Debugging Python Applications

Introduction

Writing code is only one part of software development.

The other important skill is finding out why code does not behave as expected.

Your program may:

- Crash with an error
- Produce the wrong result
- Work sometimes but fail in other situations
- Stop at a particular point
- Return unexpected data
- Become extremely slow
- Work locally but fail somewhere else

The process of finding the cause of these problems and fixing them is called **debugging**.

Debugging is not simply:

```
See error → change random code → run again
```

Good debugging is a structured process:

```
Observe
  ↓
Reproduce
  ↓
Understand
```

-
Isolate
-
Inspect
-
Fix
-
Verify

The goal is not just to make the error disappear.

The goal is to understand **why the problem happened**.

---

1. What Is Debugging?

Debugging is the systematic process of finding and fixing problems in software.

Suppose you write:

```
price = 100
quantity = 3

total = price + quantity

print(total)
```

The program runs.

There is no syntax error.

But the result is:

`103`

when you actually wanted:

`300`

This is a bug.

The program is syntactically valid, but its behavior is incorrect.

Debugging helps you discover the reason:

`total = price * quantity`

---

2. What Is a Bug?

A **bug** is a problem that causes software to behave differently from what is intended.

For example:

```
age = 17

if age > 18:
    print("Adult")
```

The code runs.

But if the intended rule is "18 or older", the condition is wrong.

It should be:

```
if age >= 18:
    print("Adult")
```

The important idea:

> A bug is about incorrect behavior, not necessarily a program crash.

---

3. Error vs Bug

These terms are related but not identical.

Error

An error is a problem detected by Python or another system.

Example:

```
print(name)
```

If `name` does not exist:

```
NameError
```

Bug

A bug can exist even when Python reports no error.

Example:

```
total = price + quantity
```

when multiplication was intended.

Think:

```
Error
→
Something went wrong technically

Bug
→
Program behavior is incorrect
```

A program can have bugs without crashing.

---

4. Three Common Categories of Python Problems

A useful starting classification is:

```
Syntax Errors
Runtime Errors
Logic Errors
```

Understanding which category you are dealing with changes how you debug it.

---

5. Syntax Errors

A syntax error means Python cannot understand the structure of your code.

Example:

```
if age >= 18
    print("Adult")
```

Python expects a colon.

You may see:

```
SyntaxError
```

Correct:

```
if age >= 18:
    print("Adult")
```

The program cannot execute normally until the syntax problem is fixed.

---

6. Runtime Errors

A runtime error occurs while the program is executing.

Example:

```
numbers = [10, 20, 30]

print(numbers[5])
```

Python understands the syntax.

But index `5` does not exist.

You may get:

`IndexError`

Another example:

`result = 10 / 0`

This produces:

`ZeroDivisionError`

---

7. Logic Errors

Logic errors are often more difficult.

The program runs.

No exception occurs.

But the result is wrong.

Example:

```
marks = [80, 90, 70]

average = sum(marks) / 2
```

```
print(average)
```

The code runs.

But the correct calculation is:

```
average = sum(marks) / len(marks)
```

Logic errors require you to compare:

```
What the program does
      vs
What the program should do
```

---

8. The First Rule of Debugging

Do not immediately change code.

First ask:

> **What exactly is going wrong?**

For example:

```
Expected:
100

Actual:
70
```

That gives you something concrete to investigate.

Without knowing the expected behavior, you cannot reliably determine whether something is wrong.

---

9. Reproduce the Problem

A bug that cannot be reproduced is harder to investigate.

Suppose someone says:

> "The program sometimes crashes."

Ask:

- What input caused it?
- What steps were performed?
- Which environment?
- Which Python version?
- Does it happen every time?
- What happened immediately before the crash?

Try to create a reliable reproduction.

For example:

```
Input:
age = -5
```

```
Action:
calculate_discount()

Result:
ValueError
```

Now you have a specific case to investigate.

