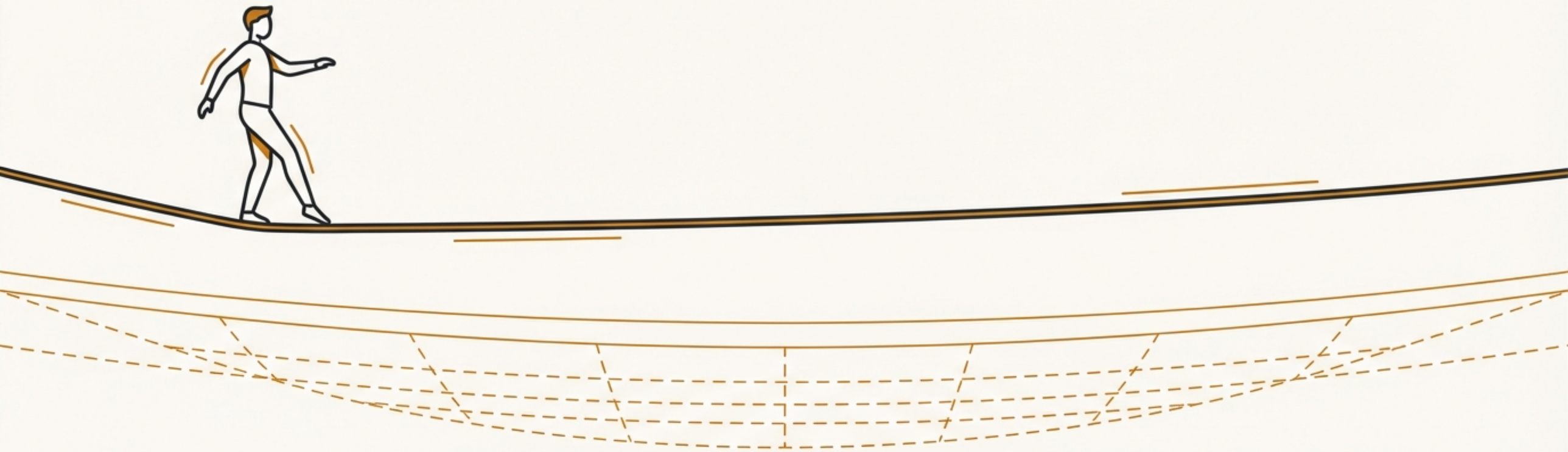
# From Fragile to Robust: Mastering Python Exception Handling

A guide to turning errors from a source of chaos into a tool for control.



# The Inevitable Slip: When Good Code Goes Wrong

Unhandled exceptions are the single biggest cause of unexpected program crashes. They turn stable applications into fragile ones.

## The Code

```python
# A simple calculation
numerator = 100
denominator = int(input("Enter a number to divide by: "))

result = numerator / denominator
print(f"The result is {result}")

# This line is never reached if an error occurs.
print("Program finished successfully.")
```

