# HOW TO AUDIT YOUR PYTHON CODE:

## CHECKLIST

## &

## SAMPLE REPORT

# CHECKLIST

Feel free to use this checklist whenever you run a code audit yourself, or to investigate whether an already prepared report is comprehensive and not missing any essential parts!

Each subsection consists of a list of questions you should ask yourself while auditing code. It also serves as an outline for your report. While you might experiment with the structure of the report itself, make sure to investigate and include all of the necessary elements.

## Code repository

- ❑ Is there a version control system that tracks and provides changes to the source in place?
- ❑ Is it maintainable?

## Technology choices

- ❑ Are all applications named according to the LTS version?
- ❑ Are they up-to-date?
- ❑ Are all the components well-tested?
- ❑ Do they work well with each other?

## Deployment configuration

- ❏ Which services are used to support the application (hosting services, hosting method)?
- ❏ Does the application contain files connected to the virtualization of the project?
- ❏ Does the README file contain all the necessary elements:
  - ❏ instructions for configuration
  - ❏ instructions for installation
  - ❏ user's manual
  - ❏ a manifest file (with an attached list of files)
  - ❏ information on copyrights and licenses
  - ❏ contact details for the distributors and developers
  - ❏ known bugs and malfunctions
  - ❏ problem solving
  - ❏ changelog (for developers)
  - ❏ news (for end-users)
- ❏ Does your project catalog include files responsible for continuous integration and deployment?
- ❏ Are you sure the project configuration doesn't contain any passwords that a third person could find?
- ❏ Is there an Error Tracking System in place?

## Coding best practices

- ❏ Is the code compliant with the PEP8 style guide and PEP257 docstring conventions?

- ❏ Is your code free of:
  - ❏ bugs?
  - ❏ bottlenecks?
  - ❏ performance issues?
  - ❏ security vulnerabilities?
  - ❏ dangers connected with maintaining the application?

# A SAMPLE CODE AUDIT REPORT

Below, you will find a sample code audit report.

It will provide you with an overview of what a well-performed, comprehensive code audit looks like. Get acquainted with the structure and the reviewed elements, however, pay in mind that your end result might be much different.

# <YOUR_APP_NAME>—Code Audit

**Author: <YOUR_NAME>**

**Date: XXXX-XX-XX**

## Overview

This document contains the conclusions from the code audit performed by the author on the <YOUR_APP_NAME> codebase.

## Document Structure

This document is divided into three sections:

- **Software Configuration Management:** this section describes how the project is managed from a technical point of view. This is where the Version Control, project configuration, and deployment strategies are reviewed.
- **Software Architecture:** This section deals with the architecture details, from technology choices to live deployment configuration.
- **Programming Best Practices:** This section lists generic programming best practices that are not specific to Python, such as documentation, code segmentation, and general performance optimization. Then, Python and Django best practices are described.

Each section lists good practices found within the code, as well as possible missteps and recommendations on how to fix them. Additionally, in the subsection called “Section Summary” you will find information on the number of comments left for each reviewed part.

# Software Configuration Management

## Code Repository

### Hosting and VCS

The code is hosted in bitbucket.org, a solid VCS repository provider, using GIT as the Version Control System, which is the current standard.

### Branching

The GIT branching system is also being used with separated branches for development and staging, which is a good practice.

Having a separate branch for production is, in my opinion, highly advisable, but not mandatory.

However, branches are not properly closed using the GIT feature, which is disadvantageous. It makes the branching more complex as time goes on.

### Section Summary

- Bitbucket.org: 0
- GIT: 0
- Branching usage: 0
- Separated dev and staging branches: 0
- No production branch: 1
- Old branches not closed: 1

# Software Architecture

## Technology Choices

### Python and Django

I won't make any extended comments about the usage of Django and Python, apart from stating that they seem to be a very good option for this project.