---

10. Read the Error Message

When Python raises an exception, it usually provides useful information.

Example:

```
numbers = [10, 20, 30]

print(numbers[5])
```

You may see:

```
Traceback (most recent call last):
  File "app.py", line 3, in <module>
    print(numbers[5])
IndexError: list index out of range
```

Do not ignore this message.

It tells you:

```
Where?
What operation?
```

What type of problem?

---

11. Understanding a Traceback

A traceback is Python's record of how execution reached the error.

Consider:

```
def calculate():
    return process()


def process():
    return divide(10, 0)


def divide(a, b):
    return a / b


calculate()
```

The traceback may show several function calls.

Read it from the bottom.

The final line often tells you the actual exception:

```
ZeroDivisionError: division by zero
```

The lines above show how execution reached that point.

---

12. Read the Last Line First

For many Python exceptions, start with the final line.

Example:

```
ValueError: invalid literal for int() with base 10: 'abc'
```

This tells you:

```
Exception:
ValueError

Reason:
Python could not convert "abc" into an integer
```

Then move upward to find where it happened.

---

13. Then Find Your Code

A traceback can include:

```
Python/library code
Your application code
```

Focus first on the lines belonging to your project.

Example:

```
File "app.py", line 27
    age = int(user_input)
```

This is immediately useful.

Now inspect:

```
user_input
```

Why is it:

```
"abc"
```

instead of:

```
"25"
```

---

14. Use the Smallest Reproduction

If a large application has a problem, isolate it.

Instead of debugging:

```
5,000 lines
```

try to reduce the problem to:

```
value = "abc"

number = int(value)
```

If the small example still fails, you have isolated the problem.

This is called creating a **minimal reproducible example**.

---

15. Print Debugging

One of the simplest debugging techniques is temporarily printing values.

Example:

```
price = 500
quantity = 3

print("price:", price)
print("quantity:", quantity)

total = price * quantity

print("total:", total)
```

You can inspect what the program is actually doing.

---

16. Debug Prints Should Answer Questions

Do not randomly add:

```
print("here")
print("here2")
print("hello")
```

Instead, print information that helps answer a specific question.

For example:

```
print("User ID:", user_id)
print("Cart items:", cart_items)
print("Total before discount:", total)
```

The question might be:

> Is the cart actually receiving the expected items?

---

17. `repr()` Is Useful for Debugging

When debugging strings and values, `repr()` can reveal hidden characters.

Example:

```
name = "Hafsa\n"

print(name)
print(repr(name))
```

Output may look like:

```
Hafsa

'Hafsa\n'
```

The `\n` becomes visible.

This is useful when debugging:

- Spaces
- Newlines
- Tabs
- Empty strings
- Special characters

---

18. Check the Type

A large number of Python bugs are caused by unexpected types.

Example:

```
age = "25"

print(age + 5)
```

This fails because:

`"25"`

is a string, not an integer.

Debug with:

```
print(type(age))
```

Output:

```
<class 'str'>
```

Now the problem becomes obvious.

---

19. Inspect Multiple Values

You can inspect both value and type:

```
print("age:", repr(age), type(age))
```

For example:

```
age: '25' <class 'str'>
```

This is often much more useful than:

```
print(age)
```

---

20. Check Assumptions

A major debugging skill is questioning assumptions.

You may think:

```
users is always a list
```

But actual data might be:

`users = None`

You may think:

`price is always a number`

But actual input might be:

`price = "500"`

Instead of asking:

> Why is Python broken?

ask:

> Which assumption about my data is wrong?

---

21. Debugging With Assertions

Python's `assert` statement can check assumptions.

Example:

```
age = 25

assert age >= 0
```

If the condition is false, Python raises:

`AssertionError`

You can write:

`assert isinstance(age, int)`

This checks that:

`age`

is an integer.

