Don’t be afraid to commit Documentation

Release 0.3

Daniele Procida

Sep 12, 2017

Contents

CHAPTER 1

Introduction

A workshop/tutorial for Python/Django developers who would like to contribute more to the projects they use, but need more grounding in some of the tools required.

The workshop will take participants through the complete cycle of identifying a simple issue in a Django or Python project, writing a patch with documentation, and submitting it.

The purpose of the workshop

Don't be afraid to commit will help put you in a position to commit successfully to collaborative projects.

You'll find it particularly useful if you think you have some good coding ideas, but find that managing the development process sometimes gets in the way of your actual development.

What's covered

virtualenv and **pip** will help you manage your own work in a more streamlined and efficient way.

Git and **GitHub** will also help you manage your own workflow and development, and will make it possible for you to collaborate effectively with others. The Django Project, like many other open projects, uses both.

Documentation - being able to create, manage and publish documentation in an efficient and orderly way will make your work more accessible and more interesting to other people.

Contributing - how to submit your work

CHAPTER 2

Contents

Prerequisites

What you need to know

The tutorial assumes some basic familiarity with the commandline prompt in a terminal.

You'll need to know how to install software. Some of the examples given refer to Debian/Ubuntu's `apt-get`; you ought to know what the equivalent is on whatever operating system you're using.

You'll also need to know how to edit plain text or source files of various kinds.

It will be very useful to have some understanding of Python, but it's not strictly necessary. However, if you've never done any programming, that will probably be an obstacle.

Software

The tutorial assumes that you're using a Unix-like system, but there should be no reason why (for example) it would not work for Windows users.

You'll need a suitable text editor, that you're comfortable using.

Other software will be used, but the tutorial will discuss its installation.

However, your environment *does* need to be set up appropriately, and you *will* need to know how to use it effectively.

If your system already has Django and/or Python packages running that you've installed, you probably already have what you need and know what you need to know. All the same:

Platform-specific notes

GNU/Linux

Please make sure that you know how to use your system's package manager, whether it's `aptitude` or `YUM` or something else.

Mac OS X

There are two very useful items that you should install.

- Command Line Tools for Xcode, various useful components (requires registration)
- Homebrew, a command line package manager

Python

You'll need a reasonably up-to-date version of Python installed on your machine. 2.6 or newer should be fine.

Git

Please do check you can install Git:

```
sudo apt-get install git # for Debian/Ubuntu users
```

or:

```
brew install git # for Mac OS X users with Homebrew installed
```

There are other ways of installing Git; you can even get a graphical Git application, that will include the commandline tools. These are described at:

> http://git-scm.com/book/en/Getting-Started-Installing-Git

Virtualenv and pip

In this section you will:

- use pip to install packages
- install virtualenv
- create and destroy, activate and deactivate virtual environments
- use `pip freeze` to show installed items

What is virtualenv?

Virtualenv lets you create and manage virtual Python environments.

If you're running a Python project for deployment or development, the chances are that you'll need more than one version of it, or the numerous other Python applications it depends upon, at any one time.

For example, when a new version of Django is released, you might want to check to see if your project is still compatible. You don't want to have to set up a whole new server with a different version of Django to find out.

With virtualenv, you can quickly set up a brand new Python environment, and install your components into it - along with the new version of Django, without touching or affecting what you already have running.

You can have literally dozens of virtualenvs on the same machine, all running different versions of your Python software, all independently of each other, and can safely make changes to one without affecting anything else.

pip goes hand-in-hand with virtualenv; in fact, it comes with virtualenv (as well as separately). It's an installer, and is the easiest way to install things into a virtualenv.

Installing pip

You will most probably find that pip is already installed on your system.

Run:

```
pip --version
```

to find out.

If it's not, you have various options.

On Debian/Ubuntu systems

```
sudo apt-get install python-pip
```

On Debian you probably will not be authorised to use `sudo`. In this case use:

```
su -
```

to switch to the root user before installing pip.

Use get-pip.py

Another option is to use the official get-pip.py script.

Install virtualenv

Try:

```
virtualenv --version
```

Keep it up-to-date:

```
sudo pip install --upgrade virtualenv
hash -r # purge shell's PATH, though this may not be necessary for you
```

If you got a "Command not found" when you tried to use virtualenv, try:

```
sudo pip install virtualenv
```

or:

```
sudo apt-get install python-virtualenv # for a Debian-based system
```

If that fails or you're using a different system, you might need more help:

Virtualenv installation documentation

Create and activate a virtual environment

```
virtualenv my-first-virtualenv
cd my-first-virtualenv
source bin/activate
```

Note: **Windows users** should run `Scripts\activate` instead of `source bin/activate`.

Notice how your command prompt tells you that the virtualenv is active (and it remains active even while you're not in its directory):

```
(my-first-virtualenv)~/my-first-virtualenv$
```

Using pip

pip freeze

`pip freeze` lists installed Python packages:

```
(my-first-virtualenv)daniele@v029:~/my-first-virtualenv$ pip freeze
argparse==1.2.1
distribute==0.6.24
pyasn1==0.1.7
virtualenv==1.9.1
wsgiref==0.1.2
```

pip install

Earlier, you may have used `sudo pip install`. You **don't** need `sudo` now, because you're in a virtualenv. Let's install something.

```
pip install rsa
```

pip will visit PyPI, the Python Package Index, and will install Python-RSA (a "Pure-Python RSA implementation"). It will also install its dependencies - things it needs - if any have been listed at PyPI.

Now see what `pip freeze` reports. You will probably find that as well as Python-RSA it installed some other packages - they were ones that Python-RSA needed.

And try:

```
(my-first-virtualenv)~/my-first-virtualenv$ python
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import rsa
```

To uninstall it:

```
pip uninstall rsa
```

To install a particular version:

```
pip install rsa==3.0
```

To ugrade the package to the latest version:

```
pip install --upgrade rsa
```

Where packages get installed

Your virtualenv has a site-packages directory, in the same way your system does. So now rsa can be found in:

```
~/my-first-virtualenv/lib/python2.7/site-packages/rsa
```

(It's possible that you'll have a different version of Python listed in that path.)

Dependencies

Python-RSA doesn't have any dependencies, but if it did, and if those dependencies had dependencies, pip would install them all.

So if all the package authors have done a good job of informing PyPI about their software's requirements, you can install a Django application, for example, and pip will will install it, and Django, and possibly dozens of other pieces of software, all into your virtualenv, and without your having to make sure that everything required is in place.

Managing virtualenvs

Create a second virtualenv

```
cd ~/ # let's not create it inside the other...
virtualenv my-second-virtualenv
```

When you activate your new virtualenv, it will deactivate the first:

```
cd my-second-virtualenv
source bin/activate
```

Note: **Windows users**: don't forget to use `Scripts\activate` rather than `source bin/activate`.

`pip freeze` will show you that you don't have Python-RSA installed in this one - it's a completely different Python environment from the other, and both are isolated from the system-wide Python setup.

Deactivate a virtualenv manually

Activating a virtualenv automatically deactivates one that was previously active, but you can also do this manually:

```
deactivate
```

Now you're no longer in any virtualenv.

–system-site-packages

When you create a virtualenv, it doesn't include any Python packages already installed on your system. But sometimes you do want to install all packages. In that case you'd do:

```
virtualenv --system-site-packages my-third-virtualenv
```

remove a virtualenv

virtualenvs are disposable. You can get rid of these:

```
cd ~/
rm -r  my-first-virtualenv my-second-virtualenv my-third-virtualenv
```

And that's pretty much all you need to get started and to use pip and virtualenv effectively.

