Code Analysis: Bank Account Transaction Processor

What Makes Code Bad and How to Fix It

AISE501 – AI in Software Engineering I

Dr. Florian Herzog

Spring Semester 2026

Contents

1 Overview

This document analyses two implementations of a bank account transaction processor. Both read account state and transactions from JSON files, validate each transaction, apply valid ones, reject invalid ones, and write results. Both produce identical output, but `bank_bad.py` violates many PEP 8 and clean code principles, while `bank_good.py` follows them consistently.

2 Violation 1: Unused Imports and Import Formatting

Bad Code

```
import json,sys,os,copy;from datetime import datetime
```

Clean Code

```
import json
from typing import TypedDict, Optional
```

What is wrong:

- `sys`, `os`, `copy`, and `datetime` are imported but **never used**.
- All imports are **on a single line** separated by commas, with a semicolon joining two import statements.
- PEP 8 requires each import on its own line and groups separated by blank lines (standard library, third-party, local).

Principles Violated

- **PEP 8 – Imports**: Imports should be on separate lines. Remove unused imports.
- **KISS**: Unused imports add noise and suggest false dependencies.

3 Violation 2: No Documentation or Docstrings

Bad Code

The file has **no module docstring** and **no function docstrings**. The only comment in the entire file is:

```
    # find account
    ...
    # print results
```

These comments describe *what* the next line does (which is already obvious from the code), not *why*.

Clean Code

```
"""Bank account transaction processor.

Reads account state and a list of transactions from JSON files,
validates and applies each transaction, then writes updated account
state and a transaction log (accepted / rejected) to output files.
"""
```

Every function has a docstring:

```
def validate_common(
    account: Optional[Account],
    amount: float,
) -> Optional[str]:
    """Run validations shared by all transaction types.

    Returns an error message string, or None if valid.
    """
```

Principles Violated

- **PEP 257**: All public modules and functions should have docstrings.
- **Clean Code – Comments**: Don't add noise comments that just restate the code. Comments should explain *why*, not *what*.

4 Violation 3: Implicit Data Model

Bad Code

The bad version operates on raw dictionaries with no type declarations. A reader must trace through the JSON file and every dictionary access to understand the data shape:

```
def proc(accs,txns):
    for t in txns:
        tp=t['type'];aid=t['account_id'];amt=t['amount'];tid=t['id']
        a=None
        for x in accs:
            if x['account_id']==aid:a=x
```

What fields does `t` have? What fields does `a` have? There is no way to know without reading the JSON file.

Clean Code

The good version defines explicit data types:

```
class Account(TypedDict):
    """A bank account with its current state."""
    account_id: str
    holder: str
    balance: float
    currency: str
    status: str             # "active" or "frozen"

class Transaction(TypedDict, total=False):
    """A financial transaction to be processed."""
    id: str
    type: str               # "deposit", "withdrawal", or "transfer"
    account_id: str
    amount: float
    description: str
    to_account_id: str      # only for transfers
    status: str             # added after processing
    reason: str             # added on rejection
```

All function signatures carry type annotations:

```
def find_account(accounts: list[Account], account_id: str) ->
    Optional[Account]:
```

Principles Violated

- **Zen of Python**: "Explicit is better than implicit."
- **Clean Code – Readability**: A reader should understand the data contract without tracing through runtime data.
- **PEP 484 / PEP 589**: Use type hints and `TypedDict` to document the structure of dictionary-based data.

5 Violation 4: Poor Naming

Bad Code

```
def loadJ(p):               # "J" for JSON? "p" for path?
def saveJ(p,d):             # "d" for data?
def proc(accs,txns):        # "proc" does what exactly?
    ok=[];bad=[]            # acceptable vs. rejected
    tp=t['type']            # "tp" is unpronounceable
    aid=t['account_id']     # "aid" looks like "aid" (help)
    amt=t['amount']         # "amt" -- abbreviation
    tid=t['id']             # "tid" -- never used again!
    a=None                  # "a" for account
    ta=None                 # "ta" for target account
    for x in accs:          # "x" for what?
    D=loadJ(...)            # capital "D" for a local variable
    T=loadJ(...)            # capital "T" for a local variable
```

