Module 5 - Files, Exceptions, and Errors

Welcome to Module 5! In the previous modules, you mastered basic programming constructs, functions, and modules. Now, we're diving into two vital aspects of real-world programming: **error and exception handling** (making your programs robust) and **file handling** (making your programs persistent).

Chapter 1: Understanding Errors and Exceptions

Even the best programmers make mistakes, and real-world conditions are unpredictable. Errors are inevitable, but knowing how to anticipate and handle them gracefully is a hallmark of good programming.

1.1 What are Errors and Exceptions?

In Python, problems that occur during the execution of a program can be broadly categorized into:

1. **Syntax Errors:**
    - These are mistakes in the structure of your code that violate Python's grammar rules.
    - The Python interpreter catches these *before* your code even starts running (at "compile time" or "parse time").
    - Your program **will not run** if it has a syntax error.
    - Examples: Missing colons, unclosed parentheses, incorrect keywords.

    Python

    ```
    >>> if x > 5 # Missing colon
    SyntaxError: expected ':'

    >>> print("Hello" # Missing closing parenthesis
    SyntaxError: unexpected EOF while parsing
    ```

2. **Exceptions (Run-time Errors):**
    - These errors occur *during* the execution of your program, even if the syntax is perfectly valid.
    - They indicate that something unexpected or problematic happened that prevents the program from continuing normally.
    - If not handled, an exception will cause your program to **crash** and terminate abruptly.
    - Python provides various built-in exception types to describe different kinds of problems.

    Let's look at some common exception types:

    - **`NameError`**: Trying to use a variable or function name that hasn't been defined.

Python

```
>>> print(undefined_variable)
NameError: name 'undefined_variable' is not defined
```

- **`TypeError`**: An operation is performed on an inappropriate type.

  Python

  ```
  >>> "hello" + 5
  TypeError: can only concatenate str (not "int") to str
  ```

- **`ValueError`**: A function receives an argument of the correct type but an inappropriate value.

  Python

  ```
  >>> int("abc") # 'abc' is a string, but cannot be converted to
  int
  ValueError: invalid literal for int() with base 10: 'abc'
  ```

- **`ZeroDivisionError`**: Attempting to divide a number by zero.

  Python

  ```
  >>> 10 / 0
  ZeroDivisionError: division by zero
  ```

- **`IndexError`**: Trying to access an index that is out of bounds for a sequence (list, string, tuple).

  Python

  ```
  >>> my_list = [1, 2, 3]
  >>> my_list[3]
  IndexError: list index out of range
  ```

- **`KeyError`**: Trying to access a non-existent key in a dictionary.

  Python

  ```
  >>> my_dict = {"name": "Alice"}
  >>> my_dict["age"]
  KeyError: 'age'
  ```

- **`FileNotFoundError`**: Trying to open a file that does not exist. (We'll see this in file handling!)
- **`FileNotFoundError`**:

  Python

  ```
  >>> open("non_existent_file.txt", "r")
  ```

```
FileNotFoundError: [Errno 2] No such file or directory:
'non_existent_file.txt'
```

3. **Module Errors (`ModuleNotFoundError`, `ImportError`):**

- These occur when Python cannot find a module you are trying to import or encounters an issue during the import process itself.
- `ModuleNotFoundError` (more specific, introduced in Python 3.6+) typically means the module file simply doesn't exist or isn't in Python's search path.
- `ImportError` is a broader category that can include `ModuleNotFoundError` or other issues preventing a successful import (e.g., a syntax error *inside* the module being imported).

Python
```
>>> import non_existent_module
ModuleNotFoundError: No module named 'non_existent_module'

>>> from math import power # 'power' is not a direct function in math
module (use math.pow)
ImportError: cannot import name 'power' from 'math'
```

4. **Logic Errors:**

- These are the most insidious type of error because they do *not* cause your program to crash or raise an exception.
- Your program runs successfully from Python's perspective, but it produces **incorrect or unintended results**.
- This happens when your code logically does something different from what you intended it to do.
- Debugging logic errors often requires careful testing, tracing variable values, and reviewing your algorithm.

