

# Coding Standards

## 1. Consistency

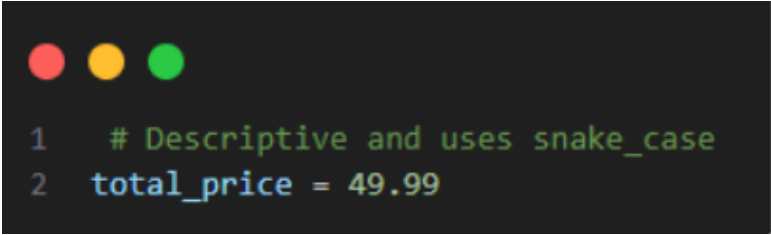
- Ensure a uniform coding style throughout the entire codebase to improve readability and maintainability.
  - Use consistent naming conventions, indentation, and formatting throughout the codebase.
- 

## 2. Naming Conventions

### 2.1 Variables

- Use **snake\_case** for variable names (all lowercase with underscores).
- Choose descriptive and meaningful names that convey the variable's purpose.
- Avoid single-letter or ambiguous names unless used in small scopes (e.g., loop counters).
- Leading underscore can indicate internal or “private” variables.

#### Example:



```
1  # Descriptive and uses snake_case
2  total_price = 49.99
```

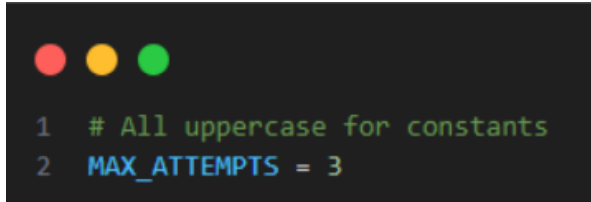
---

### 2.2 Constants

- Define constants at the module level.
- Use **UPPERCASE\_SNAKE\_CASE** for constants to indicate immutability by convention.

- Python does not enforce immutability but follow this convention for clarity.

**Example:**

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays two lines of Python code: a comment and a constant assignment.

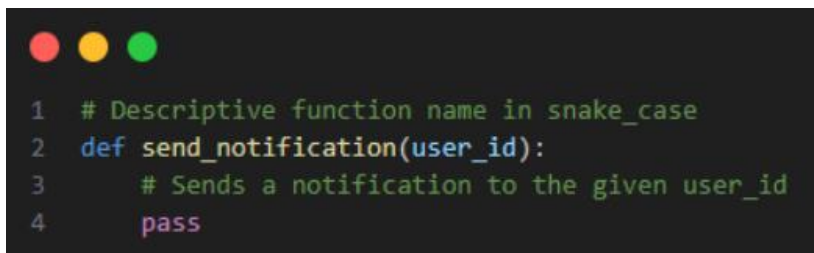
```
1 # All uppercase for constants
2 MAX_ATTEMPTS = 3
```

---

## 2.3 Functions

- Use **snake\_case** for function names.
- Function names should be **action-oriented** and descriptive, representing what the function does.

**Example:**

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It displays four lines of Python code: a comment, a function definition, another comment, and a pass statement.

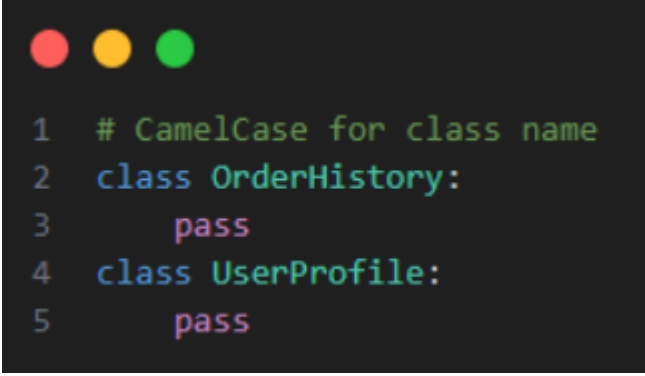
```
1 # Descriptive function name in snake_case
2 def send_notification(user_id):
3     # Sends a notification to the given user_id
4     pass
```

---

## 2.4 Classes

- Use **CamelCase** for class names.
- Class names should start with an uppercase letter and be nouns or noun phrases.

