Exceptions

Table of contents

Outline . . . 1
Syntax errors vs. exceptions . . . 2
Common run-time exceptions . . . 2
Try/except . . . 3
Catching specific vs. all exceptions . . . 4
Exercise: robust input reading . . . 4
General try/except framework . . . 5
`else` and `finally` . . . 6
Raising . . . 8
Two exception-handling philosophies . . . 9
Summary . . . 9

Lecture 7A. See csse1001[1] for course logistics and python-exceptions[2] for the full reference on `try`/`except` and raising errors.

Outline

- Syntax errors vs. run-time errors (exceptions)
- Common built-in exception types
- `try`/`except`, `else`, `finally`
- Catching specific vs. all exceptions
- Exercise: robust input reading
- Raising exceptions

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-exceptions.pdf

Syntax errors vs. exceptions

When Python *parses* a file, it returns a **syntax error** if the contents don't correspond to valid Python code. Code that passes parsing transitions to **run-time**, where errors (**exceptions**) can occur — these differ from syntax errors because *you cannot detect them until they trigger*.

As a pipeline: `source → parse → (SyntaxError, or run-time) → exception → execution aborts`.

Run-time errors are bad because they **abort the execution of the program** and all **intermediate work is lost** (unless saved to a file).

Common run-time exceptions

Exception	Description
`AssertionError`	an assertion fails
`IOError`	file does not exist
`IndexError`	index out of range
`KeyError`	key in dict does not exist
`NameError`	variable does not exist
`TypeError`	unexpected type is given to a function
`ValueError`	correct type, but inappropriate value
`ZeroDivisionError`	division by zero attempted

Four of them are shown below at the REPL. An uncaught exception prints a **traceback** before the error line; the traceback is shown in full once below and elided in the later transcripts.

```
>>> xs = [1, 2, 3]
>>> xs[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> xs = {'a': 1}
>>> xs['b']
KeyError: 'b'
```

`TypeError`'s message depends on which operand comes first, because it is the *left* operand's `+` that is tried first:

```
>>> "two" + 2
TypeError: can only concatenate str (not "int") to str
>>> 2 + "two"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The integer casting function has the type contract `int(xs: str) -> int`, and thus `int("ten")` should *not* throw a type error, because `"ten"` is indeed a string — it is the **value** that is wrong:

```
>>> int("ten")
ValueError: invalid literal for int() with base 10: 'ten'
```

Try/except

Catching an exception lets the program handle it gracefully.

```
try:
    <code>   # this code may throw an error
except ExceptionName:
    <code>   # this code is run when the error is of kind ExceptionName
```

(Also called *try-catch* in other languages, e.g. C++.) Typically used in user-facing layers of your software and/or when accessing/interacting with external resources — files, databases, API calls, network calls, and so on.

```
>>> x = 0
>>> 1/x               # without try-except
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
>>> x = 0
>>> try:
...     1/x
... except ZeroDivisionError:      # it doesn't halt the execution
...     print("Please don't divide by zero...")
Please don't divide by zero
```

The `except ZeroDivisionError:` clause doesn't halt the execution. That is the whole point of the construct: the same division that aborted the program without it now just prints a message and carries on.

Catching specific vs. all exceptions

We can catch *any* exception without specifying the error type, but this can lead to undiscovered issues — it's better to catch specific error types:

```
>>> x = 0
>>> try:
...     print(y)
...     1/x
... except:
...     print("You did something fishy...")
```

It is **bad practice to ignore all errors**.

We can also catch among a list of exceptions:

```
>>> try:
...     print(y)     # this throws a name error
...     1/x
... except NameError:
...     print("You're using an undefined name...")
... except ZeroDivisionError:
...     print("You divided by zero...")
```

Exercise: robust input reading

Write a function

```
def io_double() -> int:
```

which takes no inputs, but rather prompts the user for a number and then **doubles** that number.

A first attempt:

```
>>> def io_double() -> int:
...     str_x = input("Number please: ")
...     int_x = int(str_x)
...     return 2*int_x

>>> io_double()
```

```
Number please: 2
4
>>> io_double()
Number please: two
    ...
ValueError: invalid literal for int() with base 10: 'two'
```

Wrapping the cast in a `try/except` inside a `while True:` loop lets the user retry until they succeed:

```
>>> def io_double() -> int:
...     while True:
...         str_x = input("Number please: ")
...         try:
...             int_x = int(str_x)
...             return 2*int_x   # unreachable if the line above errors
...         except ValueError:
...             print("That wasn't a number! Try again...")

