Files & Error Handling

Python Course · Module 6

ElnurBDa

Module 6 — Files & Error Handling

In this module we learn how to **read and write files** and **handle errors gracefully**.

We will learn:

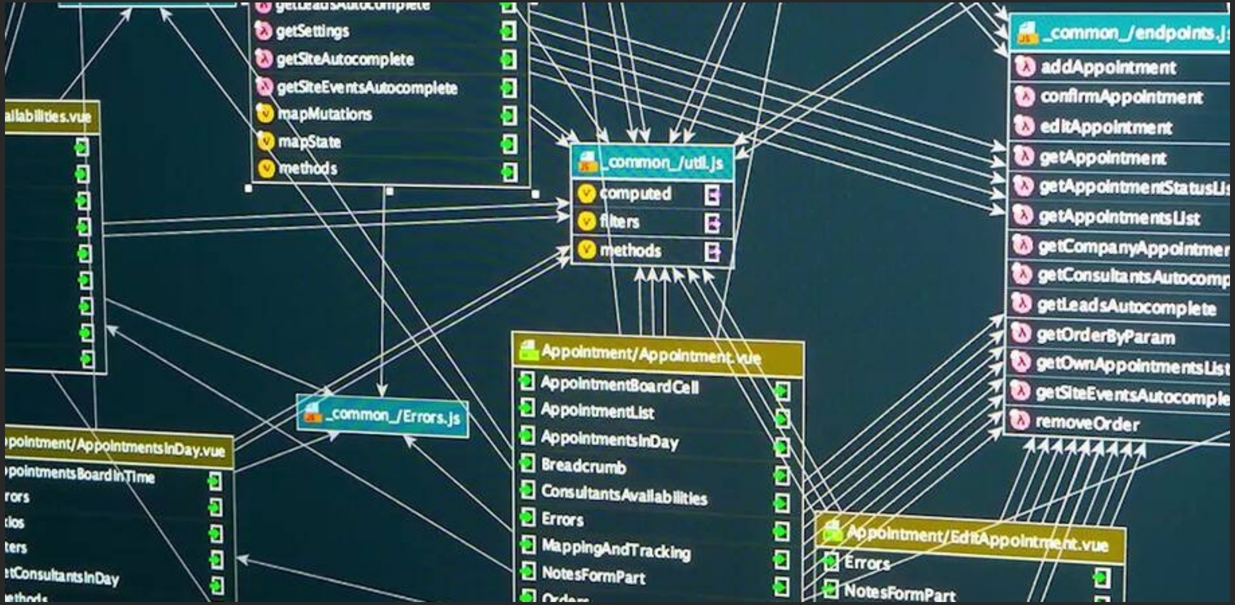
- Reading & writing files
- Handling errors with `try` / `except`
- Working with CSV files
- Best practices for error handling

Files & Persistent Storage

Programs normally store data in **RAM** (temporary).
Files store data on **disk** (persistent).

Why use files?

- Save logs
- Export reports
- Config files
- Simple data exchange



File Paths & Modes

To work with a file, Python needs:

- **Path** (where it is)
- **Mode** (what we want to do)

Common modes:

```
"r"   read (error if missing)
"w"   write, truncate file
"a"   append to end
"r+"  read & write
"rb"  read binary
"wb"  write binary
```

Always prefer **absolute paths** in bigger projects.

Opening Files Safely — with

Use with to open files — it auto-closes them.

```python
# reading a text file
path = "notes.txt"

try:
    with open(path, "r", encoding="utf-8") as f:
        content = f.read()
        print("File content:")
        print(content)
except FileNotFoundError:
    print("File not found:", path)
```

[finished]

```
File content:
some test text is here
yay!
```

Benefits:

- No need to call close()
- Works well with exceptions

Reading Files – Variants

```
with open("notes.txt", "r", encoding="utf-8") as f:
    all_text = f.read()        # whole file
    print("LEN:", len(all_text))
```

[finished]

```
LEN: 28
```

```
with open("notes.txt", "r", encoding="utf-8") as f:
    first_line = f.readline()  # one line
    print("FIRST:", first_line)
```

[finished]

```
FIRST: some test text is here

```

```
with open("notes.txt", "r", encoding="utf-8") as f:
    lines = f.readlines()      # list of lines
    print("COUNT:", len(lines))
```

[finished]

```
COUNT: 2
```

Choose based on file size and use-case.

Writing & Appending

```python
# overwrite or create
with open("log.txt", "w", encoding="utf-8") as f:
    f.write("First run\n")
    f.write("Program started successfully.\n")
```

──────────────────── [finished] ────────────────────

```python
# append
with open("log.txt", "a", encoding="utf-8") as f:
    f.write("Another run...\n")
```

──────────────────── [finished] ────────────────────

Notes:

- "w" **deletes** old content
- "a" keeps content and adds at the end

Working with CSV Files

CSV (Comma-Separated Values) is a simple text format for tables.

```python
import csv

rows = [
    ["id", "username", "score"],
    [1, "admin", 99],
    [2, "guest", 42],
]

with open("users.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerows(rows)

with open("users.csv", "r", encoding="utf-8") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

[finished]