Git and GitHub

Git and GitHub

In this section you will:

- create a GitHub account
- create your own fork of a repository
- create a new Git branch
- edit and commit a file on GitHub
- make a pull request
- merge upstream changes into your fork

What is it?

Git is a source code management system, designed to support collaboration.

GitHub is a web-based service that hosts Git projects, including Django itself: https://github.com/django/django.

The key idea in Git is that it's *distributed*. If you're not already familiar with version control systems, then explaining why this is important will only introduce distinctions and complications that you don't need to worry about, so that's the last thing I will say on the subject.

Set up a GitHub account

- sign up at GitHub if you don't already have an account

It's free.

Some basic editing on GitHub

Forking

- visit https://github.com/evildmp/afraid-to-commit/

You can do various things there, including browsing through all the code and files.

- hit the **Fork** button

A few moments later, you'll have your own copy, on GitHub, of everything in that repository, and from now on you'll do your work on your copy of it.

Your copy is at `https://github.com/<your github account>/afraid-to-commit/`.

Note: About angular brackets

Angular brackets, such as those in the line above, are used as placeholders; they do not need to be included in the command line terminal. Just write in your GitHub account name. The same applies for any angular brackets used throughout the tutorial.

You will typically do this for any Git project you want to contribute to. It's good for you because it means you don't have to sign up for access to a central repository to be permitted to work on it, and even better for the maintainers because they certainly don't want to be managing a small army of volunteers on top of all their other jobs.

Note: Don't worry about all the forks

You'll notice that there might be a few forks of https://github.com/evildmp/afraid-to-commit; if you have a look at https://github.com/django/django you'll see thousands. There'll even be forks of the forks. Every single one is complete and independent. So, which one is the real one - which one is *the* Django repository?

In a technical sense, they all are, but the more useful answer is: the one that most people consider to be the canonical or official version.

In the case of Django, the version at https://github.com/django/django is the one that forms the basis of the package on PyPI, the one behind the https://djangoproject.com/ website, and above all, it's the one that the community treats as cannonical and official, not because it's the original one, but because it's the most useful one to rally around.

The same goes for https://github.com/evildmp/afraid-to-commit and its more modest collection of forked copies. If I stop updating it, but someone else is making useful updates to their own fork, then in time theirs might start to become the one that people refer to and contribute to. This could even happen to Django itself, though it's not likely to any time soon.

The proliferation of forks doesn't somehow dilute the original. Don't be afraid to create more. Forks are simply the way collaboration is made possible.

Create a new branch

Don't edit the *master* (default) branch of the repository. It's much better to edit the file in a new branch, leaving the *master* branch clean and untouched:

1. select the **branch** menu
2. in *Find or create a branch...* enter `add-my-name`

3. hit **Create branch: add-my-name**

Note: Don't hesitate to branch

As you may have noticed on GitHub, a repository can have numerous branches within it. Branches are ways of organising work on a project: you can have a branch for a new feature, for trying out something new, for exploring an issue - anything at all.

Just as virtualenvs are disposable, so are **branches** in Git. You *can* have too many branches, but don't hesitate to create new ones; it costs almost nothing.

It's a good policy to create a new branch for every new bit of work you start doing, even if it's a very small one.

It's especially useful to create a new branch for every new feature you start work on.

Branch early and branch often. If you're in any doubt, create a new branch.

Edit a file

GitHub allows you to edit files online. This isn't the way you will normally use Git, and it's certainly not something you'll want to spend very much time doing, but it's handy for very small changes, for example typos and spelling mistakes you spot.

1. go to `https://github.com/<your github account>/afraid-to-commit`
2. find the `attendees_and_learners.rst` file
3. hit the **Edit** button
4. add your name (just your name, you will add other information later) to the appropriate place in the file. If you're following the tutorial by yourself, add your details in the *I followed the tutorial online* section.

Commit your changes

- hit **Commit Changes**

Now *your* copy of the file, the one that belongs to *your* fork of the project, has been changed; it's reflected right away on GitHub.

If you managed to mis-spell your name, or want to correct what you entered, you can simply edit it again.

What you have done now is make some changes, in a new branch, in your own fork of the repository. You can see them there in the file.

Make a Pull Request

When you're ready to have your changes incorporated into my original/official/canonical repository, you do this by making a **Pull Request**.

- go back to `https://github.com/<your github account>/afraid-to-commit`

You'll see that GitHub has noted your recent changes, and now offers various buttons to allow you to compare them with the original or make a pull request.

- hit **Compare & pull request**

This will show you a *compare view*, from which you can make your pull request.

When preparing for a pull request, GitHub will show you what's being compared:

```
evildmp:master ... <your github account>:add-my-name
```

On the left is the **base** for the comparison, my fork and branch. On the right is the **head**, your fork and branch, that you want to compare with it.

A pull request goes from the **head** to the **base** - from right to left.

You can change the bases of the comparison if you need to:

1. hit **Edit**
2. select the forks and branches as appropriate

You want your version, the **head branch** of the **head fork** - on the right - with some commits containing file changes, to be sent to my **base repo** - on the left.

1. hit the **Pull Request** button
2. add a comment if you like (e.g. "please add me to the attendees list")
3. hit **Send pull request**

You have now made a pull request to an open-source community project - if it's your first one, congratulations.

GitHub will notify me (by email and on the site, and will show me the changes you're proposing to make). It'll tell me whether they can be merged in automatically, and I can reject, or accept, or defer a decision on, or comment on, your pull request.

GitHub can automatically merge your contribution into my repository if mine hasn't changed too much since you forked it. If I want to accept it but GitHub can't do it automatically, I will have to merge the changes manually (we will cover this later).

Once they're merged, your contributions will become a part of https://github.com/evildmp/afraid-to-commit. And this is the basic lifecycle of a contribution using git: *fork* > *edit* > *commit* > *pull request* > *merge*.

Incorporate upstream changes into your master

In the meantime, other people may have made their own forks, edits, commits, and pull requests, and I may have merged those too - other people's names may now be in the list of attendees. Your own version of afraid-to-commit, *downstream* from mine, doesn't yet know about those.

Since your work is based on mine, you can think of my repository as being *upstream* of yours. You need to merge any *upstream* changes into *your* version, and you can do this with a pull request on GitHub too.

This time though you will need to switch the bases of the comparison around, because the changes will be coming from *my version* to *yours*.

1. hit **Pull Request** once more
2. hit **Edit**, to switch the bases
3. change the **head repo** on the right to *my* version, `evildmp/afraid-to-commit`, branch *master*
4. change the **base repo** to yours, and the **base branch** to *master*, so the comparison bar looks like:

   ```
   <your github account>:master ... evildmp:master
   ```

5. hit **Click to create a pull request for this comparison**

6. add a **Title** (e.g. "merging upstream master on Github) and hit **Send pull request**

You're sending a pull request to *yourself*, based on updates in my repository. And in fact if you check in your **Pull Requests** on GitHub, you'll see one there waiting for you, and you too can review, accept, reject or comment on it.

If you decide to **Merge** it, your fork will now contain any changes that other people sent to me and that I merged.

The story of your work is this: you **forked** away from my codebase, and then created a new **branch** in your fork.

Then you **committed** changes to your branch, and sent them **upstream** back to me (with a **pull request**).