### MySQL

At the database level, MySQL works well as the main Django Database. However, it's worth mentioning that pairing Django with PostgreSQL is usually the standard choice.

In any case, given the nature of this project, this choice shouldn't make any difference.

### MongoDB

I generally consider MongoDB a very good choice for DB intensive applications where scalability needs to be taken into account at all times.

However, I'm not sure if this project fits the description. Given the type of data stored in it and the type and volume of the operations, I believe that a properly configured PostgreSQL or even MySQL cluster could handle all the operations and scale properly in the future.

On the other hand, MongoDB does not implement transactions, which makes it a less ideal database than a relational one when dealing with complex business scenarios.

This, however, can be overcome. It's just worth keeping the lack of transactionality in mind when designing the data models and implementing the database interactions.

### Infusionsoft

Infusionsoft is outside of the scope of this review, so I won't comment on it.

### Section Summary

- Python: 0
- Django: 0
- MySQL: 0
- MongoDB: 1 (lack of transactionality)

## Deployment Configuration

### Configuration files

The deployment configuration can be found in the project repository, which is a good practice.

It's based on Nginx, Supervisor, and Gunicorn.

### Nginx

Nginx is a really good choice as the web server, and the configuration file is correct and properly organized.

### Supervisor

Supervisor is a valid choice as the daemon management system, but I don't consider it optimal. Whenever possible, I try to integrate the daemons in the distribution native daemon management system, such as Upstart for Ubuntu 14.04 or SystemD for Ubuntu 16.04. However, since the project is not attached to any particular distribution, I cannot strongly suggest any alternative to Supervisor.

### Gunicorn

Gunicorn is the WSGI server which I consider optimal.

However, there is a big flaw in the configuration. It uses the default "sync" worker class.

The "sync" worker class performs acceptably when processing a single request, but it handles the concurrency very poorly. If the sync worker class is used, the server can only process as many requests as Gunicorn processes there are at a time.

The alternative I suggest is using Gevent as the worker class, which makes an enormous difference in performance when multiple users access the website at the same time, or even when a single page makes multiple AJAX requests simultaneously, allowing the usage of much smaller servers to host the Django app.

The reason for this difference in performances is that the sync worker class does not pick any new request until it finishes processing the previous one. Instead, the Gevent worker class accepts all the requests at the same time and swaps between them all on every I/O operation, allowing the software to take advantage of the multiprocessing capabilities of the Database or any remote system, such as Infusionsoft.

### Deployment Automation

I cannot find any reference to a Deployment Automation system such as Ansible or Fabric in the repository.

### Continuous Integration

I cannot find any reference to a Continuous Integration system in the repository.

### Error Tracking Systems

I cannot find integration with an error tracking system like Sentry.

### Virtualization

Virtualenv is in use, which is recommendable. However, it is managed by virtualenvwrapper.

While virtualenvwrapper is a very good choice for development environments, I usually suggest using plain virtualenv for server deployment, since it keeps everything related to the deployment in a single folder.

### Scripts

There are several scripts in the repository that contain environment variables and a Python module call.

It would be advisable to keep all these Python modules within a folder and create a single command-line Python client configured using argparse to invoke them all as different tasks of the same CLI.

Optionally, they could be added as custom manage.py commands.

### Environment Variables

The web-service.sh script defines a couple of environment variables.

These variables should be kept out of the script and moved into the daemon manager configuration (supervisor, in this case).

This would allow using the same script in multiple environments.

### Django Settings

The current Django setting strategy is to have the production settings in the default settings.py module and override them using a local_settings.py.

This is a valid practice, but it could be optimized. In its current state, it demands to have a local file on each environment that is left out of the VCS with the particular settings for that environment.

The best strategy would be to have a settings module for each environment, all of them in the VCS, and have each one of them import the common settings from a base module.

```
project
├── settings
|   ├── common.py
|   ├── development.py
|   ├── __init__.py
|   └── production.py
```

This way, the production DJANGO_SETTINGS_MODULE would be “project.settings.production” while developers would be using “project.settings.development”.

