Exception Handling in Python

PALLAVI VAIDYA

Agenda

- Exceptions
- Why exception handling?
- Types of Exceptions
- Else and finally
- Raise
- Custom Exception
- Best Practices

Exceptions

An **exception** is an error that occurs during the execution of a program, disrupting the normal flow.

For example:

```
print(10 / 0)   # ZeroDivisionError
```

Why Exception Handling?

Prevents Crashes – Keeps program running smoothly.

Graceful Recovery – Allows retrying or using default values.

User-Friendly Messages – Instead of scary error traces, users see clean messages.

Error Logging – Helps record errors for debugging.

Resource Cleanup – Files, network connections, databases get closed properly (via `finally`).

Robust Applications – Real-world software (banking apps, e-commerce, servers) must handle errors safely.

Why Exception Handling?

```
pin = int(input("Enter your 4-digit PIN: "))

print("PIN accepted:", pin)
```

If the user enters `"abcd"` instead of a number → **ValueError** → ATM program crashes!

That would be terrible in real life.

Why It Matters in Real Life?

ATMs → Must not crash on wrong input.

Banking Apps → Must not crash if server/database is unavailable.

E-commerce → Must not crash if one product fails to load.

👉 Instead, the program should **handle errors gracefully**, inform the user, log the error, and continue working.

Try.. except

```
try:
    # risky code (might cause error)
except SomeException:
    # code that runs if error happens
```

Else block

The **`else` block runs only if no exception occurs** in the `try` block.

It is useful when you want to separate *risky code* (inside `try`) from *normal follow-up code* (inside `else`).

`try` → risky code

`except` → handles specific errors

`else` → runs only if no error

finally block

The `finally` block **always runs**, no matter what:

if an exception happens,

if no exception happens,

even if there's a `return` in the `try` or `except`.

Common use: **cleanup code** (closing files, database connections, releasing resources).

raise

`raise` is a Python keyword used to **manually trigger an exception.**

You can either raise:

A **built-in exception** like `ValueError`, `ZeroDivisionError`, etc.

A **custom exception** (your own class or a built-in exception with a message).

Custom Exceptions

Python has built-in exceptions like `ValueError`, `ZeroDivisionError`, `FileNotFoundError`, etc.

Sometimes, **your program has special rules** that built-in exceptions don’t cover.

You can **create your own exception class** by inheriting from Python’s `Exception` class.

Best Practices

Use Finally or Context Managers for Cleanup

Ensure resources like files, database connections, or network sockets are always released.

`finally` always runs, even if an exception occurs.

Use Else for Code That Should Run Only If No Exception Occurs

Keeps risky code separate from normal flow.

Makes code cleaner and easier to read.

Best Practices

Catch Specific Exceptions

Always handle only the exceptions you expect (e.g., `ValueError`, `FileNotFoundError`).

Avoid generic `catch-all` unless absolutely necessary.

Don’t Leave Except Blocks Empty

Always handle the error meaningfully or log it.

Silent exceptions hide bugs and make debugging difficult.

Best Practices

Provide Clear and Descriptive Error Messages

Helps understand what went wrong.

Useful for debugging and for users.

Use Custom Exceptions When Needed

For domain-specific rules (e.g., banking rules, business logic).

Makes error handling more meaningful and organized.

Best Practices

Keep Try Blocks Small

Only put code that may fail inside `try`.

Avoid wrapping too much code to prevent catching unrelated errors.

Don’t Use Exceptions for Normal Flow Control

Exceptions should signal errors, not control normal program behavior.

Best Practices

Log Exceptions in Production

Instead of printing, log exceptions to files or monitoring systems for future debugging.

Don’t Silently Suppress Exceptions

Avoid ignoring errors with empty `except` blocks.

Always handle or log them.