I **merged** your changes, and perhaps those from other people, into my codebase, and you **pulled** all my recent changes back into your *master* branch (again with a **pull request**).

So now, your *master* and mine are once more in step.

Git on the commandline

In this section you will:

- install and configure Git locally
- create your own local clone of a repository
- create a new Git branch
- edit a file and stage your changes
- commit your changes
- push your changes to GitHub
- make a pull request
- merge upstream changes into your fork
- merge changes on GitHub into your local clone

So far we've done all our Git work using the GitHub website, but that's usually not the most appropriate way to work.

You'll find that most of your Git-related operations can and need to be done on the commandline.

Install/set up Git

```
sudo apt-get install git # for Debian/Ubuntu users
brew install git # for Mac OS X users with Homebrew installed
```

There are other ways of installing Git; you can even get a graphical Git application, that will include the commandline tools. These are described at:

http://git-scm.com/book/en/Getting-Started-Installing-Git

Tell Git who you are

First, you need to tell Git who you are:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

Give GitHub your public keys

This is a great timesaver: if GitHub has your public keys, you can do all kinds of things from your commandline without needing to enter your GitHub password.

- https://github.com/settings/ssh

https://help.github.com/articles/generating-ssh-keys explains much better than I can how to generate a public key.

This tutorial assumes you have now added your public key to your GitHub account. If you haven't, you'll have to use *https* instead, and translate from the format of GitHub's *ssh* URLS.

For example, when you see:

```
git@github.com:evildmp/afraid-to-commit.git
```

you will instead need to use:

```
https://github.com/evildmp/afraid-to-commit.git
```

See https://gist.github.com/grawity/4392747 for a discussion of the different protocols.

Some basic Git operations

When we worked on GitHub, the basic work cycle was *fork* > *edit* > *commit* > *pull request* > *merge*. The same cycle, with a few differences, is what we will work through on the commandline.

Clone a repository

When you made a copy of the *Don't be afraid to commit* repository on GitHub, that was a *fork*. Getting a copy of a repository onto your local machine is called *cloning*. Copy the *ssh URL* from `https://github.com/<your github account>/afraid-to-commit`, then:

```
git clone git@github.com:<your github account>/afraid-to-commit.git
```

Change into the newly-created `afraid-to-commit` directory, where you'll find all the source code of the *Don't be afraid to commit* project.

Now you're in the **working directory**, the set of files that you currently have in front of you, available to edit. We want to know its status:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Create a new branch

Just as you did on GitHub, once again you're going to create a new branch, based on *master*, for new work to go into:

```
$ git checkout -b amend-my-name
Switched to a new branch 'amend-my-name'
```

`git checkout` is a command you'll use a lot, to switch between branches. The `-b` flag tells it to **create a new branch** at the same time. By default, the new branch is based upon whatever branch you were on.

You can also choose what to base the new branch on. A quite common thing to do is, just for example:

```
git checkout -b new-branch existing-branch
```

This creates a new branch `new-branch`, based on `existing-branch`.

Edit a file

1. find the `attendees_and_learners.rst` file in your working directory
2. after your name and email address, add your Github account name
3. save the file

`git status` is always useful:

```
$ git status
# On branch amend-my-name
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   attendees_and_learners.rst
#
no changes added to commit (use "git add" and/or "git commit -a")
```

What this is telling us:

- we're on the *amend-my-name* branch
- that we have one modified file
- that there's nothing to commit

These changes will only be applied to this branch when they're committed. You can `git add` changed files, but until you commit they won't belong to any particular branch.

Note: When to branch

You didn't actually *need* to create your new *amend-my-name* branch until you decided to commit. But creating your new branches before you start making changes makes it less likely that you will forget later, and commit things to the wrong branch.

Stage your changes

Git has a **staging area**, for files that you want to commit. On GitHub when you edit a file, you commit it as soon as you save it. On your machine, you can edit a number of files and commit them altogether.

Staging a file in Git's terminology means adding it to the staging area, in preparation for a commit.

Add your amended file to the staging area:

```
git add attendees_and_learners.rst
```

and check the result:

```
$ git status
# On branch amend-my-name
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   attendees_and_learners.rst
#
```

If there are other files you want to change, you can add them when you're ready; until you commit, they'll all be together in the staging area.

What gets staged?

It's not your files, but the **changes to your files**, that are staged. Make some further change to `attendees_and_learners.rst`, and run `git status`:

```
$ git status
# On branch amend-my-name
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   attendees_and_learners.rst
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   attendees_and_learners.rst
#
```

Some of the changes in `attendees_and_learners.rst` will be committed, and the more recent ones will not.

- run `git add` on the file again to stage the newer changes

Commit your changes

When you're happy with your files, and have added the changes you want to commit to the staging area:

```
git commit -m "added my github name"
```

The `-m` flag is for the message ("added my github name") on the commit - every commit needs a commit message.

Push your changes to GitHub

When you made a change on GitHub, it not only saved the change and committed the file at the same time, it also showed up right away in your GitHub repository. Here there is an extra step: we need to **push** the files to GitHub.

If you were pushing changes from *master* locally to *master* on GitHub, you could just issue the command `git push` and let Git work out what needs to go where.

It's always better to be explicit though. What's more, you have multiple branches here, so you need to tell git *where* to push (i.e. back to the remote repository you cloned from, on GitHub) and *what* exactly to push (your new branch).

The repository you cloned from - yours - can be referred to as **origin**. The new branch is called *amend-my-name*. So:

```
$ git push origin amend-my-name
Counting objects: 34, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (28/28), 6.87 KiB, done.
Total 28 (delta 13), reused 12 (delta 7)
To git@github.com:evildmp/afraid-to-commit.git
 * [new branch]      amend-my-name -> amend-my-name
```

Note: Be explicit!

Next time you want to push committed changes in *amend-my-name*, you won't *need* to specify the branch - you can simply do `git push`, because now *amend-my-name* exists at both ends. However, it's *still* a good idea to be explict. That way you'll be less likely to get a surprise you didn't want, when the wrong thing gets pushed.

Check your GitHub repository

- go to https://github.com/<your GitHub name>/afraid-to-commit
- check that your new *amend-my-name* branch is there
- check that your latest change to `attendees_and_learners.rst` is in it

Send me a pull request

You can make more changes locally, and continue committing them, and pushing them to GitHub. When you've made all the changes that you'd like me to accept though, it's time to send *me* a pull request.

Important: make sure that you send it from your new branch *amend-my-name* (not from your *master*) the way you did before.

And if I like your changes, I'll merge them.

Note: Keeping master 'clean'

You *could* of course have merged your new branch into your *master* branch, and sent me a pull request from that. But, once again, it's a good policy to keep your *master* branch, on GitHub too, clean of changes you make, and only to pull things into it from upstream.

In fact the same thing goes for other branches on my upstream that you want to work with. Keeping them clean isn't strictly necessary, but it's nice to know that you'll always be able to pull changes from upstream without having to tidy up merge conflicts.

Incorporate upstream changes

Once again, I may have merged other people's pull requests too. Assuming that you want to keep up-to-date with my changes, you're going to want to merge those into your GitHub fork as well as your local clone.

So:

- on GitHub, pull the upstream changes into your fork the way you did previously

Then switch back to your master branch in the usual way (`git checkout master`). Now, fetch updated information from your GitHub fork (**origin**), and merge the master:

```
git fetch
git merge origin/master
```

So now we have replicated the full cycle of work we described in the previous module.