Python
```
# Example 1: Incorrect calculation
>>> num1 = 2
>>> num2 = 2
>>> # Intention: Add num1 and num2
>>> result = num1 * num2 # Logic error: Used multiplication instead of
addition
>>> print(result)
4
# The program runs, but if the goal was 2 + 2, the result '4' is correct
for multiplication
# but incorrect for the *intended* addition. The code *should* have been:
result = num1 + num2

# Example 2: Off-by-one in a loop
>>> # Intention: Calculate sum of numbers from 1 to 5
>>> total = 0
>>> for i in range(1, 5): # Logic error: range(1, 5) goes up to 4, not 5
...     total += i
...
>>> print(total)
10 # Correct sum for 1+2+3+4, but not for 1+2+3+4+5 (which should be 15)
```

```
# Example 3: Incorrect condition
>>> age = 17
>>> if age > 18: # Intention: Check if eligible to vote (>=18). Logic
error: Used > instead of >=
...     print("Eligible to vote")
... else:
...     print("Not eligible to vote")
Not eligible to vote # Correct for condition, but wrong if 18 should be
eligible.
```

Key characteristic of logic errors: The program executes without crashing, but the *output is not what you expected* based on your problem requirements.

Understanding these different error types helps you anticipate potential issues and write code to handle them gracefully.

1.2 The Exception Handling Mechanism (`try-except`)

To prevent your program from crashing when an exception occurs, Python provides the **`try-except`** block. This mechanism allows you to "try" to execute a block of code, and if an exception occurs, to "catch" it and execute a different block of code (the exception handler) instead of crashing.

Syntax:

Python
```
try:
    # Code that might raise an exception
except SomeExceptionType:
    # Code to execute if SomeExceptionType occurs in the 'try' block
    # This is the exception handler
```

- `try`: This block contains the code that you want to monitor for exceptions.
- `except SomeExceptionType`: If an exception of `SomeExceptionType` (e.g., `ValueError`, `ZeroDivisionError`) occurs within the `try` block, the execution jumps immediately to this `except` block.
- The code inside the `except` block is executed, and then the program continues normally *after* the entire `try-except` structure.

Example: Handling `ZeroDivisionError` (Console)

Python
```
>>> try:
...     result = 10 / 0 # This line will cause a ZeroDivisionError
...     print(f"Result: {result}") # This line will NOT be executed
... except ZeroDivisionError:
...     print("Error: Cannot divide by zero!")
...
Error: Cannot divide by zero!
>>> print("Program continues here.") # Program did not crash
Program continues here.
```

Example: Handling Invalid User Input (Script)

Let's create a program that asks for a number, but robustly handles cases where the user types non-numeric input. Create `robust_input.py`:

Python
```
# robust_input.py

try:
    num_str = input("Enter an integer: ")
    number = int(num_str) # This might raise a ValueError if input is not
an int
    print(f"You entered: {number}")
except ValueError: # Catch only ValueError
    print("Invalid input! Please enter a whole number.")

print("End of program.")
```

Interaction:

```
# Scenario 1: Valid input
Enter an integer: 42
You entered: 42
End of program.

# Scenario 2: Invalid input
Enter an integer: hello
Invalid input! Please enter a whole number.
End of program.
```

1.3 Handling Specific Exceptions

It's good practice to handle specific exceptions rather than catching all possible errors. This allows you to provide precise error messages and appropriate recovery actions.

Multiple `except` Blocks:

You can use multiple `except` blocks to handle different types of exceptions in different ways. Python will check `except` blocks in order from top to bottom.

Python
```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    result = num1 / num2
    print(f"Result: {result}")
except ValueError: # Handles non-integer input
    print("Error: Please enter valid numbers.")
except ZeroDivisionError: # Handles division by zero
    print("Error: You cannot divide by zero!")
except: # A general 'except' block (catches any other unhandled exception)
    print("An unexpected error occurred.")
```

Important Note on Order: Place the most specific exception handlers first, followed by more general ones. If `except Exception:` (which catches *all* exceptions) were first, it would catch everything, and subsequent specific `except` blocks would never be reached.

Catching the Exception Object (`as e`):

You can capture the exception object itself using `as e` (or any other variable name). This object often contains useful information about the error.