## The Chaos

User enters '0' in Source Sans Pro

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
ZeroDivisionError: division by zero
```

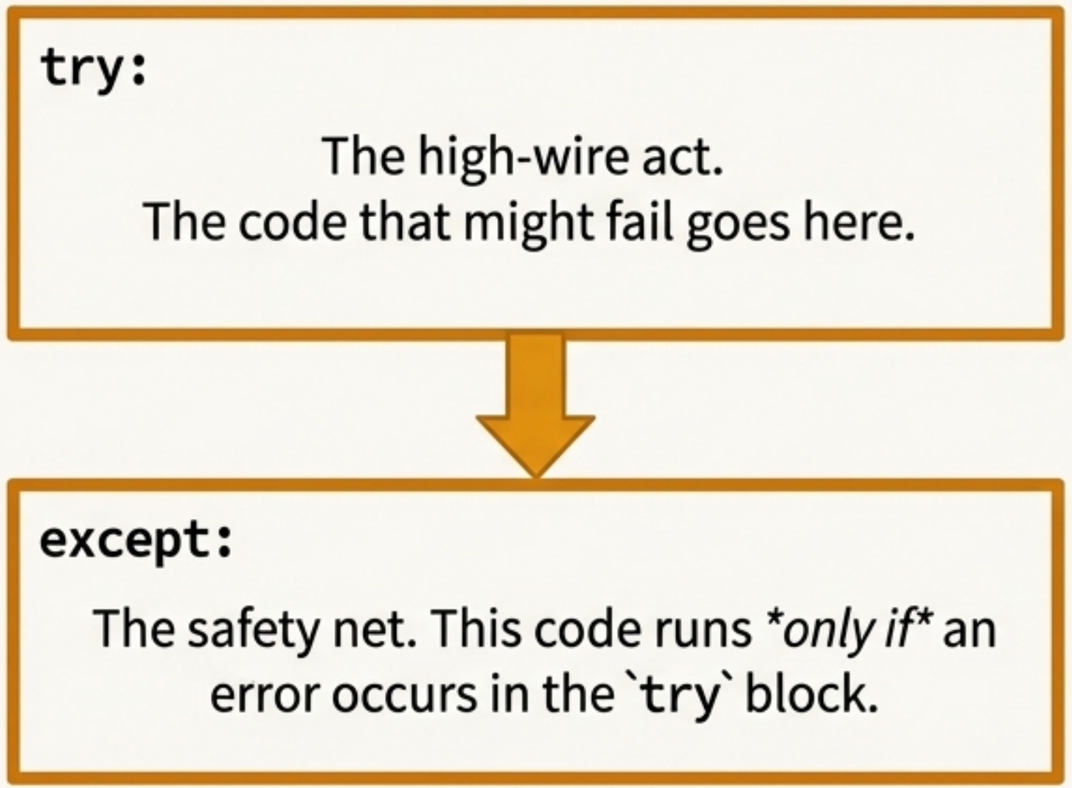
User enters 'abc' in Source Sans Pro

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'abc'
```

# The Safety Net: Catching the Fall with `try...except`

The `try...except` block allows you to run a block of code that might fail. If an error (an “exception”) occurs within the `try` block, execution immediately jumps to the `except` block, preventing a crash.

## Anatomy of a Safety Net

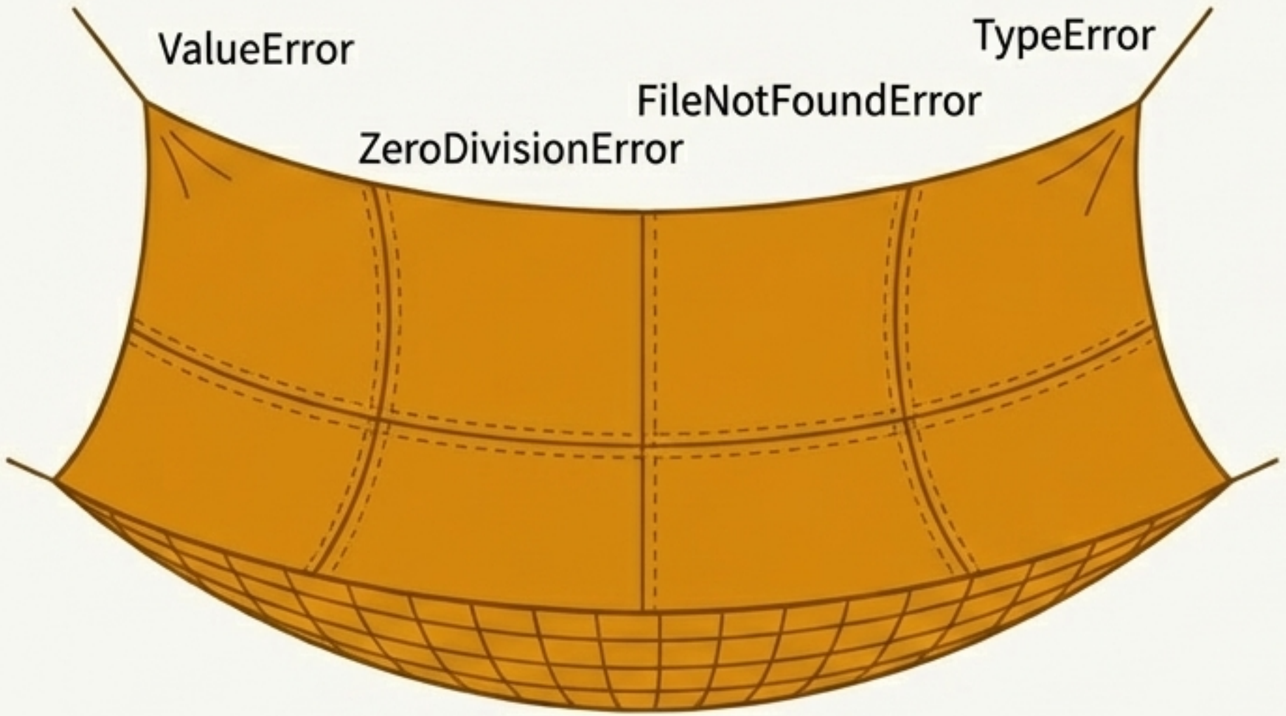



```
try:
    numerator = 100
    denominator = int(input("Enter a number
to divide by: "))
    result = numerator / denominator
    print(f"The result is {result}")
except:
    print("Something went wrong! Could not
perform the division.")

print("Program finished.") # This line is
now always reached!
```

