DEBUGGING YOUR PYTHON CODE

JASON NWAKAEZE

PRINT STATEMENTS

Using print statements at various points in your code to output values of variables or messages (or both). This can help you track the flow of your program and identify the values causing unexpected behavior.

PRINT STATEMENTS

Example

```
def calculate_total(price, quantity):
    print(f"Calculating total for price: {price}, quantity: {quantity}")
    total = price * quantity
    print(f"Total: {total}")
    return total
```

DEBUGGER

Python comes with a built-in debugger called pdb. You can insert breakpoints in your code and run it using the debugger, allowing you to step through the code line by line, inspect variables, and evaluate expressions. Use commands like n (next) and p (print) to navigate.

D E B U G G E R

Example

```
import pdb

def divide(a, b):
    result = a / b
    pdb.set_trace()  # Breakpoint
    return result
```

LOGGING

The logging module enables you to create log messages of different levels (debug, info, warning, error, etc.). This can provide insights into the flow of your program and help you pinpoint where issues are occurring.

LOGGING

Example

```
import logging

logging.basicConfig(level=logging.DEBUG)

def process_data(data):
    logging.debug(f"Processing data: {data}")
    # ...rest of the code...
```

ASSERTIONS

Use the assert statement to validate assumptions about your code. If the assertion condition is False, an exception will be raised, indicating a problem in your code.

ASSERTIONS

```
def calculate_discount(total, discount_rate):
    assert 0 <= discount_rate <= 1, "Discount rate must be between 0 and 1"
    discounted_amount = total * discount_rate
    return discounted_amount
```

TRY-EXCEPT BLOCKS

Wrap sections of your code in try-except blocks to catch and handle exceptions. This prevents your program from crashing and gives you a chance to investigate the cause of the exception.

TRY-EXCEPT BLOCKS

```
try:
    result = divide(10, 0)
except ZeroDivisionError as e:
    print(f"An error occurred: {e}")
```

INTERACTIVE DEBUGGING

Some IDEs support interactive debugging, allowing you to execute code in the context of your running program. This is particularly useful when you want to experiment with code snippets.

CODE REVIEW

Sometimes a fresh pair of eyes can catch issues you might have missed. Ask a colleague or fellow programmer to review your code.

UNIT TESTS

Write unit tests using the unittest or pytest frameworks. Tests can help you catch regressions and ensure that your code behaves as expected.

IDES AND EDITORS

Integrated Development Environments (IDEs) like PyCharm, Visual Studio Code, and editors like Sublime Text offer debugging features that can help you track and fix issues more effectively.

DEBUGGER GUIS

Some Python debuggers offer graphical user interfaces that make it easier to navigate through your code and inspect variables.

STACK TRACES

When an error occurs, Python provides a stack trace that shows the sequence of function calls leading up to the error. This can help you trace back to the source of the problem.

CODE ISOLATION:

If you're dealing with a complex program, try to isolate the problematic section of code by creating a minimal, reproducible example. This can make it easier to identify the root cause.

Debugging is like retracing your steps to find your phone—it's about retracing your code's execution to find the source of an issue.
So next time you find yourself "looking for your phone" in your code, pause, think, and apply these debugging techniques to save yourself the stress and find the solution.

Follow for more interesting content.