**ERP Code Review Document**

**Reviewer :** ANAND RAJ B

**Date :** 19-09-2023

**Branch :** master

| **S.No** | **Content** | **Page** |
| --- | --- | --- |
| 1 | Introduction | 2 |
| 2 | Code Formatting and Documenting | 3 |
| 3 | Error Handling | 6 |
| 4 | Security Improvements | 8 |
| 5 | Performance and Optimization | 10 |
| 6 | Reusability and Refactoring | 12 |
| 7 | Area of Improvement | 14 |
| 8 | Conclusion | 18 |

**1 . INTRODUCTION**

The Objective of this document to ensure the changes that are introduced to the current codebase aligns the code standards , functional requirements , system design and security best practices .

**1.1 Scope of review**

In this section , we’ll define the scope of code review document and the area of coverage are as follows ,

* **Formatting**: Ensuring guidelines and standards.
* **Functionality:** Confirming that the code accomplishes its task and functional requirement.
* **Error Handling:** Checking for better error handling.
* **Modularity and Reusability:** Assessing code structure and potential for reuse and possibility for refactoring the codebase.
* **Security :** Checking for any exposure to sensitive information.
* **Performance:** Identifying performance bottlenecks .

**1.2 Review and Feedback process**

The code review process involves under **OWASP guidelines** and includes a SANDI model for code reviewing and **CEDAR** model for feedback and improvements .

**2 . CODE FORMATTING AND DOCUMENTING**

**2.1 Code Style and Formatting**

Maintaining standard code style and code formatter for better readability of code , some of the them as follows ,

* Use **PEP8** or **black** formatter for better code style and formatting
* REF : [Link\_1](https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/) , [Link\_2](https://peps.python.org/pep-0008/)

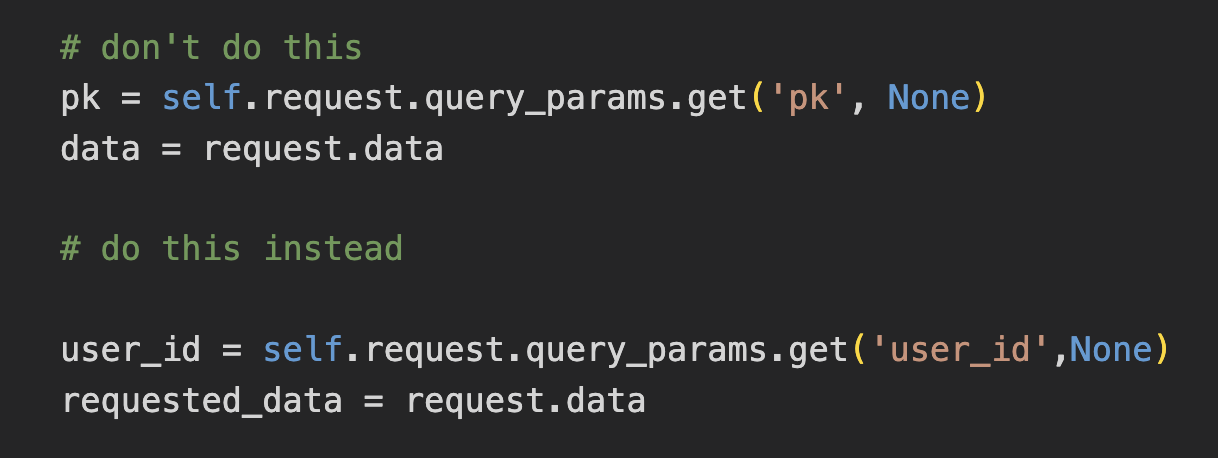
**2.2 Naming conventions**

Some of the naming errors in the code base are mentioned as follows ,

* Usage of **same variable name** in all management folder

E.g sales\_name , purchase\_order

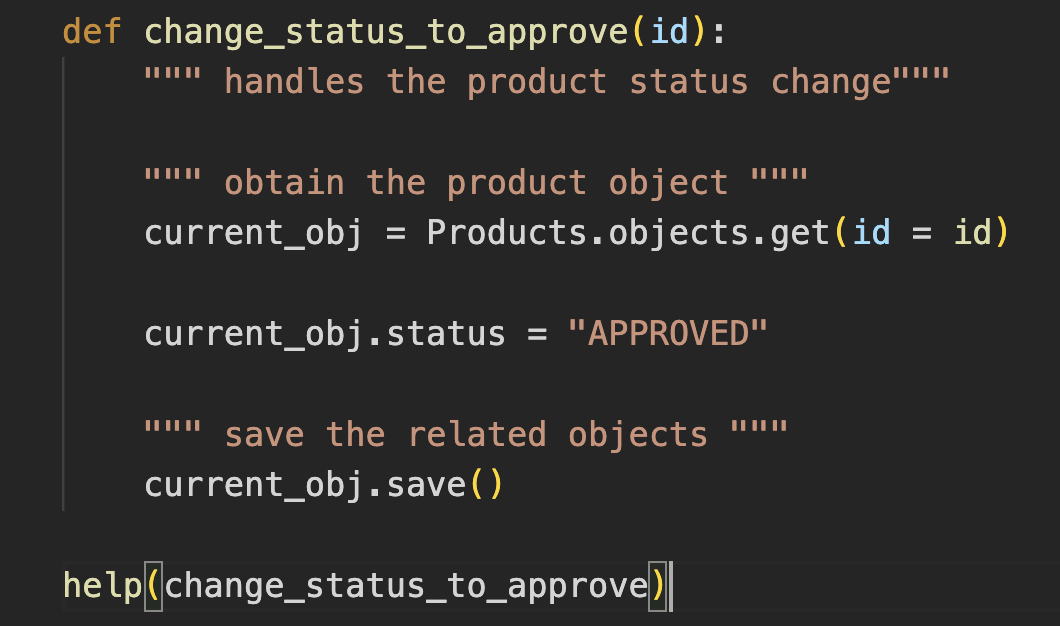
* It is recommended to follow variable name as follows ,
  + for objects : *Sales\_name\_obj*
  + for querysets : *Purchase\_order\_qryst*
* Give valid names for variables .



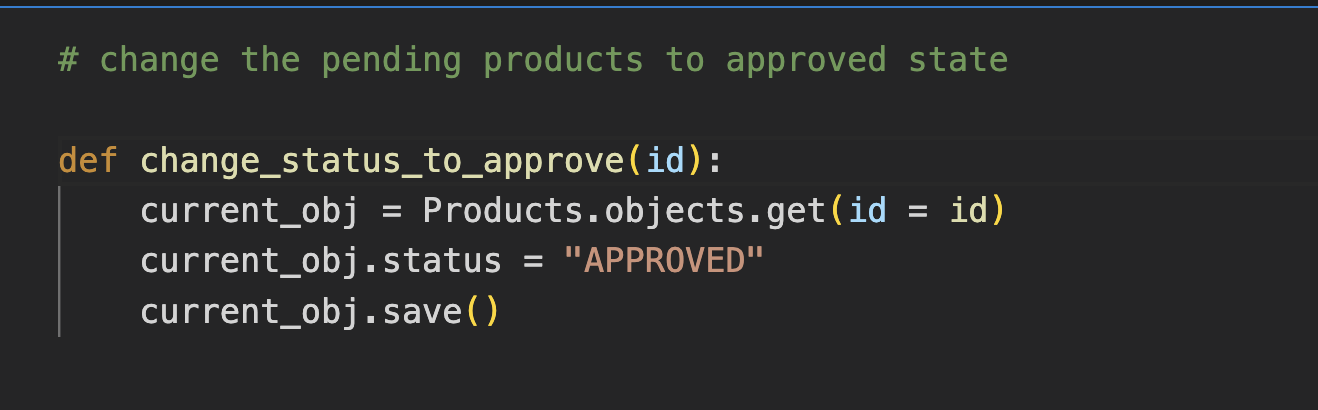
* Usage of **singular** and **plural** form of variable naming would be a best practice
  + for one or more - *Sales\_order\_items*
  + for one item - *Product\_name*
* For Constants recommended to use **Uppercase** as standard
  + E.g., CURRENT\_SALES
* For File naming use **snack case** as standard
  + E.g., api\_permission.py
* Use **Pascal case** for utility classes
  + E.g., MakeSalesReservation() , ProductManagement()
* Use plural for **folder naming** like users/ , helpers/ , forms/

