Intermediate Python Programming – Lesson 7

Facilitated by Kent State University
Topic: Error Handling and Debugging Best Practices
Duration: 1 Hour

Learning Objectives

By the end of this lesson, participants will be able to:

- Use `try`, `except`, `else`, and `finally` blocks for error handling
- Raise and define custom exceptions
- Use debugging tools such as `pdb` and `logging`
- Apply best practices for writing robust, debuggable code

Lesson 7: Error Handling and Debugging Best Practices

I. Introduction to Errors in Python (5 minutes)

Errors in Python are handled through exceptions. Exceptions interrupt normal program flow and can be caught using structured blocks of code.

Common types of errors:

- `SyntaxError`: Issues with code syntax
- `TypeError`, `ValueError`: Wrong types or values
- `IndexError`, `KeyError`: Invalid access in sequences or mappings
- `FileNotFoundError`, `ZeroDivisionError`, etc.

Python allows us to anticipate and gracefully handle these problems using structured error handling.

II. Using try, except, else, and finally (15 minutes)

Python's structured error handling lets you catch specific errors, define fallback behavior, and clean up afterward.

Syntax:

```
try:
    # code that may raise an exception
except SomeException:
    # handle the exception
else:
    # executes if no exceptions were raised
finally:
    # always executes (cleanup code)
```

Example:

```
def safe_divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Cannot divide by zero."
    else:
        return result
    finally:
        print("Attempted division.")

print(safe_divide(10, 2))  # Output: 5.0
print(safe_divide(5, 0))   # Output: Cannot divide by zero.
```

Exercise 1:

Write a function `read_file(filename)` that returns the contents of a file. Use `try`/`except` to handle missing files and print a helpful error message.

III. Raising and Defining Custom Exceptions (10 minutes)

Sometimes, the built-in exceptions don't provide the right level of semantic meaning. You can define your own exceptions by subclassing `Exception`.

Raising an exception:

```
raise ValueError("Invalid input")
```

Defining a custom exception:

```
class InvalidUserInput(Exception):
    pass

raise InvalidUserInput("User input was not valid")
```

Custom exceptions are especially useful in larger projects for conveying application-specific error conditions.

Exercise 2:

Create a function that validates an age input. If the value is negative, raise a custom `NegativeAgeError` exception.

IV. Debugging Tools and Best Practices (20 minutes)

Writing robust code is about more than just catching errors — it's also about finding and fixing them quickly.

1. The `pdb` Debugger

Python includes a built-in debugger called `pdb`.

To pause execution and enter interactive debugging:

```
import pdb; pdb.set_trace()
```

Once inside `pdb`, you can:

- Use `n` (next), `c` (continue), `l` (list code), `p` (print variable), and `q` (quit)

2. The `logging` Module

Use `logging` instead of `print()` for real-world applications.

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info("Application started")
```

Log levels include DEBUG, INFO, WARNING, ERROR, and CRITICAL. Logging can be routed to files and filtered by level.

3. Best Practices

- Catch specific exceptions (not a blanket `except:`)
- Keep `try` blocks minimal — only wrap the statements that might fail
- Use logging instead of print for error tracing
- Never silently ignore errors

Exercise 3:

Add logging to a function that reads a number from input and converts it to an integer. Log each step and handle bad input g