Lecture VII - NumPy for Scientific Computing

Programming with Python

Dr. Tobias Vlćek
Kühne Logistics University Hamburg - Fall 2026

Episode 7: The Numbers Deck

The numbers deck is due

The investor wants a numbers deck. **Two weeks of orders** sit in the system, one row each: minutes, price, zone.

. . .

Tobi's plan is a **40-tab spreadsheet**, one tab per zone, copied by hand. He looks at the pile, then at his list-of-lists, and says the only true thing he'll say all week:

> **"We need a bigger boat than a list."**

. . .

Today we get the bigger boat: **NumPy**, one object that holds a thousand numbers and does arithmetic on all of them at once.

🔥 Warm-up

Three questions from Episode 6. Commit. Hands up **before** the reveal.

Question 1

```
from math import ceil

# which line works?
```

a) `ceil(3.2)` b) `math.ceil(3.2)` c) both

Answer 1

a) **`ceil(3.2)`**: `from math import ceil` binds only the **name** `ceil`. The module `math` itself was never imported, so `math.ceil` has nothing to reach through.

Question 2

```
import random

```

```
random.seed(42)
print(random.randint(1, 20))
print(random.randint(1, 20))
print(random.randint(1, 20))
```

Tobi runs this exact script today and again tomorrow. Tomorrow's numbers are…

a) different: random is random b) an error: seed 42 was already used c) the same three numbers

Answer 2

c) the same three numbers. Every run starts from seed 42, so the stream replays from the top. That is the entire job of a seed: reproducible randomness. (Two batches *inside one run* would differ: the stream continues; a fresh run rewinds it.)

Question 3

```
import statistics

statistics.median([9, 2, 5])
```

returns…

a) `2` b) `5` c) an error: the list is not sorted

Answer 3

b) `5`: `median` sorts the values internally before picking the middle one. You never have to sort first; `5` is the middle of `2, 5, 9`.

A thousand orders at once

From a list to an array

A **NumPy array** holds many numbers under one name, like a list, but built for math. You make one from a list, then ask it about itself:

```
import numpy as np

prices = np.array([12.0, 9.0, 15.0])
print(prices)
print(prices.shape)   # how many, in each dimension
print(prices.dtype)   # what kind of number
print(prices.size)    # how many in total
```

```
[12.  9. 15.]
(3,)
float64
3
```

...

`np.array([...])` wraps a list; `import numpy as np` is the nickname everyone uses. `.shape` is `(3,)`, `.dtype` is `float64`, `.size` is `3`. One `dtype` for the **whole** array: every element shares the same type; that's part of what makes it fast.

One operation, every element

Here's the bigger boat. The deck needs **gross** prices: 19% VAT on all three. With a list you loop; with an array you just multiply:

```
# the painful way: a loop, item by item
gross = []
for p in [12.0, 9.0, 15.0]:
    gross.append(p * 1.19)
```

```
import numpy as np

prices = np.array([12.0, 9.0, 15.0])
print(prices * 1.19)   # every element, one expression
```

```
[14.28 10.71 17.85]
```

...

One operation lands on **all** elements at once: `[14.28 10.71 17.85]`. No loop, no `.append`, and on a thousand orders it's also far faster.

Arrays from scratch

Two builders make evenly-spaced arrays without typing every number, handy for axes and ranges:

```
import numpy as np

print(np.arange(0, 10, 2))     # start, stop (excluded), step
print(np.linspace(0, 1, 5))    # start, stop (included), how many
```

```
[0 2 4 6 8]
[0.   0.25 0.5  0.75 1.  ]
```

...

`arange` walks by a **step** and stops before the end, just like `range`. `linspace` splits a span into a fixed **count** of points, endpoints included.

Predict: times two

Tobi multiplies a row of counts by two. What does this print?

```
print([1, 2, 3] * 2)
```

a) `[1, 2, 3, 1, 2, 3]` b) `[2, 4, 6]` c) an error

...

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

Answer: lists repeat, arrays compute

a) **`[1, 2, 3, 1, 2, 3]`**. That's a plain **list**, and `* 2` on a list *repeats* it. Wrap it in an array and the same `* 2` does the math instead:

```
import numpy as np

print([1, 2, 3] * 2)              # list → repeated
print(np.array([1, 2, 3]) * 2)    # array → doubled
```

```
[1, 2, 3, 1, 2, 3]
[2 4 6]
```

...

Lists repeat; arrays compute. That's why we're here.

Your turn — 10 minutes

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

python.tobiasvlcek.com/notebooks/ex_07_a/

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

Asking questions of data

A comparison makes a mask

Compare an array to a number and you don't get one `True`/`False`. You get a **whole array** of them, one per element. That's a **mask**, and you can filter with it:

```
import numpy as np

times = np.array([25, 41, 18, 33])
print(times > 30)            # a True/False for every element
print(times[times > 30])     # keep only where the mask is True
```

```
[False  True False  True]
[41 33]
```

...

`times > 30` is the mask; `times[times > 30]` reads the array **through** the mask and returns just the matching values. No loop, no `if`.

Counting without a loop

A mask answers two investor questions at once. `.sum()` counts the `True`s (each counts as `1`); `.mean()` gives the **share** that are `True`:

```
import numpy as np

times = np.array([25, 41, 18, 33, 52, 29, 44, 12])
late = times > 30
print(int(late.sum()))              # how many were late
print(float(times[late].mean()))    # average of just the late ones
print(float(late.mean()))           # the SHARE that were late
```

```
4
42.5
0.5
```

...

`4` deliveries over 30 minutes, averaging `42.5`, and `late.mean()` says **half** the run was late, one line each, straight into the deck.

