

Day 15 Errors and exceptions

Self-Study Guide · About 70 minutes

By the end you will be able to

- Read a traceback from the bottom up and find the line, the function, and the cause of a crash.
- Name the common exceptions and say what kind of mistake each one reports.
- Handle errors with try, except, else, and finally, catching only the exceptions you expect.
- Raise exceptions from your own functions, including a custom exception class, so callers can handle bad input.

What to remember from earlier

- Read an error message from the bottom up. The last line says what went wrong, and the line above it shows where. ([Day 1: Setup and your first program, 1.9 Reading a NameError](#))
- `int()` stops with a `ValueError` when the text is not a whole number, such as `int("1.75")`. ([Day 2: Variables, numbers and basic types, 2.7 Converting Between Types](#))
- Dividing by zero with `/`, `//`, or `%` stops the program with a `ZeroDivisionError`. ([Day 2: Variables, numbers and basic types, 2.4 Floor Division, Remainders, and Powers](#))
- A `while True` loop repeats until a `break` inside it runs, and `continue` skips to the next pass. ([Day 5: Loops, 5.8 break and continue](#))
- Looking up a missing key with square brackets stops with a `KeyError`, while `get()` gives back a default. ([Day 8: Dictionaries, 8.4 Missing Keys: KeyError and get\(\)](#))
- Calling a function runs its body, and `return` hands a value back to the line that called it. ([Day 9: Functions, part 1, 9.4 return: Sending a Value Back](#))
- with `path.open(encoding="utf-8")` opens a file and closes it again, and `json.load` reads JSON data from it. ([Day 14: Files and data formats, 14.9 JSON Files with json.dump and json.load](#))

Make a `day15` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days.

Since Day 1 your programs have stopped with error messages: a `NameError` for a misspelled name, a `ValueError` when `int()` got a word, a `ZeroDivisionError` when a count was 0. Each time, the fix was to change the code. But some errors are not the programmer's fault. A user types `twelve` where you asked for a number. A save file gets deleted, or half written when the laptop's battery dies. A program cannot prevent those, so it must be ready for them. Days 3 and 5 promised that today you would learn how. By the end of today your programs will answer bad input with a clear message instead of a crash, and your to-do manager will survive a missing or broken save file.

15.1 Reading a Traceback from the Bottom Up

Before you can handle an error, you need to read what Python tells you about it. On Day 1 every error came from one line in the main part of a file. Since Day 9 your programs call functions, and

functions call other functions, so an error can happen several calls deep. The report then gets longer.

When a running program hits a problem it cannot get past, Python **raises an exception**: it stops what it was doing and creates an **exception**, a value that describes what went wrong. If nothing in the program deals with the exception, the program stops and prints a **traceback**, the report you met on Day 1. Its name is literal: it traces back through every function call that was in progress when the error happened.

WORKED EXAMPLE An error three calls deep

This program prints the average price of each day's receipts. Tuesday has no receipts.

receipt_average.py

```
1 def average(prices):
2     return sum(prices) / len(prices)
3
4
5 def report(day, prices):
6     print(f"{day}: {average(prices):.2f}")
7
8
9 report("Monday", [4.5, 3.0, 6.0])
10 report("Tuesday", [])
11 report("Wednesday", [2.0, 5.0])
```

```
Monday: 4.50
Traceback (most recent call last):
  File ".../day15/receipt_average.py", line 10, in <module>
    report("Tuesday", [])
    ~~~~~^~~~~~
  File ".../day15/receipt_average.py", line 6, in report
    print(f"{day}: {average(prices):.2f}")
    ~~~~~^~~~~~
  File ".../day15/receipt_average.py", line 2, in average
    return sum(prices) / len(prices)
    ~~~~~^~~~~~
ZeroDivisionError: division by zero
```

The traceback lists the calls in the order they were made, with the **most recent call last** (that is what its first line says). Each call gets its own entry, called a **frame**: a

File ..., line N, in name line saying where, then the line of code itself. Read it from the **bottom up**:

Read	Lines	What it tells you
1st	ZeroDivisionError: division by zero	What went wrong: the kind of exception and its message
2nd	line 2, in average and the code under it	Where it happened: the line that was running when it failed
3rd	line 6, in report	Who called that : report was in the middle of calling average

4th	line 10, in <module>	Who called that: the main part of the file was calling report
Last	Traceback (most recent call last):	Only the heading

Under each line of code, the marks show the part that failed. The ^ carets point at the exact operation (here the /), and the ~ tildes underline the rest of the expression around it.

Monday: 4.50 printed first, because line 9 ran without trouble. Line 11 never ran: once an exception stops the program, nothing after it happens.

So the bottom frame says **where** the program crashed, and the frames above it say **how it got there**. You need both, because the line that crashed is not always the line with the bug, as the Watch Out below shows.

The same reading works when the error happens inside one of Python's built-in functions. Here the text "four" reaches `int()` two calls deep:

split_bill.py

```

1 def parse_guests(text):
2     return int(text)
3
4
5 def split_bill(total, guests_text):
6     guests = parse_guests(guests_text)
7     return total / guests
8
9
10 print(split_bill(60, "4"))
11 print(split_bill(60, "four"))

```

15.0

Traceback (most recent call last):

File ".../day15/split_bill.py", line 11, in <module>
 print(split_bill(60, "four"))

~~~~~

File ".../day15/split\_bill.py", line 6, in split\_bill  
 guests = parse\_guests(guests\_text)

File ".../day15/split\_bill.py", line 2, in parse\_guests  
 return int(text)

ValueError: invalid literal for int() with base 10: 'four'

The bottom line is the Day 2 `ValueError`. The bottom frame is `parse_guests`, line 2, where `int(text)` ran. One frame up, `split_bill` passed it `guests_text`, and one more frame up, line 11 is where "four" came from. Python does not always draw the marks; when they would just underline the whole expression on the line, as in `return int(text)`, it leaves them out.

#### KEY IDEA

Read a traceback from the bottom up: the last line says **what** went wrong, the bottom frame says **where** the program stopped, and each frame above says which call led there. The cause can be in any of the frames.

**WATCH OUT**

**Fixing the bottom line without asking how it got there.** Faced with `ZeroDivisionError` on line 2 of `receipt_average.py`, it is tempting to change `average`. But `average` has no sensible answer for an empty list, and silently returning 0 would print `Tuesday: 0.00`, an average of receipts that never existed. The useful question is the one the upper frames answer: who sent an empty list, and what should happen then? Maybe `report` should print `Tuesday: no receipts` instead. Read every frame before you decide where the fix belongs.

Error messages change a little between versions of Python. If yours are worded differently or show no `~` and `^` marks, the last line still names the same exception.

**PAUSE AND TRY****Question 1** Tracing

What does this program print? Give the output and the last line of the traceback, and name the function that was running when the program stopped.

`trace_quiz.py`

