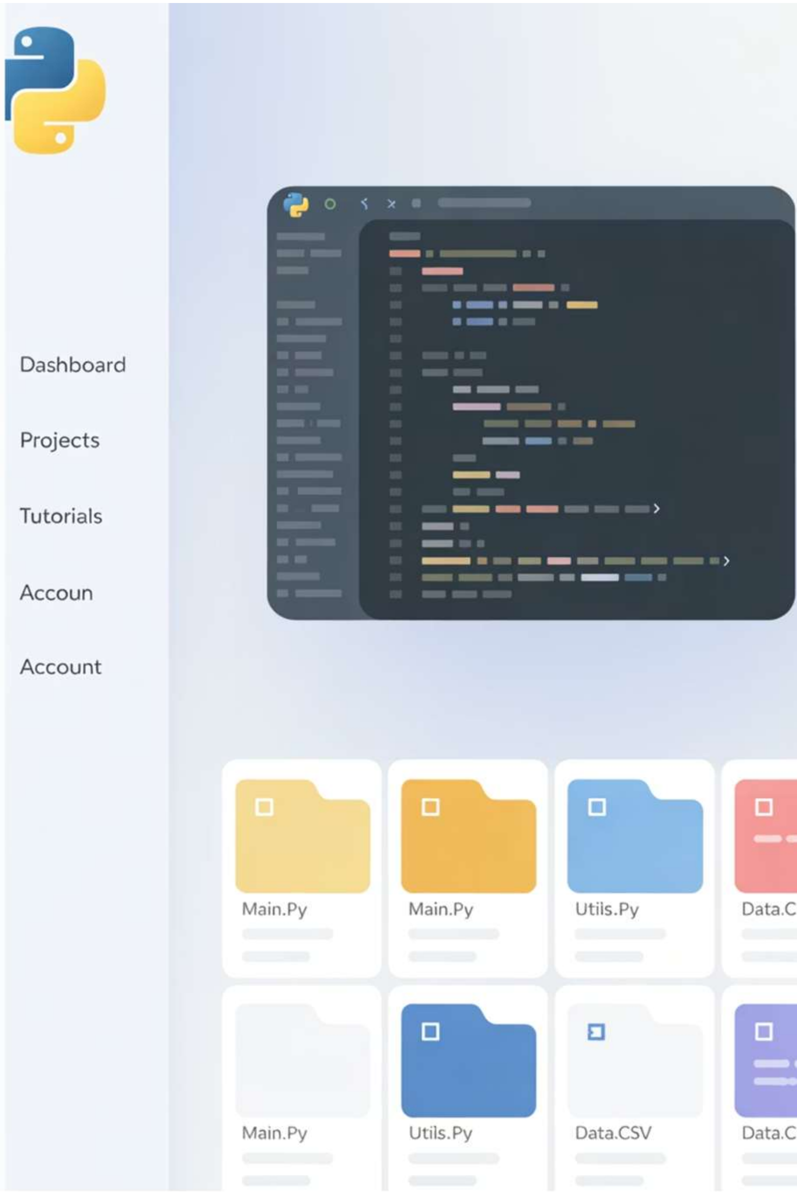
Files and Exceptions in Python: Essential Concepts and Examples

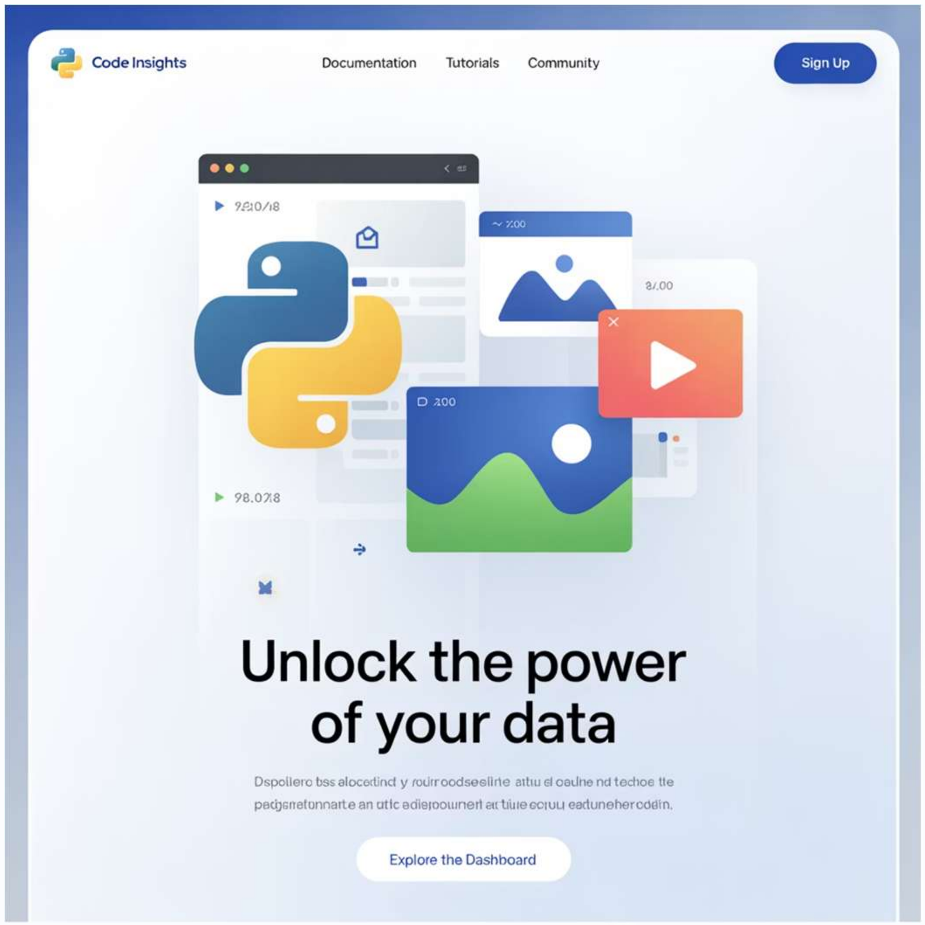
A practical guide to working with files and handling errors in Python programming



What Is File Handling in Python?