Python
```
try:
    data = [1, 2, 3]
    index = int(input("Enter an index: "))
    value = data[index] # Could raise IndexError
    print(f"Value at index {index}: {value}")
except IndexError as e: # 'e' will hold the IndexError object
    print(f"Error accessing list: {e}")
    print("The index you entered is out of range.")
except ValueError as e: # 'e' will hold the ValueError object
    print(f"Error converting input: {e}")
    print("Please enter a valid integer for the index.")
```

1.4 The `else` and `finally` Blocks

The `try-except` structure can be extended with `else` and `finally` blocks for more control over execution flow.

- **`else` Block:**
  - The code in the `else` block is executed **only if the `try` block completes successfully**, without any exceptions being raised.
  - It's a good place for code that should only run if no errors occurred.
- **`finally` Block:**
  - The code in the `finally` block is **always executed**, regardless of whether an exception occurred in the `try` block or not, and regardless of whether it was handled by an `except` block.
  - It's typically used for **cleanup operations** that *must* happen, such as closing files or releasing network connections, even if an error crashes the main logic.

Syntax with `else` and `finally`:

Python
```
try:
    # Code that might raise an exception
except SpecificException1:
    # Handler for SpecificException1
except SpecificException2:
    # Handler for SpecificException2
else:
    # Code to run if NO exception occurs in the 'try' block
finally:
    # Code that ALWAYS runs, regardless of exceptions
```

Example: Using `try-except-else-finally` (Script)

Create `full_exception_example.py`:

Python
```
# full_exception_example.py

def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Caught an error: Cannot divide by zero!")
        return None # Return None to indicate failure
    except TypeError:
        print("Caught an error: Invalid types for division (must be
numbers)!")
        return None
    else: # This block runs ONLY if no exception occurred in the 'try'
block
        print("Division successful!")
        return result
    finally: # This block ALWAYS runs
        print("--- Division attempt finished. ---")

# Test cases
print("Scenario 1: Successful division")
res1 = divide_numbers(10, 2)
if res1 is not None:
    print(f"Result: {res1}\n")

print("Scenario 2: Division by zero")
res2 = divide_numbers(10, 0)
if res2 is not None:
    print(f"Result: {res2}\n")

print("Scenario 3: Type error")
res3 = divide_numbers(10, "two")
if res3 is not None:
    print(f"Result: {res3}\n")
```

Interaction:

```
Scenario 1: Successful division
Division successful!
--- Division attempt finished. ---
Result: 5.0

Scenario 2: Division by zero
Caught an error: Cannot divide by zero!
--- Division attempt finished. ---

Scenario 3: Type error
Caught an error: Invalid types for division (must be numbers)!
--- Division attempt finished. ---
```

1.5 Raising Exceptions (`raise`)

Sometimes, you might need to force an exception to occur if a certain condition is met (or not met). The `raise` statement allows you to trigger an exception manually. This is useful for:

- **Input Validation:** If a function receives invalid arguments.
- **Custom Errors:** Creating and raising your own specific error types (more advanced).
- **Signaling Problems:** Indicating that a situation occurred that your function cannot handle.

Syntax:

Python
```
raise ExceptionType("Optional error message")
```

Example: Raising a `ValueError` for Invalid Age (Console)

Python
```
>>> def set_age(age):
...     if not isinstance(age, int): # Check if it's an integer
...         raise TypeError("Age must be an integer.")
...     if age < 0 or age > 120:
...         raise ValueError("Age must be between 0 and 120.")
...     print(f"Age set to: {age}")
...
>>> set_age(30)
Age set to: 30
>>> set_age(200) # This will raise a ValueError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in set_age
ValueError: Age must be between 0 and 120.
>>> set_age("abc") # This will raise a TypeError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in set_age
TypeError: Age must be an integer.
>>>
```

Key Takeaway: Exception handling is about making your programs resilient. Instead of crashing, they can provide meaningful feedback to the user or recover from unexpected situations.

Chapter 2: File Handling - Interacting with Files

Most useful programs need to store or retrieve data beyond the current memory of the program. This is where **file handling** comes in. It allows your Python programs to interact with files on your computer's storage (hard drive, SSD, etc.).

2.1 Introduction to File I/O (Input/Output)

- **Persistence:** Data stored in variables is lost when your program ends. Files provide a way to store data **persistently**, meaning it remains available even after your program finishes running or your computer shuts down.
- **Input/Output (I/O):**
    - **Input (Reading):** Your program reads data from a file.

