Lecture V - Errors and Debugging

Programming with Python

Dr. Tobias Vlćek
Kühne Logistics University Hamburg - Fall 2026

📋 Checkpoint 2

Before the doors open. The first 40 minutes are the checkpoint. It starts now.

- **Individual work**: no AI, no neighbors, no chat
- The **link and QR** are handed out in class. Open it and start
- ~6 short tasks: write code, trace code, fix a bug, answer a multiple choice
- It sweeps **Sessions III–IV**: functions, scope, classes, lists and dictionaries, comprehensions

When you're done

Menu → *Download* → *Download Python code* → upload the `.py` to the **"Checkpoint 2"** assignment on Moodle, before the 40 minutes are up. **No retakes**: one sitting.

. . .

> ℹ Note
>
> The green ✅ live checks are **provisional**. The final grading runs on my side. And take a breath: everything in it was rehearsed in the labs.

Episode 5: The 3-AM Checkout

After the checkpoint

Pens down. The checkpoint is behind you. Now the reason we're all here.

. . .

At 3 AM, running on his fourth energy drink, Tobi rewrote the **entire checkout** "to make it faster." He went to bed very pleased. This morning two things are true: the till is a minefield of crashes, and the **health inspector** just called to announce a surprise visit.

. . .

So today the code learns to survive things going wrong. We read a **traceback**, catch failures with try/except, and let our own code refuse bad input.

Reading the Explosion

Tobi's 3-AM checkout, live

A customer typed `generous` into the tip box. Tobi's new checkout did this:

```
tip_text = "generous"
tip = float(tip_text)     # 💥
```

...

Python stopped and printed a **traceback**, a crash report. It looks scary, but it is the most useful thing on your screen:

```
Traceback (most recent call last):
  File "checkout.py", line 12, in <module>
    tip = float(tip_text)
ValueError: could not convert string to float: 'generous'
```

Read the traceback bottom-up

You read a traceback from the **bottom up**:

- the **last line** names *what* went wrong (`ValueError`) and gives a hint
- the lines **above** show *where*: the file, the line number, the guilty code

```
      File "checkout.py", line 12, in <module>   ← WHERE it happened
        tip = float(tip_text)                    ← the exact line
    ValueError: could not convert ...               ← WHAT went wrong (read
me first)
```

...

Two facts and you know where to go: **line 12**, and it's a **`ValueError`**. The scary wall of text is really just an address and a reason.

The big five (1)

A handful of exception types cover almost everything you'll hit. The first three:

- **`ValueError`**: right *type*, senseless *value*, as in `int("lots")`
- **`TypeError`**: the wrong type entirely, like `"Bowl " + 9`
- **`KeyError`**: a dictionary key that isn't there (`table_map["C3"]`)

...

Each one is Python refusing to guess. `int("lots")` has no sensible number; `"Bowl " + 9` mixes text and a number; `table_map["C3"]` asks for a key that was never added.

The big five (2)

The other two you'll meet constantly:

- **`IndexError`**: a list position past the end, like `seats[9]` on a 3-item list

- `ZeroDivisionError`: dividing by zero, as in `88.00 / 0` when splitting a bill among no guests

...

You don't have to memorize all of Python's exceptions. Recognize these five on sight, and read the last line for the rest.

`try` / `except`: catch the fall

First, the happy path: text that **looks** like a number converts cleanly; text that doesn't, explodes. A `try` / `except` block runs the risky code and catches the failure instead of stopping the program:

```
print(float("5.50"))          # numeric text converts cleanly → 5.5

try:
    tip = float("generous")   # non-numeric text raises ValueError...
except ValueError:
    tip = 0.0                 # ...caught here, so the checkout keeps going

print(tip)
```

```
5.5
0.0
```

...

The risky line goes in the `try`; the recovery goes in the `except`. No crash. The program lands in the `except` and carries on.

Catch the *specific* type

Name the exact exception you expect. `except ValueError:` catches **only** value errors and lets anything else through:

```
try:
    tip = float(tip_text)
except ValueError:            # only this kind
    tip = 0.0
```

...

> ⚠ Warning
>
> A **bare** `except:` catches *everything*, including your own typos (a misspelled variable would vanish silently). Catch the **specific** type, so real bugs still surface.

Predict: which explosion?

Tobi builds a table label. `guests` is the number `3`. What does the **last line** do?

```
guests = 3
label = "Party of " + guests
print(label)
```

a) raises `TypeError` b) prints `Party of 3` c) raises `ValueError`

. . .

Predict first. Pick a letter, then I reveal the answer.

Answer: text and numbers won't mix

a) raises `TypeError`: `+` can glue two strings *or* add two numbers, but it refuses to mix a string and an `int`. The fix is to convert first (`str(guests)`):

```
guests = 3
try:
    label = "Party of " + guests
except TypeError as e:
    print("TypeError:", e)
```

```
TypeError: can only concatenate str (not "int") to str
```

. . .

`as e` keeps the error object in a variable, so you can print its message instead of losing it. Any name works; `e` is the habit.

Your turn — 10 minutes

Open the exercise (scan the QR or type the link):

python.tobiasvlcek.com/notebooks/ex_05_a/

First **predict** what happens, then run it.

