Module 10 of 20

Errors and exceptions

The try statement with else and finally, the hierarchy and matching, raise and chaining, custom exception classes, EAFP with suppress and sentinels, context managers as classes and generators, exception groups with except* and add_note, and assertions versus boundary validation.

Python signals every failure by raising — a missing key, the end of an iterator, a file that is not there — so exception handling is not an edge-case skill but the ordinary texture of the language. This module makes it precise: what the four clauses of `try` do, how a handler matches, what `raise` and `from` mean, how to design exception classes that callers can act on, and the EAFP style that the standard library assumes. It then covers the two structures that grew out of exceptions — context managers, which guarantee cleanup, and exception groups, which report several failures at once — and closes with the rule that separates assertions from validation.

Exceptions covers the statement, the hierarchy, `raise`, re-raising and chaining, and the difference between catching and hiding. Custom exceptions covers subclassing, data attributes, a base per package and when a built-in is the honest choice. EAFP covers the cost model, the library's defaulted forms, `contextlib.suppress`, return-versus-raise, sentinels and retries. Context managers covers `with`, `__enter__` / `__exit__`, `@contextmanager`, and the `contextlib` helpers including `ExitStack`. Exception groups covers `ExceptionGroup`, `except*`, `add_note`, the `traceback` module and warnings. Assertions covers `assert` under `-O`, boundary validation and fail-fast design.

The exercises are whole programs: a clause tracer, chained division errors, an application error hierarchy, an exception carrying data, a first-or-default with a sentinel, a capped retry loop, a commit-or-rollback transaction manager, a generator manager that always runs its teardown, a validator that collects failures into a group, traceback text from evaluated expressions, boundary validation of a configuration, and a stack machine with asserted invariants. The checkpoint adds a record loader reporting every failure through `except*`, nested savepoints, and a calculator caught at exactly the right level of its hierarchy.

CONTENTS

1 Exceptions — try, except, else, finally, raise 14 min

2	Custom exceptions — a hierarchy for your own errors	13 min
3	EAFP and exception-driven flow — suppress, retries, return versus raise	13 min
4	Context managers — with, __enter__/__exit__ and contextlib	14 min
5	Exception groups, notes and the traceback module	13 min
6	Assertions and defensive code — validate at the boundary, assert the invariant	13 min
✓	Checkpoint — Errors and exceptions	25 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Exceptions — try, except, else, finally, raise

An exception is an object that interrupts normal control flow: raised at the point of failure, it unwinds the call stack until a handler catches it, running cleanup blocks on the way, or reaches the top and ends the program with a traceback. Python uses exceptions for everything from `KeyError` to the end of a file, so handling them well is not an edge-case skill but the normal texture of the code. This lesson covers the full `try` statement, the standard hierarchy and how to choose what to catch, `raise` in its forms, re-raising and chaining with `from`, and the difference between catching an error and hiding it.

The statement

```
try:
    n = int(text)                  # the code that may raise
except ValueError:
    print("not a number")          # runs only if a ValueError was raised in t
else:
    print(n * 2)                   # runs only if nothing was raised
finally:
    print("done")                  # always runs — after the try, after any ha
```

Execution enters `try`; if an exception is raised there, the interpreter looks for a matching `except` clause in order and runs the first that matches; if none matches, the exception propagates to the caller. `else` runs when the `try` block completed without raising — it keeps code that should not be protected by the handler *out* of the `try`. `finally` runs no matter what, which makes it the place for cleanup (though `with` is usually better — lesson 4).

Keep the `try` block small: only the statements that can raise the exception you mean to handle. A `try` wrapping fifty lines catches a `ValueError` from any of them and hides the one you did not expect.

Matching exceptions

```
except ValueError:                    # this class or any subclass
except (ValueError, TypeError):       # any of several
except ValueError as e:               # bind the exception object
    print(e)                          # the message; e.args holds the construc
except Exception:                     # everything that is a normal error
except:                               # everything, including KeyboardInterrup
```

Handlers are tried top to bottom, and a handler matches an exception of its class *or any subclass*. So `except Exception` before `except ValueError` makes the second unreachable; order specific to general. A bare `except:` also catches `KeyboardInterrupt` (Ctrl-C) and `SystemExit` (from `sys.exit`), which is why it turns a program into one that cannot be stopped; `except Exception` is the widest handler that is ever appropriate, and it belongs at a boundary — a request handler, a worker loop — with logging, not in ordinary code.

The hierarchy

```
BaseException
├── SystemExit, KeyboardInterrupt, GeneratorExit
└── Exception
    ├── ArithmeticError → ZeroDivisionError, OverflowError
    ├── LookupError → IndexError, KeyError
    ├── ValueError → UnicodeError
    ├── TypeError
    ├── AttributeError, NameError
    ├── OSError → FileNotFoundError, PermissionError, TimeoutError
    ├── RuntimeError → RecursionError, NotImplementedError
    ├── StopIteration, AssertionError, ImportError, …
```

`LookupError` catches both an `IndexError` and a `KeyError`; `OSError` catches every file and network failure; `ArithmeticError` both division and overflow. Catching a parent is right when the handling is the same for all its children. `Exception` is the base for your own exception classes (next lesson).

---

raise

```
raise ValueError("age must be positive")      # an instance, with a message
raise ValueError                              # the class; an instance is cre
raise                                         # inside a handler: re-raise th
```

Raise the built-in exception whose *meaning* fits: `ValueError` for a right-typed but wrong-valued argument, `TypeError` for a wrong type, `KeyError` / `IndexError` for a missing key/position, `RuntimeError` for a state that should not happen, `NotImplementedError` for an abstract operation. The message should say what was wrong *and* what was received — `f"expected 1-10, got {n}"` — because it is what the person reading the traceback sees.

Re-raising and chaining

A handler that cannot fully deal with an exception should pass it on:

```
try:
    value = parse(text)
except ValueError:
    log("bad input")
    raise                                     # the original traceback is pres

try:
    config = json.loads(text)
except json.JSONDecodeError as e:
    raise ConfigError(f"invalid config: {e}") from e     # explicit chaining:
```

When an exception is raised *inside* a handler, Python chains them automatically ("During handling of the above exception, another exception occurred") so both tracebacks show. `raise X from e` marks `e` as the *cause*, which reads better; `raise X from None` suppresses the chain when the original is noise — the right choice when translating an internal exception into one the caller expects.

Where exceptions come from

Built-in operations raise them (`int("x")` , `d[k]` , `xs[99]` , `1/0` , `open("missing")`); library calls raise them (`json.loads` , `re.compile` , `datetime.strptime`); your own `raise` ; and `assert` . Everything that can fail in Python fails by raising — there are no error codes to check

— which is why unchecked results are not a Python bug class, and why *swallowing* exceptions is.

Catching versus hiding

```
try:
    total = compute(rows)
except Exception:
    total = 0                    # the bug is now invisible: a wrong total ins
```

A handler must do something meaningful: recover (retry, use a default the caller understands), translate (raise a better exception), report (log and re-raise), or clean up (`finally`). A handler that turns a failure into a plausible-looking result hides the bug until it corrupts something further away. If you do not know what to do with an exception, do not catch it.

Pitfalls

- A bare `except:` or a broad `except Exception:` deep inside logic.
- A `try` block that covers more than the call that can raise.
- Handlers in general-to-specific order, making the specific ones dead.
- Catching and returning `None` or `0` , converting a loud failure into a silent wrong answer.
- `raise e` inside a handler instead of `raise` — it rewrites the traceback's origin to the handler.
- Forgetting that `finally` runs even after a `return` in the `try` , and that a `return` in `finally` overrides everything.

Key takeaways

- `try` / `except` / `else` / `finally` : handle what raised, run `else` on success, `finally` always; keep the `try` small.
- Handlers match a class or its subclasses, top to bottom; specific before general; `except Exception` only at boundaries; never bare `except:` .
- Raise the built-in whose meaning fits, with a message that says what was expected and what arrived.

Bare `raise` re-raises; `raise X from e` chains with a cause; `from None` suppresses the chain.

Catch only to recover, translate, report or clean up — a handler that hides a failure is worse than none.

EXERCISE 1

Trace the clauses

For each command, run an operation inside one `try` statement that has an `except ValueError`, an `except KeyError`, an `else` and a `finally`, and print the names of the parts that actually ran, in order, on one line. `ok` runs an operation that succeeds; `value` runs `int("x")`; `key` runs `{}["k"]`; `zero` runs `1 / 0` — which no handler matches, so record `finally` and then let it propagate to an outer handler that prints `propagated ZeroDivisionError`.

Input: commands. **Output:** one line per command (two for `zero`).

```
ok
value
zero
```

prints

```
try else finally
try except:ValueError finally
try finally
propagated ZeroDivisionError
```

INPUT 1

```
ok
value
zero
```

OUTPUT 1

```
try else finally
try except:ValueError finally
try finally
propagated ZeroDivisionError
```

REFERENCE SOLUTION

```
import sys

OPS = {
    "ok": lambda: 1,
    "value": lambda: int("x"),
    "key": lambda: {}["k"],
    "zero": lambda: 1 / 0,
}


def run(op):
    parts = ["try"]
    try:
        op()
    except ValueError:
        parts.append("except:ValueError")
    except KeyError:
        parts.append("except:KeyError")
    else:
        parts.append("else")
    finally:
        parts.append("finally")
        print(" ".join(parts))
    return parts


for line in sys.stdin:
    try:
        run(OPS[line.strip()])
    except ZeroDivisionError as e:
        print(f"propagated {type(e).__name__}")
```

EXERCISE 2

Chained causes

Write `ratio(a, b)` that returns `a / b` but, on `ZeroDivisionError`, raises `ValueError("cannot divide by zero") from` the original exception. For each input line `a b`, print the result with two decimals, or — catching the `ValueError` — print its message and the type name of its `__cause__`. A non-numeric token should propagate as a plain `ValueError` with no cause: print `bad input` for that (its `__cause__` is `None`).