    - **Output (Writing):** Your program writes data to a file.
- **File Types:**
    - **Text Files:** Store data as plain text (e.g., `.txt`, `.csv`, `.py`, `.html`). This is what we'll focus on primarily. When you open a text file in a text editor, you can read its content directly.
    - **Binary Files:** Store data in a format specific to the program or system (e.g., images, videos, executables, `.docx`, `.xlsx`). You typically can't read these directly in a text editor.

2.2 Opening and Closing Files (`open()` and `close()`)

Before you can read from or write to a file, you must **open** it. After you're done, you **must close** it.

The `open()` Function:

The `open()` function is used to open a file. It returns a **file object** (sometimes called a file handle), which you then use to interact with the file.

- **Syntax:** `file_object = open(filename, mode)`
    - `filename`: A string representing the path to the file you want to open (e.g., `"my_data.txt"`, `"data/report.csv"`).
    - `mode`: A string specifying how the file will be used. This is crucial!

Common File Modes:

Mode	Meaning	Action if File Exists	Action if File Doesn't Exist	Cursor Position
`r`	**Read (default)**: Opens the file for reading.	Reads from start	Raises `FileNotFoundError`	Beginning
`w`	**Write**: Opens the file for writing. If the file exists, its content is **truncated (deleted)**. If the file doesn't exist, a new one is created.	Truncates	Creates new	Beginning
`a`	**Append**: Opens the file for writing. If the file exists, new data is written to the **end** of the file. If the file doesn't exist, a new one is created.	Appends	Creates new	End
`x`	**Exclusive Creation**: Opens the file for exclusive creation. If the file already exists, the operation fails and raises a `FileExistsError`. Useful to ensure you're creating a new file.	Raises `FileExistsError`	Creates new	Beginning

`r+`	**Read and Write**: Opens the file for both reading and writing. The cursor is placed at the beginning of the file. No truncation occurs. Use carefully: if you write, you might overwrite existing data if you don't move the cursor.	Reads from start	Raises `FileNotFoundError`	Beginning
`w+`	**Read and Write (Truncate)**: Opens the file for both reading and writing. If the file exists, it's truncated. If it doesn't exist, a new one is created. Similar to `w`, but allows reading too.	Truncates	Creates new	Beginning
`a+`	**Read and Append**: Opens the file for both reading and writing. If the file exists, new data is appended. If it doesn't exist, a new one is created. Reading starts from the beginning, but writing always happens at the end.	Appends	Creates new	Beginning (Read), End (Write)

The `close()` Method:

It is **absolutely critical** to close a file after you are done with it.

- **Syntax:** `file_object.close()`
- **Why close?**
  - **Resource Release:** It frees up system resources associated with the file.
  - **Data Integrity:** Ensures that all buffered data (data temporarily held in memory) is actually written to the disk. If a program crashes or exits without closing an opened file, data might be lost or corrupted.

Example: Opening and Closing (Script)

Create `file_open_close.py`:

Python
```
# file_open_close.py

# Open a file in write mode ('w')
# If 'my_file.txt' exists, its content will be deleted!
# If it doesn't exist, it will be created.
file_object = open("my_file.txt", "w")
print("File 'my_file.txt' opened in write mode.")

# Perform some operations (writing will come later)
# ...

# Close the file
```

```
file_object.close()
print("File 'my_file.txt' closed.")

# Now try opening in read mode ('r')
# This will raise an error if 'my_file.txt' doesn't exist (e.g., if you run
this alone)
try:
    file_object_read = open("my_file.txt", "r")
    print("File 'my_file.txt' opened in read mode.")
    file_object_read.close()
    print("File 'my_file.txt' closed.")
except FileNotFoundError:
    print("Error: 'my_file.txt' not found for reading (might not have been
created yet).")
```

Interaction:

```
File 'my_file.txt' opened in write mode.
File 'my_file.txt' closed.
File 'my_file.txt' opened in read mode.
File 'my_file.txt' closed.
```

After running, you will see a new (empty) file named `my_file.txt` in the same directory as your script.

2.3 Writing Data to a File (`write()` and `writelines()`)

Once a file is opened in a write (`'w'`) or append (`'a'`) mode, you can write data to it.

`file_object.write(string)`:

- Writes a string to the file.
- **Does not** automatically add a newline character (`\n`) at the end. You must explicitly add it if you want new lines.

