



An error travels up through the calls until an `except` that matches its class catches it, so catch the narrowest class you can actually handle. Examples use `d = {"a": 1}` and `nums = [1, 2]`.

CATCH

```
text = "abc"
d = {"a": 1}
nums = [1, 2]
```

```
try:
    n = int(text)
except ValueError:
    n = 0
catch only the error you expect
```

```
except (KeyError, IndexError):
    a tuple catches any of them
```

```
try:
    nums[5]
except IndexError as e:
    msg = str(e)
→ 'list index out of range'
as e gives you the error; str(e) is its message
```

```
try:
    n = int(text)
except ValueError:
    print("not a number")
else:
    print(n * 2)    # only if no error
finally:
    print("done")  # always runs
else: the happy path; finally: clean-up that must happen
```

RAISE

```
def set_age(age):
    if age < 0:
        raise ValueError("age < 0")
    return age
raise an instance, with a message that helps
set_age(-1) → raises ValueError

raise
inside except: re-raise, traceback intact

def load(path):
    try:
        return open(path).read()
    except OSError as e:
        raise RuntimeError(path) from e
from e keeps the first error as the cause
load("app.cfg") → raises RuntimeError
the traceback shows both errors

raise ParseError("...") from None
hides the cause; use it rarely

raise NotImplementedError
a method a subclass must fill in

sys.exit("no input file")
prints the message and quits with status 1
```

LOOK INSIDE AN ERROR

```
e = KeyError("b")

e.args → ('b',)
what was passed when it was raised

str(e) → "'b'"
a KeyError's message is the quoted key

type(e).__name__ → 'KeyError'
the class name, for messages

repr(e) → "KeyError('b')"
class and message: best for logs

isinstance(e, LookupError) → True
test the family, not type(e) ==
```

YOUR OWN EXCEPTIONS

```
class AppError(Exception):
    """Base for this app's errors."""
subclass Exception, never BaseException

class NotFound(AppError):
    pass
a body of pass is enough

issubclass(NotFound, AppError) → True
so except AppError catches both

class HTTPError(AppError):
    def __init__(self, code):
        super().__init__(f"HTTP {code}")
        self.code = code
extra data rides along as attributes

str(HTTPError(404)) → 'HTTP 404'

HTTPError(404).code → 404
```

WHAT CATCHES WHAT

```
issubclass(KeyError, LookupError) → True
LookupError also catches IndexError

issubclass(PermissionError, OSError) → True
OSError: every file, disk and network error

issubclass(SystemExit, Exception) → False
sys.exit() and Ctrl+C slip past except Exception

except Exception as e:
the widest catch to write: at the top, and log it
```

ERRORS YOU WILL MEET

every error explained: pythonmastery.io/errors.html

```
import json
d = {"a": 1}
nums = [1, 2]

int("3.5") → raises ValueError
right type, wrong value

"age: " + 3 → raises TypeError
wrong type for the operation

d["b"] → raises KeyError
d.get("b") gives None instead

nums[5] → raises IndexError
past the end; len(nums) is 2

None.upper() → raises AttributeError
often a function returned None

totl → raises NameError
a typo, or used before it is assigned

1 / 0 → raises ZeroDivisionError

next(iter([])) → raises StopIteration
next(it, default) avoids it

open("nope.txt")
→ raises FileNotFoundError
the path is relative to where python runs

json.loads("{bad}")
→ raises JSONDecodeError
a ValueError, so except ValueError works too

RecursionError
a function keeps calling itself with no way out

import nopkg
ModuleNotFoundError: pip install it, or fix the name
```

HANDY TOOLS

```
import logging, os
from contextlib import suppress
n = 1

with suppress(FileNotFoundError):
    os.remove("cache.tmp")
ignore one expected error, on purpose

e = ValueError("bad row")
e.add_note("in data.csv, line 3")
e.__notes__
→ ['in data.csv, line 3']
notes print under the traceback

try:
    1 / 0
except ZeroDivisionError:
    logging.exception("sum failed")
logs the message and the full traceback

assert n > 0, "n must be positive"
checks your own logic; python -O skips it
```

EXCEPTION GROUPS (3.11+)

```
try:
    raise ExceptionGroup("batch",
        [ValueError(1), OSError(2)])
except* ValueError:
    print("bad data")
except* OSError:
    print("disk problem")
several errors at once; each except* takes its share
```

WATCH OUT

except:
bare except catches Ctrl+C too: write `except Exception`

except Exception:
`pass`
hides every bug, silently: log it or narrow it

except Exception: ...
except ValueError: ...
the second never runs: put the narrow one first

raise AppError(str(e))
drops the original traceback: add from e

ValueError("bad").message
→ raises **AttributeError**
Python 3 has no .message: use `str(e)`

try:
40 lines
except ValueError:
hides which line failed: wrap only the risky line

assert age > 0 # user input
removed by python -O: raise **ValueError** instead

print(e) # after the except block
NameError: e is deleted when the block ends; save it

except ValueError or TypeError:
only catches **ValueError**: use a tuple

finally:
return result
swallows any error from the try: never return in finally