---

22. Assertions With Messages

You can provide a useful message:

`assert age >= 0, "Age cannot be negative"`

If the condition fails:

`AssertionError: Age cannot be negative`

This can make debugging easier.

---

23. Assertions Are Not Input Validation

Do not use `assert` as your main mechanism for validating untrusted user input.

For example:

```
assert age >= 18
```

is not a replacement for proper validation.

For application validation, use explicit checks:

```
if age < 18:
    raise ValueError("Age must be at least 18")
```

Assertions are primarily useful for checking assumptions and programmer-level invariants.

---

24. Debugging With a Debugger

Print statements are useful, but a debugger provides much more control.

A debugger lets you:

- Pause execution
- Inspect variables
- Step through code
- Execute one line at a time
- Follow function calls
- Add breakpoints
- Examine program state

Instead of:

```
Run everything
```

Crash

you can:

Run
Pause
Inspect
Step
Understand

---

25. What Is a Breakpoint?

A breakpoint tells the debugger:

> Stop the program here.

Example:

```
def calculate_total(price, quantity):
    total = price * quantity
    return total
```

You can place a breakpoint at:

```
total = price * quantity
```

When execution reaches that line, the debugger pauses.

You can inspect:

`price`
`quantity`
`total`

---

26. Stepping Through Code

Once paused, a debugger typically provides operations such as:

Step Over

Execute the current line and move to the next line.

Step Into

Enter the function being called.

Step Out

Finish the current function and return to the caller.

Continue

Resume execution until the next breakpoint.

These tools allow you to follow the program's execution.

---

27. Example of Stepping

Consider:

```
def calculate_total(price, quantity):
    subtotal = price * quantity
    tax = subtotal * 0.1
    total = subtotal + tax

    return total
```

You can inspect the values as execution moves:

```
price = 100
quantity = 2

subtotal = 200
tax = 20
total = 220
```

If you unexpectedly get:

```
total = 2020
```

you can identify the exact point where the value changed incorrectly.

---

28. Python's Built-In `breakpoint()`

Python provides a built-in function:

```
breakpoint()
```

Example:

```
def calculate_total(price, quantity):
    subtotal = price * quantity

    breakpoint()

    tax = subtotal * 0.1
    return subtotal + tax
```

When execution reaches:

```
breakpoint()
```

Python enters the debugger.

This is useful for interactive debugging.

---

29. Inspecting Variables in the Debugger

Suppose:

```
def calculate(price, quantity):
    total = price * quantity

    breakpoint()

    return total
```

When paused, you can inspect:

```
price
quantity
total
```

This lets you ask:

```
Is price correct?
Is quantity correct?
Was total calculated correctly?
```

---

30. Debugging State

A program's **state** is the current information stored in memory while it runs.

For example:

```
cart = {
   "items": 3,
   "total": 1500
}
```

At one point:

```
items = 3
total = 1500
```

Later:

```
items = 3
total = 4500
```

If the total is wrong, you need to find the point where the state changed incorrectly.

Debuggers are particularly useful for this.

---

31. Follow the Data

A powerful debugging technique is:

> Follow the value from where it is created to where it becomes wrong.

Example:

```
price = get_price()
discount = calculate_discount(price)
total = price - discount
```

If:

```
total
```

is wrong, inspect:

```
price
  □
discount
  □
total
```

Do not immediately rewrite the final calculation.

Find where the incorrect value first appeared.

---

32. Find the First Wrong Value

Suppose:

```
price = 100      
discount = 20    
total = 800      
```

The bug is probably in:

```
total = price - discount
```

But if:

```
price = 100      
discount = 800   
total = -700     
```

the real problem is earlier.

The principle:

> **Find the first incorrect state, not just the final incorrect output.**

---

33. Debugging Loops

Loops can produce subtle bugs.

Example:

```
numbers = [1, 2, 3, 4]

total = 0

for number in numbers:
    total = number

print(total)
```

Output:

```
4
```

If the goal was the sum, the correct code is:

```
total = 0

for number in numbers:
    total += number
```

Debugging the loop means inspecting:

```
number
total
```

on each iteration.