Note: `git pull`

Note that here instead of `git fetch` followed by `git merge`, you could have run `git pull`. The `pull` operation does two things: it **fetches** updates from your GitHub fork (**origin**), and **merges** them.

However, be warned that occasionally `git pull` won't always work in the way you expect, and doing things the explicit way helps make what you are doing clearer.

`git fetch` followed by `git merge` is generally the safer option.

Switching between branches locally

Show local branches:

```
git branch
```

You can switch between local branches using `git checkout`. To switch back to the *master* branch:

```
git checkout master
```

If you have a changed tracked file - a tracked file is one that Git is managing - it will warn you that you can't switch branches without either committing, abandoning or 'stashing' the changes:

Commit

You already know how to commit changes.

Abandon

You can abandon changes in a couple of ways. The recommended one is:

```
git checkout <file>
```

This checks out the previously-committed version of the file.

The one that is not recommended is:

```
git checkout -f <branch>
```

The `-f` flag forces the branch to be checked out.

Note: Forcing operations with `-f`

> Using the `-f` flag for Git operations is to be avoided. It offers plenty of scope for mishap. If Git tells you about a problem and you force your way past it, you're inviting trouble. It's almost always better to find a different way around the problem than forcing it.
>
> `git push -f` in particular has ruined a nice day for many people.

Stash

If you're really interested, look up `git stash`, but it's beyond the scope of this tutorial.

Working with remotes

In this section you will:

- add a remote repository to your local clone
- fetch remote information
- checkout a remote branch
- merge an upstream branch

Managing remotes

Your repository on GitHub is the **remote** for the clone on your local machine. By default, your clone refers to that remote as **origin**. At the moment, it's the only remote you have:

```
$ git remote
origin

$ git remote show origin
    * remote origin
      Fetch URL: git@github.com:evildmp/afraid-to-commit.git
      Push  URL: git@github.com:evildmp/afraid-to-commit.git
      HEAD branch: master
      Remote branches:
        amend-my-name tracked
        master        tracked
      Local branch configured for 'git pull':
        master merges with remote master
  Local refs configured for 'git push':
    amend-my-name pushes to amend-my-name (up to date)
    master        pushes to master        (up to date)
```

It's very useful for Git to know about other remote repositories too. For example, at the end of the previous section, we considered a conflict between your GitHub fork and and the upstream GitHub repository. The only way to fix that is locally, on the command line, and by being able to refer to both those remotes.

Now you *can* refer to a remote using its full address:

> https://github.com/evildmp/afraid-to-commit

But just as your remote is called **origin**, you can give any remote a more memorable name. Your own **origin** has an upstream repository (mine); it's a convention to name that **upstream**.

Add a remote

```
git remote add upstream git@github.com:evildmp/afraid-to-commit.git
```

Fetch remote data

The remote's there, but you don't yet have any information about it. You need to **fetch** that:

```
git fetch upstream
```

This means: get the latest information about the branches on **upstream**.

List remote branches

`git branch` shows your local branches.

To see a list of them all, including the remote branches:

```
git branch -a
```

Checkout a remote branch

You'll have seen from `git branch -a` that there's a branch called *additional-branch* on the **upstream** repository.

You can check this out:

```
git checkout -b additional-branch upstream/additional-branch
```

This means: create and switch to a new local branch called *additional-branch*, based on branch *additional-branch* of the remote **upstream**.

Managing *master* on the commandline

Until now, you have only updated your *master* on GitHub using GitHub itself. Sometimes it will be much more convenient to do it from your commandline. There are various ways to do it, but here's one:

```
git checkout master # switch back to the master branch
git fetch upstream # update information about the remote
git merge upstream/master # merge the changes referred to by
upstream/master
```

`git status` will tell you that your local master is ahead of your *master* at **origin**.

Push your changes to master:

```
git push origin master
```

And now your *master* on GitHub contains everything my *master* does; it's up-to-date and clean.

Resolving conflicts

In this section you will:

- encounter a merge conflict on GitHub
- encounter a merge conflict on the commandline
- resolve the conflict in a new temporary Git branch

Encountering a merge conflict on GitHub

Sometimes you'll discover that your GitHub fork and the upstream repository have changes that GitHub can't merge.

There's an *unmergeable-branch* at https://github.com/evildmp/afraid-to-commit. It's unmergeable because it deliberately contains changes that conflict with other changes made in *master*.

Using the GitHub interface, try creating a pull request from *unmergeable-branch* to your *master* on GitHub. If you do, GitHub will tell you:

```
We can't automatically merge this pull request.

Use the command line to resolve conflicts before continuing.
```

GitHub will in fact tell you the steps you need to take to solve this, but to understand what's actually happening, and to do it yourself when you need to, we need to cover some important concepts.

Merging changes from a remote branch

```
# make sure you have the latest data from upstream
$ git fetch upstream
# create and switch to a new branch based on master to explore the conflict
$ git checkout -b explore-conflict upstream/master
# now try merging the unmergeable-branch into it
$ git merge upstream/unmergeable-branch
Auto-merging attendees_and_learners.rst
CONFLICT (content): Merge conflict in attendees_and_learners.rst
Automatic merge failed; fix conflicts and then commit the result.
```

When there's a conflict, Git marks them for you in the files. You'll see sections like this:

```
<<<<<<< HEAD
* Daniel Pass <daniel.antony.pass@googlemail.com>
* Kieran Moore
=======
* Kermit the Frog
* Miss Piggy
>>>>>>> upstream/unmergeable-branch
```

The first section, `HEAD` is what you have in your version. The second section, `upstream/unmergeable-branch` is what Git found in the version you were trying to pull in.

You'll have to decide what the file should contain, and you'll need to edit it. If you decide you want the changes from both versions:

```
* Daniel Pass <daniel.antony.pass@googlemail.com>
* Kieran Moore
* Kermit the Frog
* Miss Piggy

$ git add attendees_and_learners.rst
$ git commit -m "fixed conflict"
[explore-conflict 91a45ac] fixed conflict
```

Note: Create new branches when resolving conflicts

It's very sensible *not* to do merging work in a branch you have done valuable work in. In the example above, your *explore-conflict* branch is based on master and doesn't contain anything new, so it will be easy to re-create if it all goes wrong.

If you had a branch that contained many complex changes however, you certainly wouldn't want to discover dozens of conflicts making a mess in the files containing all your hard work.

So remember, **branches are cheap and disposable**. Rather than risk messing up the branch you've been working on, create a new one specially for the purpose of discovering what sort of conflicts arise, and to give you a place to work on resolving them without disturbing your work so far.

You might have conflicts across dozens of files, if you were unlucky, so it's very important to be able to backout gracefully and at the very least leave things as they were.

More key Git techniques

There are a few more key Git commands and techniques that you need to know about.

.gitignore

When you're working with Git, there are lots of things you won't want to push, including:

- hidden system files
- .pyc files
- rendered documentation files
- ... and many more

`.gitignore`, https://github.com/evildmp/afraid-to-commit/blob/master/.gitignore, is what controls this. You should have a `.gitignore` in your projects, and they should reflect *your* way of working. Mine include the things that my operating system and tools throw into the repository; you'll find soon enough what yours are.

With a good `.gitignore`, you can do things like:

```
git add docs/
```

and add whole directories at a time without worrying about including unwanted files.

Starting a new Git project

You've been working so far with an existing Git project. It's very easy to start a brand new project, or turn an existing one into a Git project. On GitHub, just hit the **New repository** button and follow the instructions.

