

Troubleshooting Common Python Programming Problems



Scope: diagnosing and fixing errors in scripts, notebooks, and small apps



For learners, junior developers, analysts, and educators who write basic Python



Troubleshooting means reading tracebacks, isolating causes, testing fixes



The loop: reproduce, read the error, locate the line, inspect state, fix, retest



A structured loop beats guessing: fewer regressions and faster learning



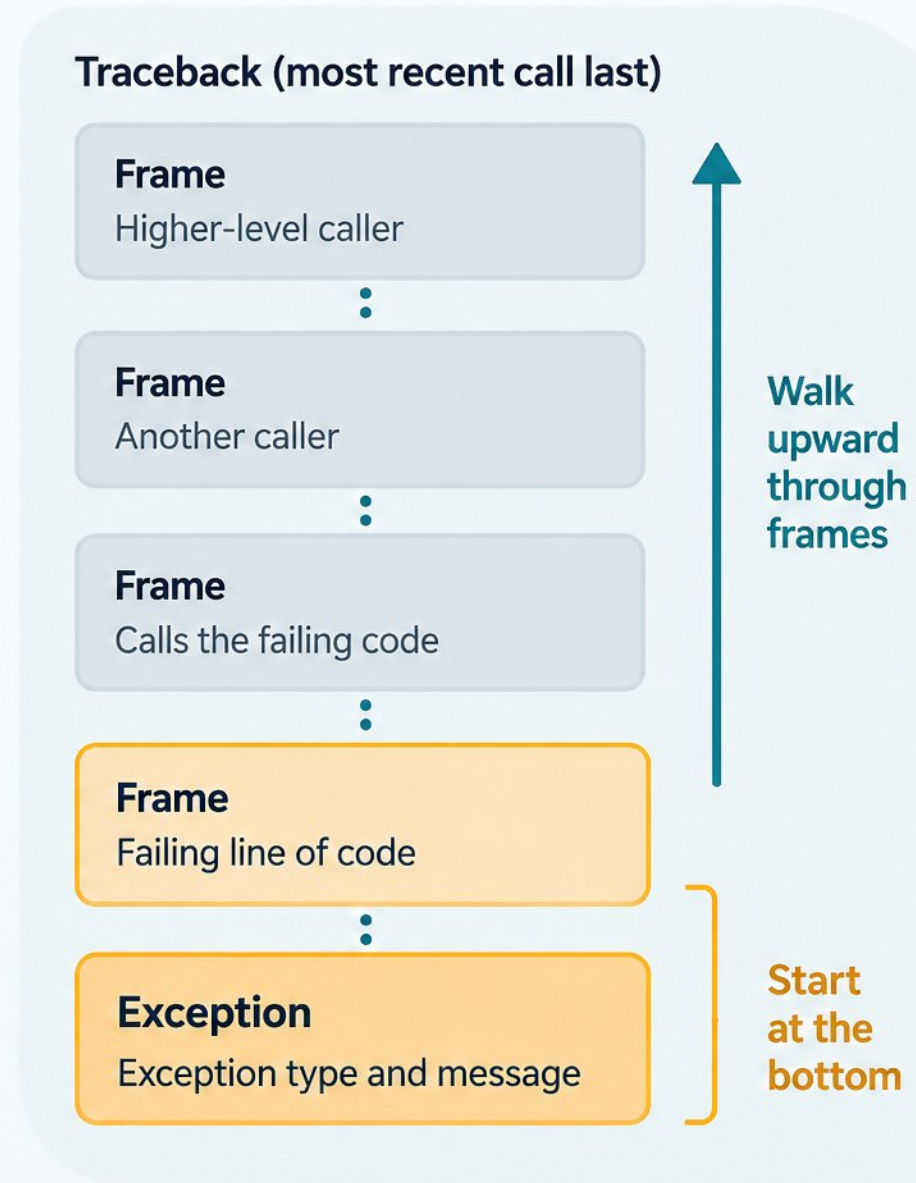
Roadmap: parse errors, runtime errors, collections, functions, imports, tools, files, lab



