/python3.4` or `python3.4`, for example), so it won't matter if a new version is available in the system.

1.3 How to install it

Several instructions to install `fades` in different platforms.

1.3.1 Simplest way

In some systems you can install `fades` directly, no needing to install previously any dependency.

If you are in debian unstable or testing, just do:

```
sudo apt-get install fades
```

For Arch Linux, you can install it from the **AUR** using any AUR helper, e.g. with `pikaur`:

```
pikaur -S fades
```

In systems with Snaps:

snap install fades –classic

(why *–classic*? Because it's the only way that *fades* could, from inside the snap, access the rest of the system in case you want to use a different Python version, or a dependency that needs compilation, etc).

For Mac OS X (and Homebrew):

brew install fades

Else, keep reading to know how to install the dependencies first, and `fades` in your system next.

1.3.2 Dependencies

Besides needing Python 3.3 or greater, fades depends also on the `pkg_resources` package, that comes in with `setuptools`. It's installed almost everywhere, but in any case, you can install it in Ubuntu/Debian with:

```
apt-get install python3-setuptools
```

And on Arch Linux with:

```
pacman -S python-setuptools
```

It also depends on `python-xdg` package. This package should be installed on any GNU/Linux OS wiht a freedesktop.org GUI. However it is an **optional** dependency.

You can install it in Ubuntu/Debian with:

```
apt-get install python3-xdg
```

And on Arch Linux with:

```
pacman -S python-xdg
```

Fades also needs the virtualenv package to support different Python versions for child execution. (see the `--python` option.)

1.3.3 For others debian and ubuntu

If you are NOT in debian unstable or testing (if you are, see above for better instructions), you can use this .deb.

Download it and install doing:

```
sudo dpkg -i fades_*.deb
```

1.3.4 Using pip if you want

```
pip3 install fades
```

1.3.5 Multiplatform tarball

Finally you can always get the multiplatform tarball and install it in the old fashion way:

```
wget http://ftp.debian.org/debian/pool/main/f/fades/fades_9.0.1.orig.tar.gz
tar -xf fades_*.tar.gz
cd fades-*
sudo ./setup.py install
```

1.3.6 Can I try it without installing it?

Yes! Branch the project and use the executable:

```
git clone https://github.com/PyAr/fades.git
cd fades
bin/fades your_script.py
```

1.3.7 What about Windows?

Windows is a platform supported by fades.

However, we don't have a proper Windows installer (a `.exe` or `.msi`), but you can install it using `pip`, or from the tarball, or try it directly from the project. All these options are properly described above.

We *do* want to have a Windows installer. If you can help us in this regard, please contact us. Also we would want a Travis running in Windows so that GitHub runs all the tests in this platform too before landing any code. Thanks!

1.4 Get some help, give some feedback

You can ask any question or send any recommendation or request to the mailing list.

Come chat with us on IRC. The #fades channel is located at the Freenode network.

Also, you can open an issue here (please do if you find any problem!).

Thanks in advance for your time.

1.5 How to develop fades itself

Quick guide to get you up and running in fades development.

1.5.1 Getting the code

Clone the project:

```
git clone git@github.com:PyAr/fades.git
```

1.5.2 Install dependencies

fades manages it's own dependencies, so there is nothing extra you need to install.

To try it, just do:

```
bin/fades -V
```

1.5.3 How to run the tests

When starting development, at all times, and specially before wrapping up a new branch, you need to be sure that all tests pass ok.

This is very simple, actually, just run:

```
./test
```

That will not only check test cases, but also that the code complies with aesthetic recommendations, and that the README document has a proper format.

If you want to run *one* particular test, just specify it. Example:

```
./test tests.test_main:DepsMergingTestCase.test_two_different
```

1.5.4 Development process

Just pick an issue from the list.

Develop, assure `./test` is happy, commit, push, create a pull request, etc.

Please, if you aim for creating a Pull Request with new code (functionality or fixes), include tests for your changes.

Thanks! Enjoy.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search