Input: lines `a b`. **Output:** `<ratio>`, or `error: <message> (caused by ZeroDivisionError)`, or `bad input`.

```
1 4
1 0
x 2
```

prints

```
0.25
error: cannot divide by zero (caused by ZeroDivisionError)
bad input
```

INPUT 1

```
1 4
1 0
x 2
```

OUTPUT 1

```
0.25
error: cannot divide by zero (caused by
ZeroDivisionError)
bad input
```

REFERENCE SOLUTION

```python
import sys


def ratio(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError("cannot divide by zero") from e


for line in sys.stdin:
    try:
        a, b = map(float, line.split())
        print(f"{ratio(a, b):.2f}")
    except ValueError as e:
        if e.__cause__ is not None:
            print(f"error: {e} (caused by {type(e.__cause__).__name__})")
        else:
            print("bad input")
```

QUIZ

1. When does the `else` clause of a `try` statement run?

A. Always

B. Only when the `try` block raised no exception

C. Only after an `except` handler ran

D. When `finally` is absent

2. What is wrong with this order?

```
try:
    ...
except Exception:
    ...
except ValueError:
    ...
```

A. Nothing

B. The `ValueError` handler is unreachable — `Exception` matches it first

C. It is a syntax error

D. `ValueError` is not an `Exception`

3. What does a bare `except:` catch that `except Exception:` does not?

A. Nothing

B. `KeyboardInterrupt` and `SystemExit`, which is why it makes programs unstoppable

C. Only `ValueError`

D. Warnings

4. Inside an `except` block, what is the difference between `raise` and `raise e`?

A. None

B. `raise` re-raises with the original traceback; `raise e` restarts the traceback at the handler

C. `raise e` is a syntax error

D. `raise` creates a new exception

5. What does `raise ConfigError("bad") from None` do?

A. Raises with no message

B. Raises `ConfigError` and suppresses the automatic 'during handling…' chain

C. Raises `None`

D. Re-raises the original exception

ANSWER KEY · EXCEPTIONS — TRY, EXCEPT, ELSE, FINALLY, RAISE

1. B `else` is the success branch; it keeps code that should not be covered by the handlers out of the `try` block.

2. B Handlers match subclasses and are tried in order; put specific classes before general ones.

3. B Those derive from `BaseException` directly; catching them by accident swallows Ctrl-C and `sys.exit`.

4. B Bare `raise` preserves where the exception was originally raised, which is what you want in a log.

5. B `from None` sets `__suppress_context__`, hiding the exception being handled when it is only noise.

LESSON 2 · 13 MIN

Custom exceptions — a hierarchy for your own errors

The built-in exceptions describe *what kind* of thing went wrong — a bad value, a missing key — but not *whose* code it went wrong in. A library or application that raises `ValueError` for everything leaves its callers unable to tell a malformed input from a failed business rule from a bug. Custom exception classes solve that: a base class for the package, subclasses for the distinct failures, fields that carry what the handler needs, and a message that reads well. This lesson covers defining them, adding data, building a small hierarchy, writing messages, and the rules for when a custom exception earns its place versus when a built-in is the honest choice.

The minimal custom exception

```
class InsufficientFunds(Exception):
    pass

def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFunds(f"balance {balance}, requested {amount}")
    return balance - amount

try:
    withdraw(10, 50)
except InsufficientFunds as e:
    print("declined:", e)          # declined: balance 10, requested 50
```

Subclass `Exception` (never `BaseException`), give it a name that ends in `Error` or reads as a condition, and it is done: the message passed to the constructor is stored in `e.args` and shown by `str(e)`. A `pass` body is enough — the class's identity is its content.

Carrying data

When handlers need to *act* on details, store them as attributes rather than making callers parse the message:

```python
class InsufficientFunds(Exception):
    def __init__(self, balance, requested):
        super().__init__(f"balance {balance}, requested {requested}")
        self.balance = balance
        self.requested = requested

    @property
    def shortfall(self):
        return self.requested - self.balance

try:
    withdraw(acct, 50)
except InsufficientFunds as e:
    offer_overdraft(e.shortfall)
```

Call `super().__init__` with the message so that `str(e)`, the traceback and `e.args` behave normally, then set your own attributes. A dataclass-style exception (`@dataclass class X(Exception)`) is possible but fiddly with `args`; the explicit `__init__` is the convention.

A hierarchy

```python
class AppError(Exception):
    """Base for every error this application raises on purpose."""

class ConfigError(AppError):
    pass

class ValidationError(AppError):
    def __init__(self, field, message):
        super().__init__(f"{field}: {message}")
        self.field = field

class NotFound(AppError):
    pass
```

One base per package or application, one subclass per *distinct handling*. The base lets a boundary catch "anything we raised deliberately" (`except AppError`) separately from bugs (`except Exception` — a `TypeError` from a typo is not an `AppError`), and the leaves let

specific code handle specific failures. Do not create a subclass for every message; create one when some caller will catch it *differently* from its siblings.

Multiple inheritance from a built-in is legitimate when the custom exception *is* a kind of built-in error too: `class ConfigKeyError(ConfigError, KeyError)` is caught by both `except ConfigError` and `except KeyError`, which keeps old callers working while new ones use the specific class.

Messages

The message is read by a human in a traceback or a log, so: say what was expected and what arrived (`f"port must be 1-65535, got {port}"`), name the thing (`f"user {user_id} not found"`), avoid duplicating what the class name already says, and do not end with a full stop or capitalise like a sentence — `ValueError: invalid literal for int() with base 10: 'x'` is the house style. Never put the fix in the message unless it is certain; put the context.

When not to define one

A built-in already means it. A bad argument is `ValueError` or `TypeError`; a missing key is `KeyError`; a missing file is `FileNotFoundError`. Callers already know how to handle those, and a `MyValueError` that nobody catches differently is noise.

Nobody will catch it. An exception that only ever ends the program with a traceback gains nothing from a custom class.

It is control flow inside one function. Use a return value or a `break`.

Define one when callers *outside your module* need to distinguish your failure from others, when handlers need structured data, or when a package wants one base to catch.

Translating at a boundary

A library that calls another library should not leak the inner one's exceptions: a caller of `load_config` wants `ConfigError`, not `json.JSONDecodeError` or `KeyError` from whatever the implementation happens to use today. The pattern is a small `try` at the boundary that catches the inner exception and raises the outer one `from` it — the cause stays visible in the traceback for debugging, but the *type* the caller must catch is stable across implementation

changes. Do the translation once, at the edge of the module; deeper code raises whatever is natural.

Documenting and testing

A function's docstring should list what it raises (`Raises ValidationError if …`), because raising is part of the contract. Tests assert the class and, when it matters, the fields:

```
try:
    validate({"age": -1})
except ValidationError as e:
    assert e.field == "age"
else:
    raise AssertionError("expected ValidationError")
```

`unittest`'s `assertRaises` and pytest's `pytest.raises` (Module 18) are the same check with less ceremony.

Pitfalls

- Subclassing `BaseException` — it escapes `except Exception` and behaves like `KeyboardInterrupt`.
- A custom `__init__` that forgets `super().__init__(message)`, leaving `str(e)` empty.
- A class per message rather than per handling.
- Callers parsing `str(e)` because the data was not stored as attributes.
- Wrapping a built-in in a custom exception without `from e`, losing the cause.
- Naming: `ErrorHappened`, `MyException` — name the condition.

Key takeaways

- Subclass `Exception`, name the condition, pass a message to `super().__init__`; a `pass` body is often enough.
- Store details handlers need as attributes; the message is for humans.
- One base class per package, one subclass per distinct handling; inherit from a built-in too when it is a kind of that error.

Messages say what was expected and what arrived, in the built-ins' lower-case style.

Prefer a built-in when it already means the failure; define your own when callers must tell it apart or need its data.

EXERCISE 1

An application hierarchy

Define `AppError(Exception)`, `ValidationError(AppError)` with a `field` attribute and a message `<field>: <problem>`, and `NotFound(AppError)`. `lookup(text)` raises `ValidationError("id", "must be a positive integer")` when the text is not a positive integer, `NotFound(f"user {n} not found")` when the integer exceeds 100, and otherwise returns `f"user {n}"`. Handle each line with three handlers — `ValidationError` (print `invalid <field>: <message>`), `NotFound` (print `missing: <message>`), then `AppError` as a fallback that never fires here — and print the result otherwise.