Reading Python Tracebacks Without Panic



- Start at the bottom: exception type, message, then the failing line
- Walk upward through frames to see who called the failing code
- The last line names the error; earlier frames explain why
- Know the usual suspects: `NameError`, `TypeError`, `ValueError`, `KeyError`, `IndexError`, `AttributeError`
- Hierarchy matters: `ModuleNotFoundError` is an `ImportError`; `UnicodeDecodeError` is a `ValueError`
- Syntax, runtime, and logical errors need different repair workflows
- Trust the last line, unless an earlier frame holds the real cause



Fix Syntax and Indentation Errors First

Missing colon

```
if x > 0
```



SyntaxError

Unmatched bracket

```
items = [1, 2, 3
```



SyntaxError

Mismatched quote

```
name = "Alex
```



SyntaxError

Mixed tab and space

```
def total():  
    ... return 42
```



TabError / IndentationError

- SyntaxError and IndentationError are parser failures: code never runs
- Common causes: missing colon, unmatched brackets, mismatched quotes, mixed tabs
- Caret marker and line number point near the problem, sometimes just past it
- TabError and IndentationError mean inconsistent block indentation: choose spaces, stay consistent
- Let linters, formatters, and a quick compile check catch these early
- Repair workflow: fix one parse error, rerun — parsers often report only the first one



Names, Types, and Values: Common Runtime Errors

UNDERSTAND • DIAGNOSE • WRITE MORE ROBUST CODE

NameError

```
print(user_name)
```

NameError:
name 'user_name' is not defined

TypeError

```
len(42)
```

TypeError:
object of type 'int' has no len()

ValueError

```
int("hello")
```

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

AttributeError

```
result = get_data()  
result.upper()
```

AttributeError:
'NoneType' object has no attribute 'upper'

KeyError

```
user = {}  
user["name"]
```

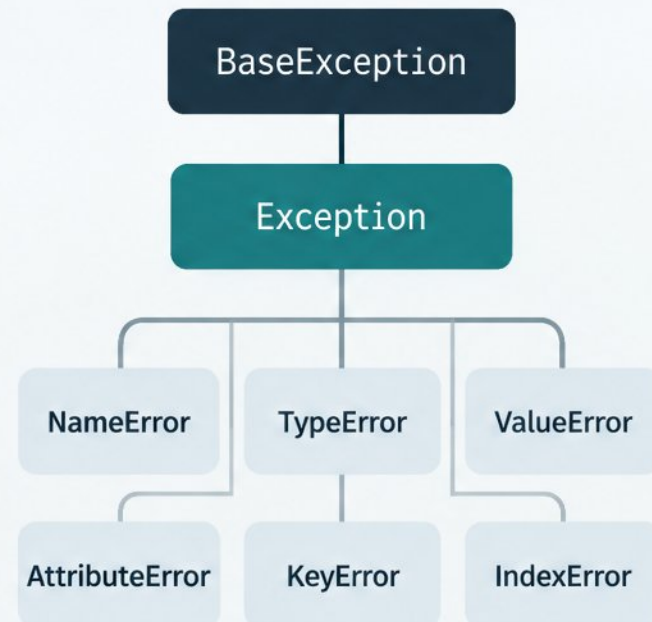
KeyError:
'name'

IndexError

```
items = []  
items[0]
```

IndexError:
list index out of range

Exception Hierarchy



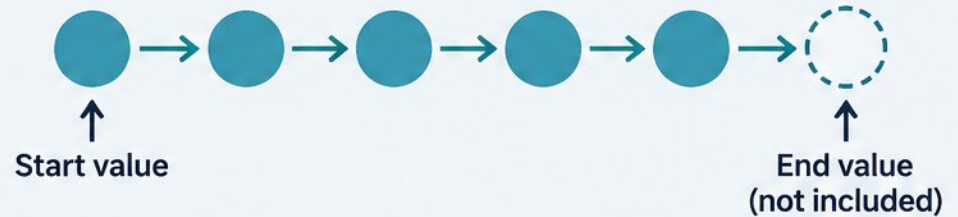
- **NameError** and **UnboundLocalError**: misspelled, unassigned, or branch-only names
- **TypeError**: wrong object type — including None used as a value
- **ValueError**: correct type, invalid value; UnicodeDecodeError is a subclass
- **AttributeError**: object lacks the attribute — often a function returned None
- **KeyError** vs. **IndexError**: missing dictionary key vs. out-of-range position
- **Inspect first**: type(), isinstance(), len(), targeted prints
- **Fix patterns**: validate input, use get(), check for None before access