>>> io_double()
Number please: two
That wasn't a number! Try again...
Number please: 3
6
```

The `return` line is unreachable if the cast above it errors. The `return` is inside the `try`, so if `int(str_x)` raises, control jumps to the `except` and the loop goes round again. Put the `return` outside the `try` and the retry behaviour disappears.

General try/except framework

```
try:
    <code>
except ExceptionName_0:
    <code>
except ExceptionName_1:
    <code>
    ...
except ExceptionName_k:
    <code>
```

The clauses are not a set — they are tried **in source order**, and the first one whose exception type matches wins. Figure 1 shows the search.

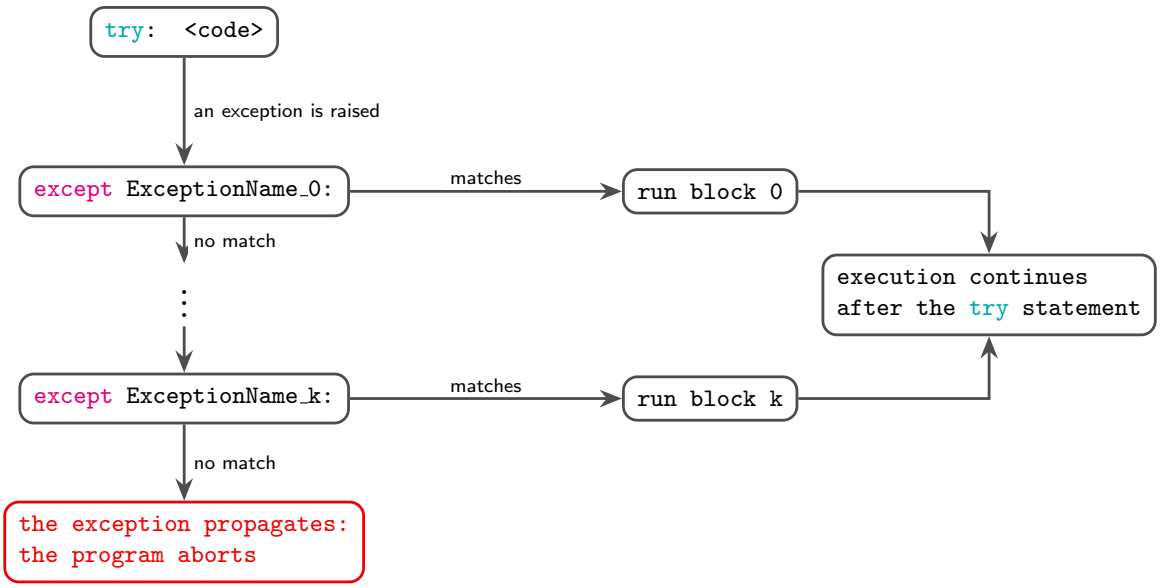

Figure 1: How a raised exception finds its handler: the except clauses are tried top to bottom, the first matching clause runs and the rest are skipped, and if none match the exception propagates and the program aborts. This is why the 'specific exceptions before general ones' rule matters — a general clause placed early swallows everything below it.

Python requires that **specific** exceptions appear before **general** ones:

```
try:
    <code>
except ZeroDivisionError:
    <code>
except:
    <code>
```

`else` and `finally`

A `try` block can carry two further blocks beyond its `except` clauses. Figure 2 shows how the four fit together.