**2.3 Docstring and Comments**

* It is recommended to add **docstring** in the current code in-order to understand the functional works of a piece of code with example as below image ,



* No docstring required if a function is **straightforward** and what it does .
* Add comments for explaining **why** and **what** of the code



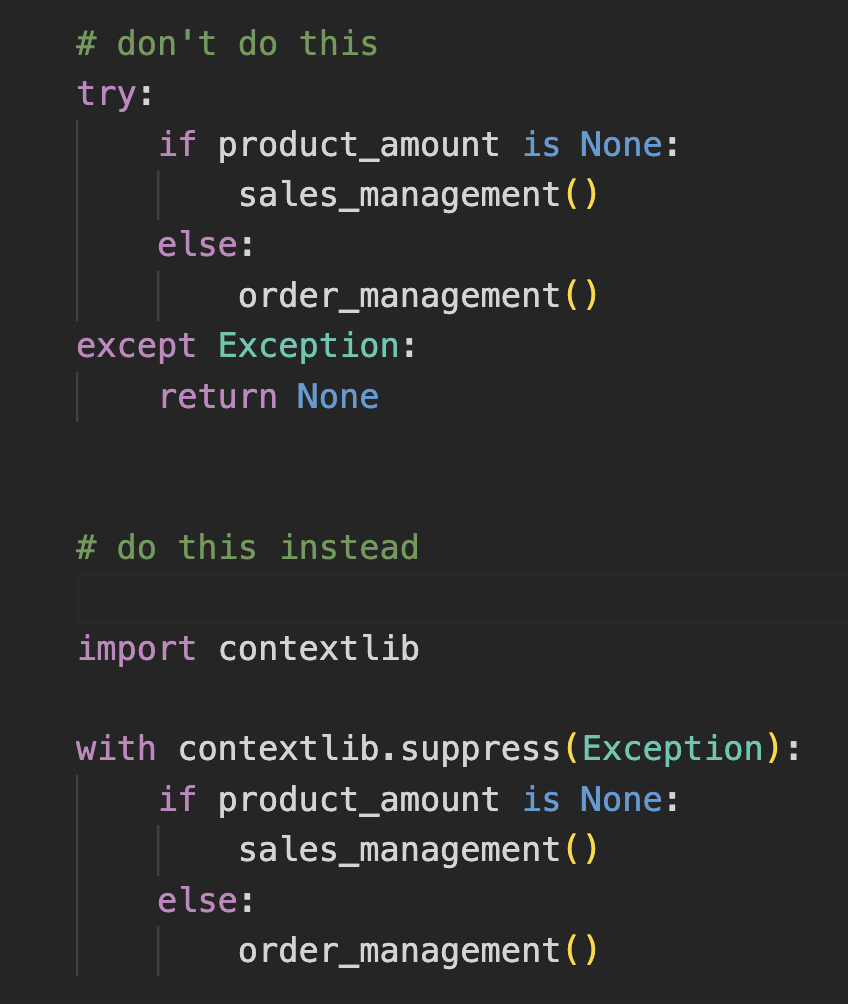
**2.4 Documenting strategies**

* Add a detailed **README.md** for
  + Installation guide
  + Technology stack
  + ER diagram
  + Authors , Maintainers and Contributors .
* Additional documents
  + Postman collection
  + Swagger api document
  + Developer documentation
  + Mermaid
* REF : <https://readme.so/editor>

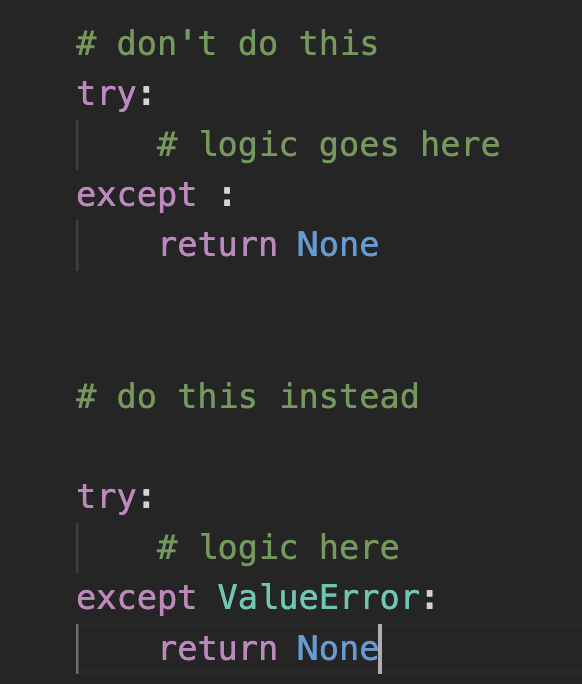
**3 . ERROR HANDLING**

**3.1 Exceptions**

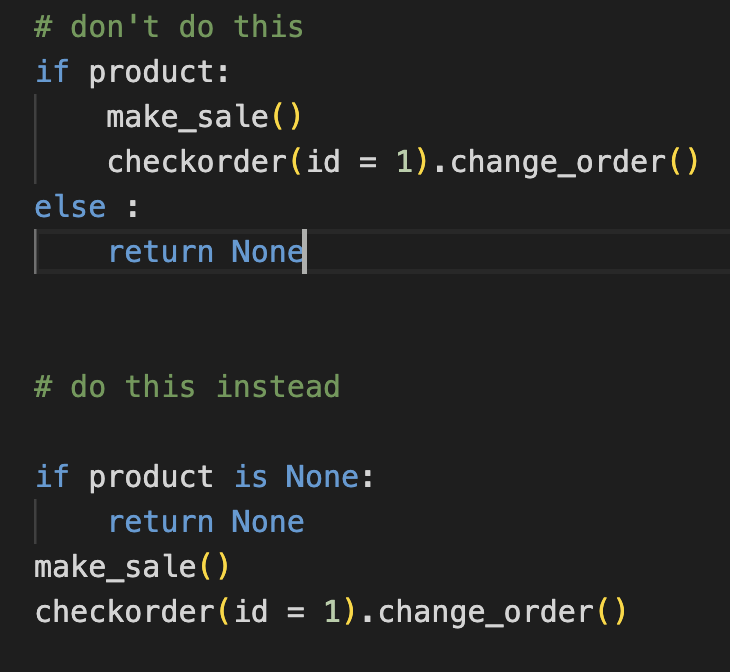
* As a result there is an event of adding try and except clauses in the current code base , it is better to avoid those exception clauses for smaller logic and use **contextlib** library .