File handling is a fundamental skill for any Python developer, enabling you to:

- Process data from external sources
- Store information persistently between program executions
- Generate reports and log program activity
- Work with configuration files



File Operations: The Basics

Opening Files

```
file = open("data.txt", "r")
```

The **open()** function takes a filename and mode parameter

Reading Files

```
content = file.read()
```

Read entire file content as a single string

Writing Files

```
file.write("Hello Python")
```

Write string data to the file

Closing Files

```
file.close()
```

Always close files to prevent resource leaks

The Context Manager (`with` Statement)

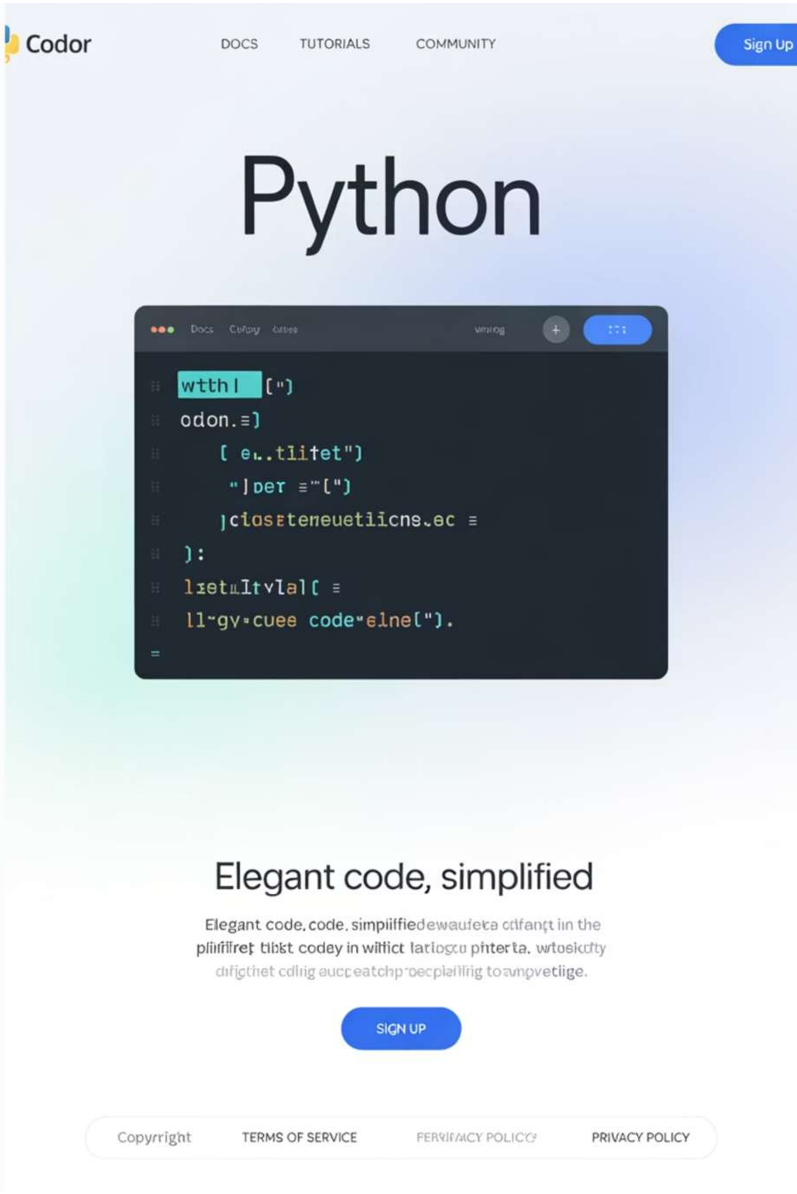
The Problem with Manual Closing

```
file = open("data.txt", "r")# If an error occurs here, file may never closecontent = file.read()file.close()  # May never execute
```

The Solution: Context Manager

```
with open("data.txt", "r") as file: content = file.read()# File automatically closes when leaving the block
```

The **with** statement ensures files are properly closed even if exceptions occur



Reading Files: Simple Example

Reading an Entire File

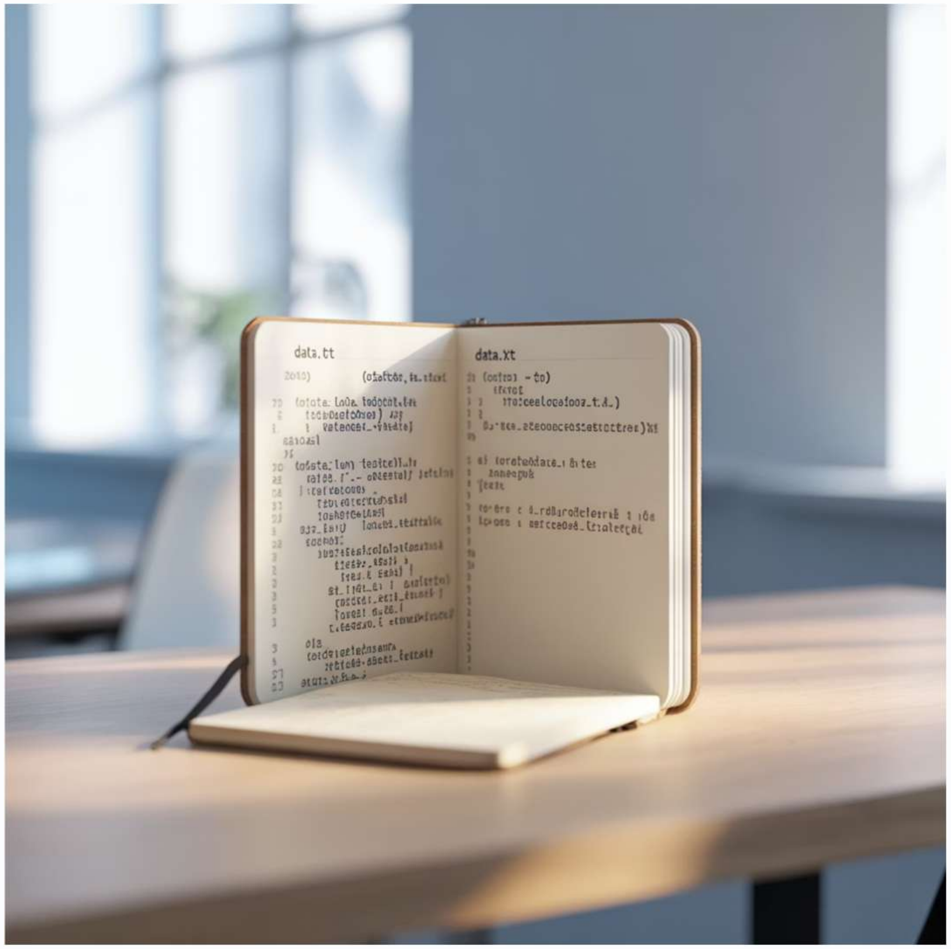
```
with open("example.txt", "r") as file:    # Read all content at once
content = file.read()    print(content)
```

