Excalibur 6738 Code Conventions

2024-2025 Season

Version 1.2

General Guidelines

Correct Code

- Consistency: Follow these conventions consistently throughout the codebase to ensure uniformity.
- Clarity: Ensure that your code is clear and self-explanatory. Use meaningful variable names and avoid ambiguous expressions.
- Modularity: Break down large functions and classes into smaller, more manageable pieces. Each function should perform a single task.
- Error Handling: Implement robust error handling to manage exceptions and unexpected conditions gracefully.
- Optimization: Optimize code for performance only after ensuring clarity and correctness. Premature optimization can lead to complex and hard-to-maintain code.
- Maintainability: Write code that is easy to maintain and modify. Anticipate changes and design accordingly.

SOLID Principles

- Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one job or responsibility.
- Open/Closed Principle (OCP): Software entities should be open for extension but closed for modification.
- Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of a subclass without affecting the functionality of the program.
- Interface Segregation Principle (ISP): No client should be forced to depend on methods it does not use.
- Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions.

Code Review

- Peer Review: All code must be reviewed by at least one other team member before merging into the main branch.
- Review Criteria: Ensure the code meets all guidelines mentioned here. Focus on readability, correctness, and adherence to SOLID principles.

Naming Conventions

- Do not use non-standard abbreviations. Always use full words unless a common abbreviation is universally understood within the team.
- Use only proper English. No Hebrew. If unsure about something, Google it.
- Use {...}s for plurals (e.g., movies, not movie).

Allowed Abbreviations Table:

שם	משמעות	שם	משמעות
k,j,i	עבור משתני אינדקס (זה בסדר גם לשכפל kk,jj,ii ע"מ למנוע בלבול עם קבועים מרוכבים)	*dest, src*	destination, source
ptr	עבור מצביעים	*fd*	file descriptor עבור
n	עבור מס' שלם (אם אין שם מתאים בעל יותר משמעות)	*str*	string עבור
		len	length עבור
num, val	עבור number, value בהתאמה	*arg*	argument עבור
in/out	עבור input/output למשל: val_out, num_in	*tmp, temp*	עבור temporary (בהיעדר שם מתאים).
max, min	עבור מינימום ומקסימום (כדאי – כחלק משם כדי למנוע התנגשות עם הפונק' המתמטיות)	*func, diff*	function, difference

Documentation

- Use docstrings (or equivalent documentation blocks) for all functions, methods, and classes.
- Comments: Use comments to explain why something is done, not what is done. The code itself should be self-explanatory for the "what".
- Inline Comments: Use inline comments sparingly, and only when necessary to explain complex logic.
- Block Comments: Use block comments to describe the purpose and functionality of a block of code.

Java Code Conventions

Naming Conventions

- Variables: Use camelCase for variable names (e.g., motorSpeed, sensorValue).
- Constants: Use UPPER_SNAKE_CASE for constants (e.g., MAX_SPEED).
- Functions/Methods: Use camelCase for function and method names (e.g., calculateDistance(), initializeRobot()).
- Classes: Use PascalCase for class names (e.g., RobotController, SensorManager).
- For class variables, use the prefix m_ (e.g., m_var).

Code Style

- Indentation: Use 4 spaces per indentation level (tab = 4).
- Line Length: Limit lines to 80 characters where possible. If not, ensure readability.
- Spacing: Use spaces around operators and after commas. Avoid extraneous whitespace.
- Always use curly braces {} even if the block contains only one line.

Python (PEP 8):

Intro:

PEP 8 is the style guide for Python code. Following it helps make your code more readable and maintainable.

Indentation:

Python's way of sectioning parts like Functions and Class is by Indenting, in Java or C it will usually be done by curly braces.

In Python, you should Indent with 4 spaces or a Tab, Similar to this:

```python
def my_fnction():
	# indentation
	return None
```

Naming Conventions:

Component	Example
Module	`mymodule.py`, `data_processing.py`
Package	`mypackage`, `data_analysis`
Class	`MyClass`, `DataProcessor`
Exception	`MyCustomError`, `DataValidationError`
Global Variable/Constant	`MAX_SIZE`, `PI`
Function	`my_function`, `process_data`
Variable	`my_variable`, `data_list`
Method	`my_method`, `process_data`
Instance Variable	`self.instance_variable`, `self.data_list`
Class Variable	`class_variable`
Function/Method Argument	`arg1`, `param1`
Private Variable/Method	`_private_variable`, `_private_method`
Mangled Variable	`__mangled_variable`
Module Variable (Global)	`MAX_CONNECTIONS`, `SERVER_URL`