Clean Code

```
def load_json(file_path):
def save_json(file_path, data):
def find_account(accounts, account_id):
def validate_common(account, amount):
def process_deposit(accounts, transaction):
def process_withdrawal(accounts, transaction):
def process_transfer(accounts, transaction):
def process_all_transactions(accounts, transactions):
def print_results(accounts, accepted, rejected):
```

What is wrong:

- Function names use **abbreviations** (`loadJ`, `saveJ`, `proc`) instead of descriptive snake_case names.
- Variable names are **single letters or short abbreviations** (`a`, `t`, `x`, `tp`, `aid`, `amt`, `ta`).
- `tid` is assigned but **never used** — dead code.
- `D` and `T` use **uppercase**, suggesting constants, but they are local variables.
- The name `ok` for accepted transactions and `bad` for rejected ones is **imprecise**.

Principles Violated

- **PEP 8 – Naming**: Functions and variables use `lower_case_with_underscores`. Constants use `UPPER_CASE`.
- **Clean Code – Descriptive Names**: "Other developers should figure out what a variable stores just by reading its name."
- **Clean Code – Consistent Vocabulary**: Don't mix `ok`/`bad` with `accepted`/`rejected`.
- **Clean Code – No Abbreviations**: `amt`, `tp`, `tid` are not words.

6 Violation 5: Formatting – Semicolons and Dense Lines

Bad Code

```
f=open(p,'r');d=json.load(f);f.close();return d

tp=t['type'];aid=t['account_id'];amt=t['amount'];tid=t['id']

a['balance']=a['balance']+amt;t['status']='accepted';ok.append(t)

if a==None:
    t['reason']='account not found';bad.append(t);continue
```

Clean Code

Every statement is on its own line with proper whitespace:

```
account = find_account(accounts, transaction["account_id"])
error = validate_common(account, transaction["amount"])
if error:
    return False, error

account["balance"] += transaction["amount"]
return True, "accepted"
```

What is wrong:

- **Semicolons** pack 3–4 statements onto one line, making it nearly impossible to follow the logic.
- **No whitespace** around = and after commas.
- Control flow (`continue`) is **hidden at the end of a dense line**.
- PEP 8 explicitly states: "Compound statements (multiple statements on the same line) are generally discouraged."

Principles Violated

- **PEP 8 – Compound Statements**: Generally discouraged. Each statement on its own line.
- **PEP 8 – Whitespace**: Surround operators with spaces. Space after commas.
- **Zen of Python**: "Readability counts." "Sparse is better than dense."

7 Violation 6: No Context Managers for File I/O

Bad Code

```
def loadJ(p):
    f=open(p,'r');d=json.load(f);f.close();return d

def saveJ(p,d):
    f=open(p,'w');json.dump(d,f,indent=2);f.close()
```

If `json.load(f)` raises an exception, the file is **never closed** because `f.close()` is skipped. This is a resource leak.

Clean Code

```
def load_json(file_path: str) -> dict:
    """Read and parse a JSON file, returning the parsed data."""
    with open(file_path, "r", encoding="utf-8") as file_handle:
        return json.load(file_handle)
```

The `with` statement guarantees the file is closed even if an exception occurs.

Principles Violated

- **Pythonic Code**: Always use context managers (`with`) for resource management.
- **Clean Code – Error Handling**: Code should be robust against exceptions. Manual `open`/`close` is error-prone.

8 Violation 7: God Function – Single Responsibility Violation

Bad Code

The function `proc()` is 38 lines long and handles **all of the following** in a single function:

- Finding accounts by ID
- Validating account status
- Validating amounts
- Processing deposits
- Processing withdrawals
- Processing transfers (including finding the target account)
- Handling unknown transaction types
- Building accepted and rejected lists

```
def proc(accs,txns):
    ok=[];bad=[]
    for t in txns:
        ...  # 35 lines of nested if/elif/else with continue
    return accs,ok,bad
```