# Not All Slips Are the Same: Handling Specific Errors

**Key idea**: Catching every possible error with a generic `except:` is like using one giant, clumsy net. It is far better to have specific nets for specific types of falls. This makes your code clearer and easier to debug.




## Common Python Exceptions

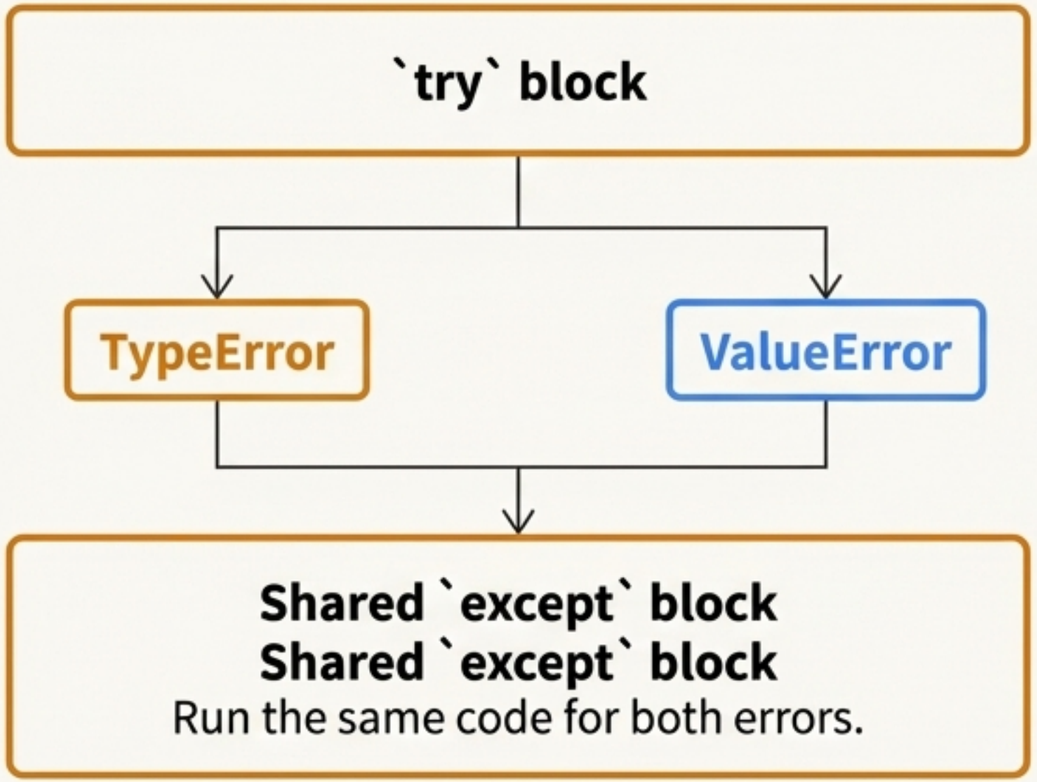
- `ValueError`
- `ZeroDivisionError`
- `FileNotFoundError`
- `TypeError`
- `IndexError`
- `KeyError`