Comprehensions, Loops, and Collection Pitfalls

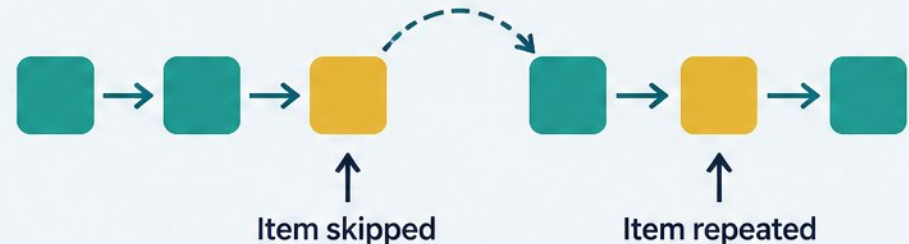


- 1 range() stops before the end value;
trace loop bounds before blaming the body
- 2 Mutating a list while iterating silently skips or repeats items — iterate a copy
- 3 IndexError: bad list or tuple position.
KeyError: missing dictionary key
- 4 Comprehension loop variables live in their own scope; filters change results
- 5 Nested comprehensions are hard to read — flatten them or use a loop
- 6 Debug: print the loop variable, shrink the example, use enumerate() and zip()

range() stops before the end value



Mutating a list while iterating can skip or repeat items



Error anatomy

What you assumed

[● ● ●]

IndexError:
bad list or tuple position.

What the object actually is

{ ■ : ● , ■ : ● }

KeyError:
missing dictionary key.

Comprehension scope and filters



Nested comprehensions

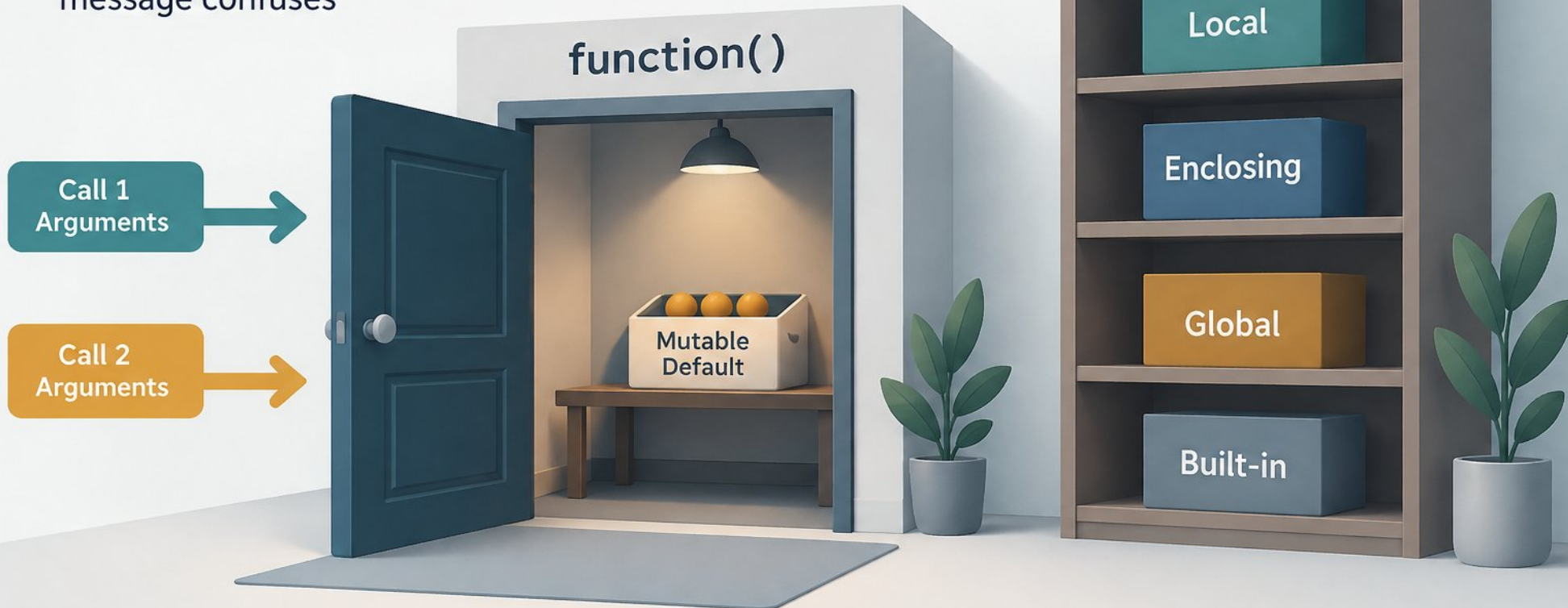
[[● for ● in ●]
for ● in ●]

Nested comprehensions are hard to read — flatten them or use a loop.

