

# Debugging and Troubleshooting

---

## Finding and Fixing Software Defects

Mehdi Lotfinejad

## Learning Objectives

By the end of this lesson, you will be able to:

1. **Define** debugging and explain its critical role in the software development lifecycle
2. **Identify and use** essential debugging tools including debuggers, breakpoints, and logging mechanisms
3. **Apply** systematic debugging techniques to isolate and identify the root cause of software defects
4. **Distinguish** between static and dynamic defect identification approaches and when to use each
5. **Execute** a step-by-step debugging process to fix common software errors in your code

# The Reality of Debugging

## Scenarios you'll face:

- Program calculates student grades but crashes with certain inputs
- Web app works perfectly locally but fails in production
- Code that worked yesterday mysteriously breaks today

**Time Investment:** Developers spend 35-50% of their time debugging and testing code—one of the most time-consuming activities in software development.

**The truth:** Every program you write will have bugs. This isn't failure—it's normal software development.

# What Is Debugging?

"Debugging is the systematic process of identifying, isolating, and fixing errors (called 'bugs' or 'defects') in computer software." — Wikipedia

## More than just "fixing broken code":

- Structured problem-solving process
- Running code step by step
- Examining how data changes
- Identifying exactly where logic went wrong

**Think of it as:** Being a medical doctor for your software—observe symptoms, run diagnostics, identify the disease, prescribe treatment.

## A Brief History: The First Bug

**1947:** Computer pioneer Grace Hopper found an actual moth causing problems in the Harvard Mark II computer.

She taped it in her logbook with the note:

"First actual case of bug being found."

**Today:** Bugs are logical errors rather than insects, but they're just as troublesome!

The term "debugging" has been part of computing since the very beginning.

# Why Debugging Matters

## Quality and Reliability:

- Bugs cause crashes, incorrect results, unpredictable behavior
- Banking apps miscalculating interest = serious consequences
- Medical devices with wrong readings = life-threatening

## User Experience:

- Nothing frustrates users more than software that doesn't work
- Every bug you fix improves user experience and builds trust

## Cost Efficiency:

- Bug found during development: 30 minutes to fix
- Same bug found in production: thousands of dollars, emergency patches, customer support

# Why Debugging Matters (continued)

## Learning and Growth:

- Debugging teaches you how software really works
- Stepping through code line by line deepens understanding
- Every bug you fix makes you a better developer

**Key Insight:** Effective debugging isn't just about fixing errors—it's about understanding why they occurred and preventing similar issues in the future.

# The Debugging Mindset: Think Like a Detective

Essential attitudes for effective debugging:

## 1. Assume Nothing

- Your code does exactly what you told it to do
- Bug exists because of mismatch between intentions and instructions
- Don't assume a function works correctly just because it "should"

## 2. Reproduce Consistently

- Make the bug happen reliably before trying to fix it
- Identify conditions that trigger it
- If bug only appears "sometimes," find the pattern



# The Debugging Mindset (continued)

## 3. Isolate the Problem

- Large programs have thousands of lines
- Narrow down where the problem occurs
- Input validation? Calculation logic? Output formatting?

## 4. Form Hypotheses

- Based on symptoms, what could be causing the problem?
- Variable not initialized? Loop runs one too many times?
- Form theories, then test them systematically

## 5. Document Your Findings

- Keep notes about what you've tried and learned
- Prevents testing the same thing twice
- Helps you see patterns

# Debugging Mindset in Action

## Example problem:

```
# Program to calculate average of student scores
def calculate_average(scores):
    total = 0
    for score in scores:
        total += score
    average = total / len(scores) # BUG: What if scores is empty?
    return average

# Test the function
student_scores = [] # Empty list - edge case!
result = calculate_average(student_scores)
print(f"Average score: {result}")

# This will crash with: ZeroDivisionError: division by zero
```

# Systematic Debugging Approach

A beginner might panic. But with the debugging mindset:

1. **Observe the symptom:** Program crashes with `ZeroDivisionError`
2. **Form hypothesis:** Division by zero means we're dividing by `len(scores)`, which must be zero
3. **Test hypothesis:** Check what happens when `scores` is empty
4. **Identify root cause:** No validation for empty input
5. **Fix the bug:** Add input validation

This systematic approach—observe, hypothesize, test, fix—is the foundation of effective debugging.