* Use exception types in exception clauses like **ValueError** , **TypeError** ,



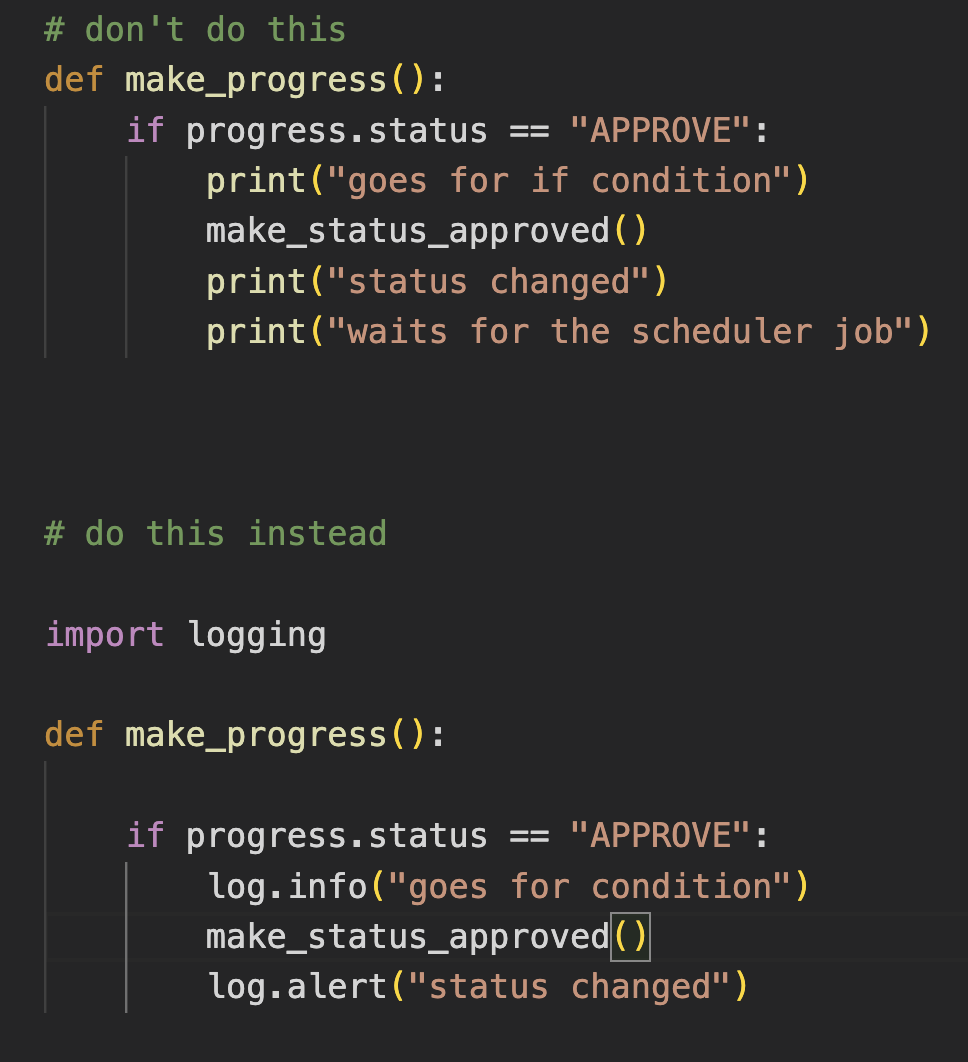
* For improved readability use **guard classes** for safe exiting .



* REF : [Link\_1](https://docs.djangoproject.com/en/4.2/ref/exceptions/) , [Link\_2](https://docs.python.org/3/library/exceptions.html)

**3.2 Logging**

* It is recommended to use a logger for printing the process status for better logging and debugging .



* For event based or daily logging use django’s inbuilt logger
  + REF : [django\_logging](https://docs.djangoproject.com/en/4.2/topics/logging/)

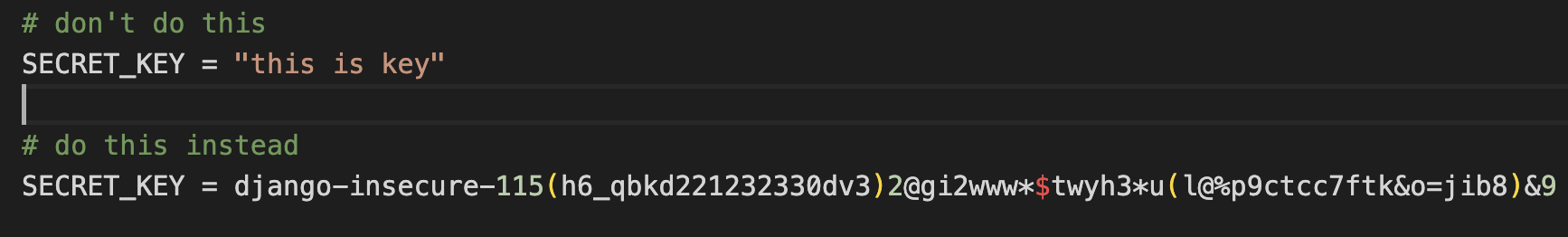
**4 . SECURITY IMPROVEMENTS**

**4.1 Validation**

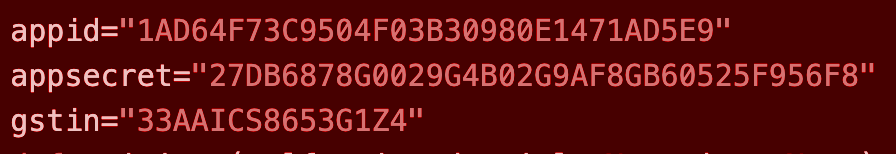
* As a result in the code review process , i could see there is vital loophole in the form validations and **input sanitization** in both client and server side
* **Expected Validations** are well used in the current code base on the server side but requires more fine tuning .
* Chances for **XSS attacks** from client side .
* Tips :
  + Add Content security Policy (**CSP**)
  + Better Input validation and HTML escaping
  + REF : [aviod\_xss](https://jscrambler.com/blog/the-most-effective-way-to-protect-client-side-javascript-applications) , [django\_security](https://docs.djangoproject.com/en/4.2/topics/security/)

**4.2 Credentials Exposure**

* **Security Key** in django is the doorway and crucial component for application security , i could see the security key is kept as below.



* Exposure of sensitive information and credentials are exposed in folder name **gst/**