Function Calls, Arguments, and Scope

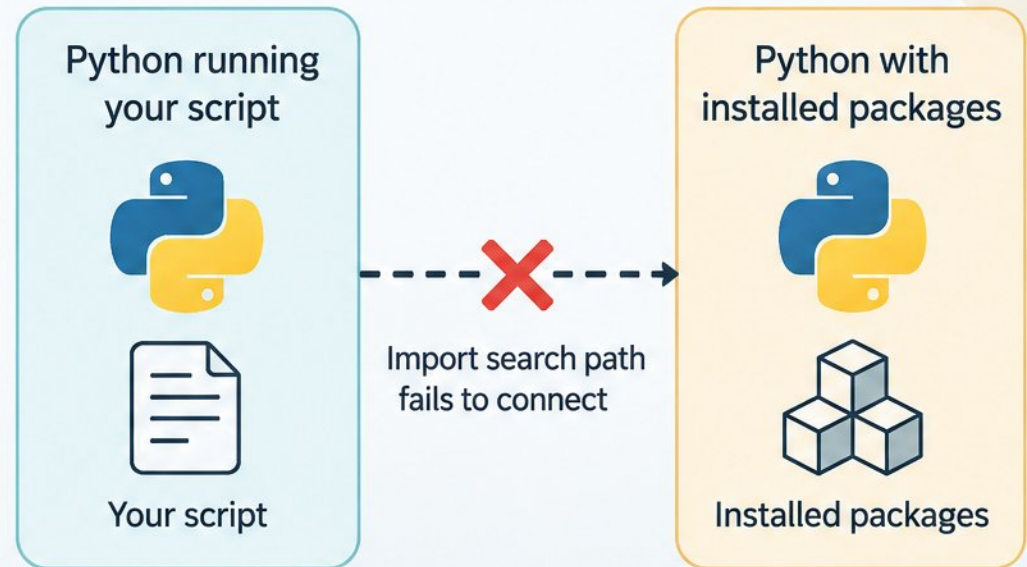


- **TypeError**: wrong argument count, unexpected keyword, or missing required argument
- Mutable default arguments are created once and shared across every call
- Use **None** as the default, then build the list or dict inside the function
- Name resolution follows LEGB: Local, Enclosing, Global, Built-in
- Most **NameErrors** and **UnboundLocalErrors** are scope mistakes
- Read the signature and the call site together when the message confuses



Import Errors and Environment Mismatches

- **ModuleNotFoundError:** package installed into a different Python than the one running your script
- **Top causes:** wrong Python on PATH, unactivated venv, pip bound to another interpreter
- **Habit:** use `python -m pip install` so pip matches the running interpreter
- **Quick diagnostics:** which python, which pip, print `sys.executable` and `sys.path`
- **Install name vs import name:** Pillow→PIL, PyYAML→yaml, beautifulsoup4→bs4
- **Local file shadowing:** a file named like a stdlib module hijacks the import — rename it
- **PEP 668:** distro Pythons refuse pip installs — use a virtual environment



```
$ python my_script.py
ModuleNotFoundError: No module named 'some_package'
```

Package not found because it's installed in a different Python environment

Debugging Tools: From Print to Breakpoints



pdb> |

Debug with breakpoint()



Logging or asserts

Replace prints with intent



Print debugging

Print values, not labels



Start here

- Print debugging: print values, not labels; remove prints after
- Replace prints with logging calls or asserts that carry intent
- breakpoint() is the modern entry point (Python 3.7+)
- Key pdb commands: n, s, c, p, pp, l, w, u, d, b, r, q
- Conditional breakpoints: probe without editing source
- Post-mortem: python -m pdb, pdb.pm(), or %debug in Jupyter
- Notebooks: restart and run all when state feels stale
- Use the lightest tool that answers the current question

```
>>> breakpoint()
> example.py(10)<module>()
-> value = compute()
(Pdb) |
```

Paused at your
code, explore with
pdb commands.