`file_object.writelines(list_of_strings)`:

- Writes a sequence (like a list) of strings to the file.
- **Does not** automatically add newline characters. Each string in the list is written consecutively.

Example: Writing Simple Text (Script)

Create `write_data.py`:

Python

```
# write_data.py

# Open in 'w' mode: creates file or overwrites if it exists
file_write = open("greetings.txt", "w")

file_write.write("Hello, Python!\n") # Add newline manually
file_write.write("This is a new line.\n")
file_write.write("We are learning file handling.\n")
```

```
lines_to_write = ["First item\n", "Second item\n", "Third item\n"]
file_write.writelines(lines_to_write) # writelines expects strings with
newlines

file_write.close()
print("Data written to greetings.txt")

# Check content of greetings.txt:
# Hello, Python!
# This is a new line.
# We are learning file handling.
# First item
# Second item
# Third item
```

Interaction:

```
Data written to greetings.txt
```

After running, a file named `greetings.txt` will be created with the specified content.

2.4 Reading Data from a File (`read()`, `readline()`, `readlines()`, Iterating)

Once a file is opened in read mode (`'r'`), you can retrieve data from it.

`file_object.read(size):`

- Reads the entire content of the file if `size` is omitted.
- If `size` is provided, it reads up to `size` characters (or bytes for binary files).
- The file cursor moves to the position after the read data.

`file_object.readline():`

- Reads a single line from the file, including the newline character (`\n`) at the end if present.
- Returns an empty string (`''`) when the end of the file is reached.

`file_object.readlines():`

- Reads all lines from the file and returns them as a list of strings. Each string in the list represents a line and includes the newline character.

Best Practice: Iterating Directly Over File Object

For reading large files, iterating directly over the file object in a `for` loop is the most memory-efficient and Pythonic way. It reads one line at a time, without loading the entire file into memory.

Python

```
for line in file_object:
    # Process each line
```

Example: Reading from a File (Script)

Create `read_data.py`:

Python
```
# read_data.py

# First, ensure greetings.txt exists with some content (from previous
example)
# Or create it if you're running this part separately
# with open("greetings.txt", "w") as f:
#     f.write("Line 1\nLine 2\nLine 3\n")

# --- Reading the entire file ---
print("--- Reading entire file ---")
file_read_all = open("greetings.txt", "r")
content = file_read_all.read()
print(content)
file_read_all.close()

# --- Reading line by line using readline() ---
print("\n--- Reading line by line (readline()) ---")
file_read_line = open("greetings.txt", "r")
line1 = file_read_line.readline()
line2 = file_read_line.readline()
print(f"First line: {line1.strip()}") # .strip() removes leading/trailing
whitespace, including '\n'
print(f"Second line: {line2.strip()}")
file_read_line.close()

# --- Reading all lines into a list (readlines()) ---
print("\n--- Reading all lines into a list (readlines()) ---")
file_read_list = open("greetings.txt", "r")
all_lines = file_read_list.readlines()
for line in all_lines:
    print(f"List item: {line.strip()}")
file_read_list.close()

# --- Best Practice: Iterating over file object (most common and efficient)
---
print("\n--- Reading efficiently (for loop) ---")
file_efficient_read = open("greetings.txt", "r")
for line in file_efficient_read:
    print(f"Processed: {line.strip()}")
file_efficient_read.close()
```

Interaction:

```
--- Reading entire file ---
Hello, Python!
This is a new line.
We are learning file handling.
First item
Second item
Third item

--- Reading line by line (readline()) ---
First line: Hello, Python!
Second line: This is a new line.
```

```
--- Reading all lines into a list (readlines()) ---
List item: Hello, Python!
List item: This is a new line.
List item: We are learning file handling.
List item: First item
List item: Second item
List item: Third item

--- Reading efficiently (for loop) ---
Processed: Hello, Python!
Processed: This is a new line.
Processed: We are learning file handling.
Processed: First item
Processed: Second item
Processed: Third item
```

2.5 Appending Data to a File (`a` mode)

When you want to add new content to an existing file without deleting its original content, you open it in **append mode (`'a'`)**. If the file doesn't exist, it will be created.

