# Lecture VI - Modules and the Standard Library

## Programming with Python

Dr. Tobias Vlćek
Kühne Logistics University Hamburg - Fall 2026

## 📋 Checkpoint 3

**Sessions I–V.** The first 40 minutes are the checkpoint. It starts now, before the investor sits down.

- **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 **everything from Sessions I–V**: variables, control flow, functions, data structures, errors

### When you're done

Menu → *Download* → *Download Python code* → upload the `.py` to the **"Checkpoint 3"** assignment on Moodle. **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 6: Due Diligence Week

### The investor walks in

Pens down. The checkpoint is behind you. Now the reason today matters.

. . .

An hour ago the **investor** walked into the shop unannounced. It's **due-diligence week**: before she signs anything, she wants to see how this place actually runs. Tobi offered her a coffee and a spreadsheet he "mostly trusts."

. . .

She didn't drink the coffee. She walked to the whiteboard, uncapped a marker, and wrote one question:

**“Why is everything built from scratch?”**

### Part II: the rules change

For five sessions you built everything by hand, on purpose. From today, that changes.

- **AI is now allowed and taught.** We work *with* it, deliberately, starting in today’s lab.
- **The disclosure habit:** every submission that used AI carries a **one-line note** saying what you used it for. Not a confession. A professional reflex.
- **The course chatbot** (sidebar widget) now gives you **full code** on request, where it used to stop at hints.

. . .

You spent five sessions learning to think without a co-pilot. Now you get one, and you’ll be the pilot.

### Two accounts this week

Both are free, both are on the AI Tools page, together about ten minutes:

- **Mistral:** a Le Chat account for questions, plus an API key. The same key later powers the AI agent inside your editor
- **Zed student plan:** sign in with the GitHub account you made in Session I (it’s old enough now) and apply with your KLU e-mail. Verification takes up to 72 hours, so the free year of Zed Pro is ready long before Session X

. . .

> ! Important
>
> Turn **off** the training-data switches in Mistral’s privacy settings before you paste coursework: one for Le Chat, one for the API (details on the AI Tools page). Nothing here needs a credit card.

## Don’t build it, import it

### What’s a module?

The investor’s question has an answer: **you shouldn’t build it from scratch.** Most of what you need is already written.

- A **module** is a toolbox of code someone already wrote and tested
- Python ships with a whole shelf of them: the **standard library**
- You `import` a module, then reach for the tools inside it with a dot: `math.ceil(...)`

. . .

Tobi has been hand-rolling arithmetic for months. The standard library did most of it before he was born.

## `import math`: stop rounding by hand

130 pastries need to ship. They go in crates of 48. How many crates? You need to round **up**: a half-full crate still ships as a whole one.

```
import math

pastries = 130
per_crate = 48
crates = math.ceil(pastries / per_crate)   # ceil = round UP
print(crates)
```

```
3
```

...

`130 / 48` is `2.7…`; `math.ceil` bumps it to **3**. No fiddling with "if there's a remainder, add one." The tool already knows.

## `from statistics import ...`

Sometimes you only want a couple of tools, not the whole box. Import them **by name** and use them directly, no `statistics.` prefix:

```
from statistics import mean, median

ratings = [4.5, 4.8, 1.0, 5.0, 4.2]
print(mean(ratings))     # the average
print(median(ratings))   # the middle value
```

```
3.9
4.5
```

...

One furious review (a 1.0) drags the mean down to 3.9. The investor asked for the **typical** rating: `median` sorts the values and hands back the middle one (4.5), unmoved by one angry customer.

## Aliases: a shorter name

Some module names are long, or you'll type them fifty times. `import ... as` gives a module a **nickname** for the rest of the file:

```
import statistics as stats

print(stats.median([4.5, 4.8, 1.0, 5.0, 4.2]))
```

```
4.5
```

. . .

`stats.median` is the same tool as `statistics.median`, just less to type. You'll meet fixed conventions soon (`import pandas as pd`); using the community's nickname makes your code instantly readable to everyone else.

## Looking inside a module

You don't have to memorize a module. Python will tell you what's in it and what each tool does:

```
import math

dir(math)          # lists every name in the module
help(math.ceil)    # prints what ceil does, and how to call it
```

. . .

> Tip
>
> `dir()` is the drawer of tools; `help()` is the little instruction card taped to each one. Between them you can explore any module without leaving your editor.

## Predict: which way does floor go?

`math.ceil` rounds up. Its partner `math.floor` rounds **down**, but *down* from a negative number is the tricky part. What does the last line print?

```
import math
print(math.floor(-2.5))
```

a) `-3` b) `-2` c) an error

. . .