Data, Files, and Encoding Troubles

- `FileNotFoundError`: check relative vs. absolute paths, working directory, use `pathlib`
- `UnicodeDecodeError`: file isn't UTF-8; byte and position reveal the cause
- BOM clues: `0xff 0xfe` UTF-16; `0xef 0xbb 0xbf` needs `encoding='utf-8-sig'`
- Excel and legacy exports are often cp1252 or latin-1, not UTF-8
- Always pass encoding; avoid `errors='ignore'` for data you care about
- `pandas.read_csv` defaults to UTF-8: set `encoding=` or `encoding_errors=`
- `json.load()` takes a file handle; `json.loads()` takes a string

First bytes can tell the story

UTF-16 BOM

`0xff 0xfe ...`



Decodes to
(UTF-16 text)

cp1252 export

`0x93 0x94 ...`



Decodes to
"smart quotes"

UTF-8 BOM

`0xef 0xbb 0xbf ...`



Decodes to
(UTF-8 text)

Error anatomy

What you assumed



It's a UTF-8 text file.

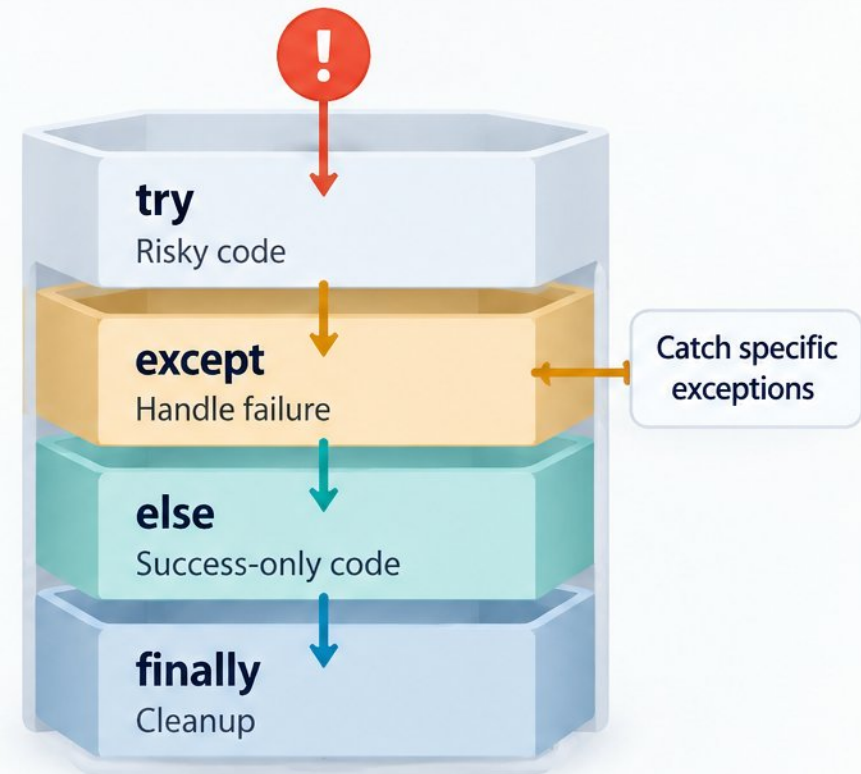
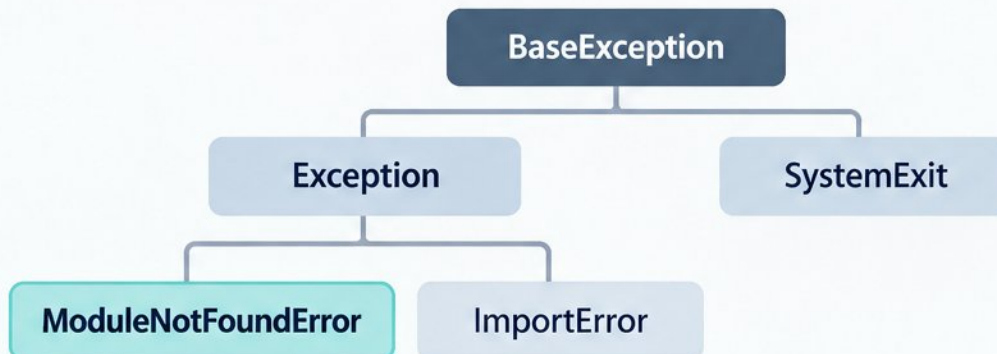
What the object actually is



It's a different encoding
(or not a text file).