Reading Line by Line

```
with open("example.txt", "r") as file:    # Loop through each line
for line in file:
          print(line.strip())
```

Reading File into list

```
with open("example.txt", "r") as file:    # Loop through each line
for line in file:
          print(line.strip())
```



Writing to Files: Basic Usage

Open in Write Mode

```
with open("output.txt", "w")
as file:
```

The "w" mode creates a new file or overwrites an existing one

Write Content

```
    file.write("Hello, Python!")
```

The write() method adds string data to the file

File Is Saved

Content is written and file is automatically closed when the with block ends

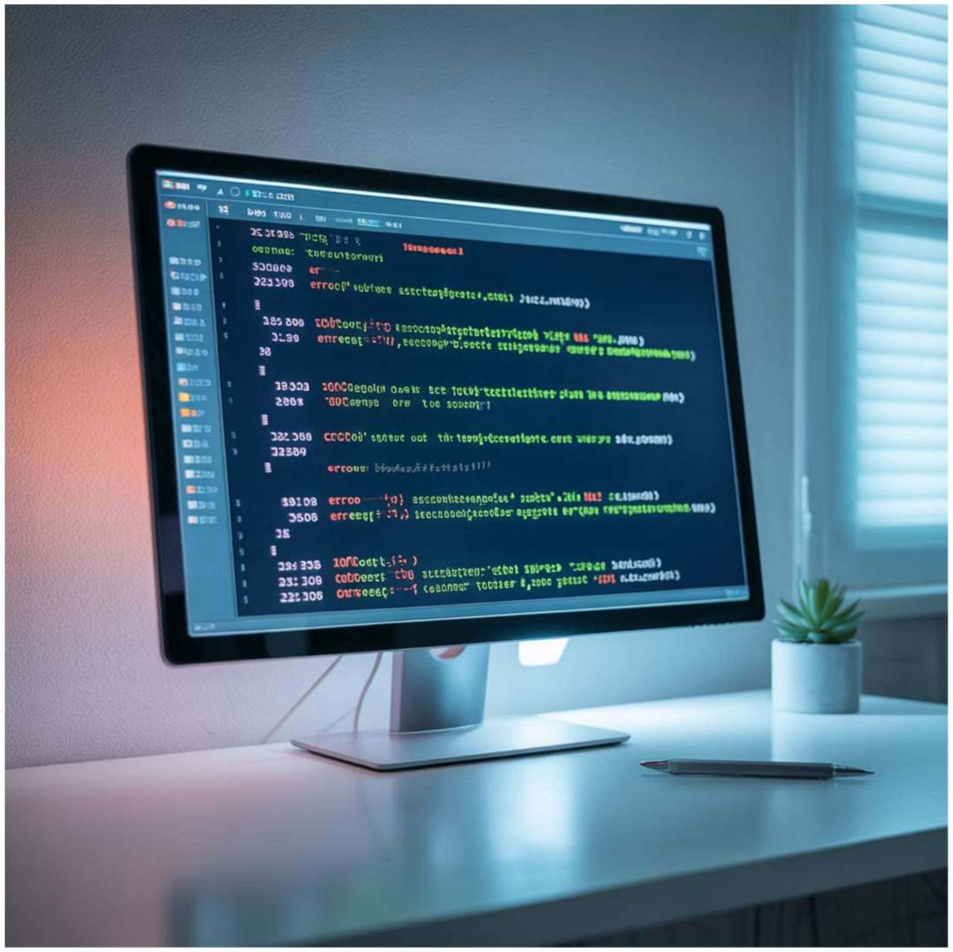
Warning: Write mode will erase any existing content in the file!

Appending Data to Files

Example: Log File

```
import datetimewith open("app_log.txt", "a") as log_file:
          timestamp = datetime.datetime.now()
          log_entry = f"{timestamp}:
                    Application started\n"
                    log_file.write(log_entry)
```

