ScopeTreasure Courses

PYTHON COURSES · LESSON PY.11 · CODE SCOPE SERIES

Errors and Exceptions

Errors are messages, not failures — learn the difference between syntax errors and runtime errors, and how try, except and finally handle them.

Introduction

Every programmer meets errors. Beginners often think an error means they are bad at coding. It does not. An error is Python's way of saying, "I stopped here, and here is why." Learning to read that message is a core programming skill — often more useful than memorising any command.

There are two big families of errors. A **syntax error** means Python could not even understand your code — a missing colon, an unclosed bracket, a misspelled keyword. The program never starts. A **runtime error**, also called an exception, means the code was understood, but something went wrong while running — dividing by zero, opening a missing file, or using a name that does not exist.

Python lets you **handle** runtime errors with `try` and `except`: try the risky code, and if a specific error happens, run a backup plan instead of crashing. That is what this lesson teaches.

(Roman Urdu: Error ka matlab yeh nahi ke aap ghalat hain — error aik paigham hai jo batata hai ke masla kahan hai, ta ke aap usay theek kar saken.)

Error handling has a golden rule that beginners break constantly: handle the error you expect, where you expect it, and nothing more. Wrapping your whole program in one giant `try` hides the very information you need — which line failed, and why. A small `try` around one risky step, with a specific `except`, keeps the program alive without hiding bugs.

Key points

- **A SyntaxError means the code itself is broken.** Python cannot read it, so nothing runs. Example: a missing colon after `if`.
- **A runtime error (exception) happens during running.** The code reads fine, but a step fails — for example, `10 / 0` or opening a missing file.
- **Common exceptions:** `ZeroDivisionError` (dividing by zero), `ValueError` (wrong value type, like `int("abc")`), `FileNotFoundError` (missing file), `NameError` (unknown variable name), `IndexError` (list index out of range), `TypeError` (wrong type for an operation).
- **A traceback is read from the bottom up.** The last line names the error; the lines above show which line of your code caused it.
- **`try` wraps the risky code.** `except` catches a specific error and runs a backup plan. Code after the whole block keeps running.
- **Catch specific errors, not everything.** `except ZeroDivisionError:` is better than a bare `except:` that hides every problem, including bugs you need to see.

- `finally` **always runs.** It is for cleanup that must happen whether or not an error occurred, like closing a resource.
- **Handling an error is not the same as fixing the cause.** A backup message keeps the program alive, but the underlying problem (a missing file, bad input) still needs attention.
- `else` **after** `try/except` **runs only when no error happened.** It holds the code that should run on success, keeping the `try` block small.
- `raise` **creates an error on purpose.** Use it when your own code spots bad data, like a negative age, so callers must deal with it.

Examples

Example 1 — A runtime error, then the same code handled

First, the crash. Run this on its own and read the last line:

```
print(10 / 0)
```

What you will get:

```
ZeroDivisionError: division by zero
```

Now the handled version:

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("Cannot divide by zero!")
print("Program continues")
```

What you will get:

```
Cannot divide by zero!
Program continues
```

The `except` block replaced the crash with a friendly message, and the program carried on to the last line. Without the handler, it would have stopped at the error.

Example 2 — Handling a missing file and a bad number

```
try:
    with open("missing.txt", "r") as f:
        print(f.read())
except FileNotFoundError:
    print("That file does not exist yet.")

try:
    number = int("abc")
    print(number)
except ValueError:
    print("That is not a valid number.")
```

What you will get:

```
That file does not exist yet.
That is not a valid number.
```

Each `try` block has its own `except` for the error it expects. Matching the error type matters: a `FileNotFoundError` handler will not catch a `ValueError`, and vice versa.

Example 3 — finally always runs

```
try:
    print("Trying...")
    print(10 / 0)
except ZeroDivisionError:
    print("Caught the error.")
finally:
    print("Cleanup: this always runs.")

try:
    print("Trying again, no error this time.")
    print(10 / 2)
except ZeroDivisionError:
    print("Caught the error.")
finally:
    print("Cleanup: this always runs too.")
```

What you will get:

```
Trying...
Caught the error.
Cleanup: this always runs.
Trying again, no error this time.
5.0
Cleanup: this always runs too.
```

Whether or not an error happened, the `finally` block ran. That is why it is used for cleanup steps that must never be skipped.

Example 4 — Several specific handlers, plus else

One risky step can fail in different ways, and each deserves its own message. The `else` block holds the success path, so the `try` stays small:

```
def safe_divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Cannot divide by zero"
    except TypeError:
        return "Numbers only, please"
    else:
        return result

print(safe_divide(10, 2))
print(safe_divide(10, 0))
print(safe_divide(10, "2"))
```

What you will get:

```
5.0
Cannot divide by zero
Numbers only, please
```

Dividing by zero hits the first handler; dividing by a string hits the second, because a number divided by text is a TypeError. The `else` runs only for the successful case, returning the real answer.

Example 5 — Raising your own error for bad data

Sometimes your code is the one that knows a value is wrong — a negative age can never be valid. `raise` stops the function with a clear message, and the caller handles it like any other error:

```
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative")
    return age

try:
    print(set_age(20))
    print(set_age(-5))
except ValueError as error:
    print("Problem:", error)
```

What you will get:

```
20
Problem: Age cannot be negative
```

`except ValueError as error` captures the error object, so its message can be printed. This pattern — validate, raise, handle — is how real programs reject bad input politely instead of crashing mysteriously.

Example 6 — Real-world case: reading a marks file safely

A class report program reads `marks.txt`, the file from PY.10. If the file is missing, the program says so plainly; if a line is malformed, it says that instead. Either way, it does not crash in front of the user:

```
try:
    with open("marks.txt", "r") as f:
        for line in f:
            name, mark = line.strip().split(",")
            print(name, int(mark))
except FileNotFoundError:
    print("marks.txt not found — create it first")
except ValueError:
    print("A line in the file is not in name,number form")
```

What you will get (with the PY.10 `marks.txt` present):

```
Ali 85
Ayesha 92
```

Delete the file and run it again, and you get the friendly missing-file message instead of a traceback. That difference — a crash versus a message — is the entire value of error handling.

Quick recap

- Syntax errors stop the program before it starts; runtime errors stop it mid-run.
- Read a traceback from the bottom: the last line names the error.
- `try` wraps risky code; `except` gives a backup plan for a specific error.
- Catch specific errors, not everything at once.
- `finally` runs whether or not an error occurred.
- `try...else` runs the `else` only when no error occurred.
- `raise` creates an error deliberately when your code spots bad data.

Common mistakes

1. **Reading a traceback from the top** — the top is mostly file and call details. Fix: read the last line first for the error name, then look up to find your line number.
2. **Catching everything with a bare `except:`** — it hides real bugs and makes debugging much harder. Fix: name the error you expect, like `except ValueError:`.
3. **Putting too much code inside `try`** — when many lines are wrapped, you cannot tell which one failed. Fix: wrap only the one risky step.
4. **Trying to handle a syntax error with try/except** — a syntax error means Python cannot read the code at all, so no handler can run. Fix: fix the syntax itself (colon, brackets, spelling) and run again.
5. **Using try/except instead of checking first** — if you can check cheaply (for example, testing a key with `.get()`), check first and keep try/except for genuinely unpredictable failures like missing files.

1. **Putting a broad handler before a specific one** — `except Exception:` catches everything, so a later `except ValueError:` can never run. Fix: list specific errors first, broad ones last (or not at all).
2. **Hiding errors with an empty handler** — `except ValueError: pass` swallows the problem silently, and the program carries on with wrong data. Fix: at least print a message, so the failure is visible.

Practice

1. Run `print(10 / 0)` on its own and copy the last line of the error. Then wrap it in try/except and confirm the program continues afterwards.
2. Write a program that tries to read a file called `diary.txt`. If it is missing, print "No diary yet — creating one is your next task." Then actually create the file and run the program again.
3. Write a program that converts the string `"42"` to a number with `int()` inside try/except, then does the same with `"hello"`. Print a friendly message for the second case.
4. Write a try/except/finally block where the `try` succeeds (no error). Confirm that `finally` still runs, and write down why that matters for cleanup.
5. Write a function `safe_int(text)` that returns `int(text)` on success and `None` on a ValueError. Test it with `"42"`, `"hello"` and `"3.5"`, and print all three results.

6. Write a program that asks (via a variable, not input) for a list of marks as text, like `["80", "bad", "90"]`, converts each one inside try/except, and prints the valid marks and a message for each bad one.
7. Write a function `set_mark(mark)` that raises a ValueError if the mark is below 0 or above 100. Test it with 85, -5 and 120, handling the error each time.

Practice quiz

1. What is a SyntaxError?

- A wrong answer
- Code written against Python's grammar rules, so it will not run at all
- A slow program
- A missing file

Answer: Code written against Python's grammar rules, so it will not run at all

2. What does try/except do?

- Finds spelling mistakes
- Catches a runtime error so the program can carry on instead of crashing
- Deletes errors
- Speeds up code

Answer: Catches a runtime error so the program can carry on instead of crashing

3. When you read a traceback, which line tells you the error type?

- The first line
- The last line, which names the error and where it happened
- The middle line
- Tracebacks have no error name

Answer: The last line, which names the error and where it happened

---

Watch the video lesson: https://youtu.be/F2HfN281idg

Read online: https://scopetreasure.com/notes/py-errors-exceptions · More free tools and notes at https://scopetreasure.com