```
1 def lookup(prices, item):
2     return prices[item]
3
4
5 def basket_total(prices, items):
6     total = 0
7     for item in items:
8         total += lookup(prices, item)
9     return total
10
11
12 prices = {"apple": 0.5, "bread": 2.25}
13 print(basket_total(prices, ["apple", "bread"]))
14 print(basket_total(prices, ["apple", "milk"]))
```

## 15.2 Common Exceptions

The last line of a traceback starts with the exception's **type**, a name such as `ValueError`. Every exception has a type, and the type tells you what **kind** of problem happened before you read the

rest of the message. A handful of types cover most beginner errors. You have met each of them on an earlier day:

```
>>> int("ten")
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: 'ten'

>>> "Score: " + 90
Traceback (most recent call last):
...
TypeError: can only concatenate str (not "int") to str

>>> prices = {"apple": 0.5}
>>> prices["pear"]
Traceback (most recent call last):
...
KeyError: 'pear'

>>> scores = [90, 85]
>>> scores[2]
Traceback (most recent call last):
...
IndexError: list index out of range

>>> open("missing_notes.txt", encoding="utf-8")
Traceback (most recent call last):
...
FileNotFoundError: [Errno 2] No such file or directory: 'missing_notes.txt'

>>> 12 / 0
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

In the REPL the traceback is shorter, because there is no file: only the last line matters. Each line asks for something impossible in a different way:

| Exception                      | What it means                                   | Typical cause                                                              |
|--------------------------------|-------------------------------------------------|----------------------------------------------------------------------------|
| <code>ValueError</code>        | The type is right but this value is not usable  | <code>int("ten")</code> : <code>int</code> takes strings, but not that one |
| <code>TypeError</code>         | The operation does not work on this type at all | <code>"Score: " + 90</code> : text plus a number                           |
| <code>KeyError</code>          | A dictionary has no such key                    | <code>prices["pear"]</code> when there is no <code>"pear"</code>           |
| <code>IndexError</code>        | A list or string has no such position           | <code>scores[2]</code> in a list of 2 items (indexes 0 and 1)              |
| <code>FileNotFoundError</code> | No file exists at that path                     | <code>open("missing_notes.txt")</code> with a misspelled name              |

|                   |                                       |                                          |
|-------------------|---------------------------------------|------------------------------------------|
| ZeroDivisionError | Dividing by zero, which has no answer | 12 / 0, or total / count when count is 0 |
|-------------------|---------------------------------------|------------------------------------------|

The difference between `ValueError` and `TypeError` trips people up, so compare them directly. `int("ten")` is a `ValueError`: `int()` happily accepts strings, so the **type** is fine, but the string `"ten"` has no whole-number meaning. `"Score: " + 90` is a `TypeError`: `+` cannot join a string and an int no matter what the values are, so the problem is the **type** itself.

When you see `FileNotFoundError`, check the spelling of the path it names and which folder the program runs from (Day 14).

#### KEY IDEA

The exception type names the kind of problem: `ValueError` for a bad value, `TypeError` for the wrong type, `KeyError` and `IndexError` for a missing key or position, `FileNotFoundError` for a missing file, `ZeroDivisionError` for dividing by zero.

#### PAUSE AND TRY

##### Question 2 Multiple choice

`ages = {"Ana": 15, "Ben": 16}`. Which line stops with a `KeyError`?

- A. `ages["Ana"] + 1`
- B. `ages["ana"]`
- C. `ages["Ana"] + "1"`
- D. `ages.get("Cy")`

##### Question 3 Short answer

Name the exception each line raises, and say in a few words why: (a) `int("3.5")`; (b) `"abc"[3]`; (c) `len(42)`.

## 15.3 Catching an Exception with try and except

Here is a ticket program in the style of Day 5. It works well until a user types a word:

`ticket_crash.py`

```
1 age = int(input("Age: "))
2 print(f"Your ticket costs {5 if age < 12 else 9} dollars.")
3 print("Enjoy the film!")
```

```
Age: twelve
Traceback (most recent call last):
  File ".../day15/ticket_crash.py", line 1, in <module>
    age = int(input("Age: "))
ValueError: invalid literal for int() with base 10: 'twelve'
```

Checking the text with an `if` before calling `int()` is harder than it looks: there are many ways to write a number (12, -3, +7) and even more ways not to (15.9 comes back to this). What you want is to **try** the conversion, and if it fails, do something sensible instead of crashing.

That is the job of the **try statement**. Python runs the code in its `try` block. If an exception of the type you named happens there, Python does not stop the program; it jumps to your `except` block instead. Running an `except` block for an exception is called **catching** or **handling** it.

```
1 try:
2     statements that might raise an exception
3 except ExceptionType:
4     statements to run if that exception happens
```

- `try:` starts the **try block**: the indented lines Python will attempt.
- `except ValueError:` is an **except clause**. It names the exception type to catch, and its indented block is the **handler**.
- After either block finishes, the program carries on with the first line after the whole statement.

### WORKED EXAMPLE A ticket program that survives a word

ticket\_try.py

```
1 try:
2     age = int(input("Age: "))
3     print(f"Your ticket costs {5 if age < 12 else 9} dollars.")
4 except ValueError:
5     print("Please type your age as a number, such as 12.")
6 print("Enjoy the film!")
```

```
Age: twelve
Please type your age as a number, such as 12.
Enjoy the film!
```

Follow what Python does, step by step:

| Step | Line | What happens                                                                                     |
|------|------|--------------------------------------------------------------------------------------------------|
| 1    | 1    | Enter the try block                                                                              |
| 2    | 2    | <code>input</code> returns "twelve"; <code>int("twelve")</code> raises a <code>ValueError</code> |
| 3    | 3    | <b>Skipped</b> : the rest of the try block is abandoned the moment the exception happens         |
| 4    | 4    | The exception is a <code>ValueError</code> , which matches this <code>except</code> clause       |
| 5    | 5    | The handler prints its message; the exception is now handled                                     |
| 6    | 6    | The program carries on normally                                                                  |

If the user types 12 instead, line 2 succeeds, line 3 prints `Your ticket costs 9 dollars.`, the `except` clause is skipped completely, and line 6 runs. Either way the program reaches the end.