The "a" mode preserves existing content and adds new data at the end of the file



Deleting Files: Simple Example

Using os Module to Check File Existence

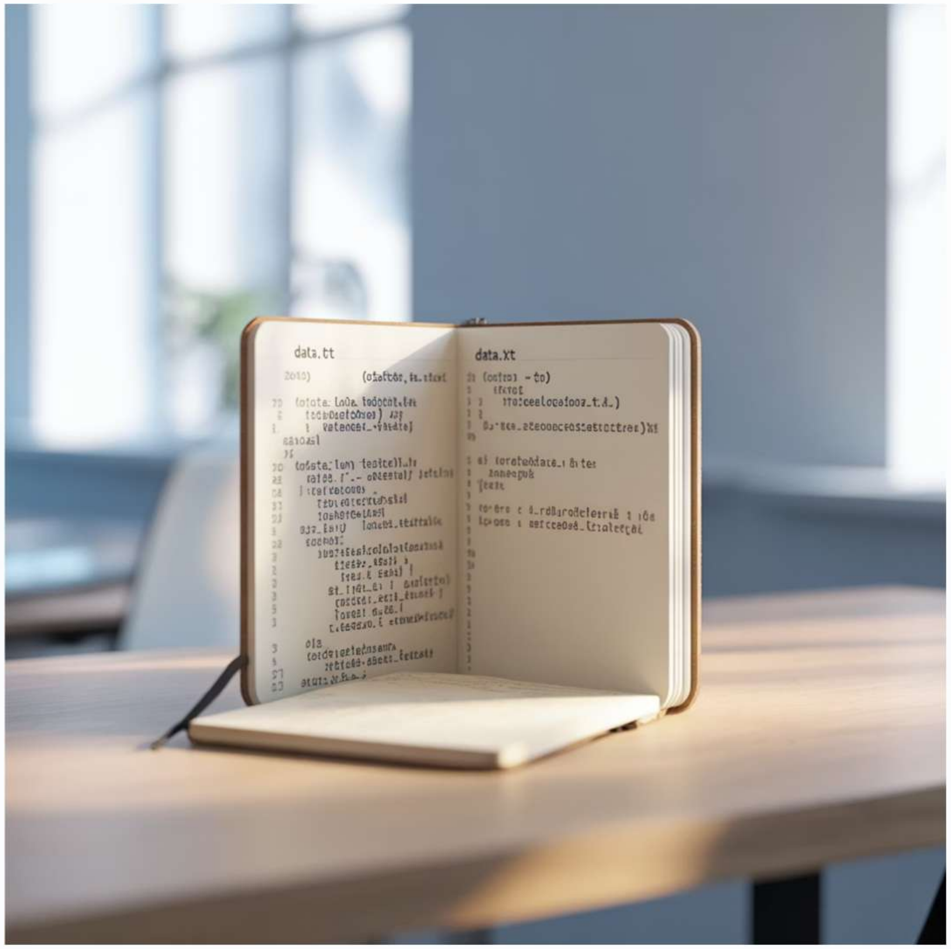
```
import os

if os.path.exists("example.txt"):
    print("File exists.")
else:
    print("File does not exist.")
```

Deleting a File

```
import os

if os.path.exists("example.txt"):
    os.remove("example.txt")
    print("File deleted.")
else:
    print("File not found.")
```

Using Try-Except While Handling Files

```
try:
    with open("example.txt", "r") as file:
        print(file.read())
except FileNotFoundError:
    print("File not found!")
```



Deleting Files: Simple Example

Downloading a file

```
import requests
import zipfile
import os

def download_file(file_url,save_path):
    bflag = False
    try:
        response = requests.get(file_url, stream=True)  # Use 
stream=True for large files
        response.raise_for_status()  # Raise an exception for bad 
status codes (4xx or 5xx)

        with open(save_path, 'wb') as file:
            for chunk in response.iter_content(chunk_size=8192):
                file.write(chunk)
        print(f"File downloaded successfully to: {save_path}")
        bflag = True
    except requests.exceptions.RequestException as e:
        print(f"Error during download: {e}")
        bflag = False
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
        bflag = False

if __name__ == '__main__':
    file_url = 
"http://ratings.fide.com/download/standard_rating_list_xml.zip"  # 
Replace with the actual URL
    save_path = "standard_rating.zip"  # Name and path for the 
downloaded file
    bflag = download_file(file_url,save_path)
```

Unzipping a file

```
def unzip_file(save_path):
    # Specify the path to your zip file
    zip_file_path = save_path

    # Specify the directory where you want to extract the contents 
(optional)
    # If not provided, contents will be extracted to the current working 
directory.
    extraction_path = '.\\'

    with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
        zip_ref.extractall(extraction_path)

    '''
    if os.path.exists("standard_rating.zip"):
        os.remove("standard_rating.zip")
        print("File deleted.")
    else:
        print("File not found.")
    '''

    print(f"Contents of '{zip_file_path}' extracted to 
'{extraction_path}’”)


if __name__ == '__main__':
        save_path = "standard_rating.zip"  # Name and path for the 
downloaded file
    if os.path.exists("standard_rating.zip"):
        print("File exists.")
        unzip_file(save_path)
    else:
        print("File does not exist.")
```

Best Practices in File Management

Always Use Context Managers

The **with** statement ensures files are properly closed even if exceptions occur

Process Large Files Efficiently

Read and process large files line by line instead of loading them entirely into memory

Use Proper Error Handling

Anticipate and handle potential file operation errors with try-except blocks

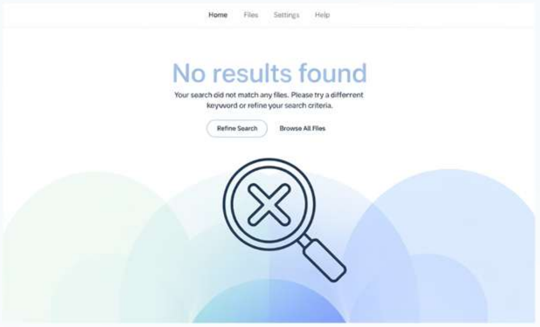
Validate File Paths

Use **os.path** or **pathlib** to check if files exist before attempting operations

Why Do Exceptions Occur in File Handling?

File Not Found

The specified file doesn't exist at the given path



Your search did not match any files. Please try a differrent keyword or refine your search criteria.