---

34. Debugging Conditions

Consider:

```
age = 20

if age > 18:
    print("Adult")
```

If your business rule is:

`18 or older`

then:

`age >= 18`

is required.

When debugging conditions, inspect:

- Input value
- Condition
- Expected branch
- Actual branch

A useful temporary debug statement:

```
print("age:", age, "condition:", age >= 18)
```

---

35. Debugging Functions

When a function produces the wrong result, check:

```
Input
 
Parameters
 
Internal state
 
Intermediate values
 
Return value
```

Example:

```
def calculate_discount(price, percentage):
    discount = price * percentage
    return price - discount
```

If:

```
calculate_discount(100, 20)
```

returns:

```
-1900
```

inspect:

```
percentage
```

Maybe the caller passed:

```
20
```

instead of:

```
0.20
```

The function may be receiving the wrong representation.

---

36. Debugging Exceptions

Suppose:

```
def get_user_name(user):
    return user["name"]
```

and:

```
user = None
```

Calling:

```
get_user_name(user)
```

causes an error.

Do not simply catch everything:

```
try:
    ...
except:
    pass
```

That hides problems.

Instead, determine why:

`user`

is `None`.

---

37. Avoid Bare `except`

This is usually a poor debugging pattern:

```
try:
    result = process_data()
except:
    pass
```

Why?

Because it hides the original problem.

The program may appear to continue while silently doing the wrong thing.

Prefer specific exception handling when recovery is appropriate:

```
try:
    result = int(value)
except ValueError:
    print("Invalid number")
```

---

38. Logging Instead of Permanent `print()`

During development:

```
print("user:", user)
```

can be useful.

For an application, logging is often more appropriate:

```
import logging

logging.basicConfig(level=logging.INFO)

logging.info("Processing user %s", user_id)
```

Logging gives you:

- Severity levels
- Consistent formatting
- Configurable output
- Better production diagnostics

---

39. Useful Logging Levels

Common levels include:

```
DEBUG
INFO
WARNING
```

ERROR
CRITICAL

Think of them as:

DEBUG
🔍
Detailed developer information

INFO
ℹ️
Normal application events

WARNING
⚠️
Something unexpected but not necessarily fatal

ERROR
❌
An operation failed

CRITICAL
🔥
A serious failure

---

40. Debug Logging

For detailed diagnostics:

```
logging.debug("User object: %r", user)
```

You can enable debug logging during development.

This is generally better than leaving dozens of temporary `print()` statements throughout production code.

---

41. Debugging File Operations

Suppose:

```
with open("users.json") as file:
    data = file.read()
```

If the application cannot find the file, ask:

```
Which directory is the program running from?
```

Check:

```
from pathlib import Path

print(Path.cwd())
```

This shows the current working directory.

The file may exist somewhere else.

---

42. Debugging Paths

Instead of assuming:

```
data/users.json
```

exists, inspect:

```
from pathlib import Path

path = Path("data/users.json")

print("Path:", path)
print("Exists:", path.exists())
print("Absolute:", path.resolve())
```

Now you have evidence.

This is much better than guessing.

---

43. Debugging APIs

Suppose your application makes an HTTP request.

Do not only inspect:

```
response.json()
```

First inspect:

```
print(response.status_code)
print(response.text)
```

You might discover:

```
404
```

or:

```
401
```

or:

```
500
```

The problem may be:

```
Wrong URL
Invalid credentials
Server error
```

rather than your JSON parsing code.

---

44. Debugging Database Problems

If a database query produces unexpected results, inspect:

```
SQL/query
parameters
connection
returned rows
data types
transaction state
```

For example:

```
print("user_id:", user_id)
print("query:", query)
```

Avoid logging passwords, tokens, or other sensitive information.

---

45. Debugging Environment Problems

Sometimes your code is correct but the environment is different.