Predict: counting the Trues

What does calling `.sum()` on the mask give?

```
print((np.array([1, 5, 3]) > 2).sum())
```

a) `True` b) `2` c) `[False, True, True]`

...

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

Answer: True counts as 1

b) `2`: `.sum()` adds the mask up, and each `True` is worth `1`, each `False` `0`. Two elements clear the bar, so the count is `2`:

```
import numpy as np

print(np.array([1, 5, 3]) > 2)            # [False  True  True]
print((np.array([1, 5, 3]) > 2).sum())    # Trues add up to 2
```

```
[False  True  True]
2
```

...

Summing a mask counts; averaging a mask shares. Same two tricks the lab asks for.

Your turn — 10 minutes

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

python.tobiasvlcek.com/notebooks/ex_07_b/

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

The days-by-zones grid

A grid of numbers

Real data isn't one row. Stack rows and you get a **2D array**: here three days (rows) across four zones (columns: Nord, Sued, Hafen, Altstadt):

```
import numpy as np

deliveries = np.array([[ 9, 14, 11,  6],
                       [15, 12,  8,  9],
                       [13, 20, 16, 11]])
print(deliveries.shape)      # (rows, columns) → (3, 4)
print(deliveries[0, 2])      # row 0, column 2
```

```
(3, 4)
11
```

...

`.shape` is now `(3, 4)`: three days, four zones. One index picks the **row**, a second the **column**: `deliveries[0, 2]` is day 0, zone 2 (Hafen).

Which way to collapse?

To sum a grid you must say **which way** to collapse it. The `axis` tells NumPy which direction disappears:

```
          Nord  Sued  Hafen  Altstadt
   day0  [  9    14    11      6  ]
   day1  [ 15    12     8      9  ]
   day2  [ 13    20    16     11  ]

   axis=0 collapses DOWN the rows:
            ↓     ↓     ↓      ↓
           37    46    35     26      one number per column (zone)

   axis=1 collapses ACROSS the columns:
   day0 → 40   ·   day1 → 44   ·   day2 → 60      one number per row (day)
```

`axis` in code

The same grid, the same two collapses, one keyword each:

```
import numpy as np

deliveries = np.array([[ 9, 14, 11,  6],
                       [15, 12,  8,  9],
                       [13, 20, 16, 11]])
print(deliveries.sum(axis=0))   # DOWN the rows → per zone
print(deliveries.sum(axis=1))   # ACROSS the columns → per day
```

```
[37 46 35 26]
[40 44 60]
```

...

`axis=0` collapses DOWN the rows, one number per column (zone): `[37 46 35 26]`. `axis=1` collapses across, one per day: `[40 44 60]`.

Where does the max sit?

The per-zone totals answer "how many?", but the investor asks "**which** zone?". `argmax` tells you **WHERE** the maximum sits, as an index:

```
import numpy as np

deliveries = np.array([[ 9, 14, 11,  6],
                       [15, 12,  8,  9],
                       [13, 20, 16, 11]])
zone_totals = deliveries.sum(axis=0)   # [37 46 35 26]
print(int(zone_totals.argmax()))        # index of the biggest
```

```
1
```

...

`max` would give the value `46`; `argmax` gives its **position**, `1`. The busiest zone is **Sued**, sitting at index 1, not at the front. Position, not value: that's the whole point of `argmax`.

Predict: which call collapses which way?

The investor wants **four** numbers, one per zone. Which call?

```
week = np.array([[ 9, 14, 11,  6],
                 [15, 12,  8,  9],
                 [13, 20, 16, 11]])
```

a) `week.sum(axis=1)` b) `week.sum()` c) `week.sum(axis=0)`

...

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

Answer: collapse the days, keep the zones

c) **`week.sum(axis=0)`**. Four zones means four numbers, so the **days** must disappear: `axis=0` collapses DOWN the rows, one number per column (zone):

```
import numpy as np

week = np.array([[ 9, 14, 11,  6],
                 [15, 12,  8,  9],
                 [13, 20, 16, 11]])
print(week.sum(axis=0))
```

```
[37 46 35 26]
```

. . .

a) would give **three** numbers (one per day); b) would give **one** number: the grand total, `144`.

Your turn — 10 minutes

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

python.tobiasvlcek.com/notebooks/ex_07_c/

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

To the Lab

After the break: the lab

- Head to the lab notebook: Episode 7 — The Numbers Deck
- You'll turn the order log into arrays, add VAT to a whole price column at once, mask out the late deliveries to count and average them, and collapse a days-by-zones grid to find the winning zone
- AI is allowed, so try the chatbot, and keep your one-line disclosure note on the submission
- 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.

Wrap-up

Three things to remember

1. **One array, one operation.** `np.array([...])` holds many numbers; arithmetic hits **every element at once**. No loop. Ask it `.shape`, `.dtype`, `.size` to know what you're holding.
2. **A comparison makes a mask.** `arr > 30` is a True/False array: `arr[mask]` filters, `.sum()` counts the Trues, `.mean()` gives their share.
3. **In 2D, pick an axis.** `axis=0` collapses DOWN the rows (one number per column), `axis=1` across the columns (one per row); `argmax` tells you where the maximum sits.

. . .

> **i** Note
>
> **Next episode** starts with **Checkpoint 4**: 40 minutes, AI allowed, everything from Episodes 6–7. Then the investor opens a data room, and Tobi lets an AI write his pandas: the arrays get column names, and eighty orders become a table you can query.

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
>
> NumPy has excellent free docs: the NumPy absolute beginner's guide covers everything in this session and a little more.

. . .

For more, see the literature list of this course.