# The Fixed Version

```
# Fixed version with proper error handling
def calculate_average(scores):
    # Validate input first - handle edge case
    if not scores: # Check if list is empty
        return 0 # Or raise an exception, depending on requirements

    total = 0
    for score in scores:
        total += score
    average = total / len(scores)
    return average

# Now it handles the edge case gracefully
student_scores = []
result = calculate_average(student_scores)
print(f"Average score: {result}") # Prints: Average score: 0
```

## Core Debugging Tools

### Your Problem-Solving Toolkit:

1. **Debuggers** → Control program execution, examine state
2. **Breakpoints** → Pause execution at specific points
3. **Logging** → Create permanent records of program behavior
4. **Print Statements** → Quick and simple diagnostic output
5. **Stack Traces** → Understand function call sequences

Mastering these tools transforms debugging from frustration into efficient problem-solving.

# Understanding Debuggers

A debugger is a specialized tool that lets you control how your program runs.

**Without Debugger:**

- Executes start to finish in milliseconds
- Can't see intermediate states
- Black box behavior

**With Debugger:**

- Pause execution anywhere
- Examine variable values
- Step through code line by line
- "Slow-motion replay"

**Available in:** Visual Studio, PyCharm, VS Code, Eclipse, and more

# Core Debugger Capabilities

## Breakpoints:

- Markers where execution should pause
- Examine program state at that exact moment

## Step Execution:

- **Step Over:** Execute current line, move to next
- **Step Into:** Enter function calls to see inside them
- **Step Out:** Finish current function, return to caller

## Variable Inspection:

- See what's stored in variables
- Examine arrays, lists, object properties
- Understand what code is actually doing

# Core Debugger Capabilities (continued)

## Call Stack:

- Shows sequence of function calls leading to current point
- If A called B, which called C, see the full chain
- Invaluable for understanding how you got somewhere

## Watch Expressions:

- Monitor specific variables continuously
- Values update automatically as you step through
- Track how data changes over time



## Python Debugger Example (pdb)

```
# Example: Finding a bug in shopping cart calculation
def calculate_total(items):
    """Calculate total price with 10% discount for orders over $100"""
    subtotal = 0

    # Calculate subtotal
    for item in items:
        subtotal += item['price'] * item['quantity']

    # Apply discount if applicable
    if subtotal > 100:
        discount = subtotal * 0.10 # 10% discount
        total = subtotal - discount

    return total # BUG: What if subtotal <= 100?
```

# Using the Debugger

```
import pdb

def calculate_total_debug(items):
    """Calculate total price with 10% discount for orders over $100"""
    subtotal = 0

    for item in items:
        subtotal += item['price'] * item['quantity']

    pdb.set_trace() # Execution will pause here

    # When paused, you can:
    # - Type 'subtotal' to see its value
    # - Type 'n' (next) to execute the next line
    # - Type 'c' (continue) to resume execution
    # - Type 'p variable_name' to print any variable

    if subtotal > 100:
        discount = subtotal * 0.10
        total = subtotal - discount

    return total # You'll discover 'total' is undefined if subtotal <= 100!
```

# The Fix

```
def calculate_total_fixed(items):  
    """Calculate total price with 10% discount for orders over $100"""  
    subtotal = 0  
  
    # Calculate subtotal  
    for item in items:  
        subtotal += item['price'] * item['quantity']  
  
    # Apply discount if applicable, otherwise use subtotal as total  
    if subtotal > 100:  
        discount = subtotal * 0.10  
        total = subtotal - discount  
    else:  
        total = subtotal # FIX: Handle case when no discount applies  
  
    return total
```

The debugger helped us see exactly where the logic fails!



## Logging: Your Program's Diary

Why logging complements debugging:

- **Debuggers:** Require active control, only work while you're running program
- **Logging:** Creates permanent record you can review later

**Think of it as:** Your program keeping a diary of what it does

"I started processing user input," "I calculated result as 42," "I encountered database error"

# When Logging Is Essential

## Production Environments:

- Can't attach debugger to customer's computer or remote server
- Logs provide visibility into what happened

## Intermittent Bugs:

- Some bugs appear only occasionally
- Logs capture information every time, helping spot patterns

## Performance Analysis:

- Timestamps show how long operations take
- Identify bottlenecks

## Audit Trails:

- Security and compliance requirements

- Record of who did what and when

# The Five Logging Levels

**DEBUG:** Detailed diagnostic information, useful during development

**INFO:** General informational messages about program flow

**WARNING:** Something unexpected happened, but program continues

**ERROR:** A serious problem occurred, some functionality failed

**CRITICAL:** Very serious error that might cause program to crash

Using appropriate levels helps you filter information based on importance.