Input: lines. **Output:** one line per input line.

```
7
abc
500
```

prints

```
user 7
invalid id: id: must be a positive integer
missing: user 500 not found
```

INPUT 1

```
7
abc
500
```

OUTPUT 1

```
user 7
invalid id: id: must be a positive
integer
missing: user 500 not found
```

REFERENCE SOLUTION

```
import sys


class AppError(Exception):
    pass


class ValidationError(AppError):
    def __init__(self, field, problem):
        super().__init__(f"{field}: {problem}")
        self.field = field


class NotFound(AppError):
    pass


def lookup(text):
    if not text.isdigit() or int(text) ⩽ 0:
        raise ValidationError("id", "must be a positive integer")
    n = int(text)
    if n > 100:
        raise NotFound(f"user {n} not found")
    return f"user {n}"


for line in sys.stdin:
    try:
        print(lookup(line.strip()))
    except ValidationError as e:
        print(f"invalid {e.field}: {e}")
    except NotFound as e:
        print(f"missing: {e}")
    except AppError as e:
        print(f"app error: {e}")
```

EXERCISE 2

An exception that carries data

Define `InsufficientFunds(Exception)` taking `balance` and `requested`, storing both, passing the message `balance <b>, requested <r>` to `super().__init__`, and exposing a `shortfall` property. Process `withdraw amount` commands against a balance read from the first line; a successful withdrawal prints the new balance, a failed one is caught and prints `declined: <message>; short by <shortfall>` — read from the exception's attributes, not by parsing the message.

Input: the starting balance, then `withdraw n` lines. **Output:** one line per command.

```
100
withdraw 30
withdraw 100
```

prints

```
70
declined: balance 70, requested 100; short by 30
```

INPUT 1

```
100
withdraw 30
withdraw 100
```

OUTPUT 1

```
70
declined: balance 70, requested 100;
short by 30
```

REFERENCE SOLUTION

```python
import sys


class InsufficientFunds(Exception):
    def __init__(self, balance, requested):
        super().__init__(f"balance {balance}, requested {requested}")
        self.balance = balance
        self.requested = requested

    @property
    def shortfall(self):
        return self.requested - self.balance


def withdraw(balance, amount):
    if amount > balance:
        raise InsufficientFunds(balance, amount)
    return balance - amount


balance = int(input())
for line in sys.stdin:
    _, amount = line.split()
    amount = int(amount)
    try:
        balance = withdraw(balance, amount)
        print(balance)
    except InsufficientFunds as e:
        print(f"declined: {e}; short by {e.shortfall}")
```

QUIZ

1. Which base should a custom exception subclass?

A. `BaseException`

B. `Exception`

C. `object`

D. `RuntimeError` always

2. What does forgetting `super().__init__(message)` in a custom `__init__` cause?

A. A syntax error

B. `str(e)` is empty and `e.args` is `()`

C. The exception cannot be raised

D. The exception is uncatchable

3. Why give a package one base exception class?

A. Python requires it

B. So a boundary can catch everything the package raises deliberately, separately from bugs

C. For faster matching

D. To avoid `finally`

4. When should details go on the exception as attributes rather than only in the message?

A. Never

B. When handlers need to act on them — parsing `str(e)` is fragile

C. Only for `ValueError`

D. When the message is long

5. What does `class ConfigKeyError(ConfigError, KeyError)` achieve?

A. Nothing — multiple inheritance is not allowed for exceptions

B. It is caught by both `except ConfigError` and `except KeyError`

C. It becomes uncatchable

D. It suppresses tracebacks

ANSWER KEY · CUSTOM EXCEPTIONS — A HIERARCHY FOR YOUR OWN ERRORS

1. B `BaseException` is reserved for interpreter-level signals; subclassing it would escape `except Exception`.

2. B The base initialiser stores the message in `args`, which `str()` and tracebacks read.

3. B `except AppError` catches intentional failures; a `TypeError` from a typo is not an `AppError` and surfaces as a bug.

4. B Attributes are structured data; the message is for humans reading a traceback.

5. B An exception that is a kind of built-in error too keeps old handlers working while offering a specific class.

LESSON 3 · 13 MIN

EAFP and exception-driven flow — suppress, retries, return versus raise

Python raises exceptions for conditions other languages treat as ordinary results: the end of an iterator is `StopIteration`, a missing key is `KeyError`, a missing attribute is `AttributeError`. The language is designed for code that *tries* and *handles*, and the `try` statement is cheap to enter — the cost is in raising, not in guarding. This lesson works through the consequences: the EAFP patterns the standard library expects, `contextlib.suppress` for the "ignore this if it happens" case, when a function should return `None` and when it should raise, sentinel results, retry loops, and the two rules that keep exception-driven code from becoming exception-obscured code.

The cost model

Entering a `try` block costs essentially nothing (since 3.11, literally zero instructions at run time — the handler table is static). Raising and catching costs on the order of a microsecond: building the exception object, unwinding, matching. So:

- A `try` around an operation that *usually succeeds* is faster than a pre-check, because the check runs every time and the raise almost never.
- A `try` around an operation that *usually fails* is slower than a check, because every failure pays for a raise.

The judgement is about frequency, not style. `int(token)` on user input that is nearly always numeric: EAFP. Testing whether half the tokens are numbers: LBYL with a cheap check, or EAFP if the check would be as expensive as the conversion.

The patterns the library expects

```
try:                                    # membership with a side effect
    value = cache[key]
except KeyError:
    value = cache[key] = compute(key)

try:                                    # optional attribute
    hook = obj.on_close
except AttributeError:
    hook = None

try:                                    # first element, or a default
    first = next(iterator)
except StopIteration:
    first = None
first = next(iterator, None)            # the same, built in

try:                                    # a file that may not exist
    with open(path) as f:
        text = f.read()
except FileNotFoundError:
    text = ""
```

The file case is the canonical argument for EAFP: `os.path.exists(path)` followed by `open(path)` has a window in which the file can vanish, and the check duplicates the work `open` does anyway. The library's own conveniences — `dict.get`, `getattr(obj, name, default)`, `next(it, default)`, `str.find` — are EAFP folded into a call, and are preferable to hand-written `try` blocks when they exist.

---

contextlib.suppress

```
from contextlib import suppress

with suppress(FileNotFoundError):
    os.remove(path)                     # delete if present; silently fine if n

with suppress(KeyError):
    del cache[key]
```

`suppress(*exceptions)` is `try: … except X: pass` as a context manager, for the one legitimate use of an empty handler: an operation whose failure means "already in the desired

state". It names the exception, so it cannot become a bare `except`, and its block should be one statement.

Return None or raise?

A function that looks something up and may not find it has two honest designs, and the choice is about what "not found" *means* to the caller:

> **Return** `None` (or an empty collection) when absence is a *normal outcome* the caller expects to test: `dict.get`, `re.match`, `find`. The return type is `X | None`, and the caller must check.
>
> **Raise** when absence is *exceptional* — the caller passed something that should have existed — so that forgetting to handle it is a loud failure rather than a `None` that surfaces three calls later as `AttributeError: 'NoneType'`.

Offer both when the function is general: `find(x)` returning `None` and `index(x)` raising, as `str` does. Never return a *plausible* value (`0`, `""`, `-1`) as an error code from a function whose real results could be that value; that is the design exceptions exist to replace.

Sentinels

When `None` is itself a valid value, a private sentinel distinguishes "absent" from "present and None":

```
_MISSING = object()

def get(self, key, default=_MISSING):
    try:
        return self._data[key]
    except KeyError:
        if default is _MISSING:
            raise
        return default
```

`object()` is unique and compares only by identity, which is exactly what a sentinel needs. The same pattern appears in `dataclasses.MISSING` and in every library that must tell "no argument given" from "argument given as `None`".

Retry loops

```python
def fetch_with_retries(fetch, attempts=3):
    for attempt in range(1, attempts + 1):
        try:
            return fetch()
        except TransientError as e:
            if attempt == attempts:
                raise
            log(f"attempt {attempt} failed: {e}")
    # unreachable: the loop either returned or re-raised
```

Retry only the exceptions that mean "try again" (a timeout, a lock conflict), never a `ValueError` that will fail identically; cap the attempts; re-raise the last failure so the caller sees a real error, not a `None`. Real systems add a delay between attempts (`time.sleep`, with backoff) — omitted here because nothing in a judged program should wait.

Two rules

Catch the narrowest exception that expresses the case you handle. `except KeyError` for a missing key, not `except Exception`; a `try` that catches more than it means to will one day catch a genuine bug and treat it as "key missing".

Keep the handler near the operation and the block small. A `try` around one call reads as "this call may fail thus"; a `try` around a function body reads as "something in here might go wrong", which tells the reader nothing and the maintainer less.

Pitfalls

- EAFP on an operation that fails half the time — measure, or check.
- `except Exception: pass` where `suppress(SpecificError)` around one statement was meant.
- Returning `None` from a function whose callers never check it.
- Returning `-1`, `0` or `""` to signal failure.
- Retrying non-transient errors, or retrying forever.
- Using `None` as a sentinel when `None` is a legitimate value.

Key takeaways

`try` is free to enter; raising costs ~1 µs — EAFP for the usually-successful case, a check for the usually-failing one.

Prefer the built-in defaulted forms (`get`, `getattr`, `next(it, default)`) to hand-written handlers.

`contextlib.suppress(X)` is the named, one-statement form of "ignore this failure".

Return `None` when absence is a normal outcome the caller will test; raise when it is exceptional; never a plausible error code.