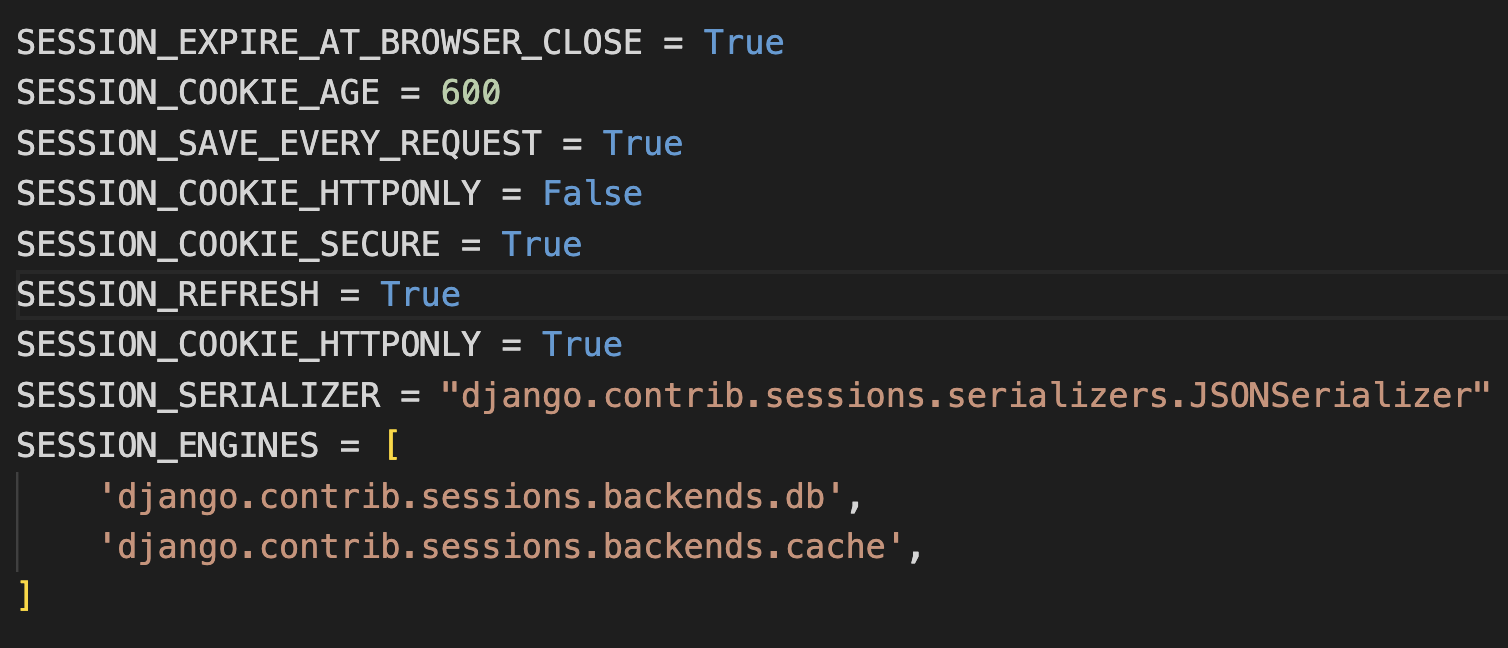


* It is recommended to **encrypt** and **decrypt** those information like

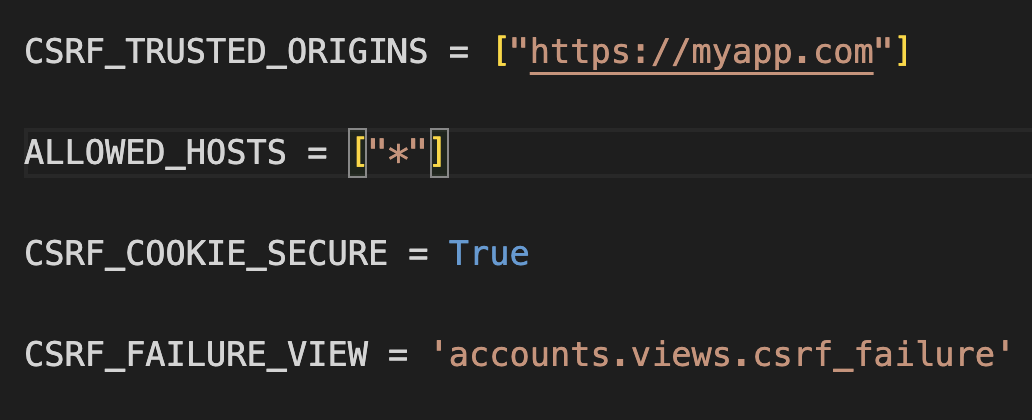
GST , passwords and tokens .

**4.3 Session and Request Management**

* Adding a custom session **middleware** or utilising the django session for better session uptime and cookie safing as below E.g., .



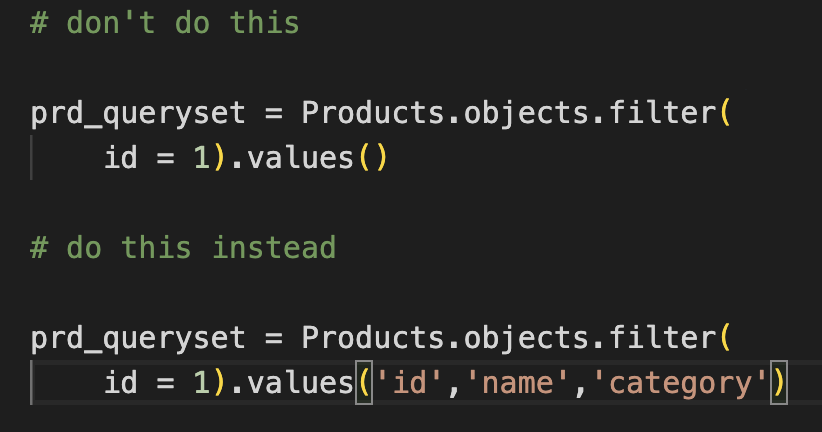
* Additional **CSRF** validation are recommended as below example

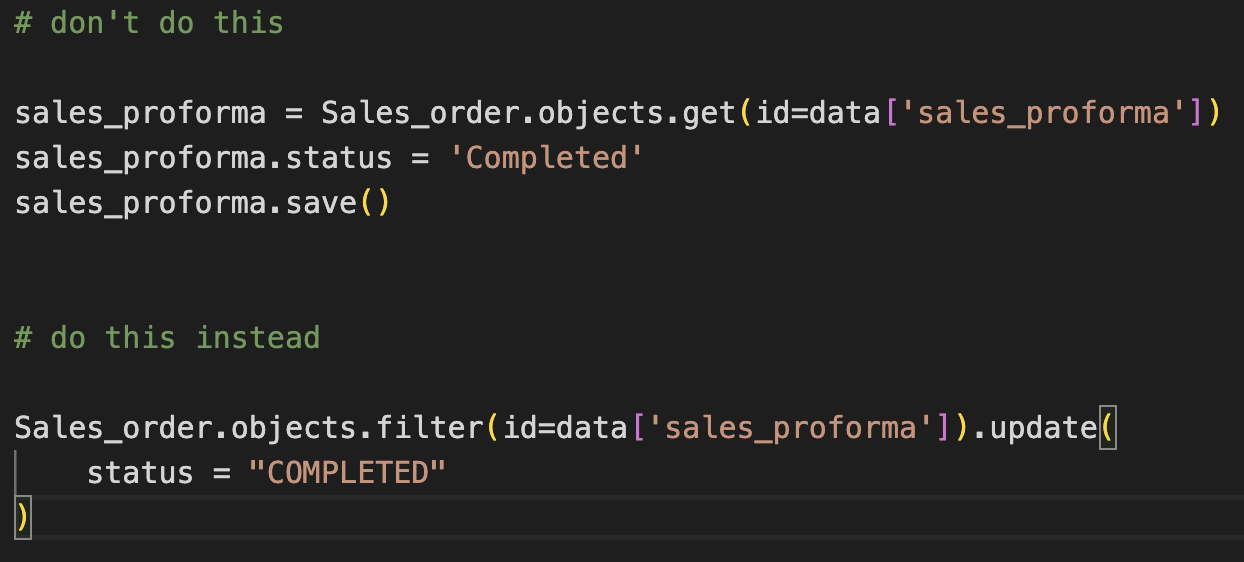


* In addition **HSTS , SSL/TLS , CORS** policy , **Proxy** configuration can be added based on the server or serverless deployment model .
* Use penetration testing tools to check the **site health** .
* REF : [pen\_tesing](https://pentest-tools.com/website-vulnerability-scanning/sql-injection-scanner-online) , [django\_checkup](https://djcheckup.com/)
* Run **python manage.py check**  for security and vulnerabilities checking .