## Code Example

```
try:
    # ... code from previous slide ...
except ValueError:
    print("Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("You cannot divide by zero.")
except Exception as e:
    # A good practice for unexpected errors
    print(f"An unexpected error occurred: {e}")
```

# Efficient Catching: Handling Multiple Exception Types at Once

When the handling logic for several different exceptions is the same, you can combine them into a single `except` block for cleaner, more maintainable code.




```
# A function that might return different errors
# depending on the input data.
def process_data(data):
    # ... function logic ...

try:
      process_data(user_input)
except (TypeError, ValueError):
    # Handle both errors in the same way
    print("Invalid data format provided. Please
check your input.")
```

# The Flawless Performance: Executing Code on Success with `else`

**Concept**

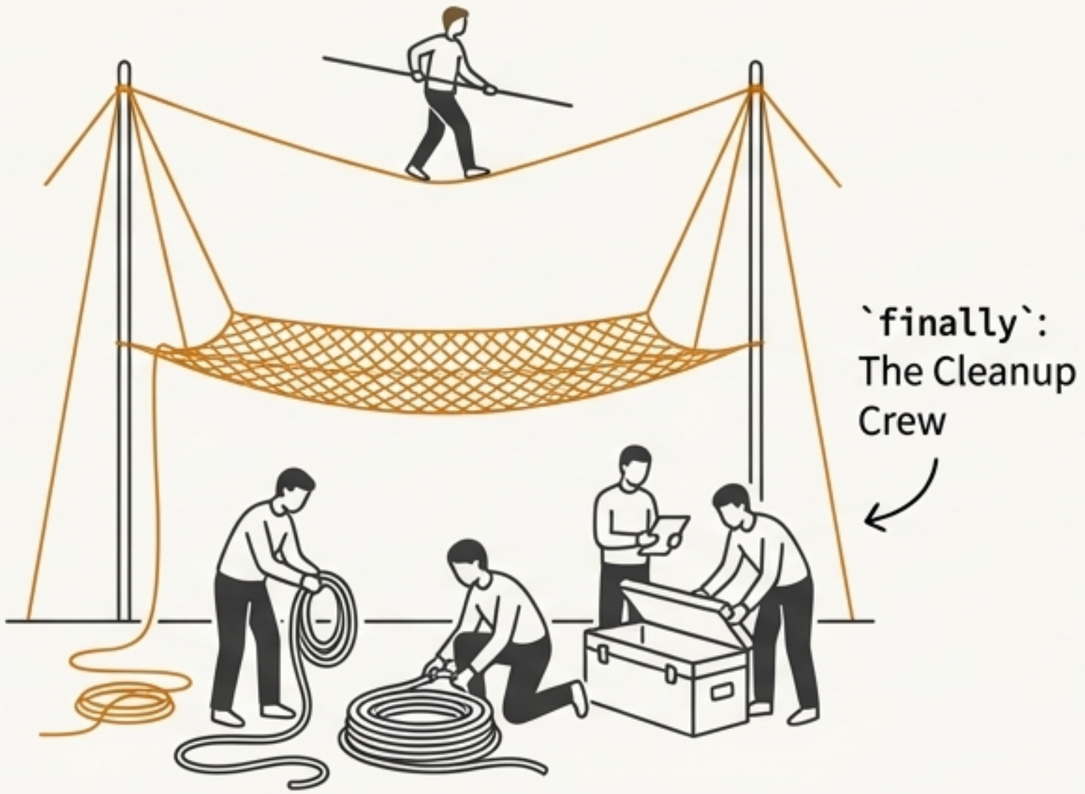
The optional `else` block contains code that is executed ***only if*** the `try` block completes successfully without raising any exceptions. This is the ideal place for logic that depends on the `try` block succeeding.

`else` block:
The success platform

```
try:
    # Action that might fail
    file = open("report.txt", "r")
except FileNotFoundError:
    print("Error: The report file was not found.")
else:
    # This only runs if the file was opened successfully
    print("File opened successfully. Processing report...")
    print(file.read())
    # file.close() would go here (but finally is better!)
```

# Win or Lose, Always Tidy Up: The `finally` Block

The `finally` block contains cleanup code that is **guaranteed** to run, regardless of whether an exception occurred in the `try` block or not.