**Example:**



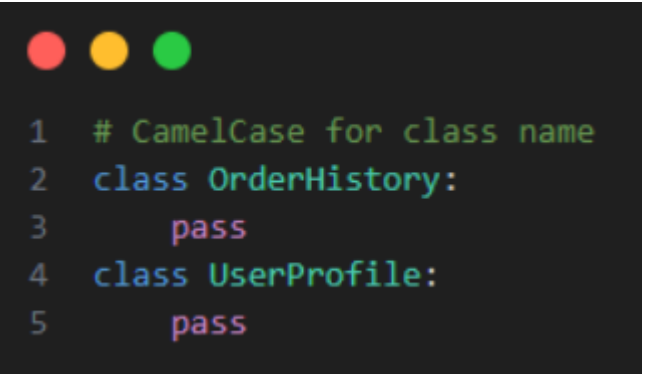
```
1  # CamelCase for class name
2  class OrderHistory:
3      pass
4  class UserProfile:
5      pass
```

---

## 2.5 Packages & Modules

- Use **short, all-lowercase** names for packages and modules.
- Avoid underscores in package names to maintain compatibility across OSs.
- Module names may use underscores if necessary for readability.

**Example:**



```
1  # CamelCase for class name
2  class OrderHistory:
3      pass
4  class UserProfile:
5      pass
```

---

## 3. Comments & Documentation

- Use **docstrings** (""" ... """) to describe modules, classes, methods, and functions.
- Docstrings should clearly explain the purpose, arguments, and return values.
- Use inline comments sparingly, only when the code's intent is not obvious.
- Avoid obvious comments that repeat what the code does.

**Example:**

```
1 def calculate_discount(price, percentage):
2     """
3     Calculate the discount amount for a given price and percentage.
4
5     Args:
6         price (float): The original price.
7         percentage (float): Discount percentage.
8
9     Returns:
10        float: The discount amount.
11    """
12    # Ensure percentage is within 0-100
13    if not 0 <= percentage <= 100:
14        raise ValueError("Percentage must be between 0 and 100.")
15    return price * (percentage / 100)
```

---

## 4. Formatting & Indentation

- Use **4 spaces** per indentation level; no tabs.
- Limit lines to **79 characters** to improve readability (PEP 8).
- Use blank lines to separate logical sections of code:
  - Two blank lines before top-level functions and classes.
  - One blank line between methods inside a class.

### Example:

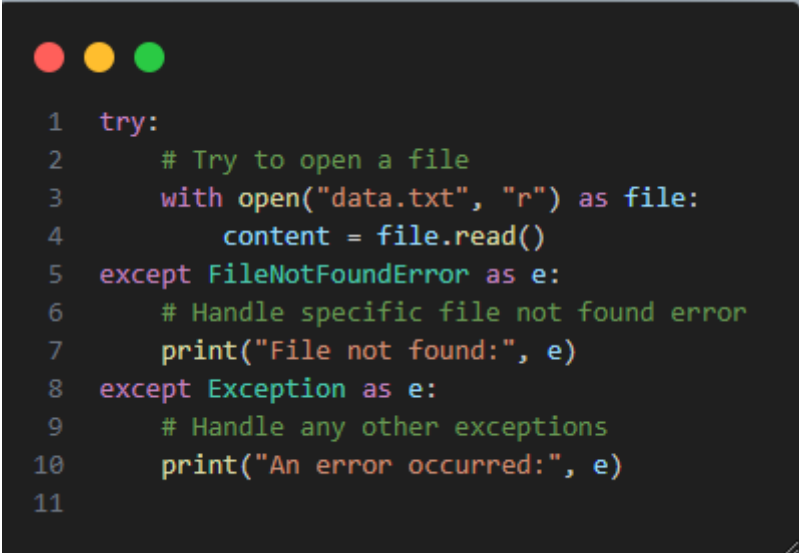
```
1 def display_user(name):
2     # Good formatting with 4 spaces per indent
3     if name:
4         print(f"Hello, {name}!")
5     else:
6         print("No user name provided.")
7
```

---

## 5. Error Handling

- Use **try-except** blocks to gracefully handle exceptions.
- Prefer **specific exceptions** over broad ones (except ValueError rather than except Exception).
- Optionally, raise meaningful exceptions with informative messages.

**Example:**

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is a Python try-except block for file handling.