Example: Adding Log Entries (Script)

Create `log_appender.py`:

Python
```
# log_appender.py
import datetime # To get current timestamp

def log_message(message):
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    log_entry = f"{timestamp} - {message}\n"

    # Open in 'a' (append) mode. Cursor starts at the end.
    log_file = open("application.log", "a")
    log_file.write(log_entry)
    log_file.close()
    print(f"Logged: {log_entry.strip()}")

print("--- Starting Application Logging ---")
log_message("Application started successfully.")
log_message("User 'Alice' logged in.")
log_message("Processing data...")
log_message("Application shutting down.")
print("--- Logging Finished ---")
```

Interaction:

```
--- Starting Application Logging ---
Logged: 2025-06-17 13:54:23 - Application started successfully.
Logged: 2025-06-17 13:54:23 - User 'Alice' logged in.
Logged: 2025-06-17 13:54:23 - Processing data...
Logged: 2025-06-17 13:54:23 - Application shutting down.
--- Logging Finished ---
```

(Timestamps will vary based on current time.) After running, check `application.log`. Each time you run the script, new lines will be added to the end of the file.

2.6 Reading and Adding Data (`r+` mode)

The `r+` mode opens a file for both reading and writing. The file pointer (cursor) is placed at the beginning. This means if you write immediately, you will overwrite the beginning of the file. To add data, you often need to read to the end or use `seek()` to move the cursor.

Important Note: This mode can be tricky for beginners because writing will happen at the current cursor position. If you want to read *then* add to the end, `a+` mode is often safer, or you need to explicitly move the cursor using `file_object.seek(0, 2)` (moves to end of file). We'll stick to a simpler case.

Example: Read content, then append a new line (simpler approach with `a+` or separate operations is usually better)

Python
```
# read_and_add.py

# Using r+ carefully
file_rplus = open("sample.txt", "r+") # Create or overwrite for clean start
file_rplus.write("Original line 1\nOriginal line 2\n")
file_rplus.seek(0) # Move cursor back to beginning to read
print("Content after initial write:")
print(file_rplus.read())

file_rplus.write("NEW LINE ADDED at start!\n") # This overwrites part of
original content!
file_rplus.seek(0) # Move cursor back to beginning to read
print("\nContent after writing in r+ mode (can overwrite):")
print(file_rplus.read())
file_rplus.close()

# Demonstrating a common pattern: Read all, then write all modified content
print("\n--- Better pattern for modifying/adding based on content ---")
try:
    with open("data_to_modify.txt", "w") as f: # Create initial content
        f.write("Item A\nItem B\n")

    # Read all content
    with open("data_to_modify.txt", "r") as f:
        existing_content = f.read()
    print("Existing content:\n", existing_content)

    # Modify/Add to content
    new_content = existing_content + "Item C (added after read)\n"

    # Write all content back (this overwrites the original file)
    with open("data_to_modify.txt", "w") as f:
        f.write(new_content)
    print("\nContent after modification and re-write:")
    with open("data_to_modify.txt", "r") as f:
        print(f.read())

except FileNotFoundError:
    print("File not found for modification.")
```

Explanation: For beginners, it's generally clearer to separate reading and writing operations, or to use `a+` if you want to read from the beginning but always write at the end. `r+` is powerful but requires careful management of the file cursor.

2.7 The `with` Statement for File Handling (Context Manager)

Managing `open()` and `close()` can be error-prone, especially if exceptions occur. If an error happens between `open()` and `close()`, the `close()` might never be called, leading to resource leaks or corrupted files.

The **`with` statement** (also known as a **context manager**) is the **recommended way** to handle files in Python. It guarantees that the file is properly closed, even if errors occur, making your code cleaner and safer.

Syntax:

Python
```
with open(filename, mode) as file_object:
    # Perform file operations here
    # The file is automatically closed when exiting this 'with' block,
    # regardless of how the block is exited (normal completion, error,
etc.)
```

Example: Rewriting Examples with `with` (Script)

Create `file_with_statement.py`:

