Clean Code Principles & Practices

A cheat sheet summarizing key principles and practices for writing clean, maintainable, and readable code.

Core Principles

Readability & Simplicity

Principle: Code should be easy to understand and modify.

- Use meaningful names for variables, functions, and classes.
- Keep functions small and focused on a single task.
- Avoid complex logic and nested conditional statements.

Benefit: Reduces cognitive load, speeds up debugging, and facilitates collaboration.

Example:

```
# Bad
def process_data(d, f):
    for i in range(len(d)):
        if f(d[i]) > 10:
            print(d[i])

# Good
def process_data(data, filter_func):
    for item in data:
        if filter_func(item) > 10:
            print(item)
```

DRY (Don't Repeat Yourself)

Principle: Avoid duplicating code. Abstract common logic into reusable functions or modules.

Benefit: Reduces redundancy, simplifies maintenance, and minimizes the risk of inconsistencies.

Example:

```
# Bad
def calculate_area_rectangle(width, height):
    return width * height

def calculate_perimeter_rectangle(width, height):
    return 2 * (width + height)

# Good
def calculate_rectangle_properties(width, height):
    area = width * height
    perimeter = 2 * (width + height)
    return area, perimeter
```

KISS (Keep It Simple, Stupid)

Principle: Favor simplicity over complexity. Choose the simplest solution that meets the requirements.

Benefit: Easier to understand, debug, and maintain. Reduces the likelihood of introducing bugs.

Example:

```
# Bad
def complex_calculation(x, y, z):
    # A very complicated formula
    result = (x**2 + y**2)**0.5 * z / (1 + x * y)
    return result

# Good
def simple_calculation(x, y):
    return x + y
```

Functions

Function Length

Guideline: Functions should be small and focused. Ideally, a function should not exceed 20-30 lines of code.

Reasoning: Shorter functions are easier to understand, test, and reuse. They promote modularity and reduce complexity.

Technique: Break down large functions into smaller, more manageable sub-functions.

Example:

```
# Bad
def process_order(order):
    # Many lines of code doing multiple things:
    # - Validate order
    # - Calculate total
    # - Apply discounts
    # - Update inventory
    # - Send confirmation email
    pass

# Good
def validate_order(order): pass
def calculate_total(order): pass
def apply_discounts(order): pass
def update_inventory(order): pass
def send_confirmation_email(order): pass

def process_order(order):
    if validate_order(order):
        total = calculate_total(order)
        discounted_total = apply_discounts(total)
        update_inventory(order)
        send_confirmation_email(order)
```

Function Arguments

Guideline: Minimize the number of function arguments. Ideally, a function should have 0-3 arguments.

Reasoning: Functions with fewer arguments are easier to call and understand. They reduce the risk of errors and improve readability.

Technique: Use objects or data structures to group related arguments. Consider using a builder pattern for functions with many optional arguments.

Example:

```
# Bad
def create_user(name, age, address, phone, email):
    # ...
    pass

# Good
class User:
    def __init__(self, name, age, address, phone, email):
        self.name = name
        self.age = age
        self.address = address
        self.phone = phone
        self.email = email

def create_user(user: User):
    # ...
    pass
```

Function Naming

Guideline: Choose descriptive and meaningful names for functions. Function names should clearly indicate what the function does.

Reasoning: Good function names improve code readability and make it easier to understand the purpose of the function.

Technique: Use verbs to name functions (e.g., `calculate_total`, `validate_input`). Follow a consistent naming convention.

Example:

```
# Bad
def x(y): # What does x do with y?
    return y * 2

# Good
def double_value(value):
    return value * 2
```

Comments and Documentation

Purpose of Comments

Guideline: Comments should explain the *why* behind the code, not the *what*. Good code should be self-documenting.

Reasoning: Comments can provide valuable context and insights into the design decisions and intent of the code.

Technique: Use comments sparingly and only when necessary to clarify complex logic or provide additional information.

Best Practice: Avoid redundant comments that simply restate the code.

Documentation

Guideline: Use documentation to describe the purpose, usage, and design of modules, classes, and functions.

Reasoning: Documentation helps other developers (and your future self) understand how to use and maintain the code.

Technique: Use docstrings, README files, and other forms of documentation to provide comprehensive information about the codebase.

Tools: Sphinx (Python), JSDoc (JavaScript), etc.

Commenting Style

Guideline: Follow a consistent commenting style throughout the codebase.

Reasoning: Consistent style improves readability and maintainability.

Technique: Use appropriate commenting syntax for the programming language (e.g., `#` for Python, `//` for JavaScript).

Example:

```
# This function calculates the total cost of an order.
def calculate_total(order):
    # Apply discounts based on customer loyalty.
    pass
```

Error Handling

Importance of Error Handling

Guideline: Implement robust error handling to prevent unexpected crashes and provide informative error messages.

Reasoning: Proper error handling improves the reliability and usability of the software.

Technique: Use try-except blocks (or equivalent) to catch exceptions and handle them gracefully.

Best Practice: Log errors for debugging and monitoring purposes.

Specific Exceptions

Guideline: Catch specific exceptions rather than general exceptions.

Reasoning: Catching specific exceptions allows you to handle different types of errors in different ways.

Technique: Identify the specific exceptions that can be raised by the code and catch them individually.

Example:

```
# Bad
try:
    # Some code that might raise an exception
    pass
except Exception as e:
    print(f"An error occurred: {e}")

# Good
try:
    # Some code that might raise an exception
    pass
except ValueError as e:
    print(f"Invalid value: {e}")
except TypeError as e:
    print(f"Invalid type: {e}")
```

Resource Management

Guideline: Ensure that resources (e.g., files, network connections) are properly released after use.

Reasoning: Failure to release resources can lead to resource leaks and performance issues.

Technique: Use try-finally blocks or context managers (e.g., `with` statement in Python) to ensure that resources are always released.

Example:

```
# Good
with open("my_file.txt", "r") as f:
    # Do something with the file
    data = f.read()
# File is automatically closed when the 'with' block exits
```