Section Summary

- Nginx: 0
- Supervisor: 1
- Gunicorn: 0
- No use of Gevent Worker Class: 5
- No deployment automation: 4
- No continuous integration: 3
- No error tracking: 3
- Virtualenv: 1
- Scripts: 1
- Environment Variables: 1
- Django Settings: 2

# Programming Best Practices

## Project Numbers

Number of python files: 207

Number of lines of code: 28255

Average lines per file: 136

Files with more than 500 lines of code: 15

Files with less than 100 lines of code: 141

Files with less than 10 lines of code: 19

This project is not too big, but it's not small either.

Perhaps the number of files is a bit excessive. It could be reduced by merging in some of the files with less than 100 lines of code.

I would recommend removing some unuseful files such as the empty models.py and admin.py files.

I also think that it's necessary to split up the 15 files with more than 500 lines of code into smaller modules.

## Documentation

Project documentation cannot be found anywhere in the repository, within the code. I don't know if external documentation exists, but including at least a README.md or similar document with some information about the project from both functional and technical points of view would definitely be helpful for new developers.

This document should at the very least outline the main use cases of the project and shed some light on the project's structure and architecture.

In-code documentation can be found all over the project, especially as docstrings, which is a good practice.

However, a few typical flaws can be found within the documentation.

### Redundant Docstrings

Some docstrings provide no other information than what is already evident from the function's name.

An example of this can be observed in multiple functions of the module `api/views/views.py`:

```
151     def set_variable(self, request, *args, **kwargs):
152         """
153         Set variable.
154         :param request:
155         :param args:
156         :param kwargs:
157         :return:
158         """
```

### Copy-pasted docstrings

Some docstrings are copy/pasted from one class or function to another, which makes them meaningless.

### Outdated in-line comments

There some comments which might have been relevant in the past but are no longer applicable, probably due to changes in the code that were not reflected in the comments.

It is visible in `api/views/views.py`, for example:

```
31          # Get the latest version of api
32          client = Client.objects.get(user_id=kwargs["client_id"])
33          data_serializer.save(client=client)
34          return Response(data=data_serializer.data)
```

### Redundant in-line comments

Some comments are clearly not necessary, like this one from `www/views/views.py`:

```
278         # additional instructions
279         context["additional_ins"] = ADDITIONAL_INSTRUCTIONS
```

## Flow Control Statements

### If...elif...else blocks

There are several points where the usage of the if..elif...else blocks is a bit abusive, making the code less efficient and less readable.

The best examples of this can be found in these modules:

- `www/templatetags/extras.py - 126 elif statements`
- `www/views/views.py - 44 elif statements`

Here are some alternatives to the long elif blocks:

#### *Constant Dicts*

When the elif conditions only compare the value of a single variable against a series of values, this can be replaced with a dict lookup.

So, this (`www/templatetags/extras.py`):

```
42 @register.filter
43 def int_to_type_rating(value):
44     return_value = "N/A"
45
46 if value == 1:
47     return_value = "Type A"
48 elif value == 2:
49     return_value = "Type B"
50 elif value == 3:
51     return_value = "Type C"
52 elif value == 4:
53     return_value = "Type D"
54 elif value == 5:
55     return_value = "Type E"
56
57 return return_value
```

Could be replaced by this:

```
42 INT_TO_TYPE_RATING = {
43     1: "Type A",
44     2: "Type B",
45     3: "Type C",
46     4: "Type D",
47     5: "Type E"
48 }
49
50
51 @register.filter
52 def int_to_type_rating(value):
53     return INT_TO_TYPE_RATING.get(value, "N/A")
```

*Constant Lists*

In the previous case in particular, the values are consecutive and exhaustive, so if this code was to be executed very often and its performance was critical, a list could also be used to improve execution times:

```
42 INT_TO_TYPE_RATING = [
43     "Type A",
44     "Type B",
45     "Type C",
46     "Type D",
47     "Type E"
48 ]
```

```
49
50
51 @register.filter
52 def int_to_type_rating(value):
53     try:
54         INT_TO_TYPE_RATING[value - 1]
55     except IndexError:
56         return "N/A"
```

### While True...continue...break

Creating loops using `while True` and managing them with `break` and `continue` statements tends to result in more complicated code that's difficult to read, often full of duplications and requiring the usage of state variables to control the context status.

