Python Exceptions

Table of contents

Common built-in exceptions 1
Catching exceptions 2
Raising exceptions 2
LBYL vs. EAFP 3

An **exception** is a run-time error — unlike a **syntax error**, which Python reports while *parsing* the file, an exception cannot be detected until the offending line actually executes. Left uncaught it **aborts the program**, and all intermediate work is lost unless it was already written to a file.

Introduced in 2025-09-09-exceptions[1] (Lecture 7A).

Common built-in exceptions

Exception	Raised when…
`AssertionError`	an `assert` fails
`IOError`	a file does not exist
`IndexError`	a sequence index is out of range
`KeyError`	a dict key does not exist
`NameError`	a variable/name does not exist
`TypeError`	an unexpected type is given to a function/operator
`ValueError`	correct type, but an inappropriate value
`ZeroDivisionError`	division by zero

The `TypeError`/`ValueError` boundary follows the function's **type contract**: `int` has the contract `int(xs: str) -> int`, so `int("ten")` is given the right *type* and therefore raises `ValueError`, not `TypeError`.

[1] courses/csse1001/2025-09-09-exceptions.pdf

Catching exceptions

```
try:
    <code that may raise>
except SomeError:
    <handle SomeError>
except AnotherError:
    <handle AnotherError>
else:
    <runs only if no exception occurred>
finally:
    <always runs>
```

Also called **try-catch** in other languages (e.g. C++). Typically used in the user-facing layers of a program, and wherever it touches external resources — files, databases, API calls, network calls.

- The `except` clauses are tried **in source order** and the **first match wins**; the rest are skipped. If none match, the exception propagates and the program aborts.
- Consequently, **specific exceptions must be listed before general ones** (and before a bare `except:`) — Python enforces this ordering, because a general clause placed first would swallow every case below it.
- A bare `except:` catches *any* exception, but hides genuine bugs — prefer catching specific exception types. It is bad practice to ignore all errors.
- `else` runs only when the `try` block completed without raising; `finally` runs on every path, exception or not, and is the right home for cleanup.
- Keep `try` blocks short: once an exception is raised inside one, none of the remaining lines in that block execute — so a `return` placed after the risky line will simply be skipped.

Raising exceptions

```
raise SomeError                  # bare
raise SomeError("message")       # with an explanatory message
```

`raise` halts execution at the point the failure becomes knowable, rather than letting a bad value travel further into the program. It is the standard way for a **setter** to reject an invalid value before it becomes an object's state — see python-classes-and-objects[2].

[2] courses/csse1001/python-classes-and-objects.pdf

LBYL vs. EAFP

Two philosophies for guarding against errors:

- **LBYL** (Look Before You Leap) — check the condition with an `if` before attempting the risky operation.
- **EAFP** (Easier to Ask for Forgiveness than Permission) — attempt the operation inside a `try`, and handle the resulting exception if it fails.

Only use a `try/except` when it is *absolutely necessary*: an exception handler that is really a misplaced `if` makes the ordinary path harder to read.