CS1010X: Programming Methodology I
Chapter 5: Debugging

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 5: Debugging

Learning Objectives

By the end of this chapter, **students will be able to**:

- **Explain** what bugs are, and why debugging is important.
- **Identify** common "low-class" programming errors (syntax, name, type, argument count, arithmetic, control flow, floating-point, logic).
- **Apply** a systematic debugging workflow to reproduce, localise, and fix a bug in a small Python program.
- **Use** debugging tools (print statements, debugger, Python Tutor) to inspect control flow and variable values.
- **Diagnose** bugs caused by numerical imprecision and fix them using a) small epsilon values, b) working directly with integers where possible.
- **Design** simple tests and "extra" helper functions that check correctness of individual functions (unit tests).
- **Adopt** good programming practices (planning, incremental development, clear interfaces) that reduce the amount of debugging needed.

5.1 Why Debugging Matters

Humans make mistakes. Programming is no exception: even experienced developers constantly introduce bugs in their code. Therefore, we should

- expect bugs to happen,
- have a systematic way to find and fix them, and
- structure our programs to make debugging easier.

In practice, debugging often takes **more time** than typing out the first version of the code. It is therefore worth *investing* in techniques that make debugging less painful and more effective.

5.2 Programming Stages

A healthy development cycle looks roughly like this:

1. **Environment:** Set up your programming environment so that experimentation is easy:
   - a text editor / IDE you are comfortable with,
   - a way to run code quickly (REPL, notebook, terminal),
   - (optionally) easy access to debugger / visualiser.
2. **Experiment:** Before writing the full solution, play with small pieces: test built-in functions, check edge cases, confirm language features behave as you expect.
3. **Plan:** Use *functional abstraction* and *incremental development*:
   - break the problem into small functions, each doing 1 clear task (wishful thinking),
   - choose appropriate data structures for each task.
4. **Code & Debug:** Implement one small function at a time. For each function:
   - write it,
   - test it on simple and boundary cases,
   - debug until it works,
   - only then use it as a building block in a larger function.

   This is **incremental development**.

A small up-front cost here pays off by dramatically reducing debugging time later.

Extra: Continuous Integration/Continuous Delivery/Deployment (CI/CD)

In large software projects, debugging and testing are continuously done. Teams *continuously* check that the codebase still works whenever new changes are added. This is usually done through a process known as CI/CD.

Continuous Integration (CI)

- Developers commit small changes to a shared repository frequently.
- Every commit automatically triggers:
  - compilation / linting,
  - unit tests and integration tests.
- If any test fails, the change is rejected or flagged immediately.

Continuous Delivery/Deployment (CD)

- Once all automated checks pass, the new version can be:
  - **delivered** to a staging environment for humans to test further, or
  - **deployed** automatically to production.

Why this matters for you (even in CS1010X):

- Think of "running your tests after every small change" as a mini CI system.
- Writing simple test functions and running them regularly is like an early, personal version of CI/CD.
- The more often you integrate and test, the earlier you catch bugs, and the easier they are to debug.

5.3 What Is Debugging?

- **Bug:** Any behaviour of the program that does not match your *intended specification*.
- **Debugging:** The process of locating, understanding, and fixing bugs.

After debugging, the program is usually *less buggy*, but not necessarily bug-free; the remaining bugs are simply harder to find.

5.3.1 Model Mismatch

The computer always does exactly what your program tells it to do, not what you *meant* it to do. Debugging is about:

1. discovering what the program *actually* does,
2. comparing that with what you *intended*,
3. changing the code (or the specification!) so that they match.

5.3.2 Think Like a Detective

Effective debugging is like detective work:

- collect clues: error messages, unusual outputs, variable values,
- narrow down suspects: eliminate code that cannot be responsible,
- construct and test hypotheses about what might be wrong.

A typical debugging loop:

1. **Reproduce** the bug reliably (same input $\Rightarrow$ same failure).
2. **Localise** the bug: shrink the suspicious region of code.
3. **Inspect** control flow and variable values in that region.
4. **Fix** the cause, not just the symptom.
5. **Retest** on original and new cases to avoid reintroducing the bug.

Non-Reproducible Bugs

If you cannot reproduce the bug, you cannot reliably tell whether it is fixed. Try to:

- remove randomness (fix random seeds, use fixed inputs),
- log inputs that caused previous failures,
- construct small artificial inputs that still trigger the bug.

Looking Forward: Race Conditions

Some of the most frustrating *non-reproducible* bugs arise from **race conditions**. These occur in concurrent programs (multiple threads / processes) when the result depends on the timing or ordering of operations.

- Two pieces of code access the same data *at the same time*.
- At least one of them *writes* to that data.
- The program is correct only for some interleavings of their steps.

5.4 Low-Class Errors

We start with "low-class" errors: simple mistakes that occur frequently and are usually easy to fix once recognised. Getting good at spotting these quickly frees your brain for harder logic bugs later.