A prime example of this in the codebase can be found in the module `www/helpers/connector.py`

This situation can usually be avoided with some of the following approaches:

#### *For loops*

In most cases, the `while True` loop is not truly infinite and has some limit on the number of iterations that it can do:

```
result = None
tries = 0
while True:
    result = do_something()
    if result:
        break

    tries +=1
    if tries >= max_retries:
        log_an_error()
        break
```

In these cases, using a for loop with the max_retries as the loop limit and breaking it if we succeed before is much cleaner:

```
result = None
for _ in range(max_retries):
    result = do_something()
    if result:
        break

if not result:
    log_an_error()
```

*Return Statements*

In most cases, a break statement can even be avoided by properly segmenting the code and using a simple return to jump out of the function.

Not only does this make the code much cleaner and more readable, but it also allows us to reuse the loop later on with only a simple line.

```
def try_to_do_something(some_params):
    for _ in range(max_retries):
        result = do_something(some_params):
        if result:
            return result

    #     reachable   only   if   do_something   never
    succeeded log_an_error()

a_result = try_to_do_something(some_params)
another_result = try_to_do_something(other_params)
```

*Condition Function*

In some cases, the condition to break the loop is something that might not depend on our actions.

For example, we might want to modify something in the database, but other concurrent processes might also modify the document before us.

In these cases, using a function as the `while` condition makes the code cleaner:

```
we_modified_it = False
def continue_loop():
    return modified or someone_else_modified_it()
while continue_loop():
```

```
    we_modified_it = modify_something_in_db()
```

# Python Best Practices

## PEP Compliance

One of Python's characteristics is the existence of Python Enhancement Proposals (PEPs) which establish a series of coding and styling standards. While not mandatory, they provide a set of rules which make the work of the Python developers easier.

In this code audit, I cover only one of them, PEP8. It is usually considered the most important one. However, it's advisable for the developers to also be familiar with PEP20 and PEP257.

## PEP8

PEP8 establishes the most important styling rules, from naming to indentation conventions.

The compliance with the PEP8 conventions can be checked using the pep8 package.

The codebase follows all the rules. However, there are a couple of points which should be improved.

### Imports

PEP8 establishes a couple of rules related to imports which are not respected in the codebase.

- **Import statements should always be at the top of the file, never within the code**

  There are some cases, like in module `www/views/views.py`, where imports occur within a function. This should always be avoided unless absolutely necessary, such

as having to wait for a monkey patching process which is executed somewhere else.

- **Wildcard imports should be avoided**

  Most of the mongoengine imports use wildcard. This should be changed to import explicit names.

- **Import grouping**

  Most of the imports in the codebase appear in random order. This makes it very difficult to control what is already imported and what isn't, leading to duplicated imports and possible overrides.

  As an example of this, the sentence `from django.contrib.auth.models import User` can be found three times in the module `www/views/views.py`

  PEP8 suggests making three groups of import statements, separated by one empty line and always in this order:

  1. Standard library imports
  2. Third package imports (things installed using pip)
  3. Imports from the same project

  And it also suggests putting all the `import package_name.object_name` before the `from package_name import object_name` statements.

As an additional rule, following the Google Python Style Guide, it's worth enforcing alphabetic order within each block of imports. This ensures a unique import order and helps avoid clashes and discrepancies when having multiple contributors manipulating the same file.