The key detail is step 3. An exception does not finish the try block; it **leaves** it at once. To see both paths in one run, this loop tries three strings:

convert\_each.py

```
1 for text in ["12", "twelve", "7"]:
2     print(f"Trying {text}")
3     try:
4         age = int(text)
5         print(f"  Next year: {age + 1}")
6     except ValueError:
7         print("  Not a whole number")
8 print("All done")
```

```
Trying 12
  Next year: 13
Trying twelve
  Not a whole number
Trying 7
  Next year: 8
All done
```

For "twelve", line 4 raises, so line 5 never runs for it. The handler prints its message, and the loop goes on to "7" as usual. A handled exception affects only the try statement it happened in.

#### KEY IDEA

try: runs its block; if a listed exception happens there, Python abandons the rest of the block and runs the matching except handler, then carries on after the statement. If nothing goes wrong, the handler is skipped.

#### WATCH OUT

**Using a variable the failed line never made.** When `int(text)` fails, the assignment to `age` never happens:

skipped\_assignment.py

```
1 text = "twelve"
2 try:
3     age = int(text)
4 except ValueError:
5     print("Not a number.")
6 print(f"Next year you will be {age + 1}.")
```

```
Not a number.
Traceback (most recent call last):
  File "../day15/skipped_assignment.py", line 6, in <module>
    print(f"Next year you will be {age + 1}.")
                                ^^^
NameError: name 'age' is not defined
```

The handler printed its message, the program carried on, and line 6 used `age`, which was never created. Catching the first error just moved the crash to the next line. Code that **needs** the result belongs where it only runs on success: inside the try block after the risky line (as in `convert_each.py`), or in an `else` block, which 15.5 shows. Another fix is a handler that leaves things usable, for example by setting a default value.

#### PAUSE AND TRY

##### Question 4 Tracing

What does this program print? Track `total`.

`try_trace.py`

```
1 total = 0
2 for text in ["5", "five", "2.5", "10"]:
3     try:
4         total += int(text)
5         print(f"added {text}")
6     except ValueError:
7         print(f"skipped {text}")
8 print(f"total {total}")
```

## 15.4 Catching Specific Exceptions

One try block can fail in more than one way, and each way usually needs a different message. This program shares pizza slices: the text might not be a number, or the number of people might be 0.

**WORKED EXAMPLE** One except clause per kind of problem

share\_pizza.py

```

1 def share(slices_text, people_text):
2     try:
3         slices = int(slices_text)
4         people = int(people_text)
5         print(f"{slices // people} slices each")
6     except ValueError:
7         print("Type whole numbers, such as 12 and 4.")
8     except ZeroDivisionError:
9         print("Nobody to share with: people must be at least 1.")
10
11
12 for slices_text, people_text in [("12", "4"), ("12", "four"), ("12", "0")]:
13     share(slices_text, people_text)

```

```

3 slices each
Type whole numbers, such as 12 and 4.
Nobody to share with: people must be at least 1.

```

A try statement can have several except clauses. When an exception happens, Python checks them **from top to bottom** and runs the **first** one whose type matches. The others are skipped.

| Call                | What happens in the try block   | Handler that runs         |
|---------------------|---------------------------------|---------------------------|
| share("12", "4")    | Nothing goes wrong; prints 3    | None                      |
| share("12", "four") | Line 4 raises ValueError        | except ValueError:        |
| share("12", "0")    | Line 5 raises ZeroDivisionError | except ZeroDivisionError: |

Line 12 uses tuple unpacking from Day 7: each pair in the list is split into `slices_text` and `people_text`.

When two exceptions deserve the **same** handling, list them in a tuple, and add `as` with a name to get hold of the exception itself:

share\_pizza\_as.py

```

1 def share(slices_text, people_text):
2     try:
3         slices = int(slices_text)
4         people = int(people_text)
5         print(f"{slices // people} slices each")
6     except (ValueError, ZeroDivisionError) as error:
7         print(f"Could not share: {error}")
8
9
10 for slices_text, people_text in [("12", "4"), ("12", "four"), ("12", "0")]:
11     share(slices_text, people_text)

```

```

3 slices each
Could not share: invalid literal for int() with base 10: 'four'
Could not share: division by zero

```

`except (ValueError, ZeroDivisionError) as error:` catches either type and stores the exception in `error`. Putting `error` in an f-string (or calling `str(error)`) gives its **message**, the text after the colon on a traceback's last line. For a `KeyError`, the message is the missing key in quotes, such as `'pear'`.

What about an exception you did **not** list? It is not caught. It carries on exactly as if there were no `try` statement, and stops the program:

share\_pizza\_uncaught.py

```
1 def share(slices_text, people_text):
2     try:
3         slices = int(slices_text)
4         people = int(people_text)
5         print(f"{slices // people} slices each")
6     except ValueError:
7         print("Type whole numbers, such as 12 and 4.")
8
9
10 share("12", "4")
11 share("12", "0")
12 print("Done sharing")
```

3 slices each

Traceback (most recent call last):

File ".../day15/share\_pizza\_uncaught.py", line 11, in <module>

share("12", "0")

~~~~~^~~~~~

File ".../day15/share_pizza_uncaught.py", line 5, in share

print(f"{slices // people} slices each")

~~~~~^~~~~~

ZeroDivisionError: division by zero

This is a feature, not a gap. You catch the problems you expect and know how to answer. A problem you did not expect is probably a bug, and a bug should stop the program loudly, with a traceback that points straight at it.

#### KEY IDEA

Name the exact exception types you expect: `except ValueError:`, or `except (ValueError, ZeroDivisionError) as error:` to treat several alike and read the message. Python runs the first matching clause; any other exception still stops the program.