Check:

```
python --version
```

and:

```
python -m pip --version
```

Also verify the Python interpreter being used.

A project may work in one virtual environment and fail in another because:

```
Different Python version
Different package versions
Missing dependency
Different environment variables
```

---

46. "Works on My Machine"

This common situation means:

```
Developer environment
    
```

```
Other environment
```

Possible causes:

- Python version
- Package version
- Operating system
- Environment variables
- File paths
- Missing system dependency
- Configuration difference

Good project configuration reduces these problems.

---

47. Debugging With Version Information

When investigating a problem, record relevant versions.

For example:

```
python --version
```

and:

```
python -m pip list
```

For a specific package:

```
python -m pip show requests
```

This helps determine whether the issue is environment-specific.

---

48. Reproduce Before Fixing

A useful debugging cycle is:

```
1. Reproduce
2. Observe
3. Isolate
4. Hypothesize
5. Test
6. Fix
7. Verify
```

Do not skip the reproduction step when possible.

Otherwise you may "fix" something without actually knowing whether the bug is gone.

---

49. Form a Hypothesis

Good debugging involves hypotheses.

Suppose an API returns:

```
401 Unauthorized
```

Possible hypothesis:

```
The API token is missing or invalid.
```

Test it.

Check:

```
print(api_key is not None)
```

without printing the actual secret.

If the key exists, investigate another possibility.

This is much more effective than randomly changing code.

---

50. Change One Thing at a Time

Suppose you change:

- API URL
- authentication
- database query
- validation
- response parsing

all at once.

If the problem disappears, you do not know which change fixed it.

Instead:

```
Change one thing
```

Run
Observe
Keep or revert

This makes debugging scientific rather than random.

---

51. Use Binary Search for Large Problems

If a large section of code is involved, you can narrow the problem quickly.

Suppose:

```
Step 1
Step 2
Step 3
Step 4
Step 5
Step 6
Step 7
Step 8
```

The output becomes wrong somewhere.

Instead of inspecting every line, check halfway:

```
After Step 4: correct?
```

If yes:

`Bug is probably Step 5–8`

If no:

`Bug is probably Step 1–4`

Continue narrowing.

This idea can save significant time.

---

52. Use Version Control During Debugging

Git can help identify when a bug appeared.

Suppose the application worked yesterday.

You can inspect recent changes:

```
git log
```

You can inspect changes:

```
git diff
```

You can compare versions:

```
git diff HEAD~1
```

If you know the bug was introduced somewhere in a range of commits, tools such as:

`git bisect`

can help identify the problematic commit.

---

53. `git bisect` Concept

The basic idea:

```
Known good commit
      
      ?
      ?
      ?
      
Known bad commit
```

Git tests commits between the two points to narrow down which change introduced the bug.

This is especially useful for large projects.

---

54. Debugging Tests

Tests are valuable debugging tools.

Suppose:

```
def calculate_total(price, quantity):
    return price + quantity
```

A test might reveal:

```
def test_total():
    assert calculate_total(100, 3) == 300
```

The test fails:

```
Expected: 300
Actual: 103
```

Now the bug is clearly isolated.

Tests don't just prevent bugs.

They help locate them.

---

55. Debugging With Smaller Inputs

Large inputs can make bugs difficult to understand.

Suppose:

```
process_orders(100000)
```

fails.

Try:

```
process_orders(3)
```

A smaller dataset can make the behavior easier to inspect.

You can then gradually increase complexity.

---

56. Debugging Boundary Cases

Many bugs appear at boundaries.

Test values such as:

```
0
1
-1
empty string
empty list
None
maximum expected value
minimum expected value
```

For example:

```
def calculate_discount(price):
    return price * 0.10
```

What happens when:

```
price = 0
```

or:

```
price = -100
```

or:

```
price = None
```

Boundary testing often reveals hidden assumptions.

