Code Analysis: Arithmetic Expression Calculator

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 the same program — an arithmetic expression calculator that parses and evaluates strings like `"3 + 5 * 2"` without using Python's `eval()`. Both produce correct results, but the first version (`calculator_bad.py`) violates numerous PEP 8 and clean code principles, while the second (`calculator_good.py`) follows them consistently.

The analysis is structured by violation category, with side-by-side comparisons of the bad and good code and references to the specific PEP 8 rules or clean code principles that apply.

2 Violation 1: Unused and Poorly Formatted Imports

Bad Code

```
import sys,os,re;from typing import *
```

What is wrong:

- `sys`, `os`, and `re` are imported but **never used** anywhere in the code.
- Multiple imports are crammed onto **one line separated by commas**, violating PEP 8's rule that imports should be on separate lines.
- A **semicolon** joins two import statements on one line.
- `from typing import *` is a **wildcard import** that pollutes the namespace.

Clean Code

The good version has **no imports at all** — the calculator uses only built-in Python features.

Principles Violated

- **PEP 8 – Imports**: "Imports should usually be on separate lines." Wildcard imports (`from X import *`) should be avoided.
- **KISS**: Unused imports add unnecessary complexity.
- **Clean Code**: Dead code (unused imports) confuses readers about dependencies.

3 Violation 2: No Module Docstring or Documentation

Bad Code

```
# calculator program
def scicalc(s):
```

The only "documentation" is a single vague comment. No module docstring, no function docstrings.

Clean Code

```
"""Simple arithmetic expression calculator with a recursive-descent
   parser.

Supported operations: +, -, *, / and parentheses.
Does NOT use Python's eval().

Grammar:
    expression  = term (('+' | '-') term)*
    term        = factor (('*' | '/') factor)*
    factor      = NUMBER | '(' expression ')'
"""
```

The good version opens with a module docstring that explains the purpose, supported operations, and even the formal grammar. Every function also has a docstring.

Principles Violated

- **PEP 257**: All public modules, functions, classes, and methods should have docstrings.
- **Clean Code – Documentation**: Good documentation helps current and future developers understand the intent.

4 Violation 3: Poor Naming Conventions

Bad Code

```
def scicalc(s):         # What does "scicalc" mean?
def doPlusMinus(s,a,b):# camelCase, not snake_case
def doMulDiv(s,a,b):   # "do" is vague
def getNum(s, a,b):    # inconsistent spacing
    t=s[a:b]            # "t" for what?
    c=t[i]              # "c" for what?
    L=doPlusMinus(...)  # uppercase "L" for a local variable
    R=doMulDiv(...)     # uppercase "R" for a local variable
    r=doPlusMinus(...)  # "r" for result?
```

Clean Code

```
def tokenize(expression_text):
def parse_expression(tokens, position):
def parse_term(tokens, position):
def parse_factor(tokens, position):
def calculate(expression_text):
    character = expression_text[position]
    operator = tokens[position]
    right_value, position = parse_term(tokens, position)
    result, final_position = parse_expression(tokens, 0)
```

What is wrong in the bad version:

- Function names use **camelCase** (`doPlusMinus`) instead of **snake_case**.

- Variable names are **single letters** (`s`, `a`, `b`, `t`, `c`, `r`) — impossible to understand without reading every line.
- `L` and `R` use **uppercase** for local variables, which PEP 8 reserves for constants.
- Names like `scicalc` are **abbreviations** that are not pronounceable or self-explanatory.
- The list of test data is called `Data` (capitalised like a class) and results `Res`.

Principles Violated

- **PEP 8 – Naming**: Functions and variables use `lower_case_with_underscores`. Constants use `UPPER_CASE`.
- **Clean Code – Descriptive Names**: Names should reveal intent. A reader should know what a variable holds without tracing its assignment.
- **Clean Code – Pronounceable Names**: `scicalc` is not a word anyone would say in a conversation.
- **Clean Code – No Abbreviations**: `doPlusMinus` is better than `dPM`, but `parse_expression` communicates the actual operation.

5 Violation 4: Formatting and Whitespace

Bad Code

```python
def scicalc(s):
  s=s.replace(' ','')      # 2-space indent
  if s=='':return 0        # no spaces around ==
  r=doPlusMinus(s,0,len(s))
  return r

def doPlusMinus(s,a,b):
    t=s[a:b]; level=0; i=len(t)-1  # 4-space indent, semicolons
    while i>=0:                     # no space around >=
        if level==0 and(c=='*' or c=='/'): # missing space before (
            L = doMulDiv(s,a,a+i); R = getNum(s,a+i+1,b)
```

Clean Code

```python
def parse_expression(tokens, position):
    result, position = parse_term(tokens, position)

    while position < len(tokens) and tokens[position] in ("+", "-"):
        operator = tokens[position]
        position += 1
        right_value, position = parse_term(tokens, position)
```

What is wrong:

- **Inconsistent indentation**: `scicalc` uses 2 spaces, other functions use 4 spaces. PEP 8 requires 4 spaces consistently.
- **Semicolons** to put multiple statements on one line (`t=s[a:b]; level=0; i=len(t)-1`).
- **Missing whitespace** around operators: `s=s.replace`, `i>=0`, `level==0 and(c==...`.