Handling and Raising Exceptions Effectively

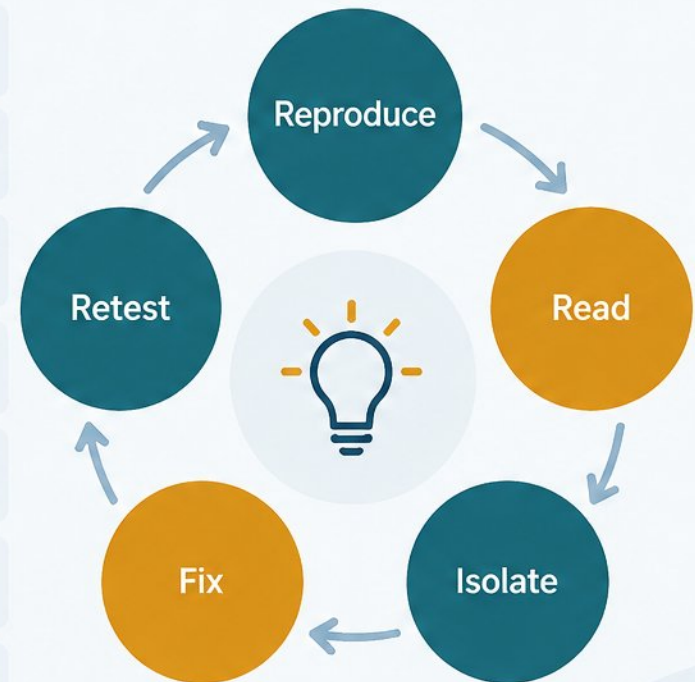
- try/except/else/finally: risky code, handle failure, success-only code, cleanup
- Catch specific exceptions, not bare except — `ModuleNotFoundError` before `ImportError`
- Raise or wrap with context to preserve the original cause
- Chain with 'raise ... from ...' so tracebacks show the real cause
- Use `add_note()` to attach diagnostic context visible in the traceback
- Name the operation and offending value in error messages
- Never swallow exceptions silently — hidden root causes get harder to debug



Guided Lab: Fix a Broken Script End to End



- 1 **Scenario:** small script reads CSV, transforms rows, writes JSON summary
- 2 **Planted bugs:** syntax, indentation, `NameError`, `IndexError`, `import`, and encoding errors
- 3 **Step 1:** read each traceback bottom-up; state the error in plain words
- 4 **Step 2:** fix parser errors first, rerun; do not chase runtime errors yet
- 5 **Step 3:** use `breakpoint()` or targeted prints; compare actual values to assumptions
- 6 **Step 4:** run `which python`, `which pip`, then print `sys.executable` and `sys.path`
- 7 **Step 5:** check first bytes, pick correct encoding, verify decoded rows
- 8 **Step 6:** remove scaffolding, rerun, confirm output file is correct



Pitfalls to Avoid When Troubleshooting

Clearer thinking leads to faster, more reliable fixes.



Common pitfall



Guessing without reproducing:
you are debugging in the dark



Fixing symptoms, not the wrong
assumption that caused them



Bare except blocks hide the
traceback you needed



PYTHONPATH and errors='ignore'
trade loud errors for silent bugs



Leaving breakpoint() or diagnostic
prints in production code



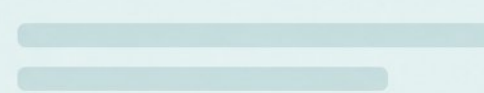
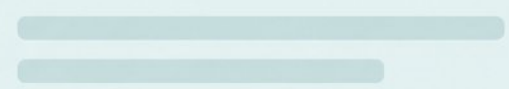
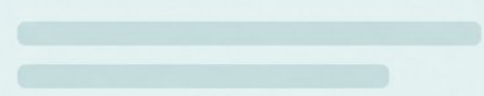
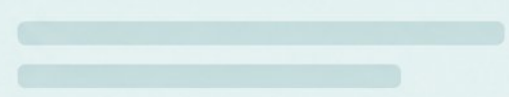
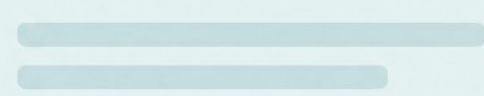
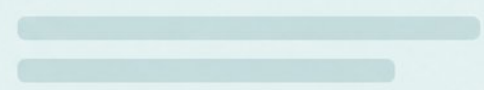
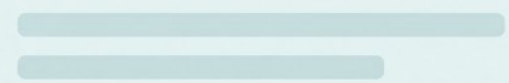
Changing several things at once
hides which fix worked