A private `object()` sentinel tells absent from `None`; retry only transient failures, capped, and re-raise the last.

EXERCISE 1

First, or a default, or an error

Write `first(iterable, default=_MISSING)` using a private sentinel `_MISSING = object()`: return the first element; if the iterable is empty, return `default` when one was given (even `None`) or raise `ValueError("empty")` when it was not. Implement it with `next()` and `StopIteration`. Each input line is a list of integers optionally followed by `| <default>` where the default may be the word `None`.

Input: lines. **Output:** the first element, the default, or `ValueError`.

```
3 1 2
| 9
| None
```

prints

```
3
9
None
ValueError
```

INPUT 1

```
3 1 2
| 9
| None
```

OUTPUT 1

```
3
9
None
ValueError
```

REFERENCE SOLUTION

```
import sys

_MISSING = object()


def first(iterable, default=_MISSING):
    try:
        return next(iter(iterable))
    except StopIteration:
        if default is _MISSING:
            raise ValueError("empty") from None
        return default


for line in sys.stdin:
    head, sep, tail = line.rstrip("\n").partition("|")
    items = [int(t) for t in head.split()]
    tail = tail.strip()
    try:
        if not sep:
            print(first(items))
        else:
            print(first(items, None if tail == "None" else int(tail)))
    except ValueError as e:
        print(type(e).__name__)
```

EXERCISE 2

Retry with a cap

Simulate an unreliable operation: it raises `TransientError` on its first `fails` calls and then succeeds, returning `"done"`. Write `with_retries(fn, attempts)` that calls `fn` up to `attempts` times, printing `attempt <n> failed` for each transient failure, returns the result on success (print `success on attempt <n>`), and re-raises the last `TransientError` when every attempt failed (the caller prints `gave up after <attempts>`). A `PermanentError` must not be retried: it propagates immediately (the caller prints `permanent`).

Input: one line `fails attempts`; a negative `fails` means the operation raises `PermanentError` at once. **Output:** the attempt log and the outcome.

```
2 3
```

prints

```
attempt 1 failed
attempt 2 failed
success on attempt 3
```

INPUT 1

```
2 3
```

OUTPUT 1

```
attempt 1 failed
attempt 2 failed
success on attempt 3
```

REFERENCE SOLUTION

```
class TransientError(Exception):
    pass


class PermanentError(Exception):
    pass


def make_operation(fails):
    calls = 0

    def operation():
        nonlocal calls
        if fails < 0:
            raise PermanentError("broken")
        calls += 1
        if calls ⩽ fails:
            raise TransientError(f"try again ({calls})")
        return "done"

    return operation


def with_retries(fn, attempts):
    for attempt in range(1, attempts + 1):
        try:
            result = fn()
        except TransientError:
            print(f"attempt {attempt} failed")
            if attempt == attempts:
                raise
        else:
            print(f"success on attempt {attempt}")
            return result


fails, attempts = map(int, input().split())
try:
    with_retries(make_operation(fails), attempts)
except TransientError:
    print(f"gave up after {attempts}")
except PermanentError:
    print("permanent")
```

QUIZ

1. Which is true of the cost of exceptions in CPython 3.11?

A. Entering a `try` is expensive

B. Entering a `try` is free; raising and catching costs about a microsecond

C. Raising is free

D. Both are free

2. Which built-in already expresses 'first element or a default'?

A. `iter(x, default)`

B. `next(iterator, default)`

C. `x.first(default)`

D. `x[0] or default`

3. What is `contextlib.suppress(FileNotFoundError)` equivalent to?

A. `try: … except: pass`

B. `try: … except FileNotFoundError: pass` around the block

C. `if os.path.exists(...)`

D. `finally: pass`

4. When should a lookup function return `None` rather than raise?

A. Always

B. When absence is a normal outcome the caller expects to test

C. When the caller is another module

D. Never

5. Why use `_MISSING = object()` as a default?

A. It is faster than `None`

B. To distinguish 'no argument given' from an explicit `None`

C. Because `None` cannot be a default

D. To make the function hashable

ANSWER KEY · EAFP AND EXCEPTION-DRIVEN FLOW — SUPPRESS, RETRIES, RETURN VERSUS RAISE

1. B The handler table is static, so `try` costs nothing at run time; the cost is in building and unwinding a raised exception.

2. B `next` takes a default returned when the iterator is exhausted, avoiding a hand-written `StopIteration` handler.

3. B It names the exception, so it cannot degrade into a bare `except`, and reads as intent.

4. B Raise when absence is exceptional, so a forgotten check fails loudly rather than as a later `NoneType` error.

5. B A fresh `object()` is unique and compared by identity, so it can never collide with a real value.

LESSON 4 · 14 MIN

Context managers — with, __enter__/__exit__ and contextlib

`with open(path) as f:` closes the file when the block ends — on success, on `return`, on an exception — and that guarantee is the *context manager protocol*: an object with `__enter__` and `__exit__`. Files, locks, database transactions, temporary directories, timers, redirected output and "restore this setting afterwards" are all the same shape: set something up, run a block, tear it down no matter how the block ends. This lesson covers the statement, the two methods and what `__exit__` receives, writing a manager as a class and as a generator with `@contextmanager`, the helpers in `contextlib`, and the relationship to `try` / `finally`.

The statement

```
with open("data.txt") as f:
    data = f.read()
# f is closed here, whether read() succeeded or raised

with open("a.txt") as src, open("b.txt", "w") as dst:      # several, entered
    dst.write(src.read())
```

`with expr as name:` calls `expr.__enter__()`, binds its return value to `name` (a file's `__enter__` returns the file itself), runs the block, then calls `expr.__exit__(...)`. The `as` part is optional when the entered object is not needed (`with lock:`). Since 3.10 several managers may be grouped in parentheses across lines.

`with` is exactly this `try` / `finally`:

```
manager = expr
value = manager.__enter__()
try:
    name = value
    ...block...
except BaseException as e:
    if not manager.__exit__(type(e), e, e.__traceback__):
        raise
else:
    manager.__exit__(None, None, None)
```

Writing one as a class

```
class Timer:
    def __enter__(self):
        self.start = time.perf_counter()
        return self                          # what `as` binds

    def __exit__(self, exc_type, exc, tb):
        self.elapsed = time.perf_counter() - self.start
        return False                         # do not suppress exceptions

with Timer() as t:
    work()
print(f"{t.elapsed:.3f}s")
```

`__exit__` receives the exception's type, instance and traceback — all `None` when the block completed normally. Returning a truthy value **suppresses** the exception; returning `False` / `None` lets it propagate after the cleanup. Suppressing is rare and deliberate (`contextlib.suppress` is that, generalised); most managers clean up and return `False`. `__exit__` runs even if the block executed `return`, `break` or `continue`.

A commit-or-rollback manager is the classic use:

```
class Transaction:
    def __init__(self, db):
        self.db = db

    def __enter__(self):
        self.db.begin()
        return self.db

    def __exit__(self, exc_type, exc, tb):
        if exc_type is None:
            self.db.commit()
        else:
            self.db.rollback()
        return False
```

Writing one as a generator

`contextlib.contextmanager` turns a generator into a manager: everything before `yield` is `__enter__`, the yielded value is what `as` binds, everything after is `__exit__`:

```
from contextlib import contextmanager

@contextmanager
def cd(path):
    old = os.getcwd()
    os.chdir(path)
    try:
        yield
    finally:
        os.chdir(old)

@contextmanager
def transaction(db):
    db.begin()
    try:
        yield db
    except Exception:
        db.rollback()
        raise
    else:
        db.commit()
```

The `try` / `finally` around the `yield` is what makes the teardown run when the block raises — the exception is re-raised *at the yield* inside the generator. Without it, an exception in the block skips the code after `yield`. This form is shorter than the class for anything that is just setup/teardown; the class form is right when the manager has methods or state callers use.

---

contextlib helpers

Helper	Use
`suppress(*exc)`	ignore listed exceptions in the block
`closing(obj)`	call `obj.close()` on exit, for objects with `close` but no `__exit__`
`redirect_stdout(f)`, `redirect_stderr(f)`	send prints to another stream — `io.StringIO()` to capture output in a test
`nullcontext(x)`	a no-op manager that yields `x`; for "optionally use a manager" code paths
`ExitStack()`	enter a variable number of managers and exit them all in reverse

```
import io
from contextlib import redirect_stdout, ExitStack

buf = io.StringIO()
with redirect_stdout(buf):
    print("captured")
buf.getvalue()                          # 'captured\n'

with ExitStack() as stack:
    files = [stack.enter_context(open(p)) for p in paths]   # however many; a
    ...
```

`ExitStack` is the answer to "open N files" and to conditional managers; it also takes `stack.callback(fn, *args)` for arbitrary cleanup. `asynccontextmanager` and `AsyncExitStack` are the async equivalents (Module 17).

Reusable and reentrant managers

A manager built with `@contextmanager` is single-use: the generator runs once, so entering the same object twice raises `RuntimeError`. A class-based manager can be reused (each `with` calls `__enter__` afresh) and, if it keeps no per-entry state, entered again while already active — `threading.RLock` is the standard reentrant example. Keep the per-entry state on the instance in `__enter__` (as `Timer` does with `start`) and the manager is safe to use in a loop; store nothing and it is safe to nest.