---

57. Debugging `None`

`None` causes many Python bugs.

Example:

```
user = find_user(user_id)

print(user["name"])
```

If `find_user()` returns:

```
None
```

the next line fails.

Instead of only fixing:

```
print(user["name"])
```

investigate:

```
Why did find_user() return None?
```

The root cause may be earlier.

---

58. Debugging Mutable Objects

Consider:

```
def add_item(items):
    items.append("Python")

items = []

add_item(items)

print(items)
```

The function modifies the original list.

This may be intended or unexpected.

When debugging objects, ask:

```
Is this function modifying the object?
Is another part of the program sharing this object?
```

Understanding mutability is essential when debugging Python.

---

59. Debugging State Changes

Consider:

```
user = {
    "balance": 100
}

user["balance"] -= 20
user["balance"] -= 50
```

Final state:

```
30
```

If the expected value is:

```
80
```

you need to find which operation changed the state unexpectedly.

Debugging is often about tracking these state transitions.

---

60. A Practical Debugging Workflow

When a Python application breaks:

Step 1: Read the error

`What exception occurred?`

Step 2: Locate the line

`Where did it happen?`

Step 3: Reproduce it

Can I make it happen consistently?

Step 4: Inspect inputs

Are the values what I expect?

Step 5: Inspect types

Are the types correct?

Step 6: Follow the data

Where did the first wrong value appear?

Step 7: Form a hypothesis

What do I think is causing this?

Step 8: Test the hypothesis

Change or inspect one thing.

Step 9: Fix the root cause

Do not merely hide the symptom.

Step 10: Verify

Run the original failing case and relevant tests.

---

61. Root Cause vs Symptom

Suppose your application crashes here:

`print(user["name"])`

You could write:

```
if user:
    print(user["name"])
```

This may prevent the crash.

But perhaps the real problem is:

`find_user()`

failed to find a user that should exist.

The first solution handles the symptom.

The second investigates the root cause.

Always ask:

> **Why did this state occur?**

---

62. Example Debugging Session

Suppose:

`def calculate_total(price, quantity, discount):`

```
    subtotal = price * quantity
    discount_amount = subtotal * discount
    return subtotal - discount


total = calculate_total(100, 2, 0.10)

print(total)
```

Expected:

```
180
```

Actual:

```
190
```

First inspect:

```
subtotal = price * quantity
```

Result:

```
200
```

Then:

```
discount_amount = subtotal * discount
```

Result:

```
20
```

But the return statement is:

```
return subtotal - discount
```

The bug is here.

Correct:

```
return subtotal - discount_amount
```

This is a classic logic bug.

---

63. Debugging Checklist

When something goes wrong, ask:

```
□ What was expected?
□ What actually happened?
□ Can I reproduce it?
□ What does the traceback say?
□ What line failed?
□ What are the input values?
□ What are their types?
□ What assumptions am I making?
□ Where does the first incorrect value appear?
□ Can I create a smaller reproduction?
□ What is my hypothesis?
□ Can I test it?
□ Did I fix the root cause?
□ Does the original problem now work?
□ Did the fix break anything else?
```

---

64. Mini Project: Debug a Broken Order System

Start with this intentionally broken code:

```
orders = [
    {"price": 100, "quantity": 2},
    {"price": 50, "quantity": 3},
    {"price": 200, "quantity": 1},
]


def calculate_total(order):
    return order["price"] + order["quantity"]


def calculate_orders_total(orders):
    total = 0

    for order in orders:
        total = calculate_total(order)

    return total


print(calculate_orders_total(orders))
```

There are multiple bugs.

Your task:

1. Find the incorrect calculation.
2. Find the incorrect accumulation logic.

3. Calculate the expected total.
4. Fix the code.
5. Add tests.
6. Use the debugger or print statements to inspect intermediate values.