#### WATCH OUT

**The bare `except`.** Python lets you write `except:` with no type at all. It catches **every** exception, including the ones that mean your code is wrong. This function has a typo on line 5:

## bare\_except.py

```

1 def total_cost(price_text, quantity_text):
2     try:
3         price = float(price_text)
4         quantity = int(quantity_text)
5         return price * quanti
6     except:
7         print("Please type numbers.")
8
9
10 print(total_cost("2.50", "4"))

```

```

Please type numbers.
None

```

The user typed perfectly good numbers, and the program blamed them. Line 5 raised a `NameError` for `quanti`, the bare `except` caught it, and the real cause disappeared. The function then ended without a `return`, so it gave back `None` (Day 9). A bare `except` even catches **Ctrl+C**: Python reports Ctrl+C as an exception called `KeyboardInterrupt`, so a loop with a bare `except` inside can refuse to stop when you press it.

Name the exception you expect instead, and the typo shows itself:

## bare\_except\_fixed.py

```

1 def total_cost(price_text, quantity_text):
2     try:
3         price = float(price_text)
4         quantity = int(quantity_text)
5         return price * quanti
6     except ValueError:
7         print("Please type numbers.")
8
9
10 print(total_cost("2.50", "4"))

```

```

Traceback (most recent call last):
  File ".../day15/bare_except_fixed.py", line 10, in <module>
    print(total_cost("2.50", "4"))
    ~~~~~^~~~~~
 File ".../day15/bare_except_fixed.py", line 5, in total_cost
 return price * quanti
 ^^^^^^^
NameError: name 'quanti' is not defined. Did you mean: 'quantity'?

```

Now the traceback points at the misspelled name, and fixing it to `quantity` makes the program print `10.0`. Never write a bare `except`. Writing `except Exception:` is almost as bad: it still catches every ordinary error, typos included.

**PAUSE AND TRY****Question 5** Multiple choice

What does this program print?

pick\_except.py

```
1 stock = {"pens": 12, "pads": 0}
2
3 for item in ["pens", "pads", "rulers"]:
4 try:
5 per_box = 24 // stock[item]
6 print(f"{item}: {per_box}")
7 except ZeroDivisionError:
8 print(f"{item}: out of stock")
9 except KeyError:
10 print(f"{item}: we do not sell these")
11 print("Report done")
```

- A. pens: 2, pads: out of stock, rulers: we do not sell these, Report done
- B. pens: 2, pads: out of stock, then a KeyError traceback
- C. pens: 2, pads: 0, rulers: we do not sell these, Report done
- D. pens: 2, then a ZeroDivisionError traceback

## 15.5 else and finally

15.3 left a question open: where should the code go that uses a successful result? You could put it all in the try block, but then every line there is "protected", and an unexpected `ValueError` from some later line would be caught by a handler written for the conversion. Good try blocks are **small**: only the line that can fail in the way you expect. A try statement has two more optional clauses for the rest.

- An **else clause** runs only if the try block finished **without** an exception. Put the code that uses the result there.
- A **finally clause** runs **always**, whether the try block succeeded, failed and was handled, or failed and was not handled. Put cleanup there: anything that must happen no matter what.

The full shape, in the only order Python accepts:

```
1 try:
2 risky line
3 except SomeError:
4 runs if SomeError happened
5 else:
6 runs if nothing went wrong
7 finally:
8 runs every time
```

**WORKED EXAMPLE** Every path through a try statement

else\_finally.py

```

1 for text in ["88", "eighty"]:
2 print(f"Reading {text}")
3 try:
4 score = int(text)
5 except ValueError:
6 print(" except: not a number")
7 else:
8 print(f" else: doubled is {score * 2}")
9 finally:
10 print(" finally: checked")

```

```

Reading 88
 else: doubled is 176
 finally: checked
Reading eighty
 except: not a number
 finally: checked

```

| text     | try block             | except  | else    | finally |
|----------|-----------------------|---------|---------|---------|
| "88"     | succeeds, score is 88 | skipped | runs    | runs    |
| "eighty" | raises ValueError     | runs    | skipped | runs    |

except and else are opposites: exactly one of them runs (when the exception is one you listed). finally runs in both rows. The else block also fixes the Watch Out in 15.3: score is used only where it is certain to exist.

finally runs even when the exception is **not** caught. Here there is no except clause at all, which is allowed:

finally\_uncaught.py

```

1 try:
2 print("Saving...")
3 share = 10 / 0
4 finally:
5 print("Cleanup runs anyway")
6 print("Never printed")

```

```

Saving...
Cleanup runs anyway
Traceback (most recent call last):
 File ".../day15/finally_uncaught.py", line 3, in <module>
 share = 10 / 0
    ~~~^~~~
ZeroDivisionError: division by zero

```

The division fails, Python runs the finally block on its way out, and then the program stops with the traceback. Never printed never prints. finally also runs when the try block is left by return or

`break`. Here the function has already worked out its return value, but `finally` prints before the value is handed back:

`finally_return.py`

```
1 def first_letter(word):
2     try:
3         return word[0]
4     finally:
5         print(f"checked {word}")
6
7
8 print(first_letter("kiwi"))
```

```
checked kiwi
k
```

On Day 14, `with open(...)` closed the file for you even if something went wrong inside the `with` block. Now you can see what that means: `with` does the same job as a `finally` that closes the file, built in.

#### KEY IDEA

Keep the `try` block to the risky line. `else` runs when the `try` block raised nothing, so it is where the result gets used; `finally` runs on every path, even through an uncaught exception or a `return`.

#### WATCH OUT

**Clauses in the wrong order.** The order is fixed: `try`, then every `except`, then `else`, then `finally`. Anything else is a syntax error, so nothing runs at all:

`wrong_order.py`

```
1 try:
2     count = int("7")
3 finally:
4     print("Finished")
5 except ValueError:
6     print("Not a number")
```

```
File ".../day15/wrong_order.py", line 5
    except ValueError:
    ^^^^^
```

```
SyntaxError: invalid syntax
```

Once `finally` has appeared, the `try` statement is over, so Python sees an `except` with nothing to belong to. Move the `finally` block to the end. It also helps to remember what `else` means here. It is not the `else` of an `if`: it runs when the `try` block **succeeded**, not when something failed.

**PAUSE AND TRY****Question 6** Tracing

What does this program print?

else\_trace.py

```
1 line = ""
2 for text in ["4", "x", "0"]:
3     try:
4         number = int(text)
5         result = 12 // number
6     except ValueError:
7         line += "V"
8     except ZeroDivisionError:
9         line += "Z"
10    else:
11        line += str(result)
12    finally:
13        line += "|"
14 print(line)
```

## 15.6 Exceptions Travel Up Through Functions

In 15.1 the traceback listed every function call that was in progress. Those same calls decide where an exception can be caught. When a function raises an exception and does not handle it, the function stops at once (no `return` happens), and the exception arrives at the line that **called** the function, as if that line had raised it. If that line is in a try block with a matching except clause, the exception is caught there. If not, it carries on up to the next caller, and so on. Passing up through the calls like this is called **propagating**. Only if it reaches the main part of the file with nothing to catch it does the program stop with a traceback.