Following this approach, the import block of the module `www/views/views.py` should be the following:

```
import logging
from datetime import date, datetime, timedelta
from collections import OrderedDict
```

```
from django.conf import settings
from django.contrib.auth.mixins import LoginRequiredMixin, UserPassesTestMixin
from django.contrib.auth.models import User

from  django.core.urlresolvers  import  reverse_lazy
from django.http import Http404, HttpResponseRedirect
from django.shortcuts import render
from django.views.generic import FormView, RedirectView, TemplateView
from mongoengine.queryset.visitor import Q

from www.client import Manager
from www.forms.treatment_form import Form, DraftForm
from www.mongo_models.regimens import DraftRegimen
```

### Line length

PEP8 states that the standard line length is 79, but that any developing team can agree to increase this size if they find it convenient. However, it establishes 99 as the maximum acceptable length in order to prevent line wrapping or horizontal scrolling when seeing two files side by side (for example, when seeing file differences during a merge).

This rule can obviously be skipped, but it's worth mentioning that 910 lines break this rule within the codebase.

## Python Features

### Class Methods and Static Methods

Python has three kinds of methods: instance methods, which are the normal methods defined in a class, and class and static methods, which are defined by adding the corresponding decorator on the regular instance method.

Using them properly might help optimize the code, especially in terms of memory usage if many class instances exist at the same time.

In order to see the difference between them, we need to remember that a method is basically a function that is assigned to another object as an attribute and that in Python, all functions are also objects which need their share of memory to exist.

- **Instance Methods** are the regular methods defined in a class without any decorator, and a copy of them exists in every instance of the class, apart from the one which is assigned to the class itself. When called, the class instance will be passed to them as the first argument, so they always need to have at least one argument.
- **Class Methods** are the ones created using the `@classmethod` decorator, and only a copy of them exists: the one which is assigned to the class. It is never copied even if the class is derived. When called, the class (not the instance) will be passed to them as the first argument, so they always need to have at least one argument.
- **Static Methods** are the ones created using the `@staticmethod` decorator. Neither the class nor the instance is ever passed to them, so they can have no parameters.

In order to make the code efficient in terms of resources, the main rule is: always try to use as many class and static methods as possible.

Here are the most important rules:

- If the method uses **nothing** from the class or instance, it must be a **static method**.
- If the method uses **only class attributes** (a static method is also a class attribute), it must be a **class method.**
- If the method uses **instance attributes**, it must be an **instance method**.
- If the method uses other **instance methods**, it must be an **instance method**.

Here is an example class with proper usage of static, class and instance methods:

```
class ExampleClass(object):
    # All instances will see this attribute as of their own
    # but they all will see the same copy of the value
    class_attribute = 'a class attribute'
```

```
def __init__(self, value):
    # instance_attribute will exist only in the instance
    # and each instance will have its own value
    self.instance_attribute = value

def instance_method(self):
    # This needs to be an instance method because it uses
    # an attribute which only exists in the instance
    print(self.instance_attribute)

def another_instance_method(self):
    # This needs to be an instance method because it uses
    # another instance method, so it cannot be class method
    self.instance_method()

@staticmethod
def static_method():
    # This can be a static method because it uses nothing
    # from neither the class nor the instance
    print('something unrelated to the class')

@classmethod
def class_method(cls):
    # This cannot be a static method because it uses an attribute
    # but it does not need to be a instance method because
    # it only uses class attributes, so it is a class method
    cls.static_method()
    print(cls.class_attribute)
```

# Django Best Practices

## Settings

The Django settings module structure has already been discussed in the Deployment Configuration section. However, there are a couple of points unrelated to the deployment configuration which call for commentary.

### Settings Modules Structure

The current settings module contains many aspects that are unrelated to Django behavior mixed up with things that are.

So, apart from having different settings modules for each environment, separating the settings which are unrelated to Django in different modules would also make it more maintainable.