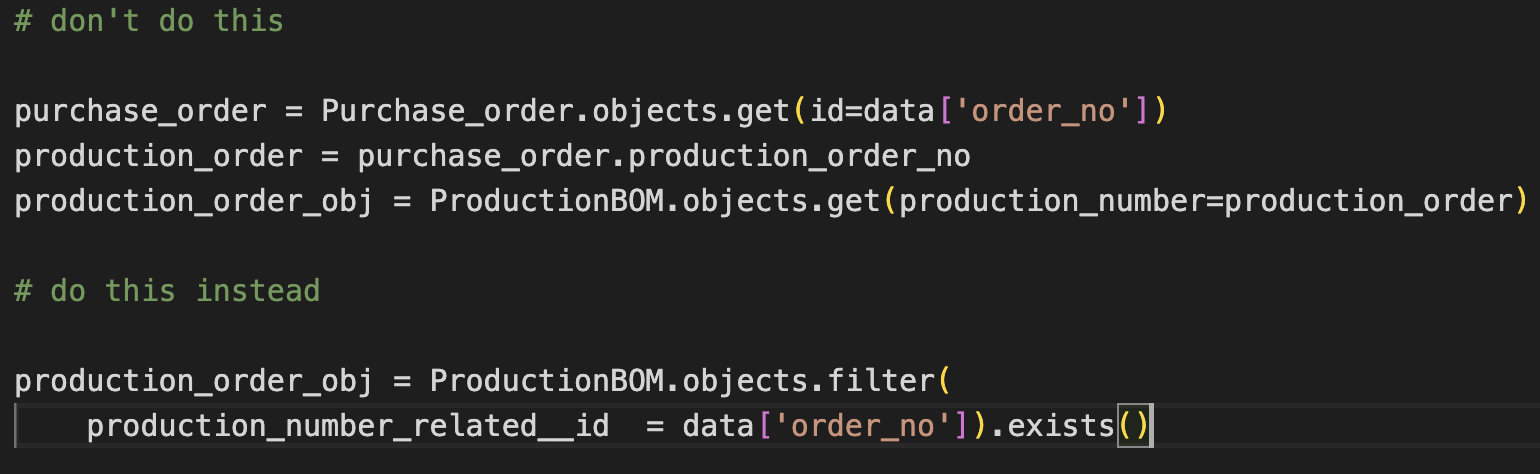
**5 . PERFORMANCE AND OPTIMIZATION**

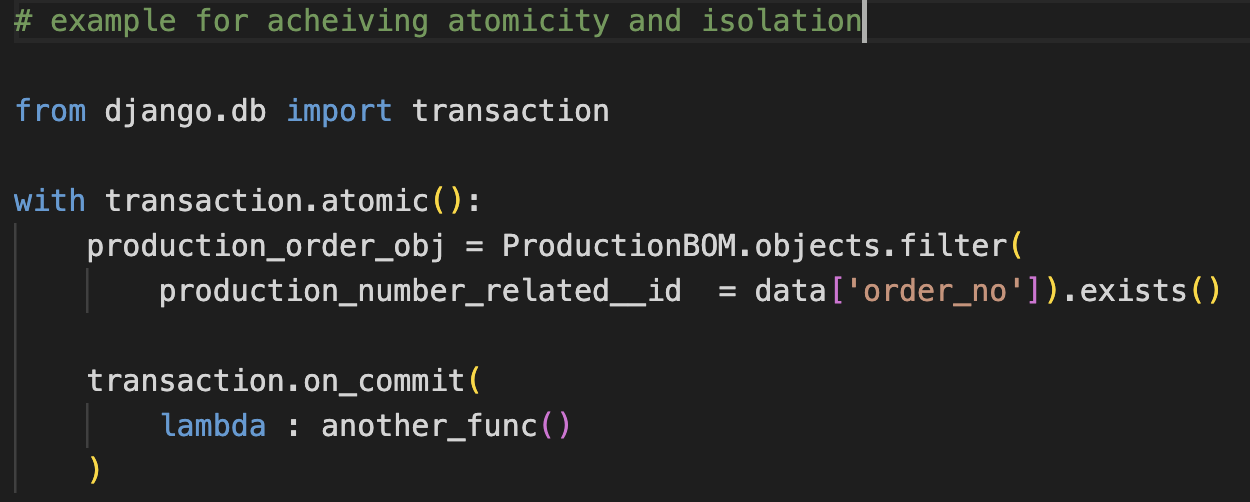
**5.1 Query Optimization**

* The current code base has better optimised queries and solves the N + 1 queries .
* Tips :
  + *Get what you need :* Instead of retrieving all the fields , get the required fields from the DB
  + 
  + *Don’t multiply :* calling the instance object multiple times calls the query multiple times .



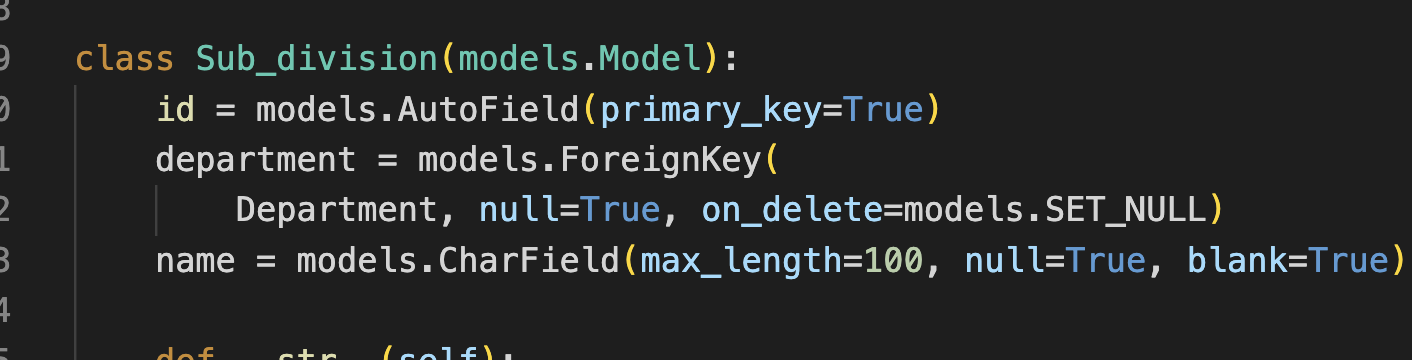
* Forward and reverse **relationships** aren’t used in the codebase .
  + E.g.,



* No **ACID** properties fulfilled in the code base .
* There is a high chance of database **concurrency** problems .

**5.2 Schemas and Indexing**

* Never ever use **underscore** in schema definition



* Add **indexing** for schemas for faster access .
* Add defined **string representation** for human readability .