Resources that need this

Files (`open`), locks (`threading.Lock` — `with lock:`), sockets and connections, `tempfile.TemporaryDirectory()` , `decimal.localcontext()` , `warnings.catch_warnings()` , `unittest.mock.patch` , `subprocess.Popen` , `zipfile.ZipFile` , database connections and cursors. If an object has `close()` , it almost certainly supports `with` ; if it does not, `closing()` makes it. The rule: any resource that must be released is acquired in a `with` .

Pitfalls

- Opening a file without `with` and relying on garbage collection to close it (unreliable timing, a `ResourceWarning` under `-X dev`).
- `@contextmanager` without `try` / `finally` around the `yield` , so teardown is skipped on error.
- `__exit__` returning `True` accidentally (a helper's return value leaking) and silently swallowing exceptions.
- Doing work that can fail in `__enter__` after acquiring a resource — if it raises, `__exit__` does not run; acquire last.
- Nesting `with` blocks deeply when one line with commas, or `ExitStack` , would do.
- A generator manager that yields twice (`RuntimeError`).

Key takeaways

- `with` calls `__enter__` , runs the block, and always calls `__exit__` — the `try` / `finally` written once, on the type.
- `__exit__(exc_type, exc, tb)` gets the exception or three `None` s; return `False` to propagate, `True` to suppress.
- `@contextmanager` on a generator: setup, `yield` , teardown in a `finally` .
- `suppress` , `closing` , `redirect_stdout` , `nullcontext` , `ExitStack` cover the common needs; `ExitStack` for a variable number of managers.
- Every resource that must be released is acquired in a `with` .

EXERCISE 1

Commit or roll back

Write a class-based context manager `Transaction(account)` where `account` is a one-element list holding a balance. `__enter__` records the starting balance and returns the account; `__exit__` restores it if the block raised (and lets the exception propagate) and otherwise keeps the changes; in both cases it prints `commit <balance>` or `rollback <balance>`. Input blocks run between `begin` and `end`: `add n` changes the balance, `fail` raises `RuntimeError` inside the block (caught outside the `with`, printing `caught`).

Input: commands. **Output:** one line per block end, plus `caught` after a failed block.

```
begin
add 5
add -3
end
begin
add 10
fail
end
```

prints

```
commit 2
rollback 2
caught
```

INPUT 1

```
begin
add 5
add -3
end
begin
add 10
fail
end
```

OUTPUT 1

```
commit 2
rollback 2
caught
```

REFERENCE SOLUTION

```
import sys


class Transaction:
    def __init__(self, account):
        self.account = account

    def __enter__(self):
        self.saved = self.account[0]
        return self.account

    def __exit__(self, exc_type, exc, tb):
        if exc_type is None:
            print(f"commit {self.account[0]}")
        else:
            self.account[0] = self.saved
            print(f"rollback {self.account[0]}")
        return False


account = [0]
lines = [line.split() for line in sys.stdin]
i = 0
while i < len(lines):
    if lines[i] and lines[i][0] == "begin":
        j = i + 1
        block = []
        while lines[j][0] != "end":
            block.append(lines[j])
            j += 1
        try:
            with Transaction(account) as acct:
                for cmd in block:
                    if cmd[0] == "add":
                        acct[0] += int(cmd[1])
                    elif cmd[0] == "fail":
                        raise RuntimeError("failed")
        except RuntimeError:
            print("caught")
        i = j + 1
    else:
        i += 1
```

EXERCISE 2

A generator manager with guaranteed teardown

Write `section(name)` with `@contextlib.contextmanager`: it prints `begin <name>`, yields, and prints `end <name>` — **even when the block raises** (the teardown belongs in a `finally` around the `yield`). Each input line is `<name> ok` or `<name> fail`; a failing block raises `ValueError` inside the `with`, which the caller catches and reports as `caught <name>`.

Input: lines. **Output:** `begin` / `end` pairs, with `caught` lines after failures.

```
load ok
parse fail
```

prints

```
begin load
end load
begin parse
end parse
caught parse
```

INPUT 1

```
load ok
parse fail
```

OUTPUT 1

```
begin load
end load
begin parse
end parse
caught parse
```

REFERENCE SOLUTION

```
import sys
from contextlib import contextmanager


@contextmanager
def section(name):
    print(f"begin {name}")
    try:
        yield
    finally:
        print(f"end {name}")


for line in sys.stdin:
    name, outcome = line.split()
    try:
        with section(name):
            if outcome == "fail":
                raise ValueError(name)
    except ValueError as e:
        print(f"caught {e}")
```

QUIZ

1. What does `__exit__` receive when the block completed normally?

A. Nothing

B. Three `None` values

C. The return value of the block

D. `True`

2. What does returning `True` from `__exit__` do?

A. Re-raises the exception

B. Suppresses the exception so the code after the `with` continues

C. Marks the block as committed

D. Nothing

3. In a `@contextmanager` generator, why wrap the `yield` in `try` / `finally`?

A. It is required syntax

B. Because an exception in the block is raised at the `yield`, and without `finally` the teardown after it is skipped

C. To return a value

D. To allow two yields

4. Which helper enters a variable number of context managers and exits them all?

A. `suppress`

B. `ExitStack`

C. `closing`

D. `nullcontext`

5. Does `__exit__` run when the block executes `return`?

A. No

B. Yes — on every way out of the block

C. Only with `finally`

D. Only if the manager is a generator

ANSWER KEY · CONTEXT MANAGERS — WITH, __ENTER__/__EXIT__ AND CONTEXTLIB

1. B `exc_type`, `exc` and `tb` are all `None` on normal completion; on an exception they describe it.

2. B A truthy return swallows the exception; most managers return `False` (or `None`) to let it propagate after cleanup.

3. B The generator resumes with the exception thrown in; `finally` guarantees the cleanup runs regardless.

4. B `ExitStack.enter_context` registers each manager; they are exited in reverse order when the stack exits.

5. B `with` is a `try`/`finally` in disguise; `return`, `break`, `continue` and exceptions all trigger the exit.

LESSON 5 · 13 MIN

Exception groups, notes and the traceback module

Python 3.11 added two things to exceptions. `ExceptionGroup` carries *several* exceptions at once — the failures of several concurrent tasks, or every validation error in a form — and `except*` handles them by type without losing the rest. `add_note()` attaches context to an exception as it travels up the stack, so the traceback says *which* record was being processed when the `ValueError` happened. This lesson covers both, then the `traceback` module for capturing and formatting exceptions as text, `sys.exc_info`, and the `warnings` module for the failures that are not errors yet.

ExceptionGroup

```
errors = []
for i, row in enumerate(rows):
    try:
        validate(row)
    except ValueError as e:
        e.add_note(f"row {i}")
        errors.append(e)
if errors:
    raise ExceptionGroup("validation failed", errors)
```

An `ExceptionGroup(message, [exceptions])` is itself an exception (a subclass of `Exception`; `BaseExceptionGroup` for the `BaseException` family) whose `.exceptions` tuple holds the members. Raised, it prints a tree traceback showing every member. It replaces the older habits of raising only the first error or stuffing a list into one exception's message. `asyncio.TaskGroup` (Module 17) raises one when several tasks fail.

except*

```
try:
    process_all(rows)
except* ValueError as group:
    print(f"{len(group.exceptions)} value errors")
except* KeyError as group:
    print(f"{len(group.exceptions)} key errors")
```

`except*` matches *members* of a group by type: each clause receives a new `ExceptionGroup` containing only the matching members (even when only one matched), and every clause whose type matches something runs — unlike `except`, where exactly one handler runs. Members that match no clause are re-raised as a smaller group after the handlers. `except*` also handles a plain, non-group exception by wrapping it. `except` and `except*` cannot be mixed in one `try`.

Two methods help without `except*`: `group.subgroup(ValueError)` returns the matching part (or `None`), `group.split(ValueError)` returns `(matching, rest)`. Nested groups are flattened by both.

---

add_note

```
def load(path):
    try:
        return parse(open(path).read())
    except ValueError as e:
        e.add_note(f"while loading {path}")
        raise
```

`add_note(text)` appends to `e.__notes__`, and every note prints under the exception in the traceback. It is the right tool for adding context *without* changing the exception's type or wrapping it — the caller still catches `ValueError`, and the traceback still says where it was raised, but now also says what was being done. Before 3.11 the same need was met by re-raising a new exception `from e`, which changed the type callers had to catch.

Handling a group without except*

Code that must also run on 3.10 catches the group as an ordinary exception and inspects it: `except ExceptionGroup as g:` then `for e in g.exceptions:` to walk the members (nested

groups have their own `.exceptions`), or `g.subgroup(ValueError)` to keep one type. `except*` is the cleaner spelling, but the group is a normal object and everything it offers is reachable by hand — including re-raising a filtered part with `raise g.subgroup(...)` after handling the rest.

The traceback module

```
import traceback, sys

try:
    risky()
except Exception:
    text = traceback.format_exc()          # the full traceback as a string
    log(text)

traceback.print_exc()                      # print it to stderr
traceback.format_exception(e)              # a list of lines from an exceptio
"".join(traceback.format_exception_only(e))   # just the last line: 'ValueErr
```

`format_exc()` inside a handler is how a program records a failure it is going to survive — a worker that must keep processing the queue, a server that answers the next request. `sys.exc_info()` returns `(type, value, tb)` for the exception being handled; since 3.11 `sys.exception()` returns just the instance. `e.__traceback__` is the traceback object on the exception itself; `e.with_traceback(tb)` attaches one.