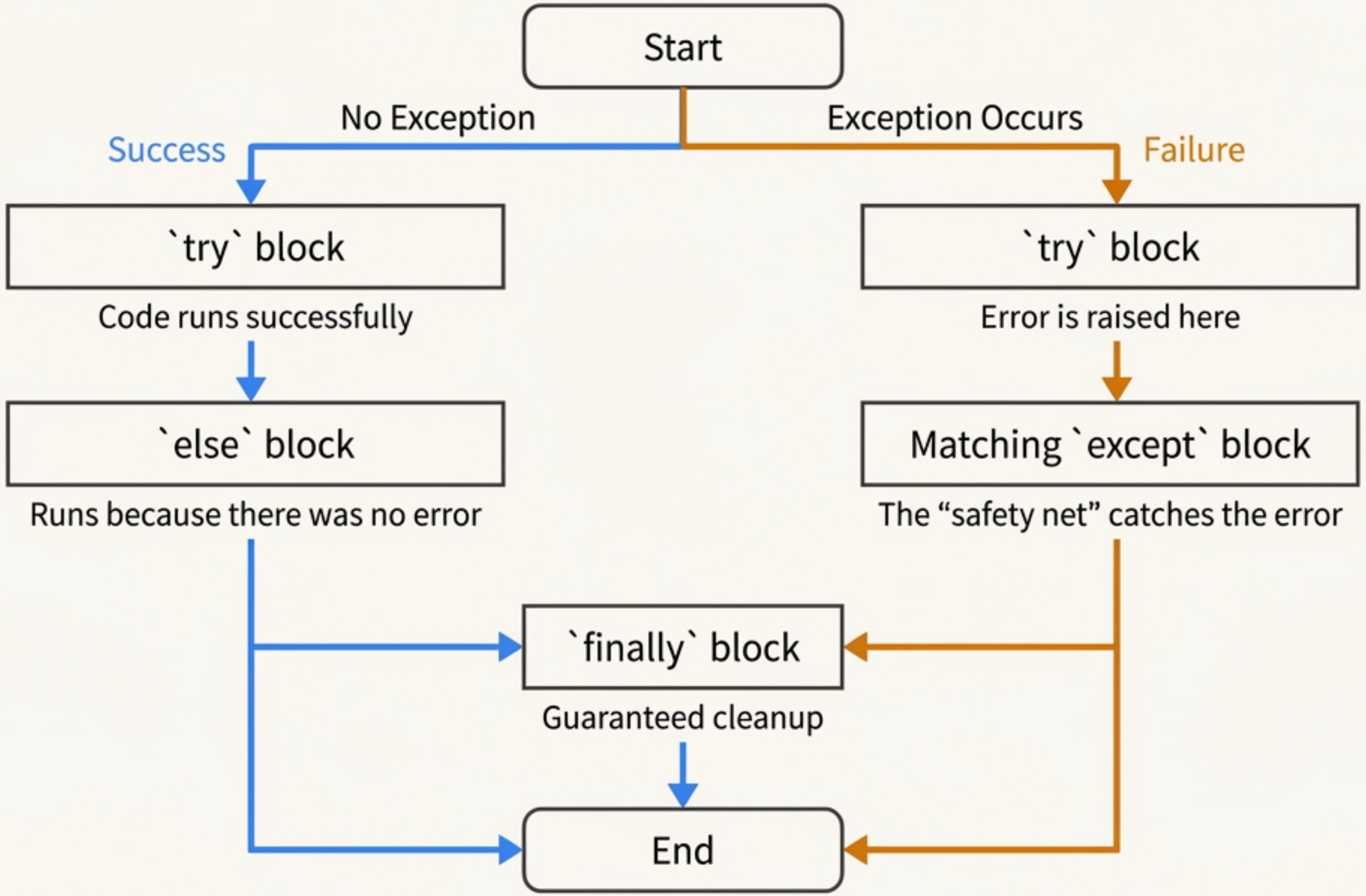



## Why It's Critical

Essential for releasing resources like closing files, disconnecting from databases, or releasing network sockets to prevent memory leaks and other issues.

```
file = None # Define outside the try block
try:
    file = open("data.txt", "r")
    # ... process the file ...
except FileNotFoundError:
    print("File could not be found.")
finally:
    # This code ALWAYS runs
    if file:
        file.close()
        print("File closed. Resources released.")
```

# Mapping the Full Performance: The Two Paths of Execution



# Taking Command: Manually Raising Exceptions with `raise`

You are not limited to handling exceptions Python raises automatically. Using the `raise` keyword, you can trigger exceptions yourself when your program's logic dictates that an error state has been reached.