Debugging and Defending

The debugging loop

When something is wrong, you don't guess randomly. You follow a loop:

1. **Read** the traceback: it hands you the *what* and the *where*
2. **Reproduce** the crash: make it happen on demand
3. **Isolate**: narrow down to the smallest failing piece
4. **Fix**: change one thing, then run again

. . .

The inspector debugs the same way: find the broken thing, make it repeat, corner it, fix it. Panic is not a step.

`print()` — the beginner's flashlight

The simplest debugger is a well-placed `print()`. When you can't see what a value *is*, shine a light on it:

```
def line_total(qty, price):
    print("DEBUG qty =", qty, "price =", price)   # the flashlight
    return round(qty * price, 2)

print(line_total(3, 5.50))
```

```
DEBUG qty = 3 price = 5.5
16.5
```

. . .

Real debuggers with breakpoints exist and are wonderful, but a `print()` in the right spot solves most beginner bugs in seconds. Remove it once you've seen enough.

`raise` — when your code should refuse

Sometimes *your own* code should reject bad input on the spot. The inspector's rule is non-negotiable: **every price ≥ 0**. `raise` throws an error deliberately:

```
def charge(amount):
    if amount < 0:
        raise ValueError("price cannot be negative")
    return amount

try:
    charge(-4.50)                  # your code refuses it...
except ValueError as e:
    print("refused:", e)           # ...on your terms
```

```
refused: price cannot be negative
```

...

A bad value stops *here*, at the door, instead of poisoning the books three screens later.

`assert` — a tripwire for invariants

An `assert` guards an *invariant*, a fact that must **always** hold. If it's true, nothing happens; if it's false, the program stops right there with an `AssertionError`:

```
subtotal = 9.60
assert subtotal >= 0, "subtotal went negative: bug upstream"
print("passed the tripwire:", subtotal)
```

```
passed the tripwire: 9.6
```

...

Think of it as a note to yourself, checked automatically: *"if this is ever false, something broke earlier. Stop before it spreads."*

Try, then fall back

A friendly, common pattern: **try** the risky thing; if it fails, **fall back** to a sensible default so the program keeps moving.

```
def seats(text):
    try:
        return int(text)
    except ValueError:
        return 2               # a table seats two, unless told otherwise
```

```
print(seats("6"))              # a clean number → 6
print(seats("full"))           # nonsense → the safe default, 2
```

```
6
2
```

...

The caller never has to worry about a crash. Garbage in, sane default out. Not every check needs a `try`: `isinstance(x, (int, float))` asks *is this a number?* and returns `True`/`False`, so an `if` can skip bad values before they explode.

Predict: what happens after the `except`?

`to_price` catches a bad value and returns `0.0`. After that, does the program reach the **next line**?

```
def to_price(text):
    try:
        return float(text)
    except ValueError:
        return 0.0

print(to_price("bad"))
print("checkout still running")
```

a) it never runs: the program already stopped b) the `try` block runs a second time c) it runs normally: the program carried on

...

Predict first. Pick a letter, then I reveal the answer.

Answer: it carries on

c) it runs normally. That's the whole point of `try` / `except`. It *handles* the failure; once the `except` has run, control simply drops to the code after it, as if nothing had gone wrong:

```
def to_price(text):
    try:
        return float(text)
    except ValueError:
        return 0.0

print(to_price("bad"))           # 0.0, recovered
print("checkout still running")  # ...and we get here
```

```
0.0
checkout still running
```

Your turn — 10 minutes

Open the exercise (scan the QR or type the link):

python.tobiasvlcek.com/notebooks/ex_05_b/

First **predict** what happens, then run it.

To the Lab

After the break: the lab

- Head to the lab notebook: Episode 5 — The 3-AM Checkout
- You'll read a traceback, write a price box that won't crash, fix Tobi's 3-AM receipt, make your code **refuse** negative prices with `raise`, and harden the whole checkout against a batch of poisoned orders
- It runs entirely in your browser: no setup, just click and code

. . .

> ! Important
>
> **Download your `.py` before you leave.** Closing the tab without downloading loses your work, and downloading is exactly how you just handed in the checkpoint. Expect **red cells** in this lab: you'll cause errors on purpose. A red cell pauses everything below it. Fix it, and everything springs back.

Wrap-up

Three things to remember

1. A **traceback** is a crash report you read **bottom-up**: the last line names *what* broke, the lines above show *where*. That's an address and a reason, not a wall of noise
2. `try` / `except` catches a failure so the program recovers instead of stopping. Catch the specific type (`except ValueError`), never a bare `except` that also hides your own typos
3. Defend your own code: `raise` to refuse bad input (prices ≥ 0), `assert` to guard an invariant that must always hold, and fall back to a safe default when a conversion fails

. . .

> **i** Note
>
> **Next time — Episode 6** starts with Checkpoint 3, which sweeps everything from Episodes 1–5, so keep this notebook and the last four close. Then someone new walks into the shop, uncaps a marker, and writes one question on the whiteboard. Part II begins.

Literature

Books to start with

- Downey, A. B. (2024). Think Python: How to think like a computer scientist (Third edition). O'Reilly. Link to free online version
- Elter, S. (2021). Schrödinger programmiert Python: Das etwas andere Fachbuch (1. Auflage). Rheinwerk Verlag.

. . .

> **i** Note
>
> Nothing new here, but these are still great books to start with!

. . .

For more, see the literature list of this course.