A correct calculation should be:

```
100 × 2 = 200
50 × 3 = 150
200 × 1 = 200

Total = 550
```

---

65. 5 Minute Challenge

Take this function:

```
def average(numbers):
    total = 0

    for number in numbers:
        total = number

    return total / len(numbers)
```

Calling:

```
print(average([10, 20, 30]))
```

produces the wrong result.

Your tasks:

1. Identify the bug.
2. Explain why the code runs without a syntax error.
3. Find the first incorrect state.
4. Fix the function.

5. Test it with:

   - `[10, 20, 30]`
   - `[5]`
   - `[]`

Then decide what should happen when the list is empty.

---

66. Exercises

Exercise 1: Type Bug

Find the problem:

```
age = input("Age: ")

if age >= 18:
    print("Adult")
```

Questions:

- What type is `age`?
- Why does the comparison fail?
- How would you fix it safely?

Exercise 2: Logic Bug

Find the problem:

```
def calculate_area(width, height):
    return width + height
```

Expected:

```
Area = width × height
```

Exercise 3: State Bug

Find the problem:

```
balance = 100

def withdraw(balance, amount):
    balance - amount

balance = withdraw(balance, 20)

print(balance)
```

Why does the balance not change?

Exercise 4: Path Bug

Debug:

```
from pathlib import Path

path = Path("data/users.json")

print(path.exists())
print(path.resolve())
```

Determine why the expected file might not be found.

Exercise 5: Exception Bug

Debug:

```
def get_age(value):
    return int(value)

print(get_age("twenty"))
```

Determine:

- Exception type
- Cause
- Appropriate handling strategy

---

67. Mini Quiz

Question 1

What is debugging?

A. Writing code faster
B. Finding and fixing software problems
C. Installing Python
D. Creating databases

Answer: B

Question 2

Which problem occurs when Python cannot understand the structure of the code?

A. Logic error
B. Syntax error
C. Configuration error
D. Design error

Answer: B

Question 3

Which problem can occur even when the program runs successfully?

A. Logic error
B. Syntax error only
C. Installation error only
D. Import error only

Answer: A

Question 4

What does a breakpoint do?

A. Deletes code
B. Stops execution at a selected point
C. Installs a package
D. Creates a database

Answer: B

Question 5

What is a traceback?

A. A list of installed packages
B. A record showing the execution path leading to an exception
C. A Python data structure
D. A Git branch

Answer: B

Question 6

What should you usually do before changing code to fix a bug?

A. Delete the project
B. Understand and reproduce the problem
C. Reinstall Python
D. Rewrite the application

Answer: B

Question 7

Why is `print(type(value))` useful?

A. It changes the value
B. It reveals the value's data type
C. It catches all exceptions
D. It installs dependencies

Answer: B

---

68. Interview Questions

Beginner

1. What is debugging?
2. What is a bug?
3. What is the difference between an error and a bug?
4. What is a syntax error?

5. What is a runtime error?
6. What is a logic error?
7. What is a traceback?
8. How do you read a Python traceback?
9. Why are error messages useful?
10. What is a breakpoint?

Intermediate

11. What is the difference between debugging with `print()` and a debugger?
12. What is `breakpoint()`?
13. How do you inspect a variable while debugging?
14. What is a minimal reproducible example?
15. Why should you inspect types while debugging?
16. What is root cause analysis?
17. Why should you avoid bare `except`?
18. When should you use logging instead of `print()`?
19. How can Git help debug a regression?
20. What is `git bisect`?

Advanced

21. How would you debug a bug that cannot be reproduced consistently?
22. How would you debug a problem that occurs only in production?
23. How would you distinguish an application bug from an environment problem?
24. How can dependency versions cause unexpected behavior?
25. How do you debug circular imports?
26. How do you find the first incorrect state in a complex data flow?
27. How can tests help with debugging?
28. How would you debug a performance problem?
29. Why is changing one thing at a time useful?
30. How would you systematically debug a large Python application?