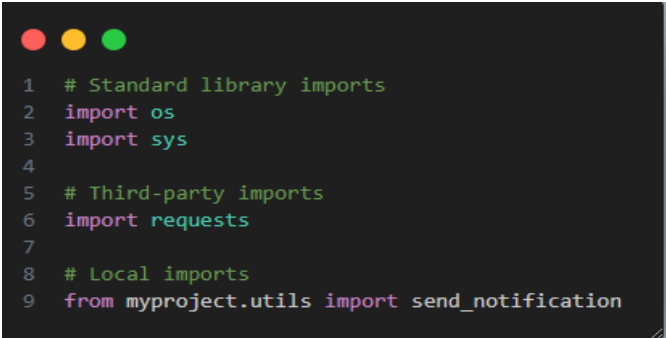
```
1  try:
2      # Try to open a file
3      with open("data.txt", "r") as file:
4          content = file.read()
5  except FileNotFoundError as e:
6      # Handle specific file not found error
7      print("File not found:", e)
8  except Exception as e:
9      # Handle any other exceptions
10     print("An error occurred:", e)
11
```

---

## 6. Import Formatting

- Write **one import statement per line**.
- Organize imports in this order with a blank line between groups:
  1. Standard library imports
  2. Third-party imports
  3. Local application imports
- Use **absolute imports** rather than relative imports for clarity.

**Example:**

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code shows imports organized into three groups with blank lines between them.

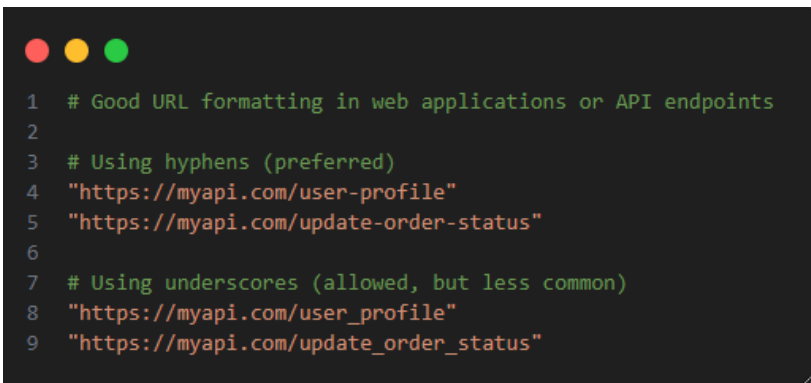
```
1  # Standard library imports
2  import os
3  import sys
4
5  # Third-party imports
6  import requests
7
8  # Local imports
9  from myproject.utils import send_notification
```

---

## 7. URL Formatting

- In web applications or APIs, use **lowercase letters** in URLs.
- Separate words with **hyphens (-)** or **underscores (\_)** to improve readability.
- Keep URLs concise and meaningful.

### Example:



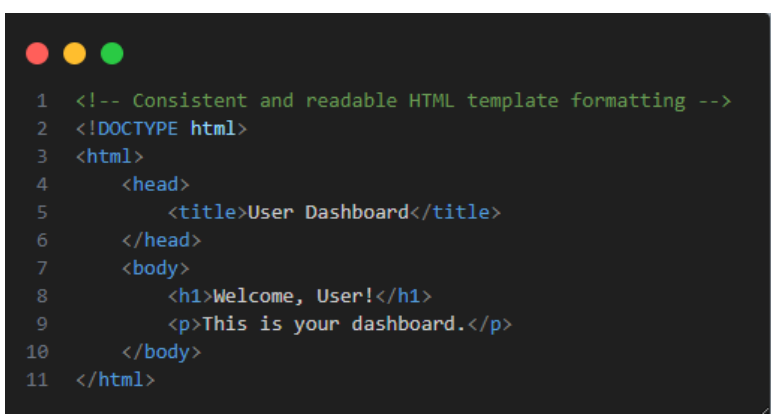
```
1 # Good URL formatting in web applications or API endpoints
2
3 # Using hyphens (preferred)
4 "https://myapi.com/user-profile"
5 "https://myapi.com/update-order-status"
6
7 # Using underscores (allowed, but less common)
8 "https://myapi.com/user_profile"
9 "https://myapi.com/update_order_status"
```

---

## 8. Template Style (HTML)

- Use consistent and readable indentation in templates (usually 2 or 4 spaces).
- Write clean, semantic HTML markup to improve accessibility and maintenance.