Permission Issues

Your program lacks the necessary rights to access or modify the file

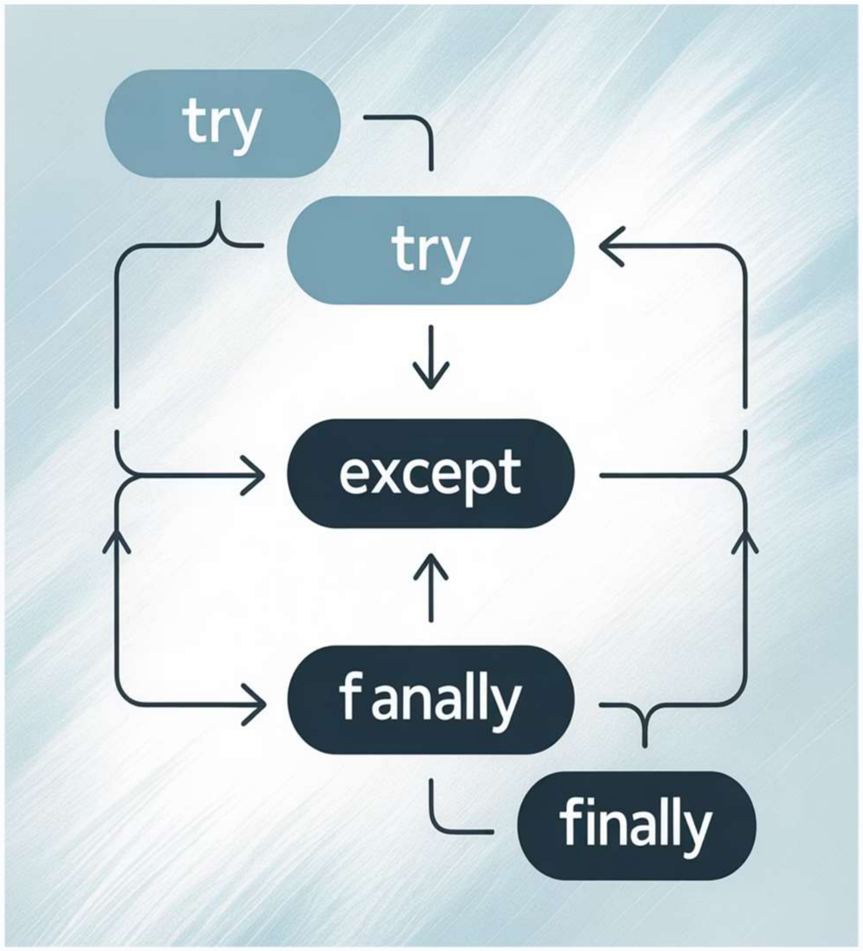
System Issues

Disk errors, network failures, or other hardware/OS problems

Introduction to Exception Handling

Basic Structure

```
try:    # Code that might cause an exception
     with open("data.txt", "r") as file:
          content = file.read()except:    # Code to execute if an exception occurs
          print("An error occurred!")
finally:    # Code that always runs
     print("Operation attempted")
```

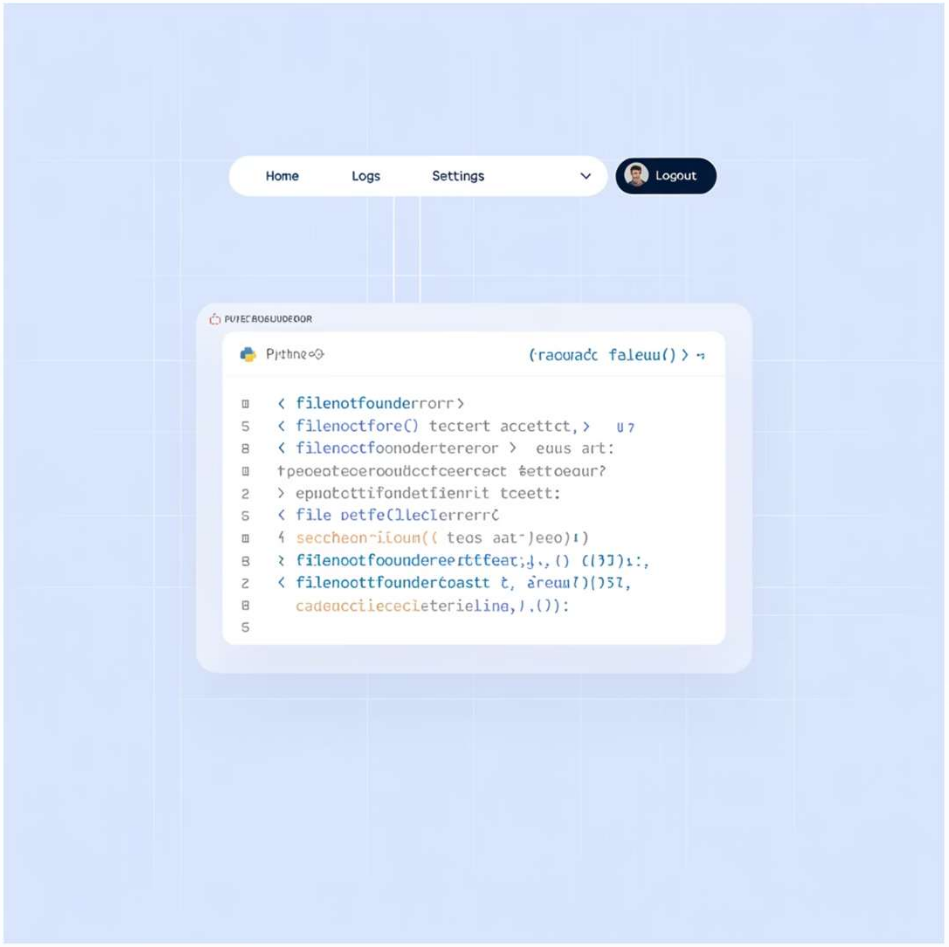


FileNotFoundError: Example and Explanation

What Triggers This Error?