**WORKED EXAMPLE** Caught two calls up

travel\_up.py

```
1 def parse_quantity(text):
2     print(f" parsing {text}")
3     quantity = int(text)
4     print(" parsed")
5     return quantity
6
7
8 def place_order(text):
9     quantity = parse_quantity(text)
10    print(f" ordered {quantity}")
11
12
13 for text in ["3", "lots"]:
14     print(f"Order: {text}")
15     try:
16         place_order(text)
17     except ValueError:
18         print(" cancelled: bad quantity")
19 print("Shop still open")
```

```
Order: 3
  parsing 3
  parsed
  ordered 3
Order: lots
  parsing lots
  cancelled: bad quantity
Shop still open
```

For "lots", follow the exception up:

1. `parse_quantity` prints `parsing lots`, then `int("lots")` on line 3 raises a `ValueError`. `parse_quantity` has no `try` statement, so it stops: line 4 (`parsed`) and the `return` never run.
2. The exception arrives at line 9 in `place_order`, which called `parse_quantity`. No `try` there either, so `place_order` stops too: `ordered` never prints.
3. The exception arrives at line 16, which called `place_order`. That line **is** inside a `try` block, and the `except` clause matches, so the handler prints `cancelled: bad quantity`.
4. The loop carries on, and the program ends normally with `Shop still open`.

This means you can catch an error **far from where it happens**, and usually you should.

`parse_quantity` knows how to convert text, but it does not know what the program should do with a bad quantity: ask again? Skip the order? Show a message in a window? The code near the user knows. A good rule: handle an exception at the place that can do something useful about it. Low-level functions let it propagate, and the code that talks to the user catches it.

**KEY IDEA**

An exception that a function does not handle stops that function and propagates to the line that called it, then to that line's caller, until a matching `except` catches it. Catch it where the program can respond sensibly.

**PAUSE AND TRY****Question 7** Tracing

What does this program print? Watch the `finally` in `outer`.

`travel_trace.py`

```
1 def inner(n):
2     print("inner start")
3     result = 10 // n
4     print("inner end")
5     return result
6
7
8 def outer(n):
9     try:
10        print("outer start")
11        value = inner(n)
12        print("outer got", value)
13    finally:
14        print("outer finally")
15
16
17 try:
18     outer(0)
19 except ZeroDivisionError:
20     print("caught at the top")
```

## 15.7 Raising Your Own Errors with `raise`

So far Python has raised every exception. But your own functions also meet values they cannot use. A function that reads a rating from 1 to 5 gets "9". `int("9")` is happy, so Python raises nothing, but 9 is still wrong. What should the function do?

It could print a message, but then it decides how the problem is shown, which is the caller's job (15.6). It could return `None`, but a caller that forgets to check will carry on with `None` and crash somewhere later, far from the cause. Python's answer is that the function raises an exception itself, exactly like `int()` does. The **raise statement** creates an exception and raises it:

```
1 raise ValueError("a message that explains the problem")
```

- `raise` is a keyword.
- `ValueError(...)` creates a new exception of that type. The text in the parentheses becomes its message.
- From that moment, everything works as in 15.6: the function stops, and the exception propagates to the caller.

### WORKED EXAMPLE A rating that must be 1 to 5

rating.py

```
1 def parse_rating(text):
2     rating = int(text)
3     if not 1 <= rating <= 5:
4         raise ValueError(f"rating must be 1 to 5, not {rating}")
5     return rating
6
7
8 for text in ["4", "9", "great"]:
9     try:
10         rating = parse_rating(text)
11     except ValueError as error:
12         print(f"Skipped {text}: {error}")
13     else:
14         print(f"Saved rating {rating}")
```

```
Saved rating 4
Skipped 9: rating must be 1 to 5, not 9
Skipped great: invalid literal for int() with base 10: 'great'
```

- `"4"`: `int` gives 4, the check on line 3 passes, line 5 returns 4, and the `else` clause saves it.
- `"9"`: `int` gives 9, but `not 1 <= 9 <= 5` is `True` (the Day 4 chained comparison), so line 4 raises a `ValueError` with a message built by an f-string. The `return` on line 5 never runs.
- `"great"`: `int("great")` raises its own `ValueError` on line 2.

The caller handles both with one `except` clause, because both are "a value that cannot be a rating". `ValueError` is the right type to raise for exactly that: the type was fine (a string, then an `int`), but the value was not usable. Raise a `TypeError` instead when a function gets the wrong type altogether.

If nobody catches a raised exception, it ends the program with a traceback like any other, and the message is the one you wrote:

## rating\_uncaught.py

```

1 def parse_rating(text):
2     rating = int(text)
3     if not 1 <= rating <= 5:
4         raise ValueError(f"rating must be 1 to 5, not {rating}")
5     return rating
6
7
8 print(parse_rating("4"))
9 print(parse_rating("9"))
10 print("Not reached")

```

```

4
Traceback (most recent call last):
  File ".../day15/rating_uncaught.py", line 9, in <module>
    print(parse_rating("9"))
    ~~~~~^~~~~~
 File ".../day15/rating_uncaught.py", line 4, in parse_rating
 raise ValueError(f"rating must be 1 to 5, not {rating}")
ValueError: rating must be 1 to 5, not 9

```

That is why a good message says what was expected **and** what arrived. Whoever reads this traceback knows what went wrong without opening the code.

**KEY IDEA**

`raise ValueError("message")` stops your function and hands the problem to its callers, exactly as Python's own errors do. Raise when your function gets input it cannot use, and let the caller decide what to do about it.