Combining Git and pip

When you used pip to install a package inside a virtualenv, it put it on your Python path, that is, in the virtualenv's `site-packages` directory. When you're actually working on a package, that's not so convenient - a Git project is the most handy thing to have.

On the other hand, cloning a Git repository doesn't install it on your Python path (assuming that it's a Python application), so though you can work on it, you can't actually use it and test it as an installed package.

However, pip is Git-aware, and can install packages *and* put them in a convenient place for editing - so you can get both:

```
cd ~/
virtualenv git-pip-test
source git-pip-test/bin/activate
pip install -e git+git@github.com:python-parsley/parsley.git#egg=parsley
```

The `-e` flag means editable; `git+` tells it to use the Git protocol; `#egg=parsley` tells it what to call it.

(Should you find that this causes an error, try using quotes around the target:

```
pip install -e "git+git@github.com:python-parsley/parsley.git#egg=parsley"
```

)

You can also specify the branch:

```
pip install -e git+git@github.com:python-parsley/parsley.git@master#egg=parsley
```

And now you will find an editable Git repository installed at:

    ~/git-pip-test/src/parsley

which is where any other similarly-installed packages will be, and just to prove that it really is installed:

```
$ pip freeze
-e git+git@github.com:python-parsley/parsley.
↪git@e58c0c6d67142bf3ceb6eceffd50cf0f8dae9da1#egg=Parsley-master
wsgiref==0.1.2
```

Git is a source code management system, designed to support collaboration.

GitHub is a web-based service that hosts Git projects, including Django itself: https://github.com/django/django.

Git is a quite remarkable tool. It's fearsomely complex, but you can start using it effectively without needing to know very much about it. All you really need is to be familiar with some basic operations.

The key idea in Git is that it's *distributed*. If you're not already familiar with version control systems, then explaining why this is important will only introduce distinctions and complications that you don't need to worry about, so that's the last thing I will say on the subject.

Documentation using Sphinx and ReadTheDocs.org

Without documentation, however wonderful your software, other potential adopters and developers simply won't be very interested in it.

The good news is that there are several tools that will make presenting and publishing it very easy, leaving you only to write the content and mark it up appropriately.

For documentation, we'll use **Sphinx** to generate it, and **Read the Docs** to publish it. GitHub will be a helpful middleman.

If you have a package for which you'd like to create documentation, you might as well start producing that right away. If not, you can do it in a new dummy project.

Set up your working environment

The virtualenv

As usual, create and activate a new virtualenv:

```
virtualenv documentation-tutorial
[...]
cd documentation-tutorial/
source bin/activate
```

The package or project

If you have an existing package to write documentation for

If your package is on GitHub already and you want to start writing documention for, clone it now using Git. And of course, start a new branch:

```
git checkout -b first-docs
```

You can merge your docs into your master branch when they start to look respectable.

If you don't have an existing package that needs docs

If you don't have a suitable existing package on GitHub, create a repository on GitHub the way you did before. Call it `my-first-docs`. Then create a Git repository locally:

```
mkdir my-first-docs
cd my-first-docs/
# Converts the directory into a git repository
git init
# Point this repo at the GitHub repo you just created
git remote add origin git@github.com:<your git username>/my-first-docs.git
# Create a new branch in which to do your work
git checkout -b first-docs
```

Create a `docs` directory

And either way, create a `docs` directory for your docs to live in:

```
mkdir docs
```

Sphinx

Install Sphinx

```
pip install sphinx
```

It might take a minute or so, it has quite a few things to download and install.

sphinx-quickstart

`sphinx-quickstart` will set up the source directory for your documentation. It'll ask you a number of questions. Mostly you can just accept the defaults it offers, and some are just obvious, but there are some you will want to set yourself as noted below:

```
sphinx-quickstart
```

Root path for the documentation `docs`

Project name `<your name>'s first docs`, or the name of your application

Source file suffix `.rst` is the default. (Django's own documentation uses `.txt`. It doesn't matter too much.)

You'll find a number of files in your `docs` directory now, including `index.rst`. Open that up.

Using Sphinx & reStructuredText

reStructuredText elements

Sphinx uses reStructuredText. http://sphinx-doc.org/rest.html#rst-primer will tell you most of what you need to know to get started. Focus on the basics:

- paragraphs
- lists
- headings ('sections', as Sphinx calls them)
- quoted blocks
- code blocks
- emphasis, strong emphasis and literals

Edit a page

Create an **Introduction** section in the `index.rst` page, with a little text in it; save it.

Create a new page

You have no other pages yet. In the same directory as `index.rst`, create one called `all-about-me.rst` or something appropriate. Perhaps it might look like:

```
############
All about me
############

I'm Daniele Procida, a Django user and developer.

I've contributed to:

*   django CMS
*   Arkestra
*   Django
```

Sphinx needs to know about it, so in `index.rst`, edit the `.. toctree::` section to add the `all-about-me` page:

```
.. toctree::
   :maxdepth: 2

   all-about-me
```

Save both pages.

Render your documentation

In the `docs` directory:

```
make html
```

This tells Sphinx to render your source pages. *Pay attention to its warnings* - they're helpful!

Note: Sphinx can be fussy, and sometimes about things you weren't expecting. For example, you well encounter something like:

```
WARNING: toctree contains reference to nonexisting document u'all-about-me'
...
checking consistency...
<your repository>/my-first-docs/docs/all-about-me.rst::
WARNING: document isn't included in any toctree
```

Quite likely, what has happened here is that you indented `all-about-me` in your `.. toctree::` with *four* spaces, when Sphinx is expecting *three*.

If you accepted the `sphinx-quickstart` defaults, you'll find the rendered pages in `docs/_build/html`. Open the `index.html` it has created in your browser. You should find in it a link to your new `all-about-me` page too.

Publishing your documentation

Exclude unwanted rendered directories

Remember `.gitignore`? It's really useful here, because you don't want to commit your *rendered* files, just the source files.

In my `.gitignore`, I make sure that directories I don't want committed are listed. Check that:

```
_build
_static
_templates
```

are listed in `.gitignore`.

Add, commit and push

`git add` the files you want to commit; commit them, and push to GitHub.

If this is your first ever push to GitHub for this project, use:

```
git push origin master
```

otherwise:

```
git push origin first-docs # or whatever you called this branch
```

Now have a look at the `.rst` documentation files on GitHub. GitHub does a good enough job of rendering the files for you to read them at a glance, though it doesn't always get it right (and sometimes seems to truncate them).

readthedocs.org

However, we want to get them onto Read the Docs. So go to https://readthedocs.org, and sign up for an account if you don't have one.

You need to **Import** a project: https://readthedocs.org/dashboard/import/.

Give it the details of your GitHub project in the **repo** field - `git@github.com:<your git username>/my-first-docs.git`, or whatever it is - and hit **Create**.

And now Read the Docs will actually watch your GitHub project, and build, render and host your documents for you automatically.

It will update every night, but you can do better still: on GitHub:

1. select **settings** for your project (not for your account) in the navigation panel on the right-hand side
2. choose **Webhooks & Services**
3. enable `ReadTheDocs` under **Add Service** dropdown

... and now, every time you push documents to GitHub, Read the Docs will be informed that you have new documents to be published. It's not magic, but it's pretty close.

Contributing

Having identified a contribution you think you can usefully make to a project, how are you actually going to make it?

For nearly every project, the best first step is:

Talk to somebody about it

You may have been eating, sleeping, dreaming and otherwise living your idea for days, but until you discuss it, no-one else knows anything about it. To them, your patch will come flying out of the blue.