## When to Use `raise`

- To validate function arguments or user input.
- To signal that a required condition has not been met.
- To enforce business rules.

```
def set_user_age(age):
    if not isinstance(age, int) or age < 0:
        # Enforce the rule: age must be a positive integer
        raise ValueError("Age must be a non-negative integer.")
    print(f"User age set to {age}.")

try:
    set_user_age(-5)
except ValueError as e:
    print(f"Error setting age: {e}")
```

# Writing the Rulebook: Building Your Own Custom Exceptions

For application-specific errors, creating your own exception classes makes your code more readable and self-documenting. A custom exception communicates a very specific problem that a generic `ValueError` cannot.

## How to Do It

Simply create a class that inherits from Python's base `Exception` class.

```
# Define a custom exception for our application
class InvalidPaymentAmountError(Exception):
    """Raised when the payment amount is invalid."""
    pass


def process_payment(amount):
    if amount <= 0:
        raise InvalidPaymentAmountError("Payment
amount must be positive.")
    # ... process the payment ...

try:
    process_payment(-50)
except InvalidPaymentAmountError as e:
    print(f"Payment failed: {e}")
```

# The Danger of Silence: Why You Should Almost Never Ignore Errors

## ⚠️ The Anti-Pattern

```
# DANGEROUS: DO NOT DO THIS
try:
    # ... some critical operation ...
except:
    pass # The error is swallowed and disappears forever
```

## Why It's So Bad

❌ **"Hides Bugs"**: The program fails silently, making it impossible to know that something went wrong.

❌ **"Complicates Debugging"**: You have no traceback, no error message, and no clue where the problem originated.

❌ **"Creates Unpredictable State"**: The program continues running in a potentially corrupted state, leading to bigger problems later on.

**The Bottom Line:** It's better for a program to crash loudly than to proceed silently with incorrect data.

# A Real-World Scenario: Robust File Handling

We need to read configuration data from a file. The file might not exist, we might not have permission to read it, or it might be empty. Our code must handle all these cases gracefully.

```
def read_config(path):
    config_data = None
    try:
        print(f"Attempting to open '{path}'...")
        f = open(path, 'r')
    except FileNotFoundError:
        print("Configuration file not found. Using default settings.")
    except PermissionError:
        print("Permission denied to read configuration file.")
    else:
        # This runs only if the file was opened successfully
        print("File opened. Reading contents.")
        config_data = f.read()
        if not config_data:
            print("Warning: Configuration file is empty.")
    finally:
        # This *always* runs to close the file if it was opened
        if 'f' in locals() and f:
            f.close()
            print("File resource closed.")
    return config_data

read_config("settings.ini")
```

# Your Developer Toolkit: Where to Deploy Exception Handling

Exception handling is not an obscure feature; it is fundamental to writing professional-grade Python code. Here are some of the most common areas where you will use it.

## File I/O

Safely opening, reading, and writing files without crashing on `FileNotFoundError` or `PermissionError`.

## Database Operations

Managing connections and handling transaction failures gracefully.

## Network & API Requests

Handling timeouts, connection errors, and unexpected API responses (e.g., 404s, 500s).

## User Input Validation

Ensuring user-provided data can be converted to the correct type (`int()`, `float()`) without `ValueError`.

# From Chaos to Control: The Power of Robust Code

We've transformed our view of errors from program-ending disasters into predictable events we can manage and control. Robust exception handling is the difference between fragile scripts and resilient applications.

## Core Principles

✅ Be specific in your `except` blocks.

✅ Use `else` for the 'success' path.

✅ Use `finally` for essential cleanup.

✅ Raise exceptions to enforce your own rules.

✅ Never let exceptions pass silently.

## Syntax Cheat Sheet

```
try:
    # Code that might fail
except SpecificError:
    # Handle that specific error
else:
    # Run only on success
finally:
    # Always run for cleanup
```