Python
```
# file_with_statement.py

# --- Writing data with 'with' ---
print("--- Writing with 'with' ---")
try:
    with open("my_notes.txt", "w") as f: # 'f' is our file object
        f.write("Meeting notes for today.\n")
        f.write("Important points:\n")
        f.write("- Discuss Module 4\n")
        f.write("- Plan next steps\n")
    print("Data written to my_notes.txt successfully.")
except IOError as e: # Catch potential errors during file operation
    print(f"Error writing to file: {e}")

# --- Reading data with 'with' ---
print("\n--- Reading with 'with' ---")
try:
    with open("my_notes.txt", "r") as f: # 'f' is our file object
        content = f.read()
        print("Content of my_notes.txt:")
        print(content)
except FileNotFoundError:
    print("Error: my_notes.txt not found.")
except IOError as e:
    print(f"Error reading from file: {e}")

# --- Appending data with 'with' ---
print("\n--- Appending with 'with' ---")
```

```
try:
    with open("my_notes.txt", "a") as f:
        f.write("\n--- Added later ---\n")
        f.write("Action item: Follow up on tasks.\n")
    print("Data appended to my_notes.txt successfully.")
except IOError as e:
    print(f"Error appending to file: {e}")

# Verify appended content
print("\n--- Verifying appended content ---")
try:
    with open("my_notes.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    print("Error: my_notes.txt not found.")
```

Interaction:

```
--- Writing with 'with' ---
Data written to my_notes.txt successfully.

--- Reading with 'with' ---
Content of my_notes.txt:
Meeting notes for today.
Important points:
- Discuss Module 4
- Plan next steps

--- Appending with 'with' ---
Data appended to my_notes.txt successfully.

--- Verifying appended content ---
Meeting notes for today.
Important points:
- Discuss Module 4
- Plan next steps

--- Added ---
Action item: Follow up on tasks.
```

Conclusion: Always use the `with` statement for file operations unless you have a very specific reason not to. It simplifies your code and handles resource management robustly.

Chapter 3: Coding Challenges for Files, Exceptions, and Errors

It's time to apply your knowledge of robust error handling and file I/O!

Challenge 1: Robust Calculator

Goal: Create a simple calculator that performs addition, subtraction, multiplication, and division. Use exception handling to make it robust against common errors.

Concepts Covered: `try-except`, `ValueError`, `ZeroDivisionError`, `input()`.

Requirements:

1. Continuously prompt the user for two numbers and an operation (`+`, `-`, `*`, `/`).
2. Use a `try-except` block to handle potential `ValueError` if the user enters non-numeric input. Print an appropriate error message.
3. Use a specific `except` block to handle `ZeroDivisionError` if the user attempts to divide by zero. Print an appropriate error message.
4. Perform the calculation and print the result.
5. Allow the user to type `exit` (case-insensitive) at any prompt to quit the calculator.

Example Interaction:

```
Simple Calculator (type 'exit' to quit)
Enter first number: 10
Enter operation (+ - * /): /
Enter second number: 0
Error: Cannot divide by zero!

Enter first number: hello
Error: Invalid input. Please enter a valid number.

Enter first number: 20
Enter operation (+ - * /): *
Enter second number: 3
Result: 60.0

Enter first number: exit
Goodbye!
```

Challenge 2: Secure File Reader

Goal: Write a program that asks the user for a filename and then tries to read and print its content. Use exception handling to gracefully manage common file-related errors.

Concepts Covered: `try-except`, `FileNotFoundError`, `IOError`, `with open()`, `read()`.

Requirements:

1. Prompt the user to enter a filename.
2. Use a `try-except` block to open and read the file's content.
3. Specifically handle `FileNotFoundError` if the file doesn't exist, printing a user-friendly message.
4. Handle a general `IOError` (or `Exception`) for any other potential file-related issues (e.g., permission errors).
5. If the file is successfully read, print its content.
6. Ensure the file is always closed, even if errors occur (use `with` statement).

Example Interaction:

```
Enter filename to read: non_existent.txt
```

```
Error: The file 'non_existent.txt' was not found. Please check the name and
try again.

Enter filename to read: my_notes.txt (assuming this file exists from
previous examples)
Content of my_notes.txt:
Meeting notes for today.
Important points:
- Discuss Module 4
- Plan next steps
--- Added later ---
Action item: Follow up on tasks.
```

Challenge 3: Daily Journal/Log Appender

Goal: Create a simple command-line journal application that allows the user to add new entries, timestamped, to a daily log file.

Concepts Covered: File appending (`'a'` mode, or `with open('filename', 'a')`), `datetime` module, `input()`.