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

## Answer: floor goes down, not toward zero

**a) `-3`**: `floor` always heads **down** the number line, toward more negative. From `-2.5`, down is `-3`, not the `-2` you'd get by rounding toward zero:

```
import math
print(math.floor(-2.5))   # down the number line → -3
```

```
-3
```

. . .

"Down" means *smaller*, and `-3` is smaller than `-2`. Keep the number line in your head, not the distance to zero.

## Your turn — 10 minutes

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

python.tobiasvlcek.com/notebooks/ex_06_a/

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

# Rehearsing luck

## The `random` toolbox

The investor wants to see how the shop copes with a **busy day**, but the busy day hasn't happened yet. So we *rehearse* it with made-up numbers. The `random` module deals them:

```
import random

print(random.random())                       # a float in [0.0, 1.0)
print(random.randint(1, 20))                 # an integer 1–20, ends included
print(random.choice(["latte", "mocha", "tea"]))  # one item, picked at random

queue = [1, 2, 3, 4, 5]
random.shuffle(queue)                        # reorders the list in place
print(queue)
```

```
0.7250656681504807
5
tea
[4, 1, 5, 3, 2]
```

...

Four tools, four flavors of luck: a raw float, a bounded integer, a pick from a list, and a reshuffle.

## "Run it again"

The investor leans over and says: **"Run it again."** Tobi does, and gets **completely different numbers**:

```
import random

print([random.randint(1, 20) for _ in range(5)])    # _ : a loop name we
never use
print([random.randint(1, 20) for _ in range(5)])   # ...and again, different!
```

```
[20, 12, 1, 2, 1]
[3, 17, 15, 5, 1]
```

...

Two runs, two answers. That's *exactly* what random is supposed to do, but it's useless for due diligence. A projection nobody can reproduce is a projection nobody can trust.

## `random.seed` makes luck repeatable

`random.seed(n)` fixes the starting point of the number stream. Same seed → **same sequence, every time**:

```
import random

random.seed(7)
print([random.randint(1, 20) for _ in range(5)])   # → [11, 5, 13, 2, 3]

random.seed(7)
print([random.randint(1, 20) for _ in range(5)])   # same seed → same list
```

```
[11, 5, 13, 2, 3]
[11, 5, 13, 2, 3]
```

...

Both lines print `[11, 5, 13, 2, 3]`. The numbers still *look* random, but now the investor can run it herself and land on the identical result.

## Predict: seeded once, built twice

Tobi seeds **once**, then builds two lists the same way, without touching the seed in between. Are `first` and `second` equal?

```
import random
```

```
random.seed(42)
first  = [random.randint(1, 20) for _ in range(3)]
second = [random.randint(1, 20) for _ in range(3)]

print(first)
print(second)
```

a) different: the second list continues where the first stopped b) equal: the seed is set, so both come out the same c) an error: the stream is empty after three draws

. . .

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

## Answer: different

**a) different**. A seed doesn't freeze `random`, it fixes the whole sequence. The first list eats the first three numbers of the stream; the second list simply continues from number four:

```
import random

random.seed(42)
first  = [random.randint(1, 20) for _ in range(3)]
second = [random.randint(1, 20) for _ in range(3)]

print(first)    # [4, 1, 9]: the stream's first three numbers
print(second)   # [8, 8, 5]: the stream carries on
```

```
[4, 1, 9]
[8, 8, 5]
```

. . .

To get the *same* list twice, you re-seed before each run, and that rewinds the stream to the start. One seed, one fixed sequence: that's the entire job of a seed.

# Your turn — 10 minutes

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

python.tobiasvlcek.com/notebooks/ex_06_b/

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

# To the Lab

## After the break: the lab

- Head to the lab notebook: Episode 6 — Due Diligence Week
- You'll `import math` and `statistics` for investor-grade counts and averages, then use `random` with a `seed` to rehearse a busy day she can reproduce
- It's the **first lab where AI is allowed**, so try the chatbot
- This week's homework: the **Mistral account** and the **Zed student plan** from the AI Tools page
- 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 handed in the checkpoint at the start of the session.

# Wrap-up

## Three things to remember

1. **Don't build it, import it.** A **module** is a toolbox someone already wrote: `import math`, `from statistics import median`, `import ... as` for a nickname. `dir()` and `help()` show you what's inside.
2. **`random` deals the luck** (`random()`, `randint`, `choice`, `shuffle`), perfect for rehearsing a day that hasn't happened yet.

3. `random.seed(n)` **makes luck repeatable.** Same seed → same sequence, every run. A projection you can reproduce is a projection an investor can trust.

. . .

> **i** Note
>
> **Next episode: the numbers deck.** The shop's data has outgrown plain lists, and NumPy turns a thousand numbers into a single, fast object.

# 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
>
> New this session: the AI Tools page: how and when to use AI in Part II, and the one-line disclosure habit that goes on every submission from here on.

. . .

For more, see the literature list of this course.