# Logging Configuration Example

```
import logging

# Configure logging (do this once at program start)
logging.basicConfig(
    level=logging.DEBUG, # Show all messages
    format='%(asctime)s - %(levelname)s - %(message)s',
    filename='app.log' # Write to file instead of console
)

def process_order(order_id, items):
    """Process a customer order with comprehensive logging"""

    # INFO: General information about program flow
    logging.info(f"Starting to process order {order_id}")
    logging.info(f"Order contains {len(items)} items")
```

# Logging in Action

```
try:
    # Calculate total
    total = 0
    for item in items:
        # DEBUG: Detailed information useful during development
        logging.debug(f"Processing item: {item['name']}, price: {item['price']}")
        total += item['price'] * item['quantity']

    logging.info(f"Order {order_id} total calculated: ${total:.2f}")

    # Validate total
    if total <= 0:
        # WARNING: Unusual but not necessarily an error
        logging.warning(f"Order {order_id} has zero or negative total")
        return False
```



# Logging Error Handling

```
# Process payment (simulated)
if total > 10000:
    # ERROR: Something went wrong that needs attention
    logging.error(f"Order {order_id} exceeds maximum: ${total:.2f}")
    return False

logging.info(f"Order {order_id} processed successfully")
return True

except KeyError as e:
    # CRITICAL: Serious error that might cause crash
    logging.critical(f"Missing required field in order {order_id}: {e}")
    return False
```

# Print Statements: Simple but Effective

Before sophisticated debuggers, developers used print statements—and they're still useful today!

```
def find_maximum(numbers):  
    """Find the maximum value in a list"""  
  
    # Debug: Check what we received  
    print(f"DEBUG: Input list: {numbers}")  
    print(f"DEBUG: List length: {len(numbers)}")  
  
    if not numbers:  
        print("DEBUG: List is empty, returning None")  
        return None  
  
    max_value = numbers[0]  
    print(f"DEBUG: Starting with max_value = {max_value}")
```

## Print Debugging Output

```
for i, num in enumerate(numbers[1:], start=1):
    print(f"DEBUG: Iteration {i}, comparing {num} with current max {max_value}")
    if num > max_value:
        max_value = num
        print(f"DEBUG: New maximum found: {max_value}")





print(f"DEBUG: Final maximum: {max_value}")
return max_value

# Test with debug output
test_numbers = [3, 7, 2, 9, 1, 5]
result = find_maximum(test_numbers)
print(f"\nResult: {result}")
```





**Output shows exactly how the algorithm works step by step!**

# Print Debugging: Pros and Cons

## Advantages:

-  Quick and simple
-  No special tools required
-  Works everywhere
-  Easy to understand

## Disadvantages:

-  Clutters code
-  Requires manual cleanup
-  Can't examine state interactively
-  Not suitable for production

**Remember:** Remove or comment out debug prints before deploying to production. Use proper logging instead.

# Static vs. Dynamic Defect Identification

Two fundamentally different approaches to finding bugs:

## Static Analysis:

- Examining code without running it
- Like proofreading an essay before submission
- Catches obvious mistakes early and cheaply

## Dynamic Analysis:

- Examining code while it runs
- Catches bugs that only appear during execution
- Reveals runtime behavior and logic errors

The most effective strategy uses both approaches together.