- **No blank lines** between logical sections within functions or between function definitions. PEP 8 requires two blank lines before and after top-level functions.

- Multiple `return` or assignment statements **on the same line** as `if`: `if s==’’:return 0`.

Principles Violated

- **PEP 8 – Indentation**: Use 4 spaces per indentation level.
- **PEP 8 – Whitespace**: Surround binary operators with single spaces. Avoid compound statements on one line.
- **PEP 8 – Blank Lines**: Two blank lines around top-level definitions.
- **Zen of Python**: “Sparse is better than dense.”

6 Violation 5: Error Handling

Bad Code

```
if R==0:print("ERROR division by zero!!!") ;return 0

try:
    x = float(t)
except:
    print("bad number: "+t);x=0
return x
```

Clean Code

```
if right_value == 0:
    raise ZeroDivisionError("Division by zero")

try:
    tokens = tokenize(expression_text)
    result, final_position = parse_expression(tokens, 0)
    ...
except (ValueError, ZeroDivisionError) as error:
    return f"Error: {error}"
```

What is wrong in the bad version:

- **Bare** `except` catches every exception including `KeyboardInterrupt` and `SystemExit` — masking real bugs.

- Errors are handled by **printing and returning a dummy value** (0), which silently produces wrong results. The caller has no way to know an error occurred.

- The error message style is inconsistent: `"ERROR division by zero!!!"` vs. `"bad number: ..."`.

What the good version does:

- Errors **raise specific exceptions** (`ValueError`, `ZeroDivisionError`) at the point of detection.

- The top-level `calculate()` function catches **only expected exceptions** and returns a formatted error string.
- Errors **propagate** rather than being silently swallowed.

Principles Violated

- **PEP 8 – Exceptions**: Catch specific exceptions, never use bare `except`.
- **Zen of Python**: "Errors should never pass silently. Unless explicitly silenced."
- **Clean Code – Error Handling**: Anticipate errors and handle them gracefully. Returning magic values (0 for an error) is an anti-pattern.

7 Violation 6: Function Structure and Single Responsibility

Bad Code

The bad version has three intertwined functions (`doPlusMinus`, `doMulDiv`, `getNum`) that each take the **entire string plus two index parameters** and internally slice the string. Parsing, tokenisation, and evaluation are all mixed together.

```
def doPlusMinus(s,a,b):
    t=s[a:b]; level=0; i=len(t)-1
    while i>=0:
        ...
        L=doPlusMinus(s,a,a+i);R=doMulDiv(s,a+i+1,b)
        ...
    return doMulDiv(s,a,b)
```

Clean Code

The good version separates **tokenisation** from **parsing**:

```
tokens = tokenize(expression_text)            # Step 1: tokenise
result, position = parse_expression(tokens, 0) # Step 2: parse
```

Each parser function has a single, clear responsibility:

- `tokenize()` – converts text to tokens
- `parse_expression()` – handles `+` and `-`
- `parse_term()` – handles `*` and `/`
- `parse_factor()` – handles numbers and parentheses
- `calculate()` – orchestrates the pipeline and error handling

Principles Violated

- **SRP (Single Responsibility Principle)**: Each function should do one thing.
- **SoC (Separation of Concerns)**: Tokenisation and parsing are different concerns.
- **Clean Code – Short Functions**: If a function takes more than a few minutes to comprehend, it should be refactored.

8 Violation 7: Missing `__main__` Guard

Bad Code

```
main()
```

The bad version calls `main()` at the module level. If another script imports this file, the calculator runs immediately as a side effect.

Clean Code

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

The good version uses the standard `__main__` guard, so the module can be safely imported without executing the calculator.

Principles Violated

- **Clean Code – Avoid Side Effects**: Importing a module should not trigger execution.
- **Python Best Practice**: The `if __name__ == "__main__"` guard is standard for all runnable scripts.

9 Violation 8: String Concatenation Instead of f-Strings

Bad Code

```
print(d+" = "+str(Res))
```

Clean Code

```
print(f"{display_expr} = {result}")
```

String concatenation with `+` and manual `str()` calls is harder to read than f-strings, which are the idiomatic Python 3.6+ way to format output.

Principles Violated

- **Pythonic Code**: Use f-strings for string formatting (readable, efficient).
- **Clean Code – Readability**: f-strings make the output format immediately visible.

10 Summary of Violations

#	Violation	Principle / PEP 8 Rule
1	Unused imports, wildcard import, one-line imports	PEP 8 Imports, KISS
2	No docstrings or documentation	PEP 257, Clean Code Documentation
3	camelCase names, single-letter variables, abbreviations	PEP 8 Naming, Descriptive Names
4	Inconsistent indent, semicolons, missing whitespace	PEP 8 Indentation & Whitespace
5	Bare except, silent error swallowing	PEP 8 Exceptions, Zen of Python
6	Mixed concerns, long tangled functions	SRP, SoC, Short Functions
7	No `__main__` guard	Avoid Side Effects
8	String concatenation instead of f-strings	Pythonic Code, Readability