Trusting an encoding detector or
stale value without verifying



Better approach

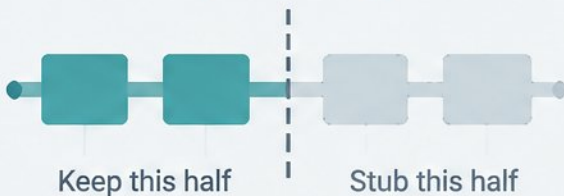


Building a Troubleshooting Habit

SMALL
STEPS
FEWER BUGS



Isolate by bisecting



Error Journal

- Exception
- Cause
- Fix
- The wrong assumption



Reusable checklist: reproduce, read, isolate, hypothesize, fix, retest



Isolate by bisecting: stub half the pipeline, confirm which half fails



Keep an error journal: exception, cause, fix, and the wrong assumption



Read Python docs with a goal: built-in exceptions, traceback, and pdb pages



Turn recurring fixes into tests, asserts, and logging



Educators: have learners explain the traceback in plain language first

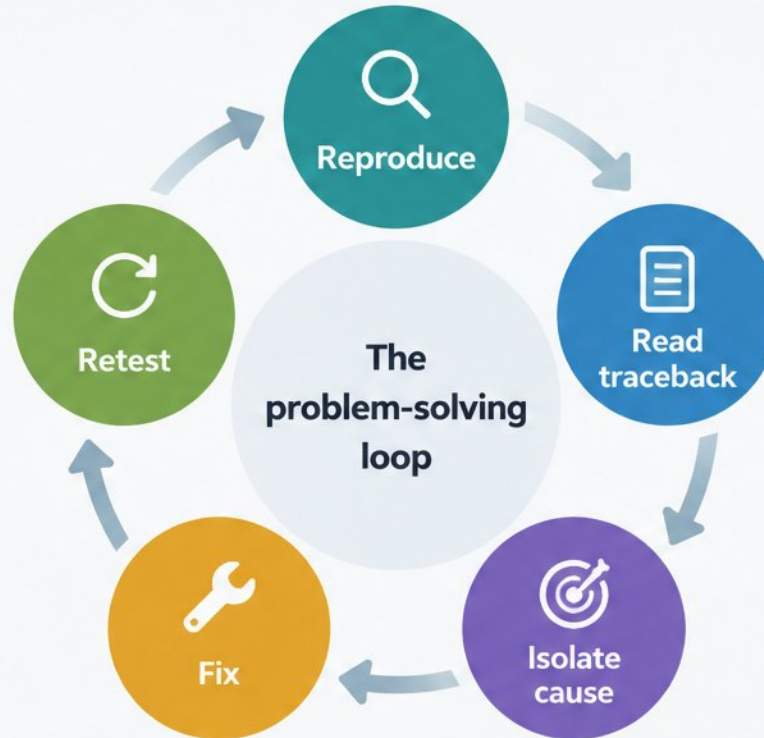
Wrap-Up, Self-Check, and Next Steps

A practical roadmap for debugging, learning, and making lasting progress.



Triage by symptom

- Parser first
- Then names
- Types
- Collections
- Imports
- Encoding



Tool ladder

- `>_` print
- `asserts`
- logging
- `breakpoint()/pdb`
- post-mortem
- IDE debuggers

- ✓ **Loop in one sentence**
reproduce, read traceback,
isolate cause, fix, retest

- ✓ **Environment check**
confirm running interpreter
matches the one holding
packages

- ✓ **Self-check**
 - reproduce a `cp1252 UnicodeDecodeError`, then
fix with correct encoding
 - explain a traceback bottom-up,
then name the exact line
to change

- ✓ **Next**
keep an error journal for
30 days, add a regression
test per fix
- 📖 **Read next**
built-in exceptions, traceback
module, and `pdb` in the
Python docs

PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/b1d8d559/public