You need - usually - to introduce your idea, and - particularly if you're new to the community - yourself.

In the case of Django, this will typically mean posting to the Django Developers email list, django-developers@googlegroups.com (sign up at http://groups.google.com/group/django-developers), or raising it on #django-dev on irc.freenode.net.

Quite apart from letting people know about what you have to offer, it's an opportunity to get some feedback on your proposal.

Don't automatically expect your proposal to be considered a great idea. Be prepared to explain the need it meets and why you think your solution is a good one. Be prepared to do some more work.

Above all, you will need to be **patient**, **polite** and **persistent**.

File it

If one doesn't exist already, lay down a formal public marker raising the issue your contribution addresses. In Django's case, this will be a ticket on https://code.djangoproject.com/. For others, it's likely to be an issue on GitHub or whatever system they use. *Mention your earlier discussion!*

Do your homework

Every project has its standards for things like code and documentation, and its ways of working. They tend to follow a general pattern, but they often have their own little quirks or preferences - so **learn them**.

If you think that sounds tedious, it's nothing compared to the potential pain of having to manage or use code and documentation written according to the individual preferences of all its different contributors.

- https://docs.djangoproject.com/en/1.7/internals/contributing/
- https://docs.djangoproject.com/en/1.7/internals/contributing/writing-code/working-with-git/

Note that the Django Project's Git guidelines ask contributors to use `rebase` - which is firstly a little unusual, and secondly explained in the documentation above better than I can here - so read that.

And be prepared to get it wrong the first few times, and even the subsequent ones. It happens to everyone.

Submit it

Now you can make your pull request. Having prepared the way for it, and having provided the accompaniments - tests and documentation - that might be required, it has a good chance of enjoying a smooth passage into the project.

What to start with?

Documentation! It's the easy way in.

Everyone loves documentation, and unlike code where incompleteness or vagueness can make it worse than useless, documentation has to be really quite bad to be worse than no documentation.

What's more, writing documentation will help you better understand the things you're writing about, and if you're new to all this, that's going to put you in a better position to understand and improve code.

Some suitable Django Project tickets

Have a look at one of the tickets specially selected for people doing this tutorial. They're not all for documentation, though most are.

Cheatsheet

virtualenv

create `virtualenv [name-of-virtualenv]`

include system's Python packages `virtualenv [name-of-virtualenv] --install-site-packages`

activate `source bin/activate`

deactivate `deactivate`

pip

install `pip install [name-of-package]`

install a particular version `pip install [name-of-package]==[version-number]`

upgrade `pip install --upgrade [name-of-package]`

uninstall `pip uninstall [name-of-package]`

show what's installed `pip freeze`

git

tell git who you are `git config --global user.email "you@example.com"`

`git config --global user.name "Your Name"`

clone a repository `git clone [repository URL]`

checkout `git checkout [branch]` switches to another branch

`git checkout -b [new-branch]` creates and switches to a new branch

`git checkout -b [new-branch] [existing branch]` creates and switches to a new branch based on an existing one

`git checkout -b [new-branch] [remote/branch]` creates and switches to a new branch based on `remote/branch`

`git checkout [commit sha]` checks out the state of the repository at a particular commit

current status of your local branches `git status`

show the commit you're on in the current working directory `git show`

commit `git commit -m "[your commit message]"`

add `git add [filename]` adds a file to the staging areas

`git add -u [filename]` - the `-u` flag will also remove deleted files

remote `git remote add [name] [remote repository URL]` sets up remote

`git remote show` lists remotes

`git remote show -v` lists remotes and their URLs

branch `git branch`

`git branch -a` to show remote branches too

fetch `git fetch` gets the latest information about the branches on the default remote

`git fetch [remote]` gets the latest information about the branches on the named remote

merge `git merge [branch]` merges the named branch into the working directory

`git merge [remote/branch] -m "[message]"` merges the branch referred to into the working directory - **don't forget to fetch the remote before the merge**

pull `fetch` followed by `merge` is often safer than `pull` - don't assume that `pull` will do what you expect it to

`git pull` fetches updates from the default remote and merges into the working directory

push `git push` pushes committed changes to the default remote, in branches that exist at both ends

`git push [remote] [branch]` pushes the current branch to the named `branch` on `remote`

log `git log` will show you a list of commits

Notes

Throughout Git, anything in the form `remote/branchname` is a reference, not a branch.

Documentation

initialise Sphinx documentation `sphinx-quickstart`

render documentation `make html`

Attendees & learners

This is a record of people who attended a *Don't be afraid to commit* workshop, or followed the tutorial in their own time.

Workshops

Pycon Zimbabwe in Harare, 24th November 2016

- Bornwell Matembudze https://github.com/bornie21
- Kudakwashe Siziva
- Akim Munthali https://github.com/amunthali @amunthali

Note: Many thanks to Charles Katuri (charle-k) for his invaluable assistance on Windows computers

PyCon Ireland in Dublin, 26th October 2015

- Simon Parker https://github.com/simonparkerdublin @SparkerDublin
- Anna Szewc, https://github.com/NannahA
- Iain Geddes iaingeddes@theiet.org https://github.com/iaingeddes
- Gearoid Ryan https://github.com/gearoid-ryan
- Jakub Pawlicki https://github.com/JakubPawlicki
- Ivin Polo Sony @ivinpolosony http://github.com/ivinpolosony/
- Lisa Cavern @anninireland https://github.com/anninireland
- Jeremie Jost https://github.com/jjst
- Richard Loy https://github.com/Richloy
- Art Knipe https://github.com/artkgithub
- Miao Li https://github.com/masonmiaoli
- Sarah Jackson
- Stefano Fedele https://github.com/stefanofedele/afraid-to-commit
- Barry Kennedy https://github.com/bazkennedy

PyCon UK in Coventry, 21st September 2015

Note: Many thanks to Helen Sherwood-Taylor (helenst) for her invaluable assistance.

- Valerio Campanella @ValerioCamp https://github.com/VCAMP/
- Laura Dreyer https://github.com/lbdreyer
- Aisha Bello https://github.com/shante66
- Paivi Suomela, https://github.com/peconia
- Neil Stoker, https://github.com/nmstoker
- Charles G Barnwell https://github.com/cgbarnwell
- Jo Williams https://githib.com/crocodile2485 fh07jw
- Sylvain Gubian
- Adam Johns https://github.com/ninjaExploder/
- Glen Davies https://github.com/glen442 @GlenDaviesDev

DjangoGirls in Portland, 27th August 2015

- Lacey Williams Henschel @laceynwilliams
- Megan Norton http://walkermacy.com
- Sara "the" Jensen https://github.com/thejensen
- TB

- Stephanie Marson

DjangoCon Europe in Cardiff, 4th June 2015

- David Bannon https://github.com/sp1ky
- Amy Lai
- Sven Groot <sven@mediamoose.nl>
- Rick de Leeuw <rick@mediamoose.nl>
- Zoe Ballard
- Jeff Doyle
- Stewart Houten
- Lukasz Wojcik
- Tom Bakx
- Marissa Zhou <https://github.com/marissazhou>
- Niels Lensink <nielslensink@gmail.nl>
- Bryan Spence
- Andraz Tori <andraz@zemanta.com>
- Gwilym Jones
- Adrienne Lowe http://codingwithkniv.es, @adriennefriend
- Zoe Ballard <https://github.com/zoe-ann-b>