# Static Defect Identification Techniques

## Code Reviews:

- Another developer reads your code
- Fresh eyes spot issues you missed
- Catches design problems early

## Linting Tools:

- Automated code analysis
- Checks style violations, potential bugs, suspicious patterns
- Examples: pylint (Python), ESLint (JavaScript), RuboCop (Ruby)

## Type Checking:

- Verify variables used correctly
- Python's mypy catches type errors before running code
- Prevents entire classes of bugs

# Static Analysis Example

```
# This code has a bug that static analysis can find
def calculate_discount(price, discount_percent):
    """Apply a percentage discount to a price"""
    # Linter warning: discount_percent is never used!
    discount_amount = price * 0.10 # BUG: Hardcoded 10%
    final_price = price - discount_amount
    return final_price

# A linter would flag: "Parameter 'discount_percent' is never used"
# This alerts you to the bug before running the code

# Corrected version:
def calculate_discount_fixed(price, discount_percent):
    """Apply a percentage discount to a price"""
    discount_amount = price * (discount_percent / 100)
    final_price = price - discount_amount
    return final_price
```

# Dynamic Defect Identification Techniques

## Testing:

- Run program with various inputs
- Check if outputs are correct
- Unit tests, integration tests, system tests

## Debugging:

- Use debugger to step through execution
- Examine runtime state

## Profiling:

- Measure performance
- Identify slow operations or memory leaks

## Monitoring:

- Observe program in production



# Dynamic Analysis Example

```
def divide_numbers(a, b):  
    """Divide two numbers"""  
    return a / b # Static analysis can't predict if b will be zero  
  
# Static analysis sees nothing wrong with this code  
# But dynamic testing reveals the problem:  
  
# Test case 1: Normal operation  
result1 = divide_numbers(10, 2) # Works fine: 5.0  
  
# Test case 2: Edge case  
result2 = divide_numbers(10, 0) # Crashes: ZeroDivisionError!  
  
# Dynamic testing found the bug. Now we fix it:  
def divide_numbers_safe(a, b):  
    """Divide two numbers with error handling"""  
    if b == 0:  
        raise ValueError("Cannot divide by zero")  
    return a / b
```

## Static vs. Dynamic Comparison

Aspect	Static Analysis	Dynamic Analysis
When	Before running	During execution
Speed	Very fast	Depends on tests
Scope	Code structure	Runtime behavior
Catches	Syntax, style, types	Logic, runtime errors
Tools	Linters, type checkers	Debuggers, tests
Cost	Low	Medium to high

**Best Practice:** Use static analysis first to catch obvious mistakes, then dynamic analysis for runtime issues.

# The Systematic Debugging Process

Professional developers follow a structured 6-step process:

1. Reproduce the Bug Reliably
2. Isolate the Problem Area
3. Understand the Root Cause
4. Form and Test Hypotheses
5. Verify the Fix
6. Document and Learn

This systematic approach is more efficient than randomly trying fixes!

# Step 1: Reproduce the Bug Reliably

Before you can fix it, make it happen consistently.

```
# Bug report: "The program sometimes crashes when processing user input"
# This is too vague. We need to reproduce it reliably.

def process_user_input(user_text):
    """Process text input from user"""
    words = user_text.split()
    first_word = words[0] # BUG: What if user_text is empty?
    return first_word.upper()

# Test to reproduce:
result1 = process_user_input("hello world") # Works: "HELLO"
result2 = process_user_input("") # Crashes: IndexError!

# Now we can reproduce reliably: happens with empty input
```

## Step 2: Isolate the Problem Area

Narrow down where the bug occurs using checkpoints:

```
def complex_calculation(data):  
    """Multi-step calculation that's failing somewhere"""  
  
    print("CHECKPOINT 1: Starting calculation")  
    step1_result = process_step_one(data)  
  
    print("CHECKPOINT 2: Step 1 complete")  
    step2_result = process_step_two(step1_result)  
  
    print("CHECKPOINT 3: Step 2 complete") # Never reaches here  
    step3_result = process_step_three(step2_result)  
  
    return step3_result  
  
# Conclusion: Bug is in process_step_two()
```

## Step 3: Understand the Root Cause

Don't just fix symptoms—understand why the bug exists:

```
def process_step_two(data):  
    """The function where our bug occurs"""  
    # Use debugger or print statements  
    print(f"DEBUG: Received data: {data}")  
    print(f"DEBUG: Data type: {type(data)}")  
  
    # Assumption: data is always a list  
    # Reality: Sometimes it's None!  
    result = []  
    for item in data: # Crashes if data is None  
        result.append(item * 2)  
  
    return result  
  
# Root cause: process_step_one() can return None  
# We assumed it always returns a list
```