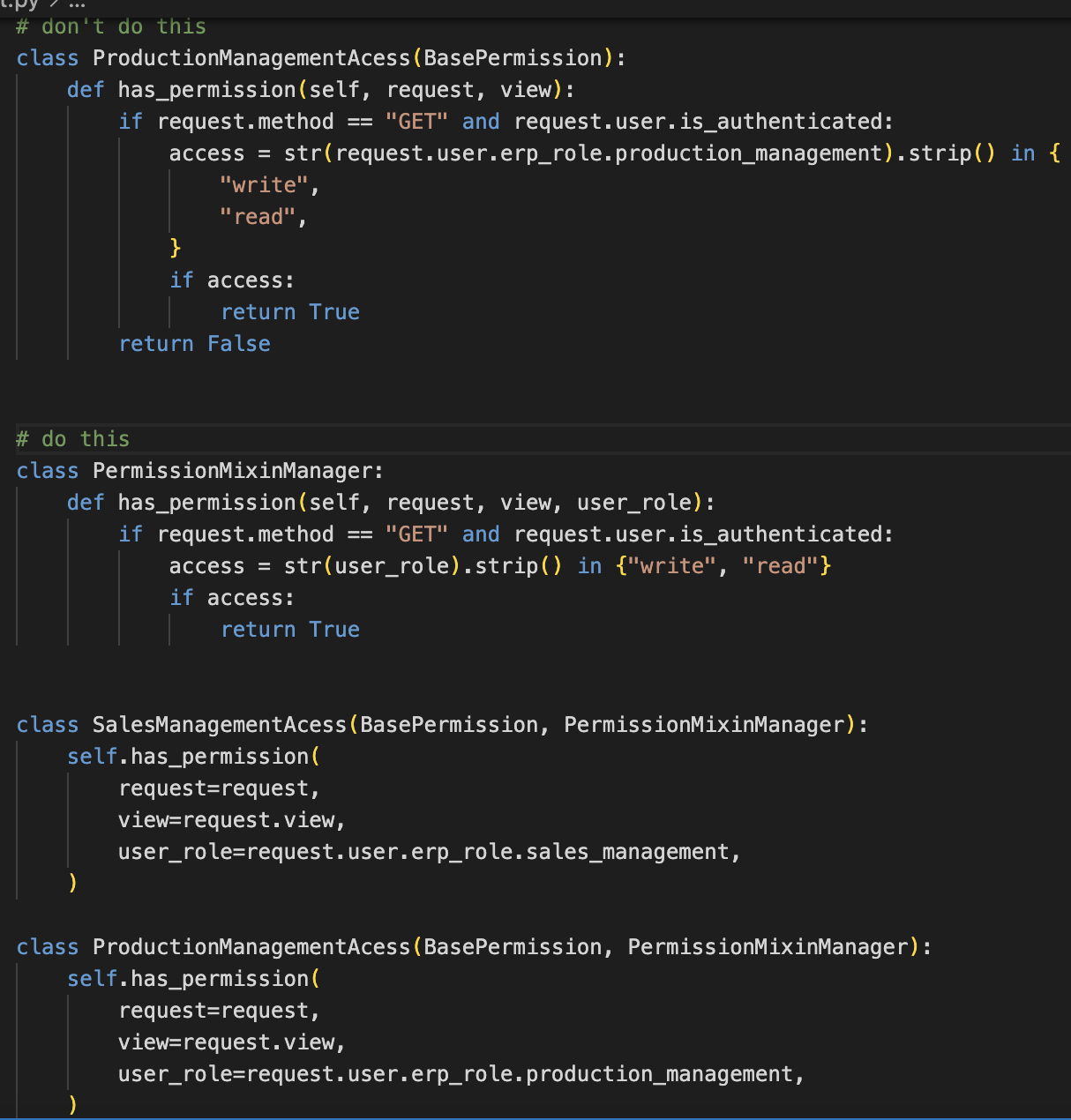
**5.3 Performance tuning**

* Implement **mem-cache** or redis cache mechanism for frequently used queries or objects .
* Use **CDN** to cache assets and to serve static data .
* Use **django silk** for profiling and query fetch time .
* Add **pagination** for large queryset and table data .
* Add rate limiting and **throttling** if required .
* Enable **Gzip** technique for data compression .
* Enable Database connection age and pooling if required .
* REF : [Link\_1](https://docs.djangoproject.com/en/4.2/topics/performance/) , [Link\_2](https://www.django-rest-framework.org/api-guide/throttling/) , [profiler](https://silk.readthedocs.io/en/latest/)

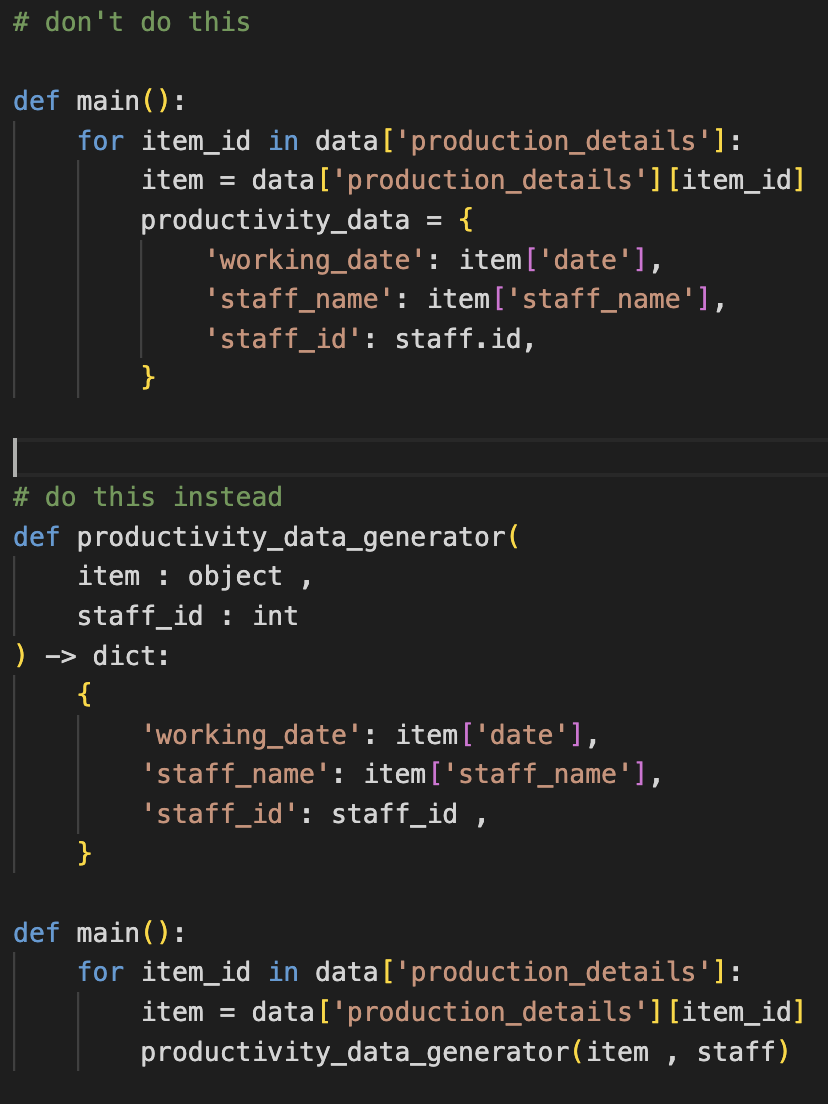
**6 . REUSABILITY AND REFACTORING**

**6.1 Modularity and Reusability**

* In the current code most of the functional logic’s aren’t reused well .
* There are some flaws with the code design and some of them are **spaghetti** code . E.g.,



* More **fatty functions** which literally makes them less readable .



**6.2 Refactoring and Organisation**

* The current codebase is a **monorepo** model , both client and server side requires refactoring of code in order to achieve scalability and **reusable** functions .
* As a result there is **tight coupling** in the codebase .
* All long and **monolithic** functions need to be cleaned and separated as chunks .
* The current code base requires a code composition and organisation strategy .
* Add **better comments** for file navigation .
* Adjust utilities , common constants and organise them in a separate folder .
* Abstract **classes** for your needs , in order to avoid code smells and anti patterns .