Dutch Django Association Sprint in Amsterdam, 7th March 2015

- Remco Kranenburg <remco@burgsoft.nl>
- Floris den Hengst
- C.T. Matsumoto <todd@l1nda.nl>
- Loek van Gent <https://github.com/gannetson>
- Nathan Schagen
- Hanna Kollo https://github.com/sztrovacsek
- Stephen Albert https://github.com/psiloLR

PyCon Ireland in Dublin, 13th October 2014

- Randal McGuckin <randal.mcguckin@gmail.com>
- Laura Duggan https://github.com/labhra
- Jenny McGee
- Conor McGee <mcgeeco@tcd.ie> https://github.com/mcgeeco
- Nadja Deininger https://github.com/machinelady
- Andrew McCarthy

- Brian McDonnell <https://github.com/brianmcdonnell/>
- Brendan Cahill (https://github.com/brencahill/)
- Adam Dickey
- Paul O'Grady (Twitter: @paul_ogrady; GITHub: paulogrady)
- Jenny DiMiceli - https://github.com/jdimiceli
- Stephen Kerr
- Wayne Tong
- Vinicius Mayer (viniciusmayer@gmail.com) https://github.com/viniciusmayer
- Dori Czapari https://github.com/doriczapari (@doriczapari)
- Karl Griffin (karl_griffin@hotmail.com) https://github.com/karlgriffin
- Vadims Briksins (https://github.com/Briksins)

PyCon UK in Coventry, 20th September 2014

- Matthew Power https://github.com/mthpower
- Brendan Oates <brenoates@gmail.com>
- Waldek Herka (https://github.com/wherka)
- Stephen Newey (@stevenewey) - https://github.com/stevenewey
- Walter Kummer (work.walter at gmail.com)
- Craig Barnes
- Justin Wing Chung Hui
- Davide Ceretti
- Paul van der Linden https://github.com/pvanderlinden
- Gary Martin https://github.com/garym
- Cedric Da Costa Faro https://github.com/cdcf
- Sebastien Charret <sebastien.charret@gmail.com> https://github.com/moerin
- Nick Smith
- Jonathan Lake-Thomas https://github.com/jonathlt
- Ben Mansbridge
- Glen Davies (@GlenDaviesDev) - https://github.com/glen442
- Mike S Collins (MikeyBoy1969)

DjangoCon US in Portland, 5th September 2014

- Joseph Metzinger (joseph.metzinger@gmail.com) https://github.com/joetheone
- Abdulaziz Alsaffar (alsaff1987@gmail.com) https://github.com/Octowl
- Patrick Beeson (@patrickbeeson) https://github.com/patrickbeeson
- Vishal Shah - https://github.com/shahv

- Kevin Daum (@kevindaum, kevin.daum@gmail.com) https://github.com/kevindaum
- Nasser AlSnayen (nasser.lc9@gmail.com) https://github.com/LC9
- Nicholas Colbert (@45cali) 45cali@gmail.com
- Chris Cauley https://github.com/chriscauley
- Joe Larson (@joelarson)
- Jeff Kile
- Orlando Romero
- Chad Hansen (chadgh@gmail.com) https://github.com/chadgh

DjangoVillage in Orvieto, 14th June 2014

- Gioele
- Christian Barra (@christianbarra) https://github.com/barrachri
- Luca Ippoliti https://github.com/lucaippo
- @joke2k (https://github.com/joke2k)
- Domenico Testa (@domtes)
- Alessio
- Diego Magrini (http://github.com/magrinidiego)
- Matteo (@loacker) https://github.com/loacker
- Simone (@simodalla) https://github.com/simodalla

DjangoCon Europe on The Île des Embiez, 16th May 2014

- Niclas Åhdén (niclas@brightweb.se) https://github.com/niclas-ahden
- Sabine Maennel (sabine.maennel@gmail.com) http://github.com/sabinem
- JB (Juliano Binder)
- Laurent Paoletti
- Alex Semenyuk (https://github.com/gtvblame)
- Moritz Windelen
- Marie-Cécile Gohier
- Isabella Pezzini
- Pavel Meshkoy (@rasstreli)

Dutch Django Association Sprint in Amsterdam, 22nd February 2014

- Stomme poes (@stommepoes)
- Rigel Di Scala (zedr) <zedr@zedr.com> http://github.com/zedr
- Nikalajus Krauklis (@dzhibas) http://github.com/dzhibas
- Ivo Flipse (@ivoflipse5) https://github.com/ivoflipse

- Martin Matusiak
- Jochem Oosterveen https://github.com/jochem
- Pieter Marres
- Nicolaas Heyning (L1NDA)
- Henk Vos h.vos@rapasso.nl https://github.com/henkvos
- Adam Kaliński @ https://github.com/adamkal
- Marco B
- Greg Chapple http://github.com/gregchapple/
- Vincent D. Warmerdam vincentwarmerdam@gmail.com
- Lukasz Gintowt (syzer)
- Bastiaan van der Weij
- Maarten Zaanen <maarten at PZvK.com><Maarten at Zaanen.net>
- Markus Holtermann (@m_holtermann)

Django Weekend Cardiff, 7th February 2014

- Jakub Jarosz (@qba73) jakub.s.jarosz@gmail.com https://github.com/qba73
- Stewart Perrygrove
- Adrian Chu
- Baptiste Darthenay

PyCon Ireland in Dublin, 14th October 2013

- Vincent Hussey vincent.hussey@opw.ie https://github.com/VincentHussey
- Padraic Harley <@pauricthelodger> <padraic@thelodgeronline.com>
- Paul Cunnane <paul.cunnane@gmail.com> https://github.com/paulcunnane
- Sorcha Bowler <saoili @ github, twitter, gmail, most of the internet>
- Jennifer Parak https://github.com/jenpaff
- Andrea Fagan
- Jennifer Casavantes
- Pablo Porto https://github.com/portovep
- Tianyi Wang <wty52133@gmail.com> @TianyiWang33
- James Heslin <program.ix.j@gmail.com> https://github.com/PROGRAM-IX
- Sorcha Bowler <saoili@gmail.com. saoili on github, twitter, most of the internet>
- Larry O'Neill (larryone)
- Samuel <satiricallaught@gmail.com>
- Frank Healy
- Robert McGivern <Robert.bob.mcgivern@gmail.com>

- James Hickey
- Tommy Gibbons

PyCon UK in Coventry, 22nd September 2013

- Adeel Younas <aedil12155@gmail.com>
- Giles Richard Greenway github: augeas
- Mike Gleen
- Arnav Khare https://github.com/arnav
- Daniel Levy https://github.com/daniell
- Ben Huckvale https://github.com/benhuckvale
- Helen Sherwood-Taylor (helenst)
- Tim Garner
- Stephen Newey @stevenewey (stevenewey)
- Mat Brunt <matbrunt@gmail.com>
- John S
- Carl Reynolds (@drcjar)
- Jon Cage & John Medley (http://www.zephirlidar.com)
- Stephen Paulger (github:stephenpaulger twitter:@aimaz)
- Alasdair Nicol
- Dave Coutts https://github.com/davecoutts
- Daley Chetwynd https://github.com/dchetwynd
- Haris A Khan (harisakhan)
- Chung Dieu https://github.com/chungdieu
- Colin Moore-Hill
- John Hoyland (@datainadequate) https://github.com/datainadequate
- Joseph Francis (joseph@skyscanner.net)
- Åke Forslund <ake.forslund@gmail.com> github:forslund
- Ben McAlister https://github.com/bmcjamin
- Lukasz Prasol <lprasol@gmail.com> github: https://github.com/phoenix85
- Jorge Gueorguiev <yefo.akira@gmail.com> https://github.com/MiyamotoAkira
- Dan Ward (danielward) (dan@regenology.co.uk)
- Kristian Roebuck <roebuck86@gmail.com> https://github.com/kristianroebuck
- Louis Fill tkman220@yahoo.com
- Karim Lameer https://github.com/klameer
- John Medley <john.medley@zephirlidar.com>