## Step 4: Form and Test Hypotheses

```
# Hypothesis: Adding a None check will fix the crash

def process_step_two_fixed(data):
    """Fixed version with proper validation"""
    print(f"DEBUG: Received data: {data}")

    # Test hypothesis: Check for None
    if data is None:
        print("DEBUG: Data is None, returning empty list")
        return [] # Handle the None case

    result = []
    for item in data:
        result.append(item * 2)

    return result

# Test the fix:
test_result = process_step_two_fixed(None) # No crash!
print(f"Result: {test_result}") # Prints: Result: []
```

## Step 5: Verify the Fix

Test thoroughly to ensure:

- Original problem is solved
- No new bugs introduced
- Edge cases are handled
- Fix works in different scenarios

```
def test_process_step_two_fixed():  
    """Test all scenarios"""  
  
    # Test 1: Normal case  
    assert process_step_two_fixed([1, 2, 3]) == [2, 4, 6]  
    print("✓ Test 1 passed: Normal input")  
  
    # Test 2: None input (the bug we fixed)  
    assert process_step_two_fixed(None) == []  
    print("✓ Test 2 passed: None input")
```



## Step 5: Verify the Fix (continued)

```
# Test 3: Empty list
assert process_step_two_fixed([]) == []
print("✓ Test 3 passed: Empty list")

# Test 4: Single item
assert process_step_two_fixed([5]) == [10]
print("✓ Test 4 passed: Single item")

print("\n✓ All tests passed! Fix is verified.")

test_process_step_two_fixed()
```

Comprehensive testing ensures the fix works and doesn't break anything else.

# Step 6: Document and Learn

Write down what caused the bug and how you fixed it:

```
def process_step_two_final(data):  
    """  
    Process data by doubling each element.  
  
    Args:  
        data: List of numbers to process, or None  
  
    Returns:  
        List of doubled numbers, or empty list if input is None  
  
    Bug History:  
        - 2024-01-15: Fixed crash when data is None. Root cause was  
          assumption that process_step_one() always returns a list,  
          but it can return None when input validation fails.  
          Added None check to handle this case gracefully.  
    """  
    if data is None:  
        return []  
  
    result = []  
    for item in data:  
        result.append(item * 2)  
  
    return result
```

# Using Breakpoints Effectively

Strategic placement matters:

```
def calculate_grade(scores):  
    """Calculate letter grade from numeric scores"""  
  
    # Breakpoint 1: Check input  
    if not scores: # <-- Place breakpoint here  
        return "No scores"  
  
    # Breakpoint 2: Before calculation  
    average = sum(scores) / len(scores) # <-- Place breakpoint here  
  
    # Breakpoint 3: At decision points  
    if average >= 90: # <-- Place breakpoint here  
        grade = "A"  
    # ... more conditions ...  
  
    # Breakpoint 4: Before return  
    return grade # <-- Place breakpoint here
```

# Conditional Breakpoints

Only pause when specific condition is true:

```
# Example: Only pause when processing a specific user
def process_transaction(user_id, amount):
    """Process a financial transaction"""

    # Regular breakpoint: Pauses for EVERY transaction
    # Conditional breakpoint: Only pause when user_id == 12345
    # This is useful when debugging issues for specific users

    if amount < 0:
        raise ValueError("Amount cannot be negative")

    # Process the transaction...
    return True

# In most debuggers, you can set condition:
# Breakpoint condition: user_id == 12345
# Now it only pauses for that specific user
```

# Watch Expressions

Monitor how specific variables change:

```
def bubble_sort(numbers):  
    """Sort numbers using bubble sort algorithm"""  
    n = len(numbers)  
  
    # Watch these expressions while debugging:  
    # - numbers (see how the list changes)  
    # - i (current outer loop iteration)  
    # - j (current inner loop iteration)  
    # - numbers[j] and numbers[j+1] (values being compared)  
  
    for i in range(n):  
        for j in range(n - i - 1):  
            if numbers[j] > numbers[j + 1]:  
                numbers[j], numbers[j + 1] = numbers[j + 1], numbers[j]  
  
    return numbers
```

# Common Pitfall 1: Changing Multiple Things

## **The Problem:**

When frustrated, changing several things simultaneously hoping one will work.

