Coding Standards for Python and Django

Assigned Member:

`Asmaul Shahana (AS)`

> Please follow the coding standards for Django Projects - Django Documentation

Best Practices:

1. Avoid abbreviating variable names.
2. Write out the function arguments name.
3. Document your class and methods for future use and easiness.
4. Add meaningful comment in your code.
5. Refactor repeated lines of code into reusable functions or methods.
6. Keep your functions and class short so that it takes up as much as of the screen size.
7. Use Flake8 to check your code quality to prevent syntax errors, typos, bad formatting etc.
8. Use explicit relative imports to avoid program bloat.

Pre-commit checks

pre-commit is a framework for managing pre-commit hooks. These hooks help to identify simple issues before committing code for review. By checking for these issues before code review it allows the reviewer to focus on the change itself.

We first install **pre-commit** and then the git hooks:

```
$ python -m pip install pre-commit
$ pre-commit install
```

Python Style

- Use EditorConfig or conform to the indentation style dictated in the **.editorconfig** file.
- Use 4 spaces for indentation with Python and 2 indentation spaces for HTML files.
- Use PEP 8 Python style as a guide unless otherwise specified.
- Use flake8 to check for problems in this area. But Django has line lengths rules that are different then PEP 8 recommends.
- Use four space hanging indentation rather than vertical alignment:

```
# Bad practice:
raise AttributeError('Here is a multiline error message '
                     'shortened for clarity.')

# Best practice
raise AttributeError(
    'Here is a multiline error message '
    'shortened for clarity.'
)
```

- Use single quotes for strings, or a double quote if the string contains a single quote.
- Use underscores, not camelCase, for variable, function and method names
- Use InitialCaps for class names (or for factory functions that return classes)
- Use in tests, assertRaiseMessage() and assertWarnsMessage() instead of assertRaises() and assertWarns() to check for the exception or warning message.
- Method definition inside a class are separated by a single blank line.
- Limit of the text per line is 79 characters(code understandability).

Don't Do

- Don't change an existing project conventions. Never pick worse name of variables to fitting code in 79 columns.

Imports

PEP-8 suggests that import should be grouped into following order:

1. Standard library import
2. Related third-party imports
3. local application/library import

Django import order

1. Standard library import
2. Import from the core django
3. Import from the third library
4. Import from local app/library

Use Explicit Relative Imports

- It is important to write code in such a way that it is easier to move, rename or version your work.
- In python, explicit import removes the hardcoding a module's package, separating individual modules from being tightly coupled to the architecture around them.

```
#pizzas/views.py
from django.views.generic import CreateView

# DO NOT DO THIS
# hardcoding of 'pizzas' package
from pizzas.models import Chocolate
from pizzas.forms import ChocolateForms
from core.view import PizzaMixin
```

Issue-1: What if you wanted to reuse your pizza app in another project,
there you have to change pizza name because that project already have app with same name.

Issue-2: What if simply wanted to change the name of app at same point.
There would be many situation in real time project which cause the problem.

- ☑ Good coding containing explicit relative imports:

```
#pizzas/views.py
from __future__ import absolute_import

from django.views.generic import CreateView

# Relative import of the 'pizzas' package
from .models import Chocolate
from .forms import ChocolateForms
from core.view import PizzaMixin
```

Use Explicit Relative Imports

- Explicitly import each module.

```
from django import forms
from django.db import models
```

- **NEVER* do the following:

```
from django.forms import *
from django.db,models import *
```

Templates

- When using Django template code, put one space between the curly brackets and the tag contents.

```
# Don't do this:
{{foo}}

# Do this:
{{ foo }}
```

View Style

- In Django views, the first parameter in a view function should be called **request**.

```
# Do this:
def my_view(request, foo):
    #...

# Don't do this:
def my_view(req, foo):
    #...
```

Model Style

- Field names should be all lowercase, using underscores instead of camelCase.

```
# Don't do this:
class Person(models.Model):
    FirstName = models.CharField(max_length=20)
    Last_Name = models.CharField(max_length=40)

# Looks way better:
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)
```

- The order of model inner classes and standard methods should be as follows (noting that these are not all required):

1. All database fields
2. Custom manager attributes
3. class Meta
4. def **str**()
5. def save()
6. def get_absolute_url()
7. Any custom methods:

```
# Not like this example:
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)
    class Meta:
        verbose_name_plural = 'people'

# This is not the right way either
class Person(models.Model):
    class Meta:
        verbose_name_plural = 'people'

    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)

# This is the way to do it:
class Person(models.Model):
    first_name = models.CharField(max_length=20)
    last_name = models.CharField(max_length=40)

    class Meta:
        verbose_name_plural = 'people'
```

- If using choices for a given model field, define each choice as a list of tuples, with an all-uppercase name as a class attribute on the model.

```
# Example with model field that has choices defined in it.
class MyModel(models.Model):
    DIRECTION_UP ='U'
    DIRECTION_DOWN = 'D'
    DIRECTION_CHOICES = [
        (DIRECTION_UP, 'Up'),
        (DIRECTION_DOWN, 'Down'),
    ]
```

Django Project Protocol

1. **Use the same database engine everywhere.**

drawback if development database is differ then production database

- You can't examine a correct database copy of production in local database. ex local database as sqlite and in production database — mysql
- Different database have different field type/ constraints
- Fixtures are magic solution here.

Fixture: fixture are great for creating simple hardcoded test data sets. Sometimes we need to populate database with fake data during deployment at early stage of project.

1. **Use pip and virtualenv**

- **pip** - install python package
- **virtualenv** - is a tool for creating isolated python environment for maintaining required libraries package for project.

use-case for virtualenv - can have more than one project with their own dependency libraries in same system.

without virtualenv — have to reinstall django with that project dependencies at each time when switching to different project.

- activate virtual env without virtualenv wrapper

```
source project_env/bin/activate
```

activate virtual env with virtualenv wrapper

```
workon project_env
```

1. Requirements.txt
   It is a list of python package that you want to install.
2. Version Control System — GIT
3. Using Isolated Docker container