### Example

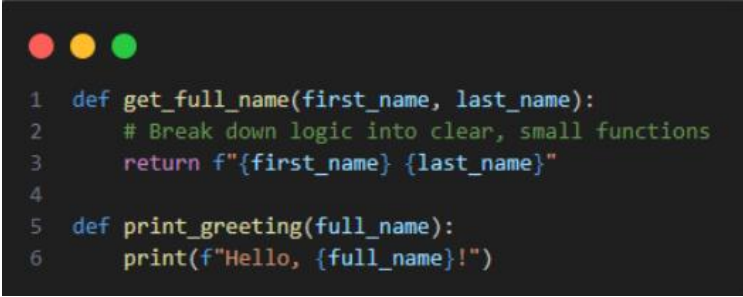


```
1 <!-- Consistent and readable HTML template formatting -->
2 <!DOCTYPE html>
3 <html>
4   <head>
5     <title>User Dashboard</title>
6   </head>
7   <body>
8     <h1>Welcome, User!</h1>
9     <p>This is your dashboard.</p>
10  </body>
11 </html>
```

## 9. Code Readability & Reusability

- Break complex tasks into smaller, reusable functions or methods.
- Use meaningful variable and function names; avoid cryptic abbreviations.
- Encapsulate reusable logic inside functions, classes, or modules to avoid duplication.

Example:



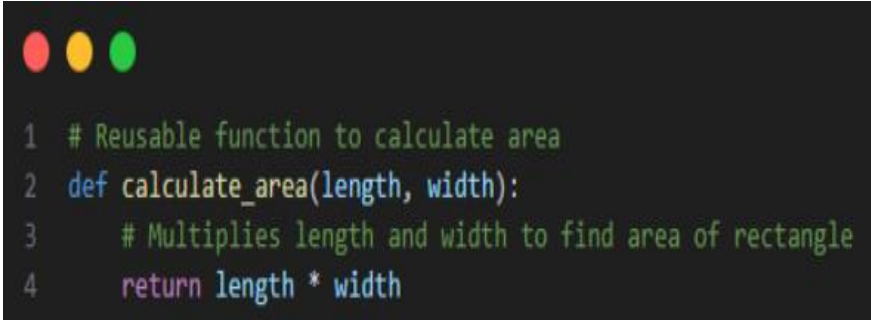
```
1 def get_full_name(first_name, last_name):
2     # Break down logic into clear, small functions
3     return f"{first_name} {last_name}"
4
5 def print_greeting(full_name):
6     print(f"Hello, {full_name}!")
```

---

## 10. Code Reusability

- Encapsulate reusable code into functions, classes, or modules to avoid duplication

Example:



```
1 # Reusable function to calculate area
2 def calculate_area(length, width):
3     # Multiplies length and width to find area of rectangle
4     return length * width
```

---

## 11. Testing and Quality Assurance

- Write **unit tests** to ensure code correctness and prevent regressions.
- Use Python testing frameworks like unittest or pytest.
- Structure tests clearly with descriptive test method names.

- Encapsulate reusable code into functions, classes, or modules to avoid duplication

### Example:

```
1 import unittest
2
3 # Unit test for calculate_area function
4 class TestAreaCalculation(unittest.TestCase):
5     def test_calculate_area(self):
6         # Checks if area calculation is correct for 5 x 10
7         self.assertEqual(calculate_area(5, 10), 50)
```

---

## 12. Security

- Always **sanitize and validate user inputs** to prevent injection and other vulnerabilities.
- Follow best practices for secure coding, especially in web and API development.

### Example:

```
# Validate user input to avoid injection or logic errors
try:
    age = int(user_input)
    if age < 0 or age > 120:
        raise ValueError("Invalid age range.")
except ValueError as e:
    print("Input error:", e)

# Flask example: sanitize input to prevent XSS
from flask import request, escape

@app.route("/submit", methods=["POST"])
def submit():
    username = escape(request.form['username']) # Prevent script injection
    return f"Hello, {username}!"
```



## 13. References & Resources

### 1. **PEP 8 – Python Style Guide**

The official Python style guide that defines conventions for code layout and formatting.

 <https://peps.python.org/pep-0008/>

### 2. **Google Python Style Guide**

Google's internal style rules, helpful for writing large-scale, maintainable Python code.

 <https://google.github.io/styleguide/pyguide.html>

### 3. **CKAN Python Contribution Guide**

Community-maintained style and contribution guide for open-source Python projects.

 <https://docs.ckan.org/en/2.9/contributing/python.html>

### 4. **Python Coding Standards by Zenesys**

Best practices and coding guidelines used in real-world enterprise development.

 <https://www.zenesys.com/python-coding-standards-best-practices>

### 5. **JU SWE Wiki – Updated Coding Standard (JU\_CSE-27)**

Customized standard used by JU High Fives for consistent academic and team projects.

 [https://github.com/JU-CSE-27/swewiki/blob/master/resources/Updated\\_coding-standard.pdf](https://github.com/JU-CSE-27/swewiki/blob/master/resources/Updated_coding-standard.pdf)