## **Why it's dangerous:**

- Don't know which change fixed the bug
- Might accidentally introduce new bugs
- Lose control and understanding

## **Best Practice:**

Change one thing at a time, test, observe result. If it doesn't work, undo before trying something else.

This systematic approach may feel slower, but it's actually much faster!

## Wrong vs. Right Approach

```
# ❌ Wrong approach: Changing multiple things
def calculate_price(quantity, unit_price):
    # Changed validation, calculation, AND return type all at once
    if quantity > 0 and unit_price > 0: # Added validation
        total = quantity * unit_price * 1.1 # Added 10% markup
        return round(total, 2) # Changed to round result
    return None # Changed to return None instead of 0

# ✅ Right approach: Change one thing at a time
# Step 1: Add validation only
# Step 2: Test
# Step 3: Add markup calculation
# Step 4: Test
# Step 5: Add rounding
# Step 6: Test
```

## Common Pitfall 2: Assuming Bug Location

### **The Problem:**

Fixating on one area of code, spending hours debugging it, when the actual bug is somewhere completely different.

### **Why it happens:**

Your assumptions about where the problem exists blind you to the real issue.

### **Best Practice:**

Follow the data. Use print statements or debugger to trace actual flow. Let evidence guide you, not assumptions.

The bug is rarely where you first think it is!



## Common Pitfall 3: Ignoring Error Messages

### The Problem:

Error messages tell you exactly what went wrong and where, yet many beginners panic and ignore them.

### Error message example:

```
# Traceback (most recent call last):  
#   File "cart.py", line 45, in calculate_total  
#     tax = subtotal * self.tax_rate  
# TypeError: unsupported operand type(s) for *: 'str' and 'float'  
  
# This tells you:  
# - WHERE: Line 45 in calculate_total  
# - WHAT: Trying to multiply a string by a float  
# - WHY: subtotal is a string when it should be a number
```

**Best Practice:** Read the entire error message including stack trace. Google it if you don't understand!

## Common Pitfall 4: No Hypothesis

### **The Problem:**

Randomly trying fixes without understanding the problem wastes time and often makes things worse.

### **Best Practice:**

Before changing code, write down your hypothesis: "I think the bug is caused by X because Y." Then test that specific hypothesis.

**If wrong:** Form new hypothesis based on what you learned.

Effective debugging requires forming theories and testing them systematically.

# Common Pitfall 5: No Version Control

## The Problem:

Making changes without version control means you can't undo them if they make things worse.

## Consequences:







- "Fix" one bug but introduce three more
- No way to get back to previous state
- Can't track what changes were made

## Best Practice:







- Use Git or another version control system
- Commit working code before debugging
- Make small commits as you fix bugs
- Can always revert to known good state

# Print Debugging vs. Interactive Debuggers

## Print Debugging:

-  Quick and simple
-  No special tools needed
-  Works in production
-  Clutters code
-  Manual cleanup
-  Not interactive

## Interactive Debuggers:

-  Powerful and flexible
-  Examine any variable
-  Step through dynamically
-  Requires IDE setup
-  Not for production
-  Doesn't work for timing bugs

# When to Use Each Approach

## Use Print Debugging for:

- Quick investigations
- Production environments
- Understanding code flow
- When debugger isn't available







## Use Interactive Debuggers for:

- Complex logic errors
- Examining object state
- Stepping through algorithms
- Testing different scenarios interactively







**Best Practice:** Use both approaches as appropriate for the situation.

# Logging vs. Debugging

## Logging:

-  Permanent records
-  Works in production
-  Track intermittent bugs
-  Performance analysis
-  Requires planning
-  Can't examine interactively







## Debugging:

-  Immediate feedback
-  Interactive examination
-  See runtime state
-  Step through code
-  Only while running
-  Not for production









# Automated Testing vs. Manual Debugging

## Automated Tests:

-  Catch bugs early
-  Prevent regressions
-  Fast feedback
-  Run automatically
-  Upfront effort
-  Requires maintenance

## Manual Debugging:

-  Flexible
-  Investigate any issue
-  No setup needed
-  Time-consuming
-  Doesn't prevent recurrence
-  Requires active work