**WATCH OUT**

**return instead of raise.** Writing `return ValueError(...)` creates the exception but does not raise it. It is returned like any other value:

## return\_not\_raise.py

```

1 def parse_rating(text):
2 rating = int(text)
3 if not 1 <= rating <= 5:
4 return ValueError(f"rating must be 1 to 5, not {rating}")
5 return rating
6
7
8 for text in ["4", "9"]:
9 try:
10 rating = parse_rating(text)
11 except ValueError as error:
12 print(f"Skipped {text}: {error}")
13 else:
14 print(f"Saved rating {rating}")

```

```
Saved rating 4
Saved rating rating must be 1 to 5, not 9
```

No exception happened, so the except clause was skipped and the else clause "saved" an exception object as if it were a rating. The f-string showed its message. Only `raise` sets off the jump to a handler.

#### PAUSE AND TRY

##### Question 8 Short answer

A classmate's function checks a percentage and, when it is above 100, does `print("Too big!")` and then return `None`. Give two reasons why raising a `ValueError` is better.

## 15.8 Defining a Custom Exception

In the to-do manager, a user can type `two` (not a number) or `7` (a number, but no such task). Both could be `ValueError`, but then the handler cannot tell them apart and cannot give a precise message. Worse, a `ValueError` from some other line, maybe a real bug, would be caught by the same handler.

You can make your **own** exception type with a name that says exactly what went wrong. A **custom exception** takes two lines. Copy this pattern and change only the name:

```
1 class TaskNumberError(Exception):
2 pass
```

Tomorrow, Day 16, explains `class` properly. For today, here is what each piece does:

- `class TaskNumberError` creates a new type named `TaskNumberError`. By convention, the names of exception types end in `Error` and are written in **CapWords** (each word starts with a capital letter, no underscores), like the built-in ones.
- `(Exception)` makes the new type a kind of exception, so it can be raised and caught. Leave it out and Python refuses to raise it.
- `pass` is a statement that does nothing. A block cannot be empty, and this one has nothing to say, so `pass` fills it.

Put the two lines near the top of the file, below the imports, so the type exists before any code raises or catches it. After that, `TaskNumberError` works like `ValueError`: you raise it with a message and catch it by name.

**WORKED EXAMPLE** Telling two problems apart

task\_number.py

```

1 class TaskNumberError(Exception):
2 pass
3
4
5 def check_task_number(tasks, number):
6 if not 1 <= number <= len(tasks):
7 raise TaskNumberError(f"there is no task {number}; choose 1 to
 {len(tasks)}")
8
9
10 tasks = [{"title": "Buy milk", "done": False}, {"title": "Walk dog", "done":
 True}]
11 for typed in ["2", "7", "two"]:
12 try:
13 number = int(typed)
14 check_task_number(tasks, number)
15 except ValueError:
16 print(f"{typed}: please type a number")
17 except TaskNumberError as error:
18 print(f"{typed}: {error}")
19 else:
20 print(f"{typed}: {tasks[number - 1]['title']}")

```

```

2: Walk dog
7: there is no task 7; choose 1 to 2
two: please type a number

```

| typed | What happens                                  | Clause that runs            |
|-------|-----------------------------------------------|-----------------------------|
| "2"   | int gives 2; the check passes                 | else: prints task 2's title |
| "7"   | int gives 7; line 7 raises TaskNumberError    | except TaskNumberError      |
| "two" | int("two") raises ValueError before any check | except ValueError           |

Each kind of problem has its own type, so each gets its own handler and message, and a `ValueError` can never be mistaken for a bad task number.

Uncaught, your exception produces a normal traceback, ending with your type's name and message:

task\_number\_uncaught.py

```

1 class TaskNumberError(Exception):
2 pass
3
4
5 def check_task_number(tasks, number):
6 if not 1 <= number <= len(tasks):
7 raise TaskNumberError(f"there is no task {number}; choose 1 to
8 {len(tasks)}")
9
10 tasks = [{"title": "Buy milk", "done": False}]
11 check_task_number(tasks, 1)
12 print("Task 1 is fine")
13 check_task_number(tasks, 3)

```

refuse it with a `TypeError` saying exceptions must derive from `BaseException`. Copy the pattern exactly, with `(Exception)`.

#### PAUSE AND TRY

##### Question 9 Tracing

What does this program print? Each seat goes through two functions before any handler.

seat\_check.py

```
1 class SeatError(Exception):
2 pass
3
4
5 def check_seat(row, seat):
6 if seat > 10:
7 raise SeatError(f"row {row} has only 10 seats")
8 return f"{row}{seat}"
9
10
11 def book(row, seat_text):
12 seat = int(seat_text)
13 label = check_seat(row, seat)
14 print(f"booked {label}")
15
16
17 for row, seat_text in [("A", "4"), ("B", "12"), ("C", "two")]:
18 try:
19 book(row, seat_text)
20 except SeatError as error:
21 print(f"sorry, {error}")
22 except ValueError:
23 print("seat must be a number")
24 print("done")
```

## 15.9 Ask Forgiveness, Not Permission

You now have two ways to deal with something that might fail. You can **check first**, and only act if the check passes. Or you can **just act**, and handle the exception if it fails. Both appear in Python code, and they have names:

- **LBYL**, "look before you leap": test the conditions with `if`, then act.

- **EAFP**, "easier to ask forgiveness than permission": act inside a `try`, and handle the failure in `except`.

### WORKED EXAMPLE The same lookup, two styles

lookup\_styles.py

```

1 stock = {"pens": 12, "pads": 5}
2
3 # Look before you leap: check first, then act.
4 item = "rulers"
5 if item in stock:
6 print(f"{item}: {stock[item]}")
7 else:
8 print(f"{item}: not stocked")
9
10 # Easier to ask forgiveness than permission: act, and handle the failure.
11 try:
12 print(f"{item}: {stock[item]}")
13 except KeyError:
14 print(f"{item}: not stocked")

```

```

rulers: not stocked
rulers: not stocked

```

Both halves give the same result. The LBYL half asks `item in stock` first (Day 8). The EAFP half looks the item up and catches the `KeyError` if it is missing.

For a dictionary, both styles are fine. But in other places, checking first is harder than it looks. Think about checking text before `int()`. These all work:

```

>>> int("-7")
-7
>>> int(" 42 ")
42
>>> int("+3")
3
>>> int("4.0")
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 10: '4.0'

```

A minus sign, spaces around the number, and a plus sign are all accepted, but `"4.0"` is not. An `if` that tried to copy `int()`'s rules would be long and still miss a case. The Day 12 to-do manager stripped the spaces off and then checked `all(ch in "0123456789" for ch in text)`: two steps to copy part of `int()`'s rules, and it still refuses `+3`, which `int()` accepts. EAFP asks the expert: let `int()` decide, and catch the `ValueError`.