- Trying to open a file that doesn't exist
- Incorrect path specification
- Typos in filename

```
# This will raise FileNotFoundErrorwith
open("missing.txt", "r") as file:
          content = file.read()
```



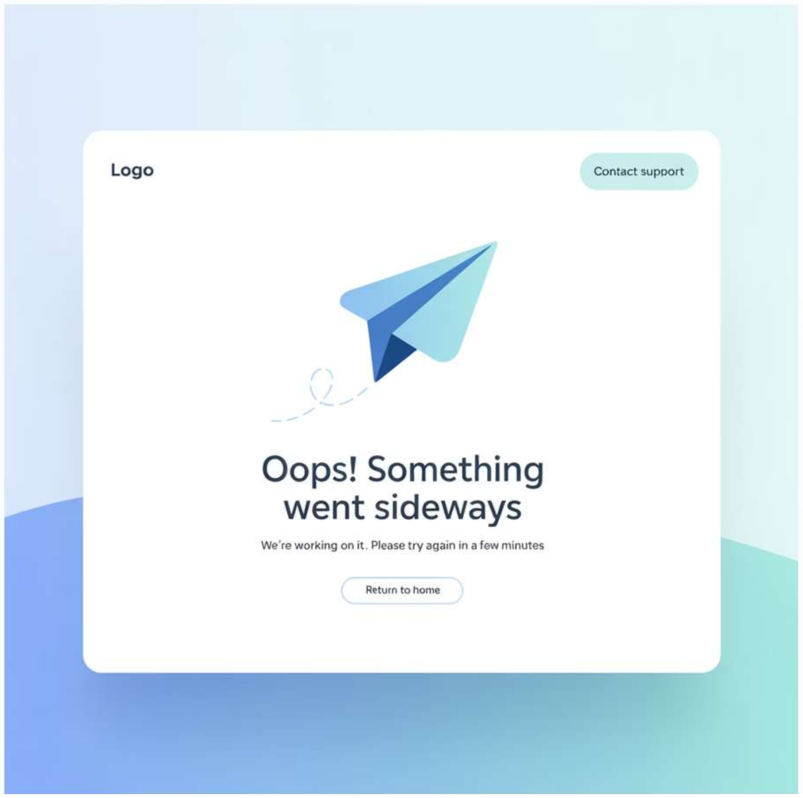
```
(racuradc faleuu() ) ⁊
< filenotfounderrorr>
< filenoctfore() tectert accettct, >   U 7
< filencctfoonoderteretor >  euus art:
tpeceotecerooudcctceercact tettoeaur?
> epuatcttifondetfienrit tceett:
< file petfe(llec lerrerrC
< seccheon-iloun(( teos aat-Jeeo)!)
< filenootfoounderee rtffeat;,l., () ((33)1:,
< filenoottfoundertoastt t, areuul)()3l,
  cadeucclieceCleterielina,).()):
```

Handling FileNotFoundError in Code

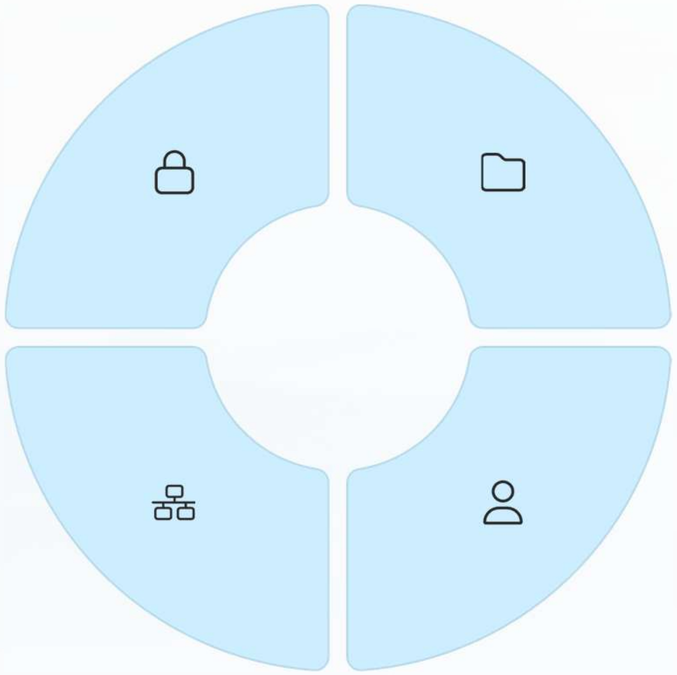
```
try:    with open("user_data.txt", "r") as file:
           user_info = file.read()
           process_user_data(user_info)
except FileNotFoundError:
           print("User data file not found!")
```

Key Benefits

- Program continues running despite the error
- User sees a helpful message instead of a crash
- Alternative actions can be taken automatically
- Opportunity to recover from the error


Logo
Contact support
Oops! Something went sideways
We're working on it. Please try again in a few minutes
Return to home

PermissionError: What and Why



Protected Files

Attempting to modify system files without admin rights

Read-Only Resources

Trying to write to a file marked as read-only

Network Restrictions

Writing to network locations without proper authentication

User Permissions

Accessing files owned by another user account

```
# Example that might raise PermissionErrorwith
open("/etc/passwd", "w") as file: # System file on Unix/Linux
       file.write("This will fail without root privileges")
```

General I/O Errors: IOError

Disk Full Errors

No space left on device to write data

Corrupted Files

File structure is damaged and cannot be read properly

Network Timeouts

Remote file operations timing out due to connection issues

Hardware Failures

Physical storage device problems preventing access

Note: In modern Python, IOError is now an alias for OSError.

