Debugging Adventures: The Quest for Bug-Free Code

Uncover Bugs and Fix Them Like a Pro

Michael Borck

Table of contents

Debugging: Fixing errors found during testing.

Testing vs. Debugging

Aspect	Testing	Debugging
Purpose	Identify errors and issues in the code	Fix errors and issues found during testing
Process	Running the code with various inputs to check for correctness	Analyzing and modifying the code to fix errors
Timing	Performed before debugging	Performed after testing detects issues
Outcome	A report of failures, errors, or issues	Corrected code without the detected issues
Tools	Testing frameworks like `unittest`, `pytest`, `doctest`	Debuggers, print statements, IDE debugging tools

Why Debugging is Important

- **Ensure Code Correctness**: Debugging helps find and fix errors, ensuring the code works as intended.
- **Improve Software Quality**: Identifying and resolving bugs leads to more reliable and robust software.
- **Optimize Performance**: Debugging can help identify performance bottlenecks and inefficiencies.
- **Enhance Maintainability**: Code that is thoroughly debugged is easier to maintain and extend.
- **Facilitate Learning**: Debugging helps programmers understand how their code works and how different parts interact.

Methods of Debugging in Python

1. **Step/Trace Through Code**
2. **Inspect Objects**
3. **Python Debugger (pdb)**

Step/Trace Through Code

- **print()**: Output variable values and program flow to the console.

```
x = 10
print(f"x: {x}")
```

- **logging()**: Use the logging module for more advanced logging capabilities.

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info(f"x: {x}")
```

Inspect Objects

- **type()**: Check the type of an object.

```
x = 10
print(type(x))
```

- **inspect module**: Provides several useful functions to help get information about live objects.

```
import inspect
print(inspect.getmembers(x))
```

Python Debugger (pdb)

- **breakpoint()**: Built-in function to pause execution and enter debugging mode.

```
def example_function():
    x = 10
    breakpoint()
    y = x + 5
    return y

example_function()
```

- **traceback**: Useful for getting detailed error reports when other methods are not enough.

```
import traceback

try:
    1 / 0
except ZeroDivisionError:
    traceback.print_exc()
```

Example: Using print() for Debugging

```
def add(a, b):
    print(f"a: {a}, b: {b}")
    return a + b

result = add(5, 3)
print(f"Result: {result}")
```

Example: Using logging for Debugging

```
import logging

logging.basicConfig(level=logging.DEBUG)

def add(a, b):
    logging.debug(f"a: {a}, b: {b}")
    return a + b

result = add(5, 3)
logging.debug(f"Result: {result}")
```

Example: Using pdb for Debugging

```
def add(a, b):
    x = a
    y = b
    breakpoint()
    return x + y

result = add(5, 3)
```

Best Practices for Debugging

- **Start Small**: Debug small sections of code before moving to larger sections.

- **Use Version Control**: Keep track of changes to easily revert to a working state.
- **Write Tests**: Combine debugging with writing tests to catch errors early.
- **Understand the Error**: Take time to understand the error message and stack trace.
- **Stay Organized**: Keep debugging sessions focused and organized.

Summary

- Debugging is the process of fixing errors found during testing.
- Various methods of debugging in Python:
  - Step/Trace through code with `print()` and `logging()`
  - Inspect objects with `type()` and `inspect`
  - Use Python debugger `pdb` with `breakpoint()` and `traceback`

Next Sessions

- Add documentation
- Distribution methods