Files make the strongest case for EAFP. On Day 14 the to-do manager checked `SAVE_FILE.exists()` before loading. But a file can be deleted **between** the check and the open, and a file that exists can still be unreadable, or not valid JSON. The check answers one question once; it cannot promise that the next line will work. Trying to open the file and catching `FileNotFoundError` covers the real moment of opening.

| Style | Looks like                                       | Good for                                                                                 |
|-------|--------------------------------------------------|------------------------------------------------------------------------------------------|
| LBYL  | <code>if key in prices: then prices[key]</code>  | Simple, cheap, complete checks; the "failure" is a normal case ( <code>get()</code> too) |
| EAFP  | <code>try: int(text) / except ValueError:</code> | Checks that are hard to write exactly, and anything outside your program, such as files  |

Python programmers lean toward EAFP, because it puts the normal path first and lets the code that does the work decide what counts as failure. Neither is wrong; choose the one whose check is honest and complete.

#### KEY IDEA

LBYL checks with `if` before acting; EAFP acts in a `try` and handles the exception. Prefer EAFP when a correct check would be complicated or could be out of date by the time you act, as with converting text and opening files.

#### PAUSE AND TRY

##### Question 10 Short answer

A program loads `settings.json`. Version 1 does `if path.exists():` and then opens and reads the file with `json.load`. Version 2 opens and reads it in a `try` with `except FileNotFoundError:` and `except json.JSONDecodeError:`. Give two situations that crash version 1 but not version 2.

## Putting it together

A teacher keeps scores in a text file, one `name,score` per line. Some lines have mistakes. The program reports the average of the good lines and explains each bad one, and it copes when the file is missing.

`class_scores.txt`:

```
Ana,88
Ben,ninety
Cy
Dee,105
Eli,72
```

`score_report.py`

```
1 from pathlib import Path
2
```

```

3
4 class ScoreRangeError(Exception):
5 pass
6
7
8 def read_lines(path):
9 with open(path, encoding="utf-8") as file:
10 return [line.strip() for line in file]
11
12
13 def parse_line(line):
14 name, score_text = line.split(",")
15 score = int(score_text)
16 if not 0 <= score <= 100:
17 raise ScoreRangeError(f"{score} is outside 0 to 100")
18 return name, score
19
20
21 def read_scores(path):
22 try:
23 lines = read_lines(path)
24 except FileNotFoundError:
25 print(f"No file named {path}: nothing to report.")
26 return []
27 scores = []
28 for number, line in enumerate(lines, 1):
29 try:
30 name, score = parse_line(line)
31 except (ValueError, ScoreRangeError) as error:
32 print(f"Line {number} skipped ({line}): {error}")
33 else:
34 scores.append(score)
35 return scores
36
37
38 for path in [Path("class_scores.txt"), Path("missing_scores.txt")]:
39 scores = read_scores(path)
40 if scores:
41 print(f"{len(scores)} scores, average {sum(scores) / len(scores):.1f}")

```

Line 2 skipped (Ben,ninety): invalid literal for int() with base 10: 'ninety'  
 Line 3 skipped (Cy): not enough values to unpack (expected 2, got 1)  
 Line 4 skipped (Dee,105): 105 is outside 0 to 100  
 2 scores, average 80.0  
 No file named missing\_scores.txt: nothing to report.

| Line        | Inside parse_line                                                        | Caught by read_scores as | Idea       |
|-------------|--------------------------------------------------------------------------|--------------------------|------------|
| Ana, 88     | Splits, converts, passes the range check                                 | else: score kept         | 15.5       |
| Ben, ninety | int("ninety") raises ValueError                                          | ValueError               | 15.3, 15.4 |
| Cy          | split(",") gives one item; unpacking it into two names raises ValueError | ValueError               | 15.2       |
| Dee, 105    | Line 17 raises ScoreRangeError                                           | ScoreRangeError          | 15.7, 15.8 |

|         |                 |                  |      |
|---------|-----------------|------------------|------|
| Eli, 72 | All checks pass | else: score kept | 15.5 |
|---------|-----------------|------------------|------|

- `read_lines` does not handle anything. When `missing_scores.txt` does not exist, its `open` raises `FileNotFoundError`, which propagates (15.6) to `read_scores`, where the `except` clause prints a message and returns an empty list. The final `if scores:` then skips the average, so there is no `ZeroDivisionError` either.
- The `try` block around `read_lines` holds one line, and the `try` block in the loop holds one line. Each catches only the exceptions it expects, named exactly (15.4), and none is bare.
- The unpacking line shows that you do not need to predict every way a line can be broken: `int()` and unpacking raise `ValueError` for you, the program checks only what Python cannot (the range), and one handler covers them all (EAFP, 15.9).
- The average is  $(88 + 72) / 2 = 80.0$ .

## Vocabulary

| Term                             | Definition                                                                         | Example                                         |
|----------------------------------|------------------------------------------------------------------------------------|-------------------------------------------------|
| <b>exception</b>                 | A value Python creates to describe a problem while a program runs                  | <code>ValueError("...")</code>                  |
| <b>raise</b> (an exception)      | Signal a problem: stop normal work and start looking for a handler                 | <code>int("ten")</code> raises one              |
| <b>traceback</b>                 | The report of the calls in progress when an uncaught exception stopped the program | Traceback (most recent call last):              |
| <b>frame</b>                     | One call in a traceback: where it was and the line running                         | line 2, in average                              |
| <b>exception type</b>            | The kind of exception, named on the traceback's last line                          | <code>KeyError</code>                           |
| <b>message</b>                   | The text after the type, explaining the problem                                    | division by zero                                |
| <b>try statement / try block</b> | Code Python attempts, with handlers ready if it fails                              | <code>try:</code>                               |
| <b>except clause / handler</b>   | Names an exception type to catch / the block that runs when it is caught           | <code>except ValueError:</code>                 |
| <b>catch / handle</b>            | Run a handler for an exception, so the program carries on                          | the ticket program                              |
| <b>bare except</b>               | An <code>except</code> clause with no type; catches everything, so never use it    | <code>except:</code>                            |
| <b>else clause</b> (try)         | Runs only if the <code>try</code> block raised nothing                             | <code>else: print(score * 2)</code>             |
| <b>finally clause</b>            | Runs on every path out of the <code>try</code> statement                           | <code>finally: print("checked")</code>          |
| <b>propagate</b>                 | An unhandled exception passing up to the caller, then its caller                   | from <code>parse_quantity</code> up to the loop |

|                         |                                                                           |                                          |
|-------------------------|---------------------------------------------------------------------------|------------------------------------------|
| <b>raise statement</b>  | Raises an exception from your own code                                    | <code>raise ValueError("too big")</code> |
| <b>custom exception</b> | Your own exception type, made with <code>class Name(Exception):</code>    | <code>TaskNumberError</code>             |
| <b>pass</b>             | A statement that does nothing, for a block with nothing to say            | <code>pass</code>                        |
| <b>LBYL</b>             | "Look before you leap": check with <code>if</code> , then act             | <code>if key in prices:</code>           |
| <b>EAFP</b>             | "Easier to ask forgiveness than permission": try, then handle the failure | <code>try: prices[key]</code>            |