Notes, the cause (`__cause__`, set by `from`), the context (`__context__`, set automatically when raising inside a handler) and the `__suppress_context__` flag (set by `from None`) are all attributes of the exception object that the traceback printer reads.

Logging exceptions

`logging.exception("message")` (Module 15) logs at ERROR level *with the traceback appended* and must be called from inside a handler. It is `log.error(..., exc_info=True)` spelled out, and it is what a boundary handler should call instead of `print(e)` — `str(e)` alone loses the type and the location.

Warnings

A warning is a condition worth telling the developer about that is not an error: a deprecated API, a probable mistake (`SyntaxWarning: "is" with a literal`), a resource left open.

```
import warnings

def old_api():
    warnings.warn("old_api is deprecated; use new_api", DeprecationWarning, st

with warnings.catch_warnings(record=True) as caught:
    warnings.simplefilter("always")
    old_api()
    print(caught[0].message)
```

Warnings go to stderr once per location by default; `python -W error` turns them into exceptions (which is how a test suite fails on deprecation); `warnings.filterwarnings` configures them; `catch_warnings(record=True)` captures them in a test. `stacklevel=2` makes the warning point at the caller rather than at the `warn` line.

Pitfalls

- Raising only the first of many failures when a group would report them all.
- Mixing `except` and `except*` in one `try` (a `SyntaxError`).
- Expecting `except* ValueError as e` to bind a `ValueError` — it binds a group.
- Wrapping an exception in a new type just to add context, when `add_note` keeps the type.
- `print(e)` at a boundary instead of `traceback.format_exc()` or `logging.exception`.
- Warnings in a library that a user cannot silence — always pass a category and `stacklevel`.

Key takeaways

- `ExceptionGroup(msg, [excs])` raises several exceptions at once; `.exceptions`, `.subgroup`, `.split` inspect it.

`except* T` handles the matching members as a sub-group; every matching clause runs; unmatched members re-raise.

`e.add_note(text)` adds context to the traceback without changing the exception's type.

`traceback.format_exc()` captures the current traceback as text; `logging.exception` logs it; `__cause__` / `__context__` / `__notes__` are the attributes behind the printout.

`warnings.warn(msg, Category, stacklevel=2)`; `-W error` promotes warnings to exceptions; `catch_warnings(record=True)` captures them.

EXERCISE 1

Collect every failure

Validate rows `name age`: a non-integer age raises `ValueError`, a negative one `RangeError(ValueError)` (define it). Validate **every** row, attach a note `row <n>` to each exception with `add_note`, and if any failed raise one `ExceptionGroup("invalid rows", errors)`. Handle it with `except*`: one clause for `RangeError` printing `range: <k>` and one for the remaining `ValueError`s printing `value: <k>`, each followed by the notes of its members, one per line. If nothing failed print `all valid`.

Input: lines. **Output:** the summary lines with their notes, or `all valid`.

```
ada 36
bob x
cy -1
dee y
```

prints

```
range: 1
row 3
value: 2
row 2
row 4
```

INPUT 1

```
ada 36
bob x
cy -1
dee y
```

OUTPUT 1

```
range: 1
row 3
value: 2
row 2
row 4
```

REFERENCE SOLUTION

```
import sys


class RangeError(ValueError):
    pass


def validate(row):
    name, age = row.split()
    try:
        value = int(age)
    except ValueError:
        raise ValueError(f"age {age!r} is not an integer") from None
    if value < 0:
        raise RangeError(f"age {value} is negative")
    return name, value


rows = [line.strip() for line in sys.stdin if line.strip()]
errors = []
for n, row in enumerate(rows, start=1):
    try:
        validate(row)
    except ValueError as e:
        e.add_note(f"row {n}")
        errors.append(e)


def report(label, group):
    print(f"{label}: {len(group.exceptions)}")
    for e in group.exceptions:
        print(e.__notes__[0])


try:
    if errors:
        raise ExceptionGroup("invalid rows", errors)
    print("all valid")
except* RangeError as g:
    report("range", g)
except* ValueError as g:
    report("value", g)
```

EXERCISE 2

The last line of a traceback

Evaluate each input line as an expression. When it raises, print exactly the last line a traceback would show — using `traceback.format_exception_only(e)` and stripping the newline — then, after adding a note `expr: <the expression>` with `add_note`, print the note count and the note. When it succeeds, print `ok <repr of the value>`.

Input: expression lines. **Output:** `ok …`, or the exception line followed by `notes 1` and the note.

```
1 + 1
int("x")
```

prints

```
ok 2
ValueError: invalid literal for int() with base 10: 'x'
notes 1
expr: int("x")
```

INPUT 1

```
1 + 1
int("x")
```

OUTPUT 1

```
ok 2
ValueError: invalid literal for int()
with base 10: 'x'
notes 1
expr: int("x")
```

REFERENCE SOLUTION

```python
import sys
import traceback

for line in sys.stdin:
    expr = line.rstrip("\n")
    try:
        value = eval(expr)
    except Exception as e:
        print("".join(traceback.format_exception_only(e)).rstrip("\n").splitlines()[
        e.add_note(f"expr: {expr}")
        print(f"notes {len(e.__notes__)}")
        print(e.__notes__[0])
    else:
        print(f"ok {value!r}")
```

QUIZ

1. What does `except* ValueError as g` bind?

A. The first `ValueError`

B. An `ExceptionGroup` holding the matching members, even if there is only one

C. A list of exceptions

D. The whole original group

2. How many `except*` clauses can run for one raised group?

A. Exactly one

B. Every clause whose type matches some member

C. None

D. At most two

3. What does `e.add_note("row 3")` change?

A. The exception's type

B. Nothing visible

C. The traceback now shows the note beneath the exception; the type and origin are unchanged

D. The message

4. Which call captures the current traceback as a string inside a handler?

A. `str(e)`

B. `traceback.format_exc()`

C. `repr(e)`

D. `sys.exc_info()[0]`

5. What does `python -W error` do?

A. Disables warnings

B. Turns warnings into exceptions

C. Prints warnings twice

D. Only affects `DeprecationWarning`

ANSWER KEY · EXCEPTION GROUPS, NOTES AND THE TRACEBACK MODULE

1. B `except*` always hands the handler a group; use `g.exceptions` for the members.

2. B Unlike `except`, the clauses partition the group by type; unmatched members are re-raised afterwards.

3. C Notes add context without wrapping or re-typing the exception, so existing handlers still match.

4. B `str(e)` loses the type and the location; `format_exc()` is what a log should record.

5. B Promoting warnings to errors is how a test suite fails on deprecations before they become breakages.

LESSON 6 · 13 MIN

Assertions and defensive code — validate at the boundary, assert the invariant

There are two kinds of "this should not happen": input from outside the program that turns out to be wrong, and a state inside the program that the code's own logic should have made impossible. They need different tools. Bad input is *expected* and is handled by validation that raises a clear exception; an impossible state is a *bug* and is caught by an `assert` that documents the assumption and fails loudly in development. This lesson separates the two, explains what `assert` is and is not (it can be switched off), gives the validation rules for a boundary, and closes with fail-fast design and the `__debug__` flag.

---

assert

```
def mean(xs):
    assert len(xs) > 0, "mean of an empty sequence"
    return sum(xs) / len(xs)
```

`assert condition, message` raises `AssertionError(message)` when the condition is false. It compiles to a check that exists **only when `__debug__` is true**: running with `python -O` strips every `assert` from the bytecode. That single fact decides what an assertion may be used for:

Yes: stating an invariant (`assert 0 ⩽ i < len(xs)`), a postcondition (`assert result ⩾ 0`), an assumption about internal state, a "this branch is unreachable" marker. If it fires, there is a bug in the program.

No: validating user input, checking a file exists, guarding against a bad argument from a caller outside your control, anything whose failure is *possible in correct code*. Under `-O` the check vanishes and the bad input sails through.

Never: an assertion with a side effect (`assert xs.pop()`), which disappears under `-O` along with the effect.

A second trap: `assert (cond, "message")` — parentheses make a non-empty tuple, which is always true, so the assertion never fires. Recent versions warn about it.

Validation at the boundary

Input arrives at a boundary — `main`, a request handler, a public function of a library — and is checked there, once, converting a vague failure into a specific exception with a helpful message:

```
def set_port(value):
    if not isinstance(value, int) or isinstance(value, bool):
        raise TypeError(f"port must be an int, got {type(value).__name__}")
    if not 1 ⩽ value ⩽ 65535:
        raise ValueError(f"port must be 1-65535, got {value}")
    self._port = value
```

`TypeError` for the wrong kind of thing; `ValueError` for the right kind with a bad value. Check the *type* first only when duck typing is not enough — usually a range or format check on the value is all that is needed, and `isinstance` is reserved for cases like `bool` masquerading as `int`. Once validated at the boundary, interior code may *assert* the property rather than re-check it, because a violation there is now a bug.