---

69. Debugging Cheat Sheet

Inspect a value

```
print(value)
```

Inspect value and type

```
print(repr(value), type(value))
```

Check an assumption

```
assert value is not None
```

Start the debugger

```
breakpoint()
```

Check current directory

```
from pathlib import Path

print(Path.cwd())
```

Check a path

```
print(path.exists())
print(path.resolve())
```

Check Python version

```
python --version
```

Check installed packages

```
python -m pip list
```

Check a package

```
python -m pip show package_name
```

Inspect Git changes

```
git diff
```

Inspect history

```
git log
```

Find a regression

```
git bisect
```

---

70. Debugging Mindset

A strong developer does not treat debugging as random trial and error.

Instead:

```
Problem
  
Evidence
  
Hypothesis
  
Experiment
  
Result
```

`Reliable reproduction`

4. Where does it first go wrong?

`First incorrect state`

5. Why did that state occur?

`Root cause`

6. Does the fix solve the original problem?

`Verification`

These questions form the foundation of effective debugging.

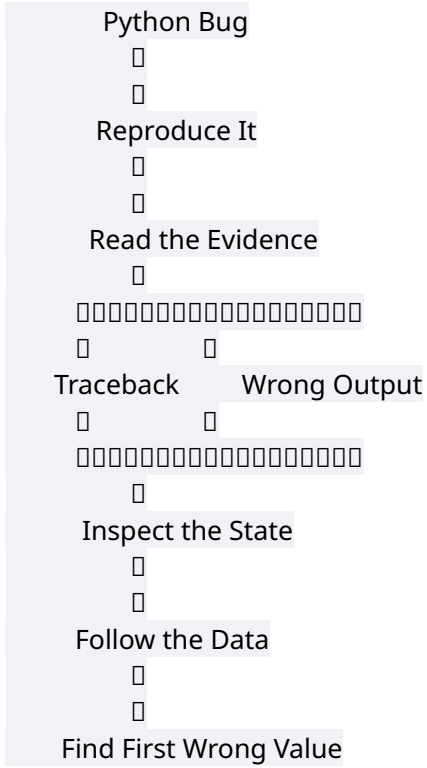
---

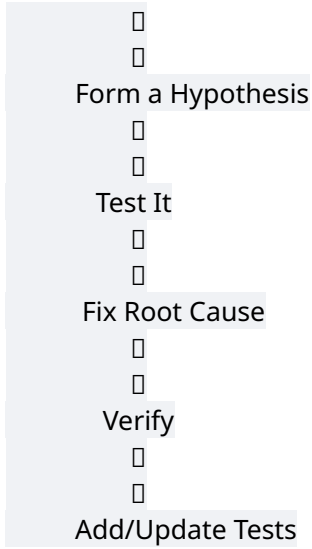
72. Key Takeaways

- Debugging is the systematic process of finding and fixing software problems.
- Bugs can exist even when programs do not crash.
- Python problems commonly involve syntax, runtime, or logic errors.
- Read tracebacks carefully instead of ignoring them.
- The final line of a traceback often identifies the exception.
- Reproduce the problem before trying to fix it.
- Inspect values and types.
- Follow data through the program.
- Find the **first incorrect state** rather than only fixing the final symptom.
- Use `print()` for quick investigation.
- Use `breakpoint()` and a debugger for deeper investigation.
- Use assertions to check programmer assumptions.
- Use logging for structured application diagnostics.

- Avoid hiding errors with broad exception handling.
- Check environment and dependency differences when behavior differs between machines.
- Use tests to reproduce and isolate bugs.
- Use Git history when investigating regressions.
- Change one thing at a time when testing a hypothesis.
- Fix root causes instead of simply hiding symptoms.
- Good debugging is a reasoning skill, not just a tool skill.

Final Mental Model





> **Debugging is not guessing what line to change. It is using evidence to discover why the program entered an incorrect state.**

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

https://dev-roast-app.vercel.app/resources