```
try:
```

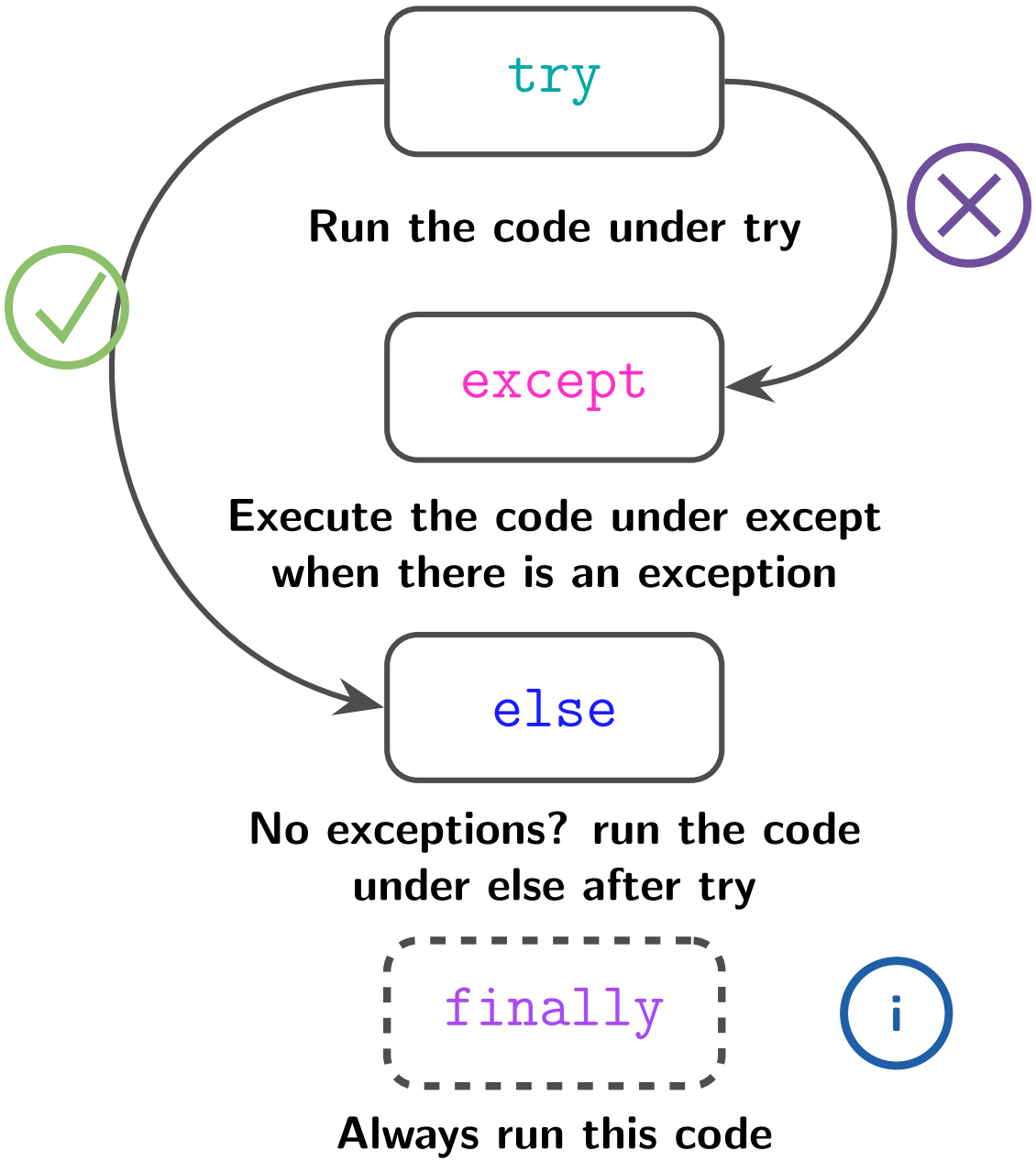

Figure 2: The four blocks associated with a try. The exception path leaves try on the right and lands in except (the purple crossed circle marks it); the no-exception path leaves on the left and skips straight past except into else (green tick). finally hangs below with no arrow into it at all, because it is not on either path — it runs either way, which is what the dashed border and the blue information glyph mark. Each keyword has its own colour: try teal, except magenta, else blue, finally purple.

```
    <code>       # run the code under try
except:
    <code>       # execute the code under except, when there is an exception
else:
    <code>       # no exceptions? run the code under else, after try
finally:
    <code>       # always run this code

try:
    x = int(input("Type a number: "))
    y = 1/x
    # note that if an exception is raised on a line of code here,
    # the following lines don't execute -- keep try blocks as short
    # as possible (don't put everything here!)
except ZeroDivisionError:
    print("Cannot enter zero")
except ValueError:
    print("Must type in a number")
else:
    print(y)
finally:
    print("this block gets always executed")
```

Raising

To **raise** an error (rather than catch it):

```
>>> def safe_div(x: int, y: int) -> float:
...     """
...     Return 1 / (x-y)
...     """
...     if x == y:
...         raise ZeroDivisionError   # halt execution as soon as zero division
...     return 1/(x-y)

>>> safe_div(1, 1)
ZeroDivisionError
```

`raise ZeroDivisionError` halts execution as soon as the zero division is detected. `raise` is the opposite move to `try`/`except` — instead of absorbing a failure it manufactures one, at the earliest point where the failure is knowable, rather than letting a bad value travel further into the program.

```
def get_applicant_age() -> int:
    age = int(input("Enter your age: "))
    if age < 15:
        raise ValueError(f"Age must be >= 15 to apply for a license -- currently it's {age}")
    else:
        return age

try:
    get_applicant_age()
except ValueError:
    print("do something - don't issue license, etc")
```

Two exception-handling philosophies

LBYL (Look Before You Leap) — test first, then act:

```
if b == 0:
    result = None
    print("zero division!")
else:
    result = a/b
```

EAFP (Easier to Ask for Forgiveness than Permission) — act, then handle the failure:

```
try:
    result = a/b
except ZeroDivisionError:
    result = None
    print("zero division!")
```

Summary

We can **catch** errors which **throw** at **runtime** with the **try-except** control structure. We should only use a try-catch when **absolutely necessary**.

Next: 2025-09-09-scope[3] (Lecture 7B).

[3] courses/csse1001/2025-09-09-scope.pdf