## Summary

- An uncaught exception stops the program with a **traceback**. Read it from the bottom up: the last line gives the **type** and **message**, the bottom frame shows where it happened, and the frames above show which calls led there; the cause may be in any of them.
- The common types each name a kind of problem: `ValueError`, `TypeError`, `KeyError`, `IndexError`, `FileNotFoundError`, and `ZeroDivisionError`.
- A **try statement** runs its block, jumps to the first matching **except** handler if a listed exception happens, runs **else** if nothing did, and runs **finally** every time. Catch only the types you expect, never with a bare `except`.
- An unhandled exception **propagates** from a function to its caller, so you can catch it where the program can respond. Your functions can **raise** exceptions too, including **custom** types made with `class Name(Exception): pass`.
- **EAFP** (try, then handle) is often clearer and safer than **LBYL** (check first), especially for converting text and opening files.

## Practice

Do these in your `day15` folder. Type each program yourself, run it, and only then open the answer.

### Question 11 Short answer

**Ask until valid.** Write a function `read_int(prompt)` that keeps asking with `input(prompt)` until the user types a whole number, then returns it as an int. After each bad answer, print a short message. Use it to ask for an age and a number of tickets.

**Question 12** Short answer

**A save file that can go wrong.** On Day 14 the to-do manager saved its tasks to `tasks.json` and loaded them at the start, checking `SAVE_FILE.exists()` first. Change `load_tasks(path)` so that a missing file means "no tasks yet" and a corrupted file (one that is not valid JSON) prints a warning and starts with no tasks, then let `main` call it without the `exists()` check. While you are there, make the task-number prompt use `try` and `int()`.

**Question 13** Short answer

**Raise and handle.** Write a function `parse_price(text)` that turns text such as `"2.50"` into a float, and raises `ValueError` with a clear message if the price is 0 or less. Then write a caller that adds up a list of price strings, skipping (with a message) any that are not usable.

## Review questions

**Question 14** Tracing

What does this program print? `finally` and `return` are both in play.

`safe_divide.py`

```
1 def safe_divide(a, b):
2 try:
3 return a / b
4 except ZeroDivisionError:
5 return None
6 finally:
7 print(f"checked {a} / {b}")
8
9
10 print(safe_divide(6, 3))
11 print(safe_divide(1, 0))
```

**Question 15** Debugging

This program should print no price for any item missing from the dictionary. It prints bread: 2.25 and then crashes. Explain the bug and fix it.

wrong\_catch.py

```
1 prices = {"apple": 0.5, "bread": 2.25}
2
3 for item in ["bread", "milk"]:
4 try:
5 print(f"{item}: {prices[item]:.2f}")
6 except ValueError:
7 print(f"{item}: no price")
```

**Question 16** Tracing

What does this program print?

review\_flow.py

```
1 def check_quantity(text):
2 quantity = int(text)
3 if quantity < 1:
4 raise ValueError("must order at least 1")
5 return quantity
6
7
8 def order(text):
9 try:
10 quantity = check_quantity(text)
11 except ValueError as error:
12 print(f"{text}: {error}")
13 else:
14 print(f"{text}: ordered {quantity}")
15 finally:
16 print("--")
17
18
19 for text in ["2", "0", "x"]:
20 order(text)
```

**Question 17** Multiple choice

What is the **third** line this program prints?

review\_tuple.py

```
1 scores = {"Ana": [90, 85], "Ben": [70]}
2
3 for name, index in [("Ana", 1), ("Ben", 1), ("Cy", 0)]:
4 try:
5 print(scores[name][index])
6 except (KeyError, IndexError) as error:
7 print(f"problem: {error}")
```

- A. problem: list index out of range
- B. problem: 'Cy'
- C. problem: Cy
- D. A KeyError traceback

**Question 18** Short answer

In the corrupted-file traceback from Practice question 12, the bottom frames are in Python's `json` module, in files you did not write. Explain how to read such a traceback to decide what to change in your own program.

**Question 19** Short answer

Day 12's to-do manager checked a task number with `all(ch in "0123456789" for ch in text)` before `int(text)`, then checked `1 <= number <= len(tasks)`. Which of these two checks would you replace with `try` and `except ValueError`, which would you keep as an `if` (or move into a function that raises `TaskNumberError`), and why?

## Go further

The official Python Tutorial covers today's topics, with a few more advanced features such as exception chaining and exception groups, in Errors and Exceptions. Its section on user-defined exceptions uses classes, which you meet tomorrow on Day 16.

## Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

☐ I can read a traceback from the bottom up and find where and why a program stopped. Reread:

15.1 Reading a Traceback from the Bottom Up

☐ I can say what each common exception type means. Reread: 15.2 Common Exceptions

☐ I can catch an exception with try and except and say which lines are skipped. Reread: 15.3

Catching an Exception with try and except

☐ I can catch specific exception types, read the message with as, and explain why a bare except is a mistake. Reread: 15.4 Catching Specific Exceptions

☐ I can use else and finally and predict when each runs. Reread: 15.5 else and finally

☐ I can follow an exception up through function calls to the handler that catches it. Reread: 15.6

Exceptions Travel Up Through Functions

☐ I can raise a ValueError with a clear message from my own function. Reread: 15.7 Raising Your Own Errors with raise

☐ I can define, raise, and catch a custom exception. Reread: 15.8 Defining a Custom Exception

☐ I can choose between checking first and trying first, and explain the choice. Reread: 15.9 Ask Forgiveness, Not Permission