Clean Code

The good version splits this into **seven focused functions**:

```
def find_account(accounts, account_id):        # lookup
def validate_common(account, amount):          # shared validation
def process_deposit(accounts, transaction):    # deposit logic
def process_withdrawal(accounts, transaction):# withdrawal logic
def process_transfer(accounts, transaction):   # transfer logic
def process_all_transactions(accounts, transactions): # orchestration
def print_results(accounts, accepted, rejected):      # output
```

A dispatch dictionary replaces the `if/elif` chain:

```
TRANSACTION_HANDLERS = {
    "deposit": process_deposit,
    "withdrawal": process_withdrawal,
    "transfer": process_transfer,
}
```

Principles Violated

- **SRP (Single Responsibility Principle)**: Each function should have one reason to change.
- **DRY (Don't Repeat Yourself)**: The amount validation (`amt<=0`) is duplicated for deposits and transfers in the bad version; `validate_common()` eliminates this.
- **Clean Code – Short Functions**: Functions should be comprehensible in a few minutes.
- **Open-Closed Principle**: Adding a new transaction type in the bad version requires modifying the `proc()` function. In the good version, you add a new handler function and register it in the dictionary.

9 Violation 8: Magic Strings Instead of Constants

Bad Code

```
if a['status']!='active':      # magic string
    ...
if tp=='deposit':              # magic string
    ...
```

The strings `'active'`, `'deposit'`, `'withdrawal'`, and `'transfer'` appear throughout the code as **literals**. If the status name ever changed, every occurrence would need to be found and updated.

Clean Code

```
ACTIVE_STATUS = "active"
...
if account["status"] != ACTIVE_STATUS:
```

Transaction types are handled via the `TRANSACTION_HANDLERS` dictionary, so the string literals appear only **once** in the handler registration.

Principles Violated

- **Clean Code – No Magic Numbers/Strings**: Use named constants for values that carry domain meaning.
- **DRY**: The same literal repeated in multiple places is a maintenance risk.

10 Violation 9: Comparison with `None`

Bad Code

```
if a==None:
    ...
if ta==None:
    ...
```

Clean Code

```
if account is None:
    ...
if target is None:
    ...
```

PEP 8 explicitly states: "Comparisons to singletons like `None` should always be done with `is` or `is not`, never the equality operators." The `is` operator checks **identity** (the correct test for `None`), while `==` checks **equality** and can be overridden by custom `__eq__` methods.

Principles Violated

- **PEP 8 – Programming Recommendations**: Use `is None`, not `== None`.

11 Violation 10: Missing `__main__` Guard and String Formatting

Bad Code

```
main()

print("  "+a['account_id']+" "+a['holder']+": "+str(a['balance'])
    +" "+a['currency']+" ("+a['status']+")")
```

Clean Code

```
if __name__ == "__main__":
    main()

print(
    f"  {account['account_id']}  {account['holder']}: "
    f"{account['balance']:.2f} {account['currency']}  "
    f"({account['status']})"
)
```

What is wrong:

- No `__main__` guard means importing the module triggers execution.
- String concatenation with `+` and `str()` is harder to read than f-strings.
- The bad version does not format numbers (`str(5000.0)` vs. `5000.00`).

Principles Violated

- **Clean Code – Avoid Side Effects**: Importing should not trigger execution.
- **Pythonic Code**: Use f-strings for string formatting.

12 Summary of Violations

#	**Violation**	**Principle / PEP 8 Rule**
1	Unused imports, one-line format	PEP 8 Imports, KISS
2	No docstrings, noise comments	PEP 257, Clean Code Documentation
3	Implicit data model (raw dicts)	Explicit > Implicit, PEP 484/589
4	Abbreviations, single-letter names	PEP 8 Naming, Descriptive Names
5	Semicolons, dense lines, no whitespace	PEP 8 Whitespace, Zen of Python
6	Manual file open/close	Pythonic Code, Context Managers
7	God function (38-line `proc`)	SRP, DRY, Open-Closed Principle
8	Magic strings	No Magic Numbers, DRY
9	`== None` instead of `is None`	PEP 8 Programming Recommendations
10	No `__main__` guard, string concat	Side Effects, Pythonic Code