```
# Create a vritualenv for active python verion
pyenv virtualenv venv34

# Activate, deactivate virtualenv
pyenv activate <name>
pyenv deactivate
```

When using virtualenvs, it is often useful to set your `PYTHONPATH` and `DJANGO_SETTINGS_MODULE` in the `postactivate` script.

```
#!/bin/sh
# This hook is sourced after this virtualenv is activated

# Set PYTHONPATH to isolate the virtualenv so that only modules installed
# in the virtualenv are available
export PYTHONPATH="/home/me/path/to/your/project_root:$VIRTUAL_ENV/lib/python3.4"

# Set DJANGO_SETTINGS_MODULE if you don't use the default `myproject.settings`
# or if you use `django-admin` rather than `manage.py`
export DJANGO_SETTINGS_MODULE="myproject.settings.dev"
```

Set your Project Path

It is often also helpful to set your project path inside a special `.project` file located in your base `<env-folder>`. When doing this, everytime you activate your virtual environment, it will change the active directory to the specified path.

Create a new file called `<env-folder>/.project`. The contents of the file should ONLY be the path of the project directory.

```
/path/to/project/directory
```

Now, initiate your virtual environment (either using `source <env-folder>/bin/activate` or `workon my_virtualenv`) and your terminal will change directories to `/path/to/project/directory`.

Single File Hello World Example

This example shows you a minimal way to create a Hello World page in Django. This will help you realize that the `django-admin startproject example` command basically creates a bunch of folders and files and that you don't necessarily need that structure to run your project.

1. Create a file called `file.py`.
2. Copy and paste the following code in that file.

```
import sys

from django.conf import settings
```

```
settings.configure(
    DEBUG=True,
    SECRET_KEY='thisisthesecretkey',
    ROOT_URLCONF=__name__,
    MIDDLEWARE_CLASSES=(
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
    ),
)

from django.conf.urls import url
from django.http import HttpResponse

# Your code goes below this line.

def index(request):
    return HttpResponse('Hello, World!')

urlpatterns = [
    url(r'^$', index),
]

# Your code goes above this line

if __name__ == "__main__":
    from django.core.management import execute_from_command_line

    execute_from_command_line(sys.argv)
```

3. Go to the terminal and run the file with this command `python file.py runserver`.

4. Open your browser and go to 127.0.0.1:8000.

Deployment friendly Project with Docker support.

The default Django project template is fine but once you get to deploy your code and for example devops put their hands on the project things get messy. What you can do is separate your source code from the rest that is required to be in your repository.

> You can find a usable Django project template on GitHub.

Project Structure

```
PROJECT_ROOT
├── devel.dockerfile
├── docker-compose.yml
├── nginx
│   └── project_name.conf
├── README.md
├── setup.py
└── src
    ├── manage.py
    └── project_name
        ├── __init__.py
```

```
        └── service
            ├── __init__.py
            ├── settings
            │   ├── common.py
            │   ├── development.py
            │   ├── __init__.py
            │   └── staging.py
            ├── urls.py
            └── wsgi.py
```

I like to keep the `service` directory named `service` for every project thanks to that I can use the same `Dockerfile` across all my projects. The split of requirements and settings are already well documented here:
Using multiple requirements files
Using multiple settings

Dockerfile

With the assumption that only developers make use of Docker (not every dev ops trust it these days). This could be a dev environment `devel.dockerfile`:

```
FROM python:2.7
ENV PYTHONUNBUFFERED 1

RUN mkdir /run/service
ADD . /run/service
WORKDIR /run/service

RUN pip install -U pip
RUN pip install -I -e .[develop] --process-dependency-links

WORKDIR /run/service/src
ENTRYPOINT ["python", "manage.py"]
CMD ["runserver", "0.0.0.0:8000"]
```

Adding only requirements will leverage Docker cache while building - you only need to rebuild on requirements change.

Compose

Docker compose comes in handy - especially when you have multiple services to run locally.
`docker-compose.yml`:

```
version: '2'
services:
  web:
    build:
      context: .
      dockerfile: devel.dockerfile
    volumes:
      - "./src/{{ project_name }}:/run/service/src/{{ project_name }}"
```