## Key Takeaways

- **Debugging is systematic problem-solving, not guessing** → Follow structured process: reproduce, isolate, understand, fix, verify, document
- **Master your debugging tools** → Breakpoints, variable inspection, logging, stack traces transform debugging into efficient problem-solving
- **Read error messages carefully** → They tell you what went wrong and where; the stack trace shows the path that led to the error
- **Combine static and dynamic analysis** → Use linters to catch obvious mistakes, testing and debugging to find runtime issues



## Key Takeaways (continued)

- **Change one thing at a time** → Test after each change to understand what actually fixed the problem
- **Logging is essential for production** → Can't attach debugger to production, but logs provide visibility into what happened
- **Prevention is better than cure** → Write clear code, validate inputs, handle edge cases, write tests

Every bug you prevent is easier than every bug you fix. Learn from each bug to avoid similar issues in the future.



## Practice Quiz: Question 1

**You're debugging a function that sometimes returns incorrect results. Which approach should you try FIRST?**

- A) Rewrite the entire function from scratch
- B) Add print statements throughout the function
- C) Try to reproduce the bug with specific inputs consistently
- D) Ask a colleague to fix it for you



## Answer 1

**Correct: C) Try to reproduce the bug with specific inputs consistently**

### **Explanation:**

Before you can fix a bug, you need to reproduce it reliably. Understanding exactly what inputs or conditions trigger the problem is the essential first step in systematic debugging.

Once you can make the bug happen consistently, you can then use print statements, debuggers, or other tools to investigate.

Rewriting without understanding often introduces new bugs. Gather information first to make help from colleagues more effective.



## Practice Quiz: Question 2

**You set a breakpoint in your code, but the debugger never pauses there. What are the most likely reasons?**

- A) The line with the breakpoint is never executed
- B) The debugger is not properly configured
- C) You're running the program normally instead of in debug mode
- D) All of the above



## Answer 2

**Correct: D) All of the above**

### **Explanation:**

All three reasons can prevent a breakpoint from working:

- If the code path isn't executed (e.g., inside an if-statement that evaluates to false), debugger won't pause
- If debugger isn't properly configured, breakpoints won't activate
- If running normally instead of debug mode, breakpoints are ignored

Test if the line is executed by adding a print statement—if it doesn't print, the line isn't being reached.



## Practice Quiz: Question 3

Which logging level should you use for a message that indicates something unexpected happened but the program can continue running?

- A) DEBUG
- B) INFO
- C) WARNING
- D) ERROR



## Answer 3

**Correct: C) WARNING**

**Explanation:**

WARNING is appropriate for unexpected situations that don't prevent the program from continuing.

**Example:** User enters invalid discount code, program continues without applying discount.

- **DEBUG:** Detailed diagnostic info during development
- **INFO:** Normal operational messages
- **ERROR:** Serious problems that caused functionality to fail
- **CRITICAL:** Very serious errors that might cause crash



## Practice Quiz: Question 4

**You've identified a bug and made a fix, but now a different part of your program is broken. What debugging principle did you likely violate?**

- A) You didn't read the error message carefully
- B) You changed multiple things at once without testing each change
- C) You didn't use a debugger
- D) You didn't write enough comments





## Answer 4

**Correct: B) You changed multiple things at once without testing each change**

### **Explanation:**

Changing multiple things simultaneously is a common debugging pitfall. When you make several changes at once, you can't tell:

- Which change fixed the original bug
- Which change introduced the new problem

**Best Practice:** Change one thing at a time, test after each change, and only proceed if the change improves the situation.

This maintains control and understanding throughout the debugging process.



## Practice Quiz: Question 5

**What is the main advantage of static analysis (like linters) over dynamic debugging?**

- A) Static analysis can find bugs without running the code
- B) Static analysis is more accurate than debugging
- C) Static analysis can fix bugs automatically
- D) Static analysis works better for complex logic errors



## Answer 5

**Correct: A) Static analysis can find bugs without running the code**

### Explanation:

The key advantage of static analysis is examining code without executing it. This means you can catch common mistakes (undefined variables, type errors, style violations) immediately while writing code.

- **Fast:** Runs instantly as you type
- **Early:** Catches mistakes before you run code
- **Integrates:** Works in your editor/IDE

However, it can't catch logic errors or runtime issues—that's where dynamic debugging excels.

Static analysis and dynamic debugging are complementary techniques!