Fail fast

The earlier a failure surfaces, the closer it is to its cause. A bad configuration should fail at start-up, not on the first request that uses it; a malformed record should fail when parsed, not when summed. Concretely: validate in constructors and setters (Module 8) so an object is never in a bad state; convert input as it is read; raise rather than substituting a default when the default would mask the problem. The cost of a loud failure is a traceback; the cost of a quiet one is a wrong answer nobody notices.

The validation toolkit

- **Ranges and enumerations:** `if not lo ⩽ x ⩽ hi: raise ValueError(...)`; `if kind not in VALID: raise ValueError(f"kind must be one of {sorted(VALID)}, got {kind!r}")`.
- **Formats:** `re.fullmatch` for shapes, `int()` / `float()` in `try` for numbers, `datetime.strptime` for dates.

Structure: `match` with class or mapping patterns for nested data (Module 3); dataclass `__post_init__` (Module 8).

Type at the boundary: `isinstance` against an ABC or protocol, so anything with the right behaviour passes.

Non-negotiable preconditions in a library: raise; the caller is another programmer and wants the traceback.

Every message names the expected and the actual. A caller reading `ValueError: invalid input` learns nothing; `ValueError: quantity must be positive, got -3` fixes the bug.

Defensive, not paranoid

Defensive code checks what it *cannot trust* — external input, the results of I/O, arguments from another module — and trusts what it has already established. Re-validating the same value at every level, wrapping every call in `try`, and testing `if xs is not None` after a function documented to return a list are not defence; they are noise that hides the checks that matter. The shape to aim for: a thin layer at the edge that validates and converts, an interior that asserts its invariants and otherwise assumes them, and exceptions that propagate from where they arise to the boundary that can report them.

---

debug and configuration

`__debug__` is `True` unless the interpreter was started with `-O`. Code that should run only in development — expensive consistency checks, verbose tracing — can be guarded by `if __debug__:`, and is removed by `-O` just as asserts are. Production deployments rarely use `-O` (the speed gain is small), but libraries must be written as if they might, which is the reason for the rule above: never assert what must be checked.

Pitfalls

`assert` to validate input.

`assert (cond, msg)` — always true.

An assertion with a side effect.

Validating deep inside instead of at the boundary, so the same value is checked in five places or none.

Substituting a default for bad input silently.

A message that says only that something is wrong.

Key takeaways

`assert` documents an invariant and fails in development; `-O` removes it, so it must never guard input or have side effects.

Validate external input once, at the boundary, with `TypeError` / `ValueError` and a message naming the expected and actual.

Fail fast: reject bad state in constructors, at parse time, at start-up — a traceback beats a wrong answer.

Interior code asserts what the boundary established and otherwise trusts it.

`assert (x, "msg")` is a tuple and never fires; `if __debug__:` guards development-only checks.

EXERCISE 1

Validate at the boundary

Write `set_port(value)` and `set_host(value)` that validate like a library boundary: `port` must be an `int` (not a `bool`) — otherwise `TypeError("port must be an int, got <type name>")` — in `1..65535` — otherwise `ValueError("port must be 1-65535, got <value>")`; `host` must be a non-empty string without spaces — `TypeError("host must be a str, got <type name>")` or `ValueError("host must be non-empty without spaces, got <repr>")`. Each input line is `port <literal>` or `host <literal>` (parsed with `ast.literal_eval`); print `ok` or `<ExceptionType>: <message>`.

Input: lines. **Output:** one line per input line.

```
port 8080
port 70000
port True
host ""
```

prints

```
ok
ValueError: port must be 1-65535, got 70000
TypeError: port must be an int, got bool
ValueError: host must be non-empty without spaces, got ''
```

INPUT 1

```
port 8080
port 70000
port True
host ""
```

OUTPUT 1

```
ok
ValueError: port must be 1-65535, got
70000
TypeError: port must be an int, got bool
ValueError: host must be non-empty
without spaces, got ''
```

REFERENCE SOLUTION

```python
import ast
import sys


def set_port(value):
    if not isinstance(value, int) or isinstance(value, bool):
        raise TypeError(f"port must be an int, got {type(value).__name__}")
    if not 1 ⩽ value ⩽ 65535:
        raise ValueError(f"port must be 1-65535, got {value}")


def set_host(value):
    if not isinstance(value, str):
        raise TypeError(f"host must be a str, got {type(value).__name__}")
    if not value or " " in value:
        raise ValueError(f"host must be non-empty without spaces, got {value!r}")


for line in sys.stdin:
    field, literal = line.split(maxsplit=1)
    value = ast.literal_eval(literal.strip())
    try:
        (set_port if field == "port" else set_host)(value)
        print("ok")
    except (TypeError, ValueError) as e:
        print(f"{type(e).__name__}: {e}")
```

EXERCISE 2

A stack machine with invariants

Run a tiny stack machine. Input errors are **validated**: `push x` with a non-integer, `pop` / `add` / `mul` on too few operands, or an unknown instruction each print `error: <message>` and continue. The machine's own invariants are **asserted**: after every instruction `assert len(stack) ⩾ 0` (trivially) and, after `add` / `mul`, that the result equals the recomputed value — so an assertion can only fail if the implementation is wrong. `print` prints the top of the stack. At the end print `size <n>`.

Input: instructions. **Output:** error and print lines, then the size.

```
push 2
push 3
add
print
pop
pop
mul
```

prints

```
5
error: pop from empty stack
error: mul needs 2 operands
size 0
```

INPUT 1

```
push 2
push 3
add
print
pop
pop
mul
```

OUTPUT 1

```
5
error: pop from empty stack
error: mul needs 2 operands
size 0
```

REFERENCE SOLUTION

```
import sys

stack = []


def need(n, op):
    if len(stack) < n:
        raise ValueError(f"{op} needs {n} operands" if n > 1 else "pop from empty st


for line in sys.stdin:
    op, *args = line.split()
    try:
        if op == "push":
            try:
                stack.append(int(args[0]))
            except (IndexError, ValueError):
                raise ValueError("push needs an integer") from None
        elif op == "pop":
            need(1, op)
            stack.pop()
        elif op in ("add", "mul"):
            need(2, op)
            b, a = stack.pop(), stack.pop()
            result = a + b if op == "add" else a * b
            stack.append(result)
            assert stack[-1] == (a + b if op == "add" else a * b), "arithmetic invar
        elif op == "print":
            need(1, op)
            print(stack[-1])
        else:
            raise ValueError(f"unknown instruction {op}")
        assert len(stack) >= 0
    except ValueError as e:
        print(f"error: {e}")
print(f"size {len(stack)}")
```

QUIZ

1. What does `python -O` do to `assert` statements?

A. Makes them raise `SystemExit`

B. Removes them entirely from the bytecode

C. Prints them as warnings

D. Nothing

2. What is wrong with `assert (x > 0, "x must be positive")`?

A. Nothing

B. The parentheses make a non-empty tuple, which is always true, so it never fires

C. It raises `SyntaxError`

D. The message is ignored

3. Which exception fits `set_age("ten")` when an int was required?

A. `ValueError`

B. `TypeError`

C. `AssertionError`

D. `KeyError`

4. Where should external input be validated?

A. At every function that touches it

B. Once, at the boundary where it enters, with a clear exception

C. Never — use `assert`

D. Only in tests

5. Why is `assert xs.pop() == 3` dangerous?

A. `pop` cannot be used in asserts

B. The side effect disappears under `-O` along with the check

C. It always fails

D. It is slow

ANSWER KEY · ASSERTIONS AND DEFENSIVE CODE — VALIDATE AT THE BOUNDARY, ASSERT THE INVARIANT

1. B Assertions exist only when `__debug__` is true, which is why they must never guard input.

2. B Write `assert x > 0, "message"` without parentheses.

3. B The wrong *kind* of object is `TypeError`; the right kind with a bad value (`set_age(-1)`) is `ValueError`.

4. B Interior code can then assert the established property rather than re-checking it everywhere.

5. B Assertions must be pure; when stripped, the `pop` never happens and the program's behaviour changes.

CHECKPOINT · 25 MIN

Checkpoint — Errors and exceptions

This checkpoint covers the whole module: the `try` statement with `else` and `finally`, the hierarchy and how handlers match, `raise` with chaining, custom exception classes with data and a base per package, EAFP patterns with `suppress` and sentinels, context managers as classes and generators with `contextlib`, exception groups with `except*` and `add_note`, and assertions versus boundary validation.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- When does the `else` clause of a `try` run, and when does `finally` run?
- Why is `except Exception` before `except ValueError` a mistake, and why is bare `except:` worse?
- What does `raise X from e` do, and what does `from None` do?
- What must a custom exception's `__init__` call, and where do handler-relevant details go?
- What does `__exit__` receive, and what does returning `True` from it mean?
- What does `except* ValueError as g` bind?
- Why must `assert` never validate input?

The three programs are a validated record loader that collects every failure into an `ExceptionGroup` with notes, a transaction context manager whose commit-or-rollback behaviour is observable, and a calculator whose custom exception hierarchy is caught at exactly the right level.

EXERCISE 1

Record loader

Load records `name;age;email`. Define `MissingField(Exception)` (raised when a line has fewer than three fields) and use `ValueError` for a non-integer or negative age. Validate every line, add a note `line <n>` to each failure, and raise a single `ExceptionGroup` if any failed. Handle it with

`except* MissingField` and `except* ValueError`, printing `missing: <n>` / `invalid: <n>` followed by the notes in order. Finally print `loaded <k>` with the number of valid records.