Spacing Conventions:

Context	Number of Blank Lines
Between top-level function and class definitions	2
Between method definitions in a class	1
Between logical sections of code	1 (sparingly)
Before imports	0 (imports should be grouped)
After imports	1 (if followed by code)

Python

```python
#Binary Operators:
x = 1 + 2
y = x - 3

Keywords and Parentheses:
print('Hello, world!')

Commas, Semicolons, and Colons:
if x == 4: print(x, y); x, y = y, x

Inside Parentheses, Brackets, or Braces:
spam(ham[1], {eggs: 2})

Keyword Arguments or Default Parameter Values:
def complex(real, imag=0.0):
    return magic(r=real, i=imag)

Trailing Blank Lines:
def my_function():
    pass

Inline Comments:
x = x + 1  # Increment x by 1

Before a Left Parenthesis:
class Foo:
    def __init__(self):
        pass

#do not use not in arguments
```

Flask (Python 10.0+)

Introduction

Flask is a lightweight and flexible web framework for Python, ideal for quickly building web applications. It provides essential features like URL routing and templates, and its modular design allows easy extension for added functionality. Flask's simplicity makes it a great choice for both beginners and experienced developers.

Component	Convention	Example
Routes	Descriptive, lowercase with underscores	`/user_profile`, `/create_user`
Functions	Descriptive, lowercase with underscores	`get_user_data`, `save_user_data`

Convention	Description
Project Structure	Organize project files and directories logically.
Application Factory	Use an application factory to initialize the app.
Configuration	Separate configuration settings for different environments.
Blueprints	Use Blueprints to organize routes and views.
Models	Define database models in a separate module.
Forms	Use Flask-WTF and define forms in a separate module.
Templates	Store HTML templates in the `templates` directory.
Static Files	Store static files in the `static` directory.
Error Handling	Define custom error pages.
Testing	Write unit tests and store them in a `tests` directory.

```python
def get_user_data(user_id):
      pass
@app.route('/user_profile')
def user_profile():
```

```
        pass
```

File Handling

Project Structure:

```
C/C++
my_flask_app/
├── app/
│      ├── __init__.py
│      ├── routes.py
│      ├── models.py
│      ├── forms.py
│      ├── templates/
│      └── static
              ├── css/
              ├── js/
              └── images/
├── config.py
├── run.py
└── requirements.txt
```

JS code conventions:

1. **File Naming**: Use camelCase for file names, and PascalCase for React component file names.

```
import MyComponent from './MyComponent';
import utils from './utils';
```

2. Indentation: Use 2 spaces for indentation.

```
function example() {
  const x = 1;
  return x;
}
```

3. Semicolons: Always use semicolons at the end of statements.

```
const x = 1;
```

4. Quotes: Use single quotes for strings.

```
const greeting = 'Hello, world!';
```

5. Braces and Spacing:

- Use braces for all multi-line blocks.
- Place opening braces on the same line as their control statement.
- Ensure there is a space before the opening brace.

```
if (condition) {
  doSomething();
} else {
  doSomethingElse();
}
```

6. Arrow Functions: Prefer arrow functions, especially for callbacks.

```
const numbers = [1, 2, 3];
const squares = numbers.map(number => number * number);
```

7. Variables: Use const for constants and let for variables that will change. Avoid using var.

```
const pi = 3.14;
let count = 0;
```

8. Destructuring: Use destructuring assignment when possible.

```
const { data, teamNumbers, teamData } = state;
```

9. Template Literals: Use template literals for string concatenation.

```
const name = 'John';
const greeting = `Hello, ${name}!`;
```

10. Avoid Inline Styles: Use CSS classes instead of inline styles for better readability and maintainability.

```
<div className="container"></div>
```

11. Hooks and Effects:

- Use hooks at the top level of your function component.
- Name state setters clearly and use appropriate initial values.

```
const [data, setData] = useState([]);
useEffect(() => {
  fetchData();
}, []);
```

12. Function and Variable Naming: Use descriptive names in camelCase for functions and variables.

```
const fetchData = async () => {
  // Fetch data logic
};

const handleInputChange = (index, event) => {
  // Handle input change logic
};
```