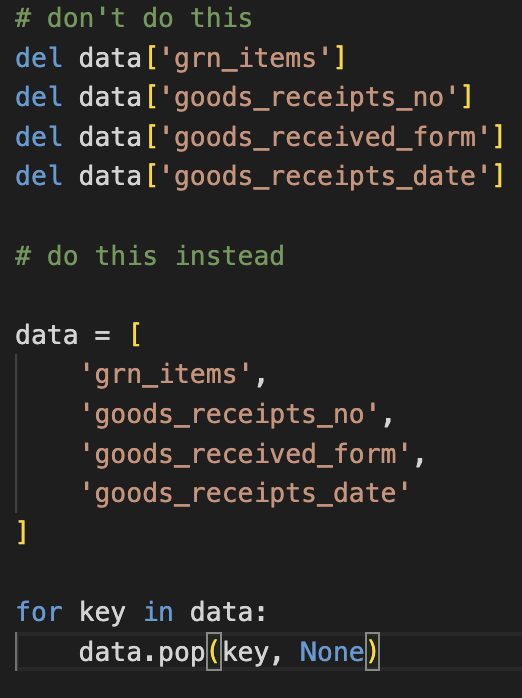
**7 . AREA OF IMPROVEMENT**

**7.1 Design**

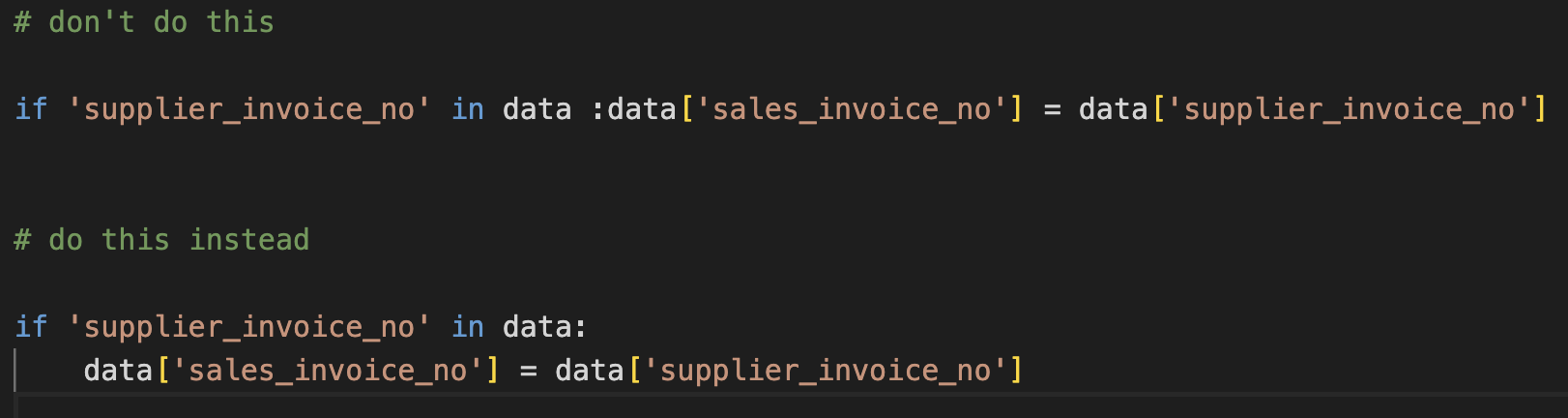
* It is essential to improve error handling and make it more **robust** .
* More dead and unused code , remove **dead** **code** or tag them as unused .
* Add a **preliminary** note on each file for better explanation of what the file does .
* The code doesn’t fulfil **OOAD** principles , all classes / functions are fatty .
* Use **DRY** and **KISS** principles .

**7.2 Functionality**

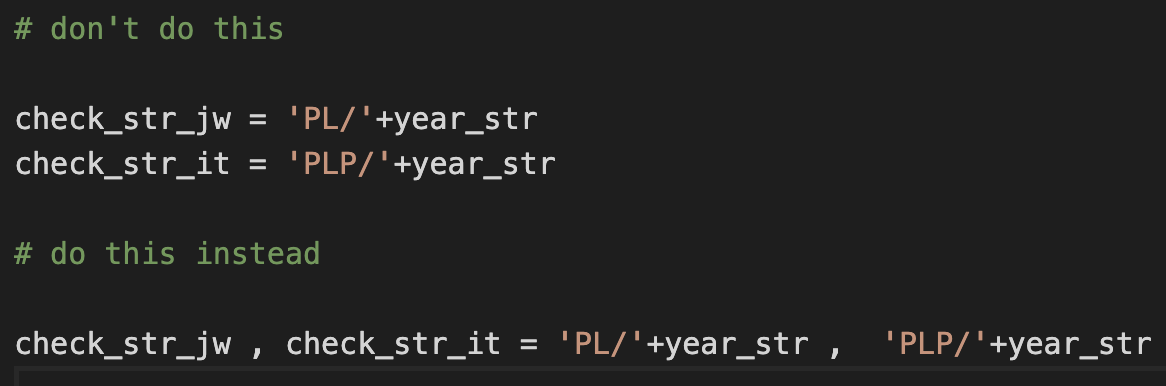
* Use **iterations** when needed



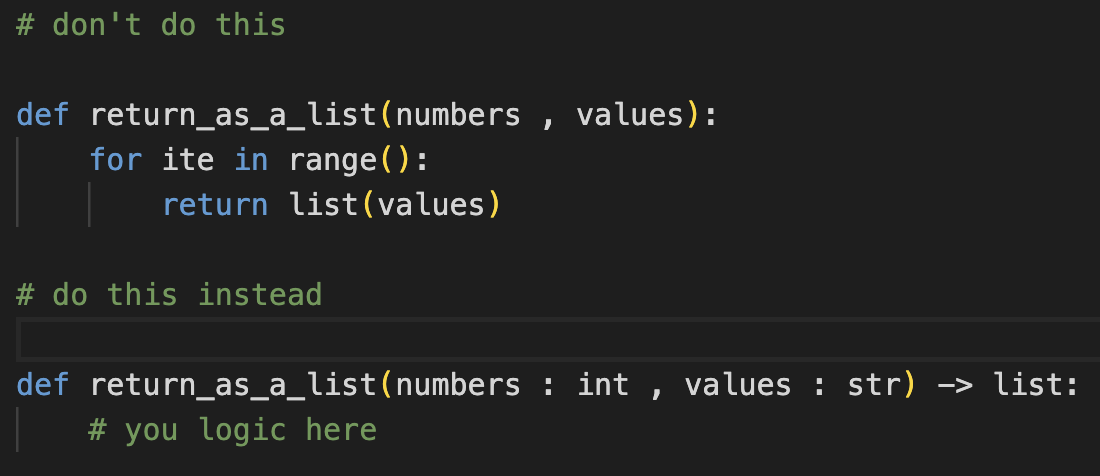
* Never use one liners , it makes them **less readable** .



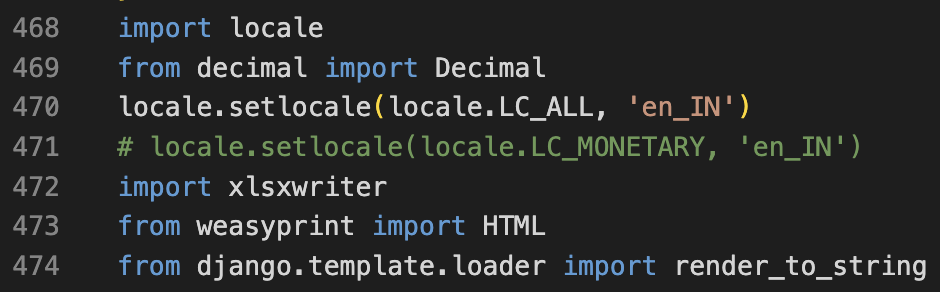
* Use **one liners** for multi assignment , data unpacking , list and dict comprehensions .