Input: lines. **Output:** the group summaries (if any), then `loaded <k>`.

```
ada;36;a@x
bob;x;b@x
cy;1
dee;-2;d@x
```

prints

```
missing: 1
line 3
invalid: 2
line 2
line 4
loaded 1
```

INPUT 1

```
ada;36;a@x
bob;x;b@x
cy;1
dee;-2;d@x
```

OUTPUT 1

```
missing: 1
line 3
invalid: 2
line 2
line 4
loaded 1
```

REFERENCE SOLUTION

```python
import sys


class MissingField(Exception):
    pass


def parse(line):
    parts = line.split(";")
    if len(parts) < 3:
        raise MissingField(f"expected 3 fields, got {len(parts)}")
    name, age, email = parts[:3]
    age = int(age)
    if age < 0:
        raise ValueError(f"age {age} is negative")
    return name, age, email


records = []
errors = []
for n, line in enumerate((l.strip() for l in sys.stdin if l.strip()), start=1):
    try:
        records.append(parse(line))
    except (MissingField, ValueError) as e:
        e.add_note(f"line {n}")
        errors.append(e)


def report(label, group):
    print(f"{label}: {len(group.exceptions)}")
    for e in group.exceptions:
        print(e.__notes__[0])


try:
    if errors:
        raise ExceptionGroup("load failed", errors)
except* MissingField as g:
    report("missing", g)
except* ValueError as g:
    report("invalid", g)
finally:
    print(f"loaded {len(records)}")
```

EXERCISE 2

Savepoints

Implement `Savepoint(state)` as a context manager over a list of integers: on entry it copies the list; on an exception it restores the copy and prints `rollback`, otherwise prints `commit`; it never suppresses the exception. Input is a nested script: `begin` opens a savepoint (nesting allowed), `end` closes it, `append n` appends, `fail` raises `RuntimeError` — which propagates through every open savepoint until the outermost `begin` … `end` block's caller catches it and prints `caught`. After the script print the state.

Input: commands. **Output:** `commit` / `rollback` / `caught` lines, then `state <values>` (or `state empty`).

```
begin
append 1
begin
append 2
fail
end
append 3
end
```

prints

```
rollback
rollback
caught
state empty
```

INPUT 1

```
begin
append 1
begin
append 2
fail
end
append 3
end
```

OUTPUT 1

```
rollback
rollback
caught
state empty
```

REFERENCE SOLUTION

```
import sys


class Savepoint:
    def __init__(self, state):
        self.state = state

    def __enter__(self):
        self.saved = list(self.state)
        return self.state

    def __exit__(self, exc_type, exc, tb):
        if exc_type is None:
            print("commit")
        else:
            self.state[:] = self.saved
            print("rollback")
        return False


def run(commands, state, i):
    """Execute from index i until the matching 'end'; return the next index."""
    while i < len(commands):
        cmd = commands[i]
        if cmd[0] == "begin":
            with Savepoint(state):
                i = run(commands, state, i + 1)
        elif cmd[0] == "end":
            return i + 1
        elif cmd[0] == "append":
            state.append(int(cmd[1]))
            i += 1
        elif cmd[0] == "fail":
            raise RuntimeError("fail")
    return i


commands = [line.split() for line in sys.stdin if line.strip()]
state = []
i = 0
while i < len(commands):
    try:
        i = run(commands, state, i)
    except RuntimeError:
        print("caught")
        depth = 0
        while i < len(commands):          # skip to the end of the outermost block
            if commands[i][0] == "begin":
                depth += 1
```

```
            elif commands[i][0] == "end":
                depth -= 1
                if depth == 0:
                    i += 1
                    break
            i += 1
print("state", " ".join(map(str, state)) if state else "empty")
```

EXERCISE 3

Calculator with a hierarchy

Define `CalcError(Exception)`, `ParseError(CalcError)` and `DivisionError(CalcError, ZeroDivisionError)`. `evaluate(line)` parses `a op b` (integers, `+ - * /`) — a malformed line raises `ParseError`, division by zero raises `DivisionError`; `/` is integer division. Handle each line with, in this order: `except DivisionError` → `division: <msg>`, `except ParseError` → `parse: <msg>`, `except CalcError` → `calc: <msg>`. Then show that the multiple inheritance works: a second `try` evaluates the same line and catches only `ZeroDivisionError`, printing `also a ZeroDivisionError` when that fires.

Input: lines. **Output:** the result or the handled message; plus the extra line for division errors.

```
6 / 3
1 / 0
1 + x
```

prints

```
2
division: division by zero
also a ZeroDivisionError
parse: bad operand 'x'
```

INPUT 1

```
6 / 3
1 / 0
1 + x
```

OUTPUT 1

```
2
division: division by zero
also a ZeroDivisionError
parse: bad operand 'x'
```

REFERENCE SOLUTION

```
import sys


class CalcError(Exception):
    pass


class ParseError(CalcError):
    pass


class DivisionError(CalcError, ZeroDivisionError):
    pass


def evaluate(line):
    parts = line.split()
    if len(parts) != 3:
        raise ParseError(f"expected 'a op b', got {line!r}")
    a, op, b = parts
    try:
        x, y = int(a), int(b)
    except ValueError as e:
        bad = a if not a.lstrip("-").isdigit() else b
        raise ParseError(f"bad operand {bad!r}") from e
    if op == "+":
        return x + y
    if op == "-":
        return x - y
    if op == "*":
        return x * y
    if op == "/":
        if y == 0:
            raise DivisionError("division by zero")
        return x // y
    raise ParseError(f"unknown operator {op!r}")


for line in sys.stdin:
    line = line.strip()
    try:
        print(evaluate(line))
    except DivisionError as e:
        print(f"division: {e}")
    except ParseError as e:
        print(f"parse: {e}")
    except CalcError as e:
        print(f"calc: {e}")
    try:
```

```
        evaluate(line)
    except ZeroDivisionError:
        print("also a ZeroDivisionError")
    except CalcError:
        pass
```

CHECKPOINT QUESTIONS

1. What does `finally` guarantee?

A. It runs only on success

B. It runs on every exit from the `try` — success, handled exception, unhandled exception, or `return`

C. It runs only when an exception was handled

D. It runs before the `try` block

2. Which handler catches both `IndexError` and `KeyError` in one clause?

A. `except IndexError`

B. `except LookupError`

C. `except ValueError`

D. `except BaseException`

3. What does `raise` with no argument do outside any `except` block?

A. Re-raises the last exception

B. `RuntimeError: No active exception to reraise`

C. Raises `Exception`

D. Nothing

4. A custom exception should store handler-relevant values as…

A. Parts of the message string

B. Attributes set in `__init__`

C. Global variables

D. Return values

5. Which is the EAFP way to use a possibly-missing attribute?

A. `if hasattr(o, "x"): use(o.x)`

B. `try: use(o.x) except AttributeError: fallback()`

C. `if "x" in dir(o): …`

D. `assert hasattr(o, "x")`

6. What must a `@contextmanager` generator do exactly once?

A. `return`

B. `yield`

C. `raise`

D. `print`

7. How does `with a, b:` order the exits?

A. `a` then `b`

B. `b` then `a` — reverse of entry

C. Simultaneously

D. Undefined

8. What happens to members of an `ExceptionGroup` matched by no `except*` clause?

A. They are discarded

B. They are re-raised as a smaller group after the handlers run

C. They are printed

D. They become warnings

9. Which is a legitimate use of `assert`?

A. Checking that user input is an integer

B. Stating a postcondition that only a bug could violate

C. Verifying a file exists

D. Popping from a list

10. What does `e.__cause__` hold after `raise B("x") from a`?

A. `None`

B. The exception `a`

C. The string `"x"`

D. The traceback

11. Why is `except Exception: return 0` inside a computation harmful?

A. It is slow

B. It converts any bug into a silent wrong answer

C. `return` is not allowed in handlers

D. It leaks memory

12. What does `contextlib.redirect_stdout(buf)` enable?

A. Faster printing

B. Capturing everything printed inside the block into `buf` — useful in tests

C. Printing to stderr

D. Disabling output permanently

ANSWER KEY · CHECKPOINT — ERRORS AND EXCEPTIONS

1. B That is what makes it the place for cleanup — and what `with` packages into a type.

2. B Both are subclasses of `LookupError`; a parent class matches its children.

3. B Bare `raise` needs an exception currently being handled.

4. B Handlers read attributes; parsing a message is fragile and breaks when wording changes.

5. B Attempt, then handle the specific failure.

6. B The single `yield` separates setup from teardown; a second one raises `RuntimeError`.

7. B Managers are nested: the last entered is the first exited.

8. B `except*` handles what it names and propagates the rest.

9. B Assertions document invariants and vanish under `-O`; input validation must raise a real exception.

10. B `from` sets the explicit cause, printed as 'The above exception was the direct cause'.

11. B A handler must recover, translate, report or clean up; hiding the failure moves the damage somewhere less visible.

12. B With an `io.StringIO`, `buf.getvalue()` holds the output after the block.

CodeKairo · Python study plan · Module 10: Errors and exceptions. Solve the exercises online to track your progress and earn XP.