Requirements:

1. Define a constant for the log filename, e.g., `JOURNAL_FILE = "daily_journal.txt"`.
2. Use a `while` loop to continuously prompt the user:
    - "Enter your journal entry (or 'quit' to exit):"
3. If the user types `quit` (case-insensitive), exit the loop and print a goodbye message.
4. For each entry:
    - Get the current timestamp (e.g., `datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")`).
    - Format the entry: `[TIMESTAMP] - [USER_ENTRY]\n`.
    - Open the `JOURNAL_FILE` in append mode (`'a'`) using a `with` statement.
    - Write the formatted entry to the file.
    - Print a confirmation message: "Entry added."

Example Interaction:

```
--- Daily Journal ---
Enter your journal entry (or 'quit' to exit): Had a great day learning
Python.
Entry added.
Enter your journal entry (or 'quit' to exit): Finished the file handling
section.
Entry added.
Enter your journal entry (or 'quit' to exit): quit
Goodbye! Journal entries saved.
```

Content of `daily_journal.txt` after interaction (timestamps will vary):

```
2025-06-17 13:54:23 - Had a great day learning Python.
2025-06-17 13:54:23 - Finished the file handling section.
```

Challenge 4: Simple Data Processor (Numbers)

Goal: Read a list of numbers from one file, calculate their sum and average, and write these results to another file.

Concepts Covered: File reading (`'r'` mode, iterating lines), file writing (`'w'` mode), `try-except` for `ValueError` during number conversion, basic arithmetic.

Requirements:

1. **Preparation:** Manually create a file named `numbers.txt` in the same directory as your script, and put some numbers in it, one per line:
2. `10`
3. `25`
4. `5`
5. `40`
6. Define `INPUT_FILE = "numbers.txt"` and `OUTPUT_FILE = "results.txt"`.
7. Read each line from `numbers.txt`:
    - Convert each line to a float. Use a `try-except ValueError` to skip any lines that are not valid numbers, printing a warning message for those.
    - Accumulate the sum of valid numbers and count how many valid numbers were processed.
8. Calculate the average. Handle `ZeroDivisionError` if no valid numbers were found.
9. Write the sum and average to `results.txt`.
    - Example format:
    - `Total sum: 80.0`
    - `Average: 20.0`
10. Use `with` statements for all file operations.

Example Interaction:

```
(Assuming numbers.txt contains: 10, 25, 5, 40)
Processing numbers from numbers.txt...
Results saved to results.txt.
```

Content of `results.txt`:

```
Total sum: 80.0
Average: 20.0
```

Consider if `numbers.txt` has invalid data (e.g., "10\nabc\n20"):

```
Processing numbers from numbers.txt...
Warning: Skipping invalid number on line: abc
Results saved to results.txt.
```

Content of `results.txt`:

```
Total sum: 30.0
Average: 15.0
```

Challenge 5: Basic User Manager (Append/Read)

Goal: Create a program that allows you to register new users (add to file) or view existing users (read from file).

Concepts Covered: File appending (`'a'`), file reading (`'r'`), `with` statement, `input()`, `if-elif-else` for menu.

Requirements:

1. Define a `USER_FILE = "users.txt"`.
2. Implement a loop that displays a menu:
    - 1. Register New User
    - 2. View All Users
    - 3. Exit
3. **Register New User:**
    - Prompt for a username and password.
    - Append `username,password\n` to `users.txt`.
    - Use a `with` statement.
    - Print confirmation.
4. **View All Users:**
    - Open `users.txt` for reading using a `with` statement.
    - Read and print each line (each user's data).
    - Handle `FileNotFoundError` if `users.txt` doesn't exist yet, printing "No users registered yet."
5. **Exit:** Break the loop and print a goodbye message.
6. Handle invalid menu choices.

Example Interaction:

```
--- User Management System ---
1. Register New User
2. View All Users
3. Exit
Enter your choice: 1
Enter username: alice
Enter password: password123
User 'alice' registered.

1. Register New User
2. View All Users
3. Exit
Enter your choice: 1
Enter username: bob
Enter password: securepass
User 'bob' registered.

1. Register New User
2. View All Users
3. Exit
Enter your choice: 2
```

```
Users:
alice,password123
bob,securepass

1. Register New User
2. View All Users
3. Exit
Enter your choice: 3
Goodbye!
```