Catching Multiple File Exceptions

```
try:
          with open("important_data.txt", "r") as file:
          data = file.read()
          process_data(data)
except FileNotFoundError:
          print("Data file is missing! Check your file path.")
          create_empty_data_file()
except PermissionError:
          print("You don't have permission to access this file.")
          request_admin_access()
except IOError:
          print("A hardware or system error occurred.")
          log_error_details()
except Exception as e:
          print(f"An unexpected error occurred: {e}")
          send_error_report()
```

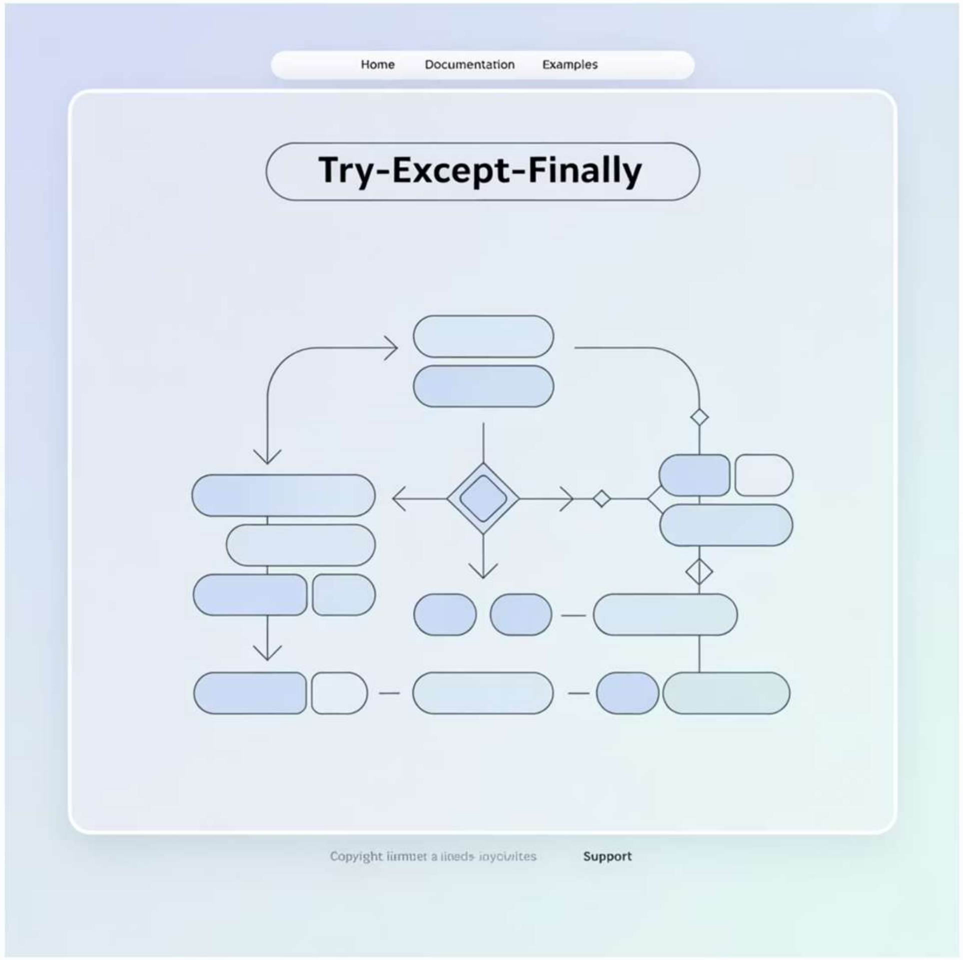
Arrange exceptions from most specific to most general. Always handle specific exceptions before catching general ones.

The Finally Block: Ensuring Cleanup

Purpose of Finally

- Execute cleanup code regardless of exceptions
- Close resources even if errors occur
- Ensure consistency in program state

```
file = None
try:
            file = open("data.txt", "r")
            content = file.read()
            process_data(content)
except FileNotFoundError:
            print("File not found!")
finally:
            if file is not None:
                        file.close()
                        print("File closed successfully")
```



The **finally** block executes whether an exception occurred or not, making it perfect for resource

Defensive Programming Strategies

1 Validate Inputs First

```
import os

if os.path.exists("data.txt"):
    with open("data.txt", "r") as f:
        data = f.read()
else:
    print("File not found. Creating empty file...")
    with open("data.txt", "w") as f:
        f.write("")
```

2 Check Permissions Before Access

```
import os

file_path = "config.ini"
if os.access(file_path, os.W_OK):
    with open(file_path, "w") as f:
        f.write("debug=True\n")
else:
    print(f"No write permission for {file_path}")
```

3 Implement Comprehensive Logging

```
import logging

logging.basicConfig(filename="app.log",
 level=logging.INFO)
try:
 with open("data.txt", "r") as f:
 data = f.read()
except Exception as e:
 logging.error(f"Error processing file: {e}")
 raise
```

Useful Real-World Patterns

Robust Batch File Processing

```
def process_files(file_list):
    results = []
    errors = []

    for filename in file_list:
        try:
            with open(filename, "r") as f:
                data = f.read()
                result = process_data(data)
                results.append((filename, result))
        except Exception as e:
            errors.append((filename, str(e)))
            # Log error but continue processing
            continue

    # Report summary
    print(f"Processed {len(results)} files successfully")
    print(f"Encountered {len(errors)} errors")

    return results, errors
```

