Debugging Techniques in Python

PALLAVI VAIDYA

Agenda

- Common Python errors and how to handle them.
- Debugging systematically using Python tools.
- Best practices for effective debugging.

Common Python Errors

Syntax Errors

Indentation Error

Name Error

Index Error

Key Error

Attribute Error

ZeroDivisionError

Import Error / ModuleNotFound Error

Syntax Error

When it happens:
Occurs when Python code is written incorrectly — for example, missing colons, parentheses, or indentation errors.

Tip:
Use a good code editor (like VS Code or PyCharm) to highlight syntax issues automatically.

Indentation Error

When it happens:

Occurs when code blocks are not indented properly.

Example:

```
for i in range(3):

print(i)
```

Fix:

```
for i in range(3):

   print(i)
```

Name Error

When it happens:

Occurs when a variable or function name is not defined.

Tip:

Check spelling and ensure variables are declared before use.

Index Error

When it happens:
Occurs when accessing a list index that doesn’t exist.

Example:

```
lst = [1, 2, 3]

print(lst[5])
```

Fix:
```
if 5 < len(lst):

    print(lst[5])

else:

    print("Index out of range!")
```

Key Error

When it happens:

Occurs when trying to access a dictionary key that doesn’t exist.

Example:

```
student = {"name": "Alice"} print(student["age"])
```

Fix:

```
print(student.get("age", "Key not found"))
```

Attribute Error

When it happens:

Occurs when you call an undefined method or attribute of an object.

Example:

```
text = "hello"

text.push("!")
```

Fix:

```
text = "hello"

text.upper()
```

ZeroDivision Error

When it happens:

Occurs when dividing by zero.

Example:

```
result = 10 / 0
```

Import Error / ModuleNotFound Error

When it happens:

Occurs when importing a non-existent module or missing package.

Example:

```
import mymodule
```

Best Practices

Always use **try/except** for risky operations (I/O, user input, APIs).

Avoid **bare except:** statements (they hide real issues).

Use **logging** instead of `print()` for debugging in production.

Write **unit tests** to catch common errors early.

Debugging Tools in Python

Debugging is the process of identifying and fixing errors or unexpected behavior in your program. Python provides several powerful debugging tools to help you trace and resolve issues efficiently.

print() Statements (Basic Debugging)

What it is:
The simplest way to debug — adding print statements to check variable values or program flow.

Example:

def add(a, b): print(f"a={a}, b={b}") # Debug info return a + b result = add(5, "2") # Causes Type Error

Pros:

✅ Quick and simple for small scripts

Cons:

❌ Not ideal for large or complex programs

assert Statements

What it is:

Used to check if a condition is true during runtime. If not, an AssertionError is raised.

x = 10

assert x > 0, "x must be positive"

Pros:

✅ Quick way to catch logical errors early

Cons:

❌ Automatically disabled when running Python in optimized mode (python -O)

pdb (Python Debugger)

A built-in interactive debugger that allows you to pause execution, inspect variables, and step through code line by line.

Pros:

✅ Built-in and powerful

✅ Interactive exploration

Cons:

❌ Command-line interface can be intimidating for beginners

Try this at the terminal

Command	Description
`p a`	Print value of `a`
`p b`	Print value of `b`
`n`	Execute the next line
`c`	Continue execution until next breakpoint or end
`q`	Quit debugging

Breakpoint()Python 3.7

What it is:

A shortcut for starting the debugger — calls pdb.set_trace() by default.

Pros:

- ✅ Easier and cleaner than import pdb
- ✅ Can integrate with other debuggers

trace Module

What it is:

A module that tracks program execution and function calls.

Use of logging

Replace print statements with structured logs for better traceability.

1. Understand the Problem

- Reproduce the error consistently.
- Read the traceback to locate the issue.
- Identify expected vs actual behavior.

2. Use print() Statements

- Add print() to track variable values and flow.
- Simple but not ideal for large programs.

Example:

```
print(f'x={x}, y={y}')
```

3. Use Logging Instead of print()

- Provides levels: DEBUG, INFO, WARNING, ERROR.
- Supports timestamps and file output.

Example:

```
import logging

logging.debug('Debug info')
```

4. Use the Python Debugger (pdb)

- Step through code interactively.
- Commands: p (print), n (next), s (step), c (continue).

Example:

```
import pdb

pdb.set_trace()
```

5. Use breakpoint()

- Shortcut for pdb.set_trace().
- Available from Python 3.7+.

6. Use Assertions

- Check assumptions during development.

Example:

```
assert x >= 0, 'x must be non-negative'
```

7. Isolate the Problem

- Test small parts of code individually.
- Comment out sections to locate bugs.

8. Check Variable Types and Values

- Type mismatches cause common errors.
- Use type() or logging.debug() for inspection.

9. Use IDE Debuggers

- Tools: VS Code, PyCharm, Spyder, Thonny.
- Features: breakpoints, step-through, variable watch.

10. Handle Exceptions Gracefully

- Use try/except to avoid crashes.

Example:

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print('Cannot divide by zero')
```

11. Use Unit Testing

- Prevent bugs by testing functions automatically.

Example:

```
assert add(2, 3) == 5
```

12. Profile and Monitor Performance

- Use cProfile to identify slow parts of code.

Command:

```
python -m cProfile myscript.py
```

13. Version Control for Debugging

- Use Git to compare versions and find bug origins.
- Command: git bisect

Summary & Best Practices

- Use print() or logging to trace code.
- Step through with pdb or IDE debuggers.
- Write tests and handle errors properly.
- Reproduce, analyze, fix, and verify.