DjangoCon US in Chicago, 2nd September 2013

- Barbara Hendrick (bahendri)
- Keith Edmiston <keith.edmiston@mccombs.utexas.edu>
- David Garcia (davideire)
- Ernesto Rodriguez <ernesto@tryolabs.com> https://github.com/ernestorx @ernestorx
- Jason Blum
- Hayssam Hajar <hayssam.hajar@gmail.com> github: hhajar

Cardiff Dev Workshop, 8th June 2013

- Daniel Pass <daniel.antony.pass@googlemail.com>
- Kieran Moore
- Dale Bradley
- Howard Dickins <hdickins@gmail.com> https://github.com/hdickins
- Robert Dragan https://github.com/rmdragan
- Chris Davies
- Gwen Williams
- Chris Lovell <chrisl1991@hotmail.co.uk> https://github.com/polyphant1
- Nezam Shah
- Gwen Williams https://github.com/gwenopeno
- Daniel Pass <daniel.antony.pass@googlemail.com>
- Bitarabe Edgar

DjangoCon Europe in Warsaw, 18th May 2013

- Amjith Ramanujam - The Dark Knight
- @zlatkoc
- larssos@github
- @erccy is my name
- Patrik Gärdeman https://github.com/gardeman
- Gustavo Jimenez-Maggiora https://github.com/gajimenezmaggiora
- Jens Ådne Rydland <jensadne@pvv.ntnu.no> https://github.com/jensadne
- Chris Reeves @krak3n
- Alexander Hansen <alexander@geekevents.org> https://github.com/wckd
- Brian Crain (@crainbf)
- Nicolas Noé <nicolas@niconoe.eu> https://github.com/niconoe
- Peter Bero

- schacki
- Michał Karzyński <djangoconwrkshp@karzyn.com> https://github.com/postrational
- @graup

I followed the tutorial online

- Daniel Quinn - 18th May 2013
- Paul C. Anagnostopoulos - 19 August 2013
- Ben Rowett - 27 August 2013
- Chris Miller, <chris@chrismiller.org> - 5th September 2013
- David Lewis - 7th September 2013
- Josh Chandler - 11th September 2013
- Richie Arnold - <richard@ambercouch.co.uk> - 22nd September 2013
- Padraic Stack - https://github.com/padraic7a
- Patrick Nsukami - <patrick@soon.pro> - lemeteore
- Can Ibanoglu - http://github.com/canibanoglu
- Pedro J. Lledó - http://github.com/pjlledo - 11th October 2013
- Ken Tam - 4th Jan 2014
- Óscar M. Lage - http://github.com/oscarmlage
- Bob Aalsma - https://github.com/BobAalsma/
- Andy Venet - https://github.com/avenet/
- Vathsala Achar - 22nd September, 2014
- Amine Zyad <amizya@gmail.com> http://github.com/amizya
- Xrispies - http://github.com/Xrispies
- Andrew Morales - October 19, 2014
- Suraj Deshmukh <surajssd009005@gmail.com> http://github.com/surajssd
- Suresh - https://github.com/umulingu/
- Chandra Bandi - 20-December 2014
- Drew A. - https://github.com/daldin - 12th December 2014
- Kumar Dheeraj-https://github.com/dhey2k-31-dec-2013
- Omar - 14-1-2015
- Surabhi Borgikar
- Cameron
- Jum - May 20, 2015
- Paul Jewell <paul@jidoka.org> July 2015 https://github.com/paul-jewell
- Alexandro Perez - https://github.com/AlexandroPerez - 6th August 2015
- Rahul bajaj - https://github.com/rahulbajaj0509 2015

- Alejandro Suárez - https://github.com/alsuga 20th October 2015
- Prathamesh Chavan
- Tad Deely
- Abhijit Chowdhury - https://github.com/achowdhury7 12th Feb 2016
- Richard Angeles - Feb 19, 2016
- Adam Shields
- Salvador Rico - April 3, 2016 - https://github.com/salvarico
- Josh Long
- Prashant Jamkhande - https://github.com/prashant0493
- Humphrey Butau - https://github.com/hbutau - 2016-11-7
- Jose Rodriguez -https://github.com/jlrods - 15/11/2016
- Michael Kortstiege - https://github.com/nodexo - Nov 19, 2016
- Steven Lee - https://github.com/stevenlee96 - 2016-11-20
- Dieter Jansen - https://github.com/dieterjansen - 2016-04-20
- Eddy Barratt
- Pooja Gadige - <poojagadige@gmail.com> - pgadige
- Jason Gardner
- Ana
- Dade Murphy
- Leticia Ulloa
- La Chilindrina
- Anselmo ~ <agprocida@gmail.com> ~ anselmoprocida

Running a workshop

If you'd like to run a workshop based on this material, please do, and please let me know about it.

Notes on running a workshop

To cover all the workshop material seems to take about four and a half hours.

Any of the following will make it take longer:

- attendees who aren't already a little familiar with the terminal and using a text editor to edit source files
- attendees whose machines aren't already suitably-configured for the workshop and need software installed
- attendees using operating systems you're not familiar with

If you're lucky, you'll find that the majority of attendees have the same expertise and the same gaps in expertise. This makes it much easier to decide which parts to dwell upon and which ones you can skim over. If you're not lucky, they will each have a completely different skillset.

You'll do a lot of running around to look at people's screens, so it helps to be able to get around the room easily.

The *Git on the commandline* section is the one where you will be in most demand - it helps great if at this stage you have one or two helpers who are familiar with Git.

Watch out for wireless network limitations - at one session the promised network turned out to block both github.com and ssh, and we had to rely on an access point created by someone's mobile telephone.

Things that might look odd

If you're experienced with things virtualenv and Git, some of the way things are done here might strike you as odd. For example:

Virtualenvs and code

The workshop has users put all the code they're working with into their virtualenv directories. This is done to help associate a particular project and set of packages with each virtualenv, and saves excessive moving around between directories.

Editing and committing on GitHub

That's certainly not what we'd normally do, but we do it here for three main reasons:

- GitHub's interface is a friendly, low-barrier introduction to Git operations
- it's easy for people to see the effects of their actions straight away
- they get to make commits, pull requests and merges as soon as possible after being introduced to the concepts

The last of these is the most significant.

Other oddities

There may be others, which might be for a good reason or just because I don't know better. If you think that something could be done better, please let me know.

CHAPTER 3

Credits

"Don't be afraid to commit" was created by Daniele Procida. Other contributors include:

- Daniel Quinn
- Brian Crain (@crainbf)
- Paul Grau
- Nimesh Ghelani https://github.com/nims11
- Robert Dragan https://github.com/rmdragan
- David Garcia https://github.com/davideire
- Jason Blum https://github.com/jasonblum
- Kevin Daum https://github.com/kevindaum

Many thanks are also due to the members of #django, #python, #git and #github on irc.freenode.net for their endless online help.

... and if I have neglected to mention someone, please let me know.

Please feel free to use and adapt the tutorial.