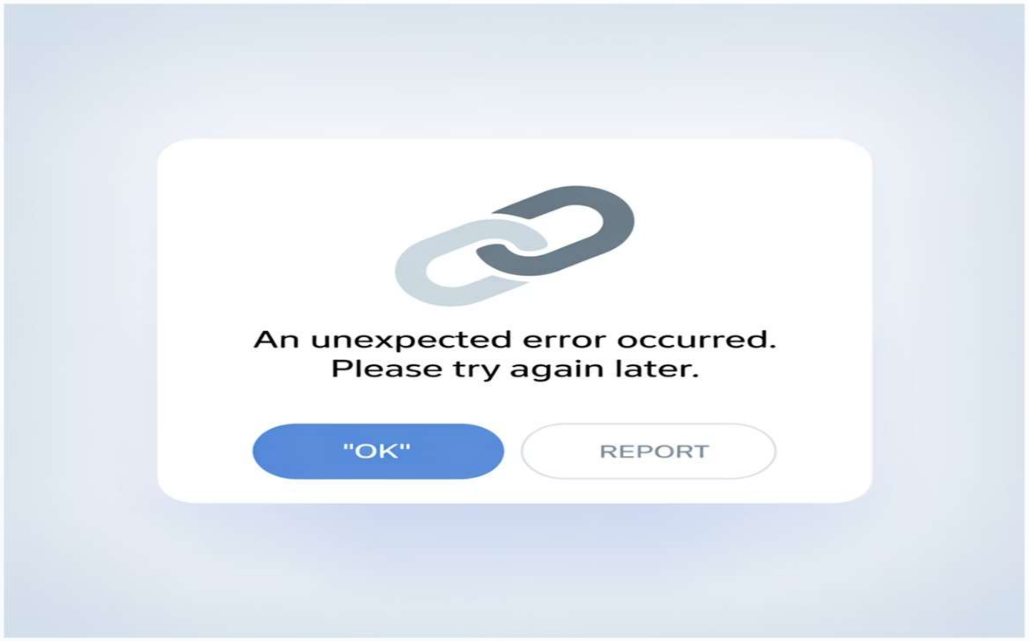
Automatic Backup Before Writing

```
import shutil
import os

def safe_write(filename, data):
 # Create backup if file exists
 if os.path.exists(filename):
 backup = filename + ".bak"
 shutil.copy2(filename, backup)
 print(f"Backup created: {backup}")

 try:
 with open(filename, "w") as f:
 f.write(data)
 return True
 except Exception as e:
 print(f"Write failed: {e}")
 # Restore from backup if write failed
 if os.path.exists(filename + ".bak"):
 shutil.copy2(filename + ".bak", filename)
 print("Restored from backup")
 return False
```

User-Defined Exceptions in File Handling

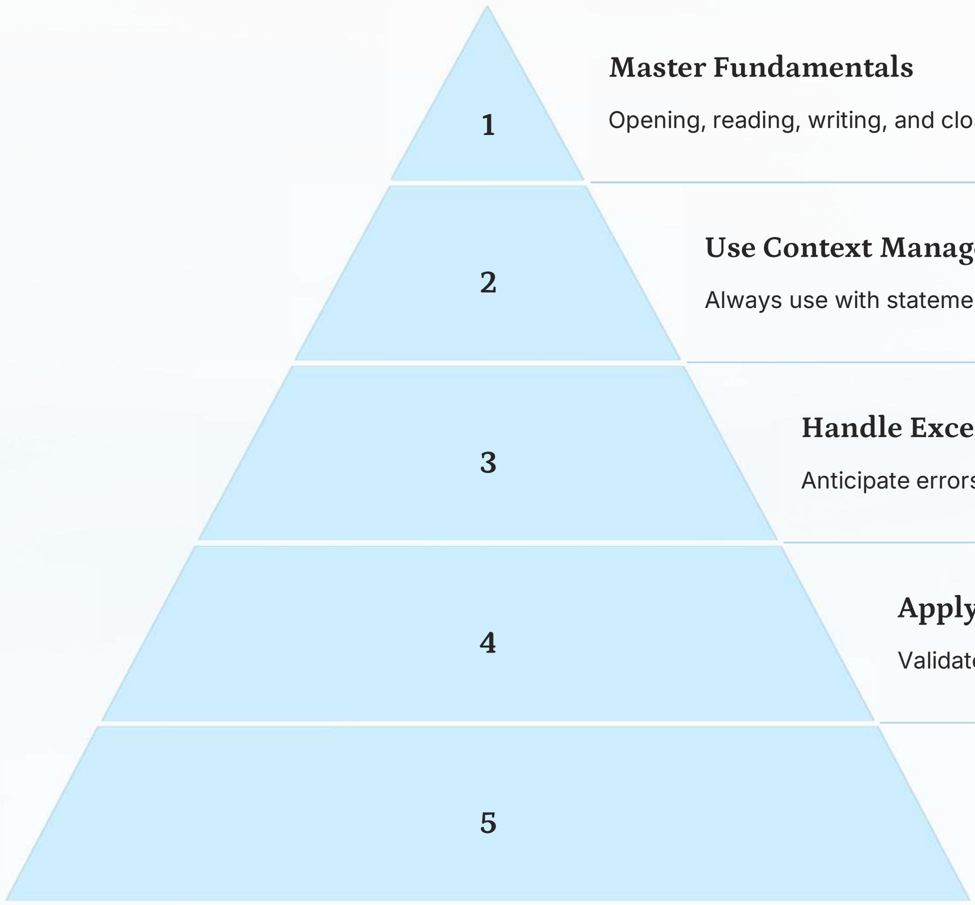
Creating Custom Exceptions

```
class FileFormatError(Exception):
    """Raised when file format is incorrect"""
    pass

class ConfigError(Exception):
    """Raised for configuration file issues"""
    pass

def read_config(filename):
    try:
        with open(filename, "r") as f:
            content = f.read()

        if not content.strip().startswith("[CONFIG]"):
            raise FileFormatError(
                "Invalid config file format")

        # Parse configuration...

    except FileNotFoundError:
        raise ConfigError("Config file not found")
    except FileFormatError as e:
        raise ConfigError(f"Format error: {e}")
```


An unexpected error occurred. Please try again later.

Custom exceptions provide several benefits:
- More specific error information
- Better organization of error types
- Application-specific error handling
- Improved debugging experience
- They are particularly valuable in larger applications where generic Python exceptions may not provide enough context about what went wrong.

Conclusion and Key Takeaways



Master Fundamentals

Opening, reading, writing, and closing files

Use Context Managers

Always use with statements to ensure proper file handling

Handle Exceptions

Anticipate errors and implement specific exception handling

Apply Best Practices

Validate paths, check permissions, and implement logging for robust file operations

Build Resilient Applications

Combine file handling with exception management to create programs that gracefully handle any file-related situation