One possible structure would be:

```
app/settings
├── common.py
├── crm
│   ├── common.py
│   ├── development.py
│   ├── __init__.py
│   └──production.py
├── development.py
├── __init__.py
└── production.py
```

With the following contents:

- `app.settings.common`:
    - Django-specific settings which are the same in all environments: INSTALLED_APPS, MIDDLEWARE, TEMPLATES, etc.
- `app.settings.crm.common`:
    - CRM-specific settings which are the same in all environments
- `app.settings.crm.{environment}`
    - CRM-specific settings that depend on the environment: hostnames, usernames and passwords, keys, etc.

- `app.settings.{environment}`
    - Django-specific settings which are environment specific: DATABASES, ALLOWED_HOSTS, DEBUG, etc.
    - `from app.settings.common import *`
    - `import app.settings.crm.{environment} as crm`

This way, all the crm settings can be accessed through the `.crm` attribute of the `django.conf.settings` module.

### apps.py, admin.py and models.py cleanup

When a new app is created using the django-admin or manage.py script, some files such as apps.py, admin.py and models.py are created by default, admin.py and models.py left empty and apps.py containing an AppConfig class which is not used at all unless manually specified in the `app __init__.py` file.

So, while leaving them be makes no actual difference on runtime, it still makes the project files list longer and can lead to confusion, especially regarding that unused AppConfig class in apps.py

For these reasons, it's advisable to always remove them if they remain unused.

## Project Structure

The current project structure is what django-admin establishes by default and the one shown in the Django Tutorial: each app, including the main one that contains the settings, lives in the base directory where manage.py is, as if they were unrelated packages.

This usually works for projects with a single app that is self-contained and without cross-references, but makes the code more complex when the project grows and each app starts using objects (such as db models) from the other apps.

When this happens, I usually prefer to move things around and make all the related apps a part of a single base package, which contains everything that they share (e.g. some models) while keeping the things specific to them (such as static or template files) in each app directory.

In the case of this codebase, it could look like this:

```
app
├── manage.py
├── static
├── templates
├── app
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── app_api
│   ├── models.py
│   ├── urls.py
│   └── views.py
├── www
│   ├── models.py
│   ├── urls.py
│   └── views.py
└── app2_api
    ├── models.py
    ├── urls.py
    └── views.py
```

Any model used both by the www and the api app would be in the module `app.models` while models only used by the www app would be in `app.www.models`, making the module relations clearer.

In this section I used the models.py module as an example, but the same rule applies for similar abstractions such as mongomodels or serializers.

## Models

Since this project uses almost no custom models and relies on mongoengine models, I will comment on the latter.

And the only comment I can make about them is that everything seemed to be correct from a technical point of view.

### Transactions

One useful feature of Django when paired with a relational Database is the `@atomic` decorator from the `django.db.transaction` module.

This, however, cannot be used with MongoDB, since it does not implement transactions, as discussed in the Software Architecture section.

As a consequence of this, if not performed carefully, some operations may lead to inconsistent data states in case of unexpected errors.

## Disclaimer

While exhaustive, this exemplary code review is not finite. There are a few more things that are worth paying attention to that aren't covered here:

- Thoroughly investigate every aspect of your framework of choice. For Django and DRF (Django REST Framework) that would be: Views, Forms, Templates, Widgets, etc.
- Best practices connected to every technology implemented in the project. For example, this example lacks information on MongoDB Best Practices.

# FINAL THOUGHTS

We hope our checklist and sample report were useful to you.

A code audit is the easiest way to improve your code. Having a fresh pair of eyes take a look at your project will help you eradicate your bad habits and practices, optimize, and make sure everything is safe and sound.

If you liked the sample code audit and you feel it would benefit your project, you're in luck. For a limited time, we're offering free code audits.

If you want to get your code professionally audited for zero money, click here to schedule your audit!