5.4.1 Syntax Errors

Python cannot even *parse* your program.

```
def proc(x)   # forgot :
    do_stuff()

'''
File "<stdin>", line 1
def proc(x)
            ^
SyntaxError: invalid syntax
'''
```

Other common syntax mistakes:

- mismatched indentation,
- missing parentheses,
- stray or extra commas, colons.

Debugging strategy:

- Read the error message carefully; it often points near the problem.
- Check the line *before* the reported one; sometimes the issue is there.
- Use an editor that highlights syntax errors as you type.

5.4.2 Name and Scope Errors

Using variables before defining them, or in the wrong scope.

```
x = 2
x + k   # k was never defined

'''
NameError: name 'k' is not defined
'''
```

Debugging strategy:

- Check for typos in variable names.
- Ensure the variable is assigned *before* it is used.
- Avoid global variables; pass needed values as parameters.

5.4.3 Type and Argument Errors

Passing arguments of the wrong type or wrong number of arguments.

```
def square(x):
    return x * x

x = 5
x + square   # forgot parentheses

'''
TypeError: unsupported operand type(s)
'''

square(3, 5) # too many arguments
'''
TypeError: square() takes 1 positional argument but 2 were given
'''
```

Debugging strategy:

- Check how the function is defined and how it is called.
- Print `type(x)` to see actual types.
- Use meaningful parameter names and docstrings.

5.4.4 Arithmetic Errors

Division by zero or similar numeric errors.

```
x = 3
y = 0
x / y

'''
ZeroDivisionError: division by zero
'''
```

Debugging strategy:

- Check denominators and ranges before division.
- Decide what should happen when the denominator is zero: error, default value, or skip?

5.4.5 Control-Flow Errors: Missing Returns and Infinite Loops

```
def square(x):
    x * x       # missing return

result = square(3)
print(result)

'''
None
'''
```

Python returns `None` by default if no `return` statement is executed.

Debugging strategy: Ensure each branch (especially in `if`/`elif`/`else`) returns a value when a function is supposed to produce one.

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

fact(2.1)   # never reaches 0
fact(-1)    # recurses forever
```

```
def fact_iter(n):
    counter, result = n, 1
    while counter != 0:
        counter, result = counter - 1, result * counter
    return result

fact_iter(-1)  # while condition never becomes False
```

Debugging strategy:

- Check that the loop's state changes in a way that will eventually make the condition false.
- Print the loop variables for a few iterations to confirm they move in the right direction.
- For recursive functions, ensure that each recursive call moves closer to a base case.

5.4.6 Numerical Imprecision

Consider:

```
def foo(n):
    counter, result = 0.0, 0.0
    while counter != n:
        counter, result = counter + 0.1, result + counter
    return result

foo(5)
```

Intuitively, this seems safe: start from 0.0, repeatedly add 0.1 until we reach 5. Yet in practice, this loop may *never terminate*. Why?

IEEE 754 Floating-Point Representation

Most modern languages (including Python) store `float` values using the IEEE 754 binary64 format:

- 1 sign bit (positive/negative),
- 11 exponent bits,
- 52 fraction (mantissa) bits.

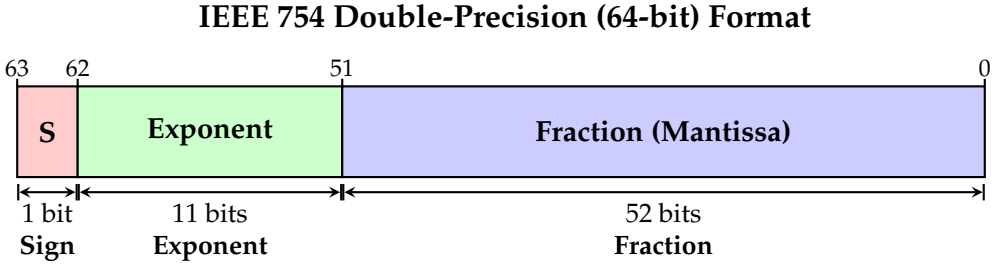

Figure 5.1: Visual representation of the IEEE 754 binary64 floating-point standard.

This representation can only exactly store fractions whose denominator is a power of 2:

$$\frac{k}{2^m}.$$

Numbers like $0.5 = 1/2$ or $0.25 = 1/4$ are exact, but $0.1 = 1/10$ is not; its binary expansion is infinite, so it gets *rounded* to the nearest representable value. As a result:

$$0.1 + 0.1 + 0.1 + 0.1 + 0.1 \neq 0.5 \quad \text{exactly},$$

even though it *prints* as `0.5` in many environments.

Inside the loop above, `counter` takes values like:

$$0.1,\ 0.2,\ 0.30000000000000004,\ \ldots$$

and may never become *exactly* 5.0, so the condition `counter != n` stays true forever.

Debugging Numerical Imprecision

When a program with floats behaves strangely:

1. **Print with high precision.** Use `repr(counter)` or formatted printing (`print(f"{counter:.17f}")`) to see the true stored value.
2. **Avoid equality comparisons with floats.** Replace `while counter != n` with an epsilon:

```
while counter < n - 1e-9:
    ...
```

   or, use an integer loop:

```
for i in range(int(10*n)):
    counter = i / 10.0
```

3. **Use alternative representations when possible:** Try to work in integers (cents instead of dollars) where possible.

5.4.7 Logic Errors and Misunderstood Syntax

Sometimes the code runs without any exceptions but produces the wrong answer.

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-1)  # BUG: should be fib(n-2)
```

Or misunderstanding an operator:

```
k = 10 / 4   # In Python 3, this is 2.5, not 2
```

Debugging strategy:

- Test on tiny inputs where you know the correct answer (e.g. $fib(0) \ldots fib(5)$).
- Draw diagrams (recursion trees, variable traces).
- Read the language documentation for confusing operators.

5.5 Debugging Tools

Different tools help you inspect what your code is doing.

5.5.1 Print Statements

The simplest (and still very effective) tool is to insert print statements to show variable values or trace function calls.

```
debug_printing_flag = True

def debug_print(x, y, z):
    if debug_printing_flag:
        print(x)
        print(y)
        print(z)

def fib(n):
    debug_print(n, None, None)
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)

debug_printing_flag = False
```

Here we wrap all debugging output in a helper function controlled by a global flag, so we can easily turn debugging on and off without deleting code.

5.5.2 Interactive Debuggers

Most IDEs (and even the Python standard library via `pdb`) provide an interactive debugger:

- **Breakpoints**: pause execution at a chosen line.
- **Step** through code line-by-line.
- **Inspect** variable values in the current frame.

These tools make it easier to visualise both the *flow of execution* and the *evolution of values*.

5.5.3 Visualisers (e.g. Python Tutor)

Tools like `Python Tutor` show a snapshot of the entire program state: stack frames, variables, lists, and so on at each step. This is particularly helpful for:

- recursion,
- aliasing and mutations on lists,
- tracing complex control flow.

5.6 Good Debugging Practices

5.6.1 Use Tools Effectively

- Use a debugger or visualiser to:
  - trace execution from the point of failure *backwards*,
  - display variable values at key points,
  - pay special attention to boundary cases (empty lists, zero, negative inputs).
- Use print-debugging for quick checks, but remember to remove or disable it afterwards.

5.6.2 Write "Extra" Functions

Sometimes it pays to write helper functions whose only purpose is to support debugging or testing:

- a function that checks whether a data structure satisfies certain invariants,
- a function that prints a complex object in a readable format,
- small unit-test functions that call your main function on several inputs and verify expected outputs.

5.6.3 Test Incrementally

- Test each function in isolation before combining them.
- Include both:
  - **typical** cases, and
  - **boundary** cases (empty input, maximum sizes, negative values, zero).

5.7 Avoiding Unnecessary Debugging

The best bug is the one that never gets written in the first place. Good programming practices can drastically reduce the number of bugs.

5.7.1 Readable Code

- Use meaningful variable and function names.
- Write clear comments where the intent is not obvious.
- State assumptions (preconditions) explicitly in docstrings.

5.7.2 Single-Responsibility Functions

Each function should do *one* coherent task. This makes:

- bugs easier to localise,
- unit tests shorter and more focused,
- reuse more likely.

Looking Forward: SOLID Principles and Maintainability

In larger projects, the main enemy is not just *bugs*, but code that is hard to understand and change. In Object-Oriented Programming, a common guideline for maintainable code is the **SOLID** principle. Although SOLID is framed for classes and objects, the ideas apply equally well to our functions and modules:

- **S: Single Responsibility**
  Each module / class / function should have *one reason to change*. This is precisely the rule above: fewer responsibilities $\Rightarrow$ fewer places where bugs can hide.
- **O: Open / Closed**
  Code should be *open for extension, closed for modification*: you add new behaviour by writing new code, not by constantly editing old code. Well-chosen function boundaries make it easier to plug in new cases without breaking existing ones.
- **L: Liskov Substitution Principle**
  If a piece of code expects an object of type T, it should also work correctly with any "subtype" of T. Violations of this principle often show up as subtle bugs when we try to reuse or extend code.
- **I: Interface Segregation**
  Many small, focused interfaces are better than one giant, "do-everything" interface. In our setting: multiple small helper functions are easier to test and reason about than one huge all-in-one function.
- **D: Dependency Inversion**
  High-level logic should not depend directly on low-level details. Instead, both should depend on *abstractions* (well-defined functions, APIs). This makes it easier to change low-level implementation without rewriting the whole program.

The more your code follows these ideas (even informally), the less surprising it becomes, and the less time you will spend in the debugger.

You will explore these principles in more detail in CS2030/S and CS2101/CS2103T.

5.7.3 Discipline Pays Off

Good discipline in programming (planning, abstraction, testing) reduces the debugging effort required later. Conversely, sloppy habits increase debugging time dramatically.

In CS1010X, we encourage you to:

- plan your solutions before coding,
- write small, testable functions,
- treat debugging as a normal, essential part of programming, not as a punishment.