```
['id', 'username', 'score']
['1', 'admin', '99']
['2', 'guest', '42']
```

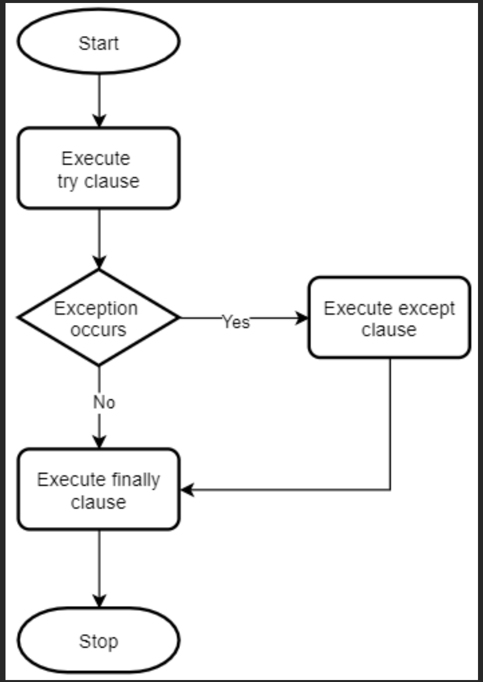
Exceptions – What & Why

An **exception** is a runtime error that stops normal execution.

Common sources:

- Invalid input
- Missing files
- Network errors
- Database problems

Goal: **fail safely**, show useful messages, keep program controllable.



Basic try / except

```python
# text = input("Enter a number: ")
text = "Salam"

try:
    n = int(text)
    print("Squared:", n * n)
except ValueError:
    print("That was not a valid integer!")
```

[finished]

```
That was not a valid integer!
```

Flow:

1. Run code in try block
2. If exception matches except, handle it
3. Program continues instead of crashing

Error Handling Patterns in Bigger Scripts

As programs grow, we want **clear places** where errors are handled.

Typical pattern:

```python
def run():
    try:
        # 1) read config
        # 2) open files / database
        # 3) main logic
        ...
    except FileNotFoundError as e:
        print("Missing file:", e)
    except ValueError as e:
        print("Bad data:", e)
```

Benefits:

- All top-level errors are handled in **one place**
- Internal functions can either handle or **raise** exceptions

In real projects we often move these messages to **logging** instead of `print()`.

0_o

```python
# text = input("Enter a number: ")
text = "3"

try:
    n = int(text)
    print("Squared:", n * n)
except ValueError:
    print("That was not a valid integer!")
```

[finished]

```
Squared: 9
```

Catching Multiple Exceptions

```python
def read_number_from_file(path):
    try:
        with open(path, "r", encoding="utf-8") as f:
            text = f.read().strip()
            return int(text)
    except FileNotFoundError:
        print("File does not exist:", path)
    except ValueError:
        print("File did not contain a valid number.")
```

You can also group them:

```python
except (FileNotFoundError, PermissionError) as e:
    print("File access problem:", e)
```

---

else and finally

```python
path = "config.txt"

try:
    f = open(path, "r", encoding="utf-8")
except FileNotFoundError:
    print("Config missing")
else:
    print("Config loaded:", f.readline().strip())
    f.close()
finally:
    print("This always runs (cleanup, logs, etc.)")
```

[finished]

```
Config missing
This always runs (cleanup, logs, etc.)
```

- `else`: runs if **no** exception happened
- `finally`: runs **always**, used for cleanup

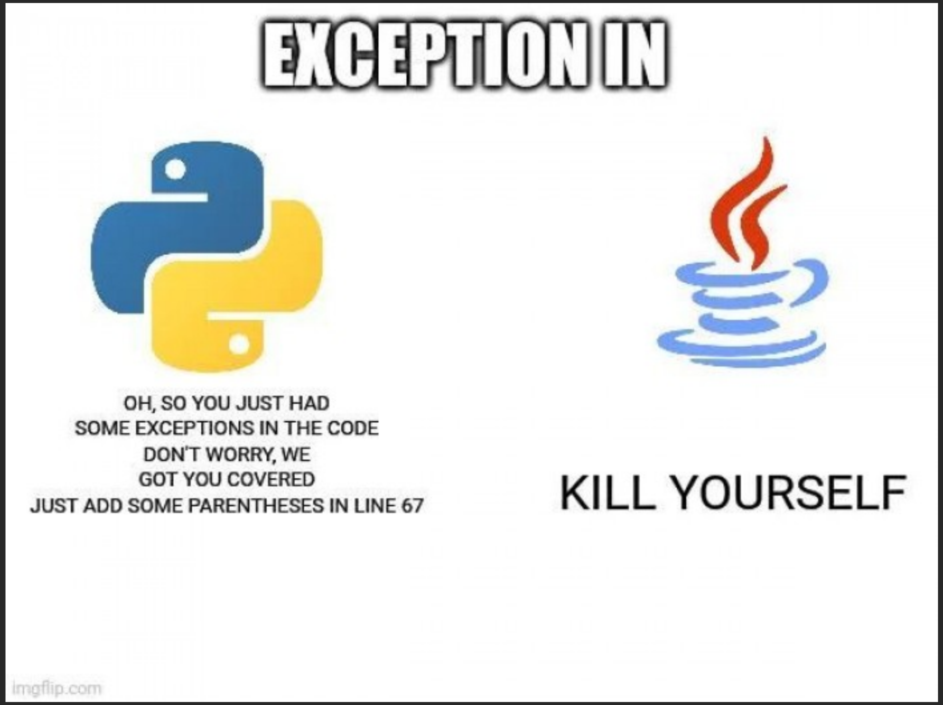
Mini Task

Create a **log tracker** program:

1. Ask user for their name and action
2. Write to activity.log with timestamp
3. Handle file errors gracefully
4. Read and display all logs
5. Add option to clear logs (with confirmation)

Bonus:

- Save logs as CSV with csv module
- Add logging levels (INFO, ERROR, WARNING)
- Filter logs by action or date


EXCEPTION IN
OH, SO YOU JUST HAD SOME EXCEPTIONS IN THE CODE DON'T WORRY, WE GOT YOU COVERED JUST ADD SOME PARENTHESES IN LINE 67
KILL YOURSELF
imgflip.com