# Debugging Tools Summary

## Essential Tools:

- **IDE Debuggers:** VS Code, PyCharm, Visual Studio
- **Command-line:** pdb (Python), gdb (C/C++), lldb (macOS)
- **Browser DevTools:** Chrome, Firefox developer tools
- **Logging Libraries:** Python logging, Log4j, Winston (Node.js)
- **Static Analysis:** pylint, ESLint, mypy, RuboCop
- **Version Control:** Git for tracking changes

Master the tools in your development environment—they're your best friends when debugging!

# The Complete Debugging Workflow

1. Reproduce Bug Reliably
- ↓
2. Isolate Problem Area
- ↓
3. Understand Root Cause
- ↓
4. Form and Test Hypotheses
- ↓
5. Verify the Fix
- ↓
6. Document and Learn
- ↓
- Bug Fixed! ✓

# Debugging Best Practices Checklist

## Before Debugging:

- ☒ Can you reproduce the bug consistently?
- ☒ Do you understand the expected behavior?
- ☒ Have you read the error message carefully?
- ☒ Is your code under version control?

## During Debugging:

- ☒ Are you changing one thing at a time?
- ☒ Are you testing after each change?
- ☒ Are you following the evidence, not assumptions?
- ☒ Are you documenting what you try?

# After Fixing the Bug

## Post-Fix Checklist:

- ☒ Did you verify the fix with multiple test cases?
- ☒ Did you check for edge cases?
- ☒ Did you write a test to prevent regression?
- ☒ Did you document what caused the bug?
- ☒ Did you commit the fix with a clear message?
- ☒ Did you learn something to prevent similar bugs?

**Remember:** Every bug is a learning opportunity!

## Additional Resources

### Official Documentation:

- Python Debugging: <https://docs.python.org/3/library/pdb.html>
- VS Code Debugging: <https://code.visualstudio.com/docs/editor/debugging>
- Chrome DevTools: <https://developer.chrome.com/docs/devtools/>

### Books:

- "Effective Debugging" by Diomidis Spinellis
- "Debugging" by David J. Agans
- "The Art of Debugging with GDB, DDD, and Eclipse"

### Online Resources:

- Stack Overflow debugging tag
- MIT OpenCourseWare: Software Construction
- Stanford CS debugging guides



# Quick Reference: Common Error Types

Error Type	Meaning	Common Cause
<b>SyntaxError</b>	Code not valid Python	Typos, missing colons
<b>NameError</b>	Variable not defined	Typo, wrong scope
<b>TypeError</b>	Wrong type used	String used as number
<b>IndexError</b>	Index out of range	List access beyond size
<b>KeyError</b>	Dictionary key missing	Key doesn't exist
<b>ZeroDivisionError</b>	Division by zero	No input validation
<b>AttributeError</b>	Attribute doesn't exist	Wrong object type

# Quick Reference: Debugger Commands

## Python pdb commands:

- `n` (next) → Execute current line
- `s` (step) → Step into function
- `c` (continue) → Resume execution
- `l` (list) → Show source code
- `p variable` → Print variable value
- `b linenum` → Set breakpoint
- `q` (quit) → Exit debugger
- `h` (help) → Show help

## Quick Reference: Logging Format

```
import logging

# Basic configuration
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    filename='app.log'
)

# Usage
logging.debug("Detailed diagnostic info")
logging.info("General informational message")
logging.warning("Warning message")
logging.error("Error message")
logging.critical("Critical error message")
```

# Final Thoughts

**Debugging is a skill that improves with practice.**

## What you've learned:







- The systematic debugging process
- Essential debugging tools and when to use them
- How to think like a detective when solving problems
- Common pitfalls and how to avoid them
- The importance of prevention and documentation

## Next steps:

- Practice debugging on real problems
- Master your IDE's debugging tools
- Write tests to prevent bugs
- Learn from every bug you encounter

# Lesson Complete!

You now understand:

-  What debugging is and why it's critical
-  Essential debugging tools (debuggers, logging, print statements)
-  Systematic debugging process (6 steps)
-  Static vs. dynamic defect identification
-  Common pitfalls and best practices
-  When to use different debugging approaches

**Remember:** Every developer debugs. The difference between beginners and experts is having systematic approaches and powerful tools. You now have both!

**Happy debugging!**