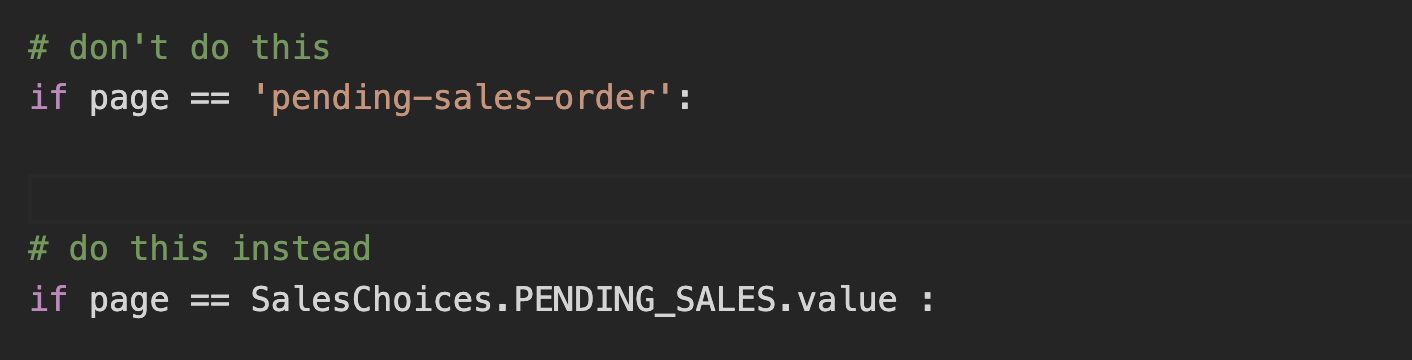
* Add necessary **annotations** and type hints in the code base



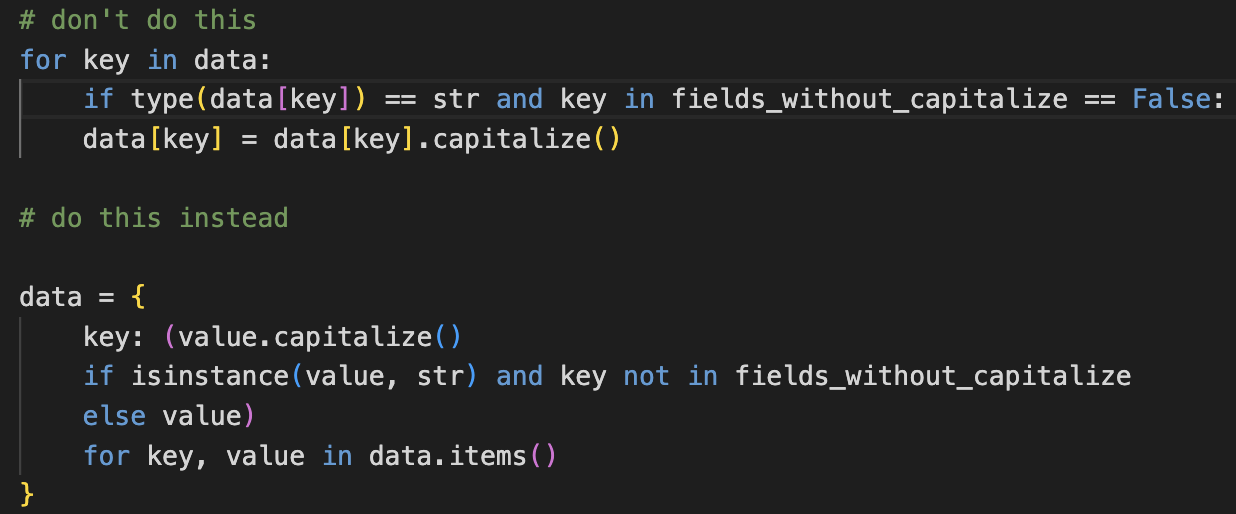
* Import **dependencies** always on top



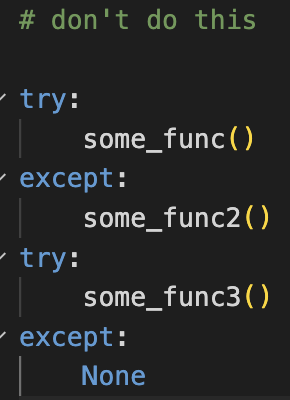
* Add named **Enum** for conditions , value setters , iterations .



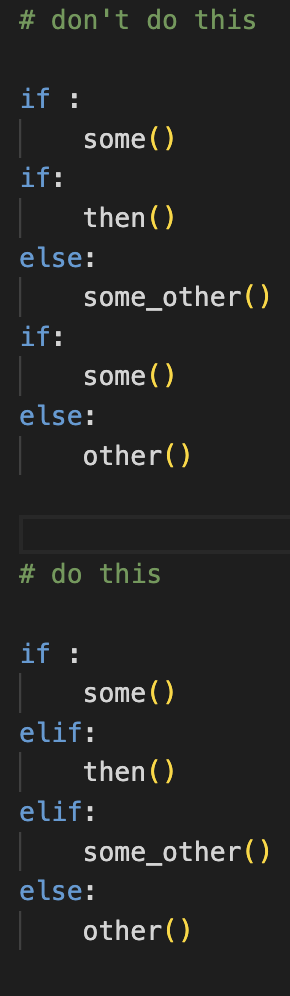
* Avoid common **iteration** mistakes



* Use exception clauses wisely not as in below image



* More global values assigned , **avoid globals .**
* Arrange conditional statements based on flow scenarios.



**7.3 Security Measures**

* Pay attention to security practices .
* Security **audit** and application vulnerability checkup need to be taken .
* Add a **honeypot** for the admin panel and check measures for SQL injection .

**7.4 Versioning**

* Maintain valid version names or branch names for future code maintenance .
* Use Semantic Versioning (**SemVer**) with version tagging for better branch management .
* E.g., :
  + DEV-v1.2
  + TEST-v1.3

**7.5 Management and Maintenance**

* Maintain a brief **change log** to document changes in releases .
* Add a document for **rollback plan** procedures for backup and restoration .
* Provide **migration** and **installation** guidelines .
* Make file modular , split all fatty functions into chunks
* Maintain at least 300 ~ 350 lines of code (**LOC**) per file .
* There is a flaw with code structure in maintaining the **MVC** architecture .

**8 . CONCLUSION**

In conclusion, this code review has provided valuable insights into the quality and maintainability of the codebase. While there are strengths and positive aspects to acknowledge ,

* There are some areas where improvements can enhance the overall **code quality** .
* It is essential to prioritise the identified improvements and implement them .
* I appreciate the **dedication** and **effort** put into this codebase, but some fine tuning needs to be done .
* Don’t take this code review as personally , knowledge sharing is not an insult , It is always better to **fail** and **fix** .
* Based on CEDAR all the improvements and issues are mentioned with **examples** and **actions** .
* I haven’t covered code **testing** content in this code review document .

Sincerely,

**Name** :ANAND RAJ B

**Role** : SDE - L3

**Date** : 19-09-2023