Cheatsheet

Programming with Python

Dr. Tobias Vlćek

One section per session, only what the course teaches. Every example runs as shown; the comment shows what it prints. Tooling (uv, git, Zed and the AI setup) is not repeated here: see the uv guide, Git Basics and the AI Tools page.

Session I: Variables, Types & f-strings

Variables and the four basic types

A variable is a name that points to a value; `=` creates it. Names start with a letter or `_`, are case-sensitive, and cannot be reserved words (`for`, `if`, `def`, ...). Comments start with `#`.

```
company_founded = 2026          # int   - whole numbers
first_employee = "Tobi"        # str   - text in quotes
price = 9.99                    # float - decimal numbers
is_open = True                  # bool  - True or False
print(type(price))              # <class 'float'>
print(type("9.99"))             # <class 'str'>  - quotes make it text
```

Arithmetic

`*` and `/` happen before `+` and `-`; parentheses override that. `/` always gives a float.

```
print(9.99 - 4.20)          # 5.79
print(300 / 8)              # 37.5  - division always gives a float
print(300 // 7)            # 42    - floor division: how many whole boxes fit
print(300 % 7)              # 6     - modulo: what is left over
print(2 + 3 * 4)            # 14    - not 20
print(round(10 / 3, 2))     # 3.33  - two decimals for money
```

Strings, conversions and f-strings

Put an `f` before the quote and write values in `{}`. `:.2f` forces two decimals, `:>8.2f` right-aligns in a width of 8, `\n` starts a new line. `str()`, `int()` and `float()` convert between types.

```
qty = 2
item = "Miso Ramen"
total = 23.0
print(f"{qty}x {item}: {total:.2f} EUR")   # 2x Miso Ramen: 23.00 EUR
```

```
print(f"Pad Thai {26.70:>8.2f}")            # Pad Thai    26.70
print("Wrap: 6.90 EUR\nPad Thai: 8.90 EUR")  # two lines
print(f"{2 * 3}")                             # 6  - braces evaluate, plain
quotes stay literal
print(str(qty) + "x " + item)                # 2x Miso Ramen  - + needs text
on both sides
print(int("123") + 1, float("5.50") * 2)    # 124 11.0
```

Session II: Control Structures

Comparisons and booleans

Six comparison operators return `True` or `False`: `<`, `>`, `<=`, `>=`, `==`, `!=`. `=` assigns, `==` compares. `and` needs both sides, `or` at least one, `not` flips.

```
delivery_hour = 21
print(delivery_hour < 22)                # True
is_weekday = True
print(is_weekday and delivery_hour < 22)  # True
```

`if` / `elif` / `else`

The first `True` branch wins, so order the ladder from strictest to loosest. The indented block (4 spaces) is what runs.

```
past_orders = 8
if past_orders >= 20:
    tier = "Gold"
elif past_orders >= 5:
    tier = "Silver"
else:
    tier = "Bronze"
print(tier)                          # Silver
```

`for` loops, the accumulator and `range`

```
prices = [5.00, 3.50, 6.50]
total = 0
for price in prices:                 # one pass per item
    total = total + price
print(total)                         # 15.0  - unindented: runs once, after
the loop
print(list(range(3)))                # [0, 1, 2]     - stops BEFORE 3
print(list(range(0, 10, 3)))         # [0, 3, 6, 9]  - start, stop, step
```

`while` loops

A `while` loop repeats as long as its condition holds. Something inside must move it toward `False`, or it never stops (and freezes the tab). `break` leaves a loop early.

```
price = 10.00
rounds = 0
while price >= 7.00:
    price = round(price * 0.8, 2)     # 20% off each round
    rounds = rounds + 1
print(rounds, price)                  # 2 6.4
```

String methods

Methods return a cleaned-up copy; the original never changes. Chain them left to right.

```
raw = "  miso ramen  "
print(raw.strip())                    # miso ramen  - outer spaces gone
print(raw.strip().title())            # Miso Ramen
print("special".upper())              # SPECIAL
print("MOIN".lower())                 # moin
print("SALE!!!".rstrip("!"))          # SALE  - trailing ! removed
```

Session III: Functions, Scope & a First Class

`def`, parameters, `return`

Parameters are the names in the definition; arguments are the values you pass. `return` hands a value back; `print` only shows it. No `return` means the function returns `None`.

```
def line_total(qty, price):
    return round(qty * price, 2)

subtotal = line_total(3, 3.20)        # catch what came back
print(subtotal)                       # 9.6

def service_fee(total, rate=0.05):    # a default, used when left out
    return round(total * rate, 2)

print(service_fee(80))                # 4.0

def label_price(price):
    print(f"{price:.2f} EUR")         # prints, but has no return

result = label_price(8.50)            # 8.50 EUR
print(result)                         # None
```

Scope

A parameter is the function's own private copy; a function cannot quietly change your variables. To keep a result, assign the return value back.

```
def bump(n):
    n = n + 10
    return n

stock = 3
bump(stock)                          # return value thrown away
print(stock)                         # 3  - untouched
print(bump(stock))                   # 13 - catch it: stock = bump(stock)
```

A class bundles data with behavior

`__init__` runs when you build an object and stores data on `self`; a method is a function inside the class that reads that data. One small class with one method is all this course uses.

```
class Delivery:
    def __init__(self, courier, distance_km):
        self.courier = courier
        self.distance_km = distance_km
        self.rate_per_km = 1.20

    def fee(self):
        return round(self.distance_km * self.rate_per_km, 2)

trip = Delivery("Nadia", 4)          # build one: __init__ runs
print(trip.courier)                  # Nadia
print(trip.fee())                    # 4.8
```

Session IV: Lists, Tuples, Dictionaries, Sets & Comprehensions

Lists and tuples

Lists are ordered and mutable. Indexing starts at `0`, `-1` is the last item, a slice's `stop` is excluded. A tuple uses `()` and cannot be changed: a fixed-length record.

```
drinks = ["Mate", "Spezi", "Ayran"]
print(drinks[0], drinks[-1])         # Mate Ayran
print(drinks[0:2])                  # ['Mate', 'Spezi']  - stop is excluded
print(drinks[-2:])                     # ['Spezi', 'Ayran'] - open end: to
the finish
drinks.append("Kombucha")            # in place
print(len(drinks))                   # 4
opening = (9, 0)                     # tuple: (hour, minute)
print(opening[0])                    # 9
```

Dictionaries

Map a key to a value. Reading a missing key with `[]` raises `KeyError`; `.get()` returns `None` or your fallback instead. Assigning to a key updates it or adds it. Dictionaries keep insertion order.

```
prices = {"Mate": 3.50, "Spezi": 3.20}
print(prices["Spezi"])                # 3.2
print(prices.get("Cola"))             # None
print(prices.get("Cola", 0))          # 0
winter = dict(prices)                 # a copy keeps the original safe
winter["Spezi"] = 3.40                # update
winter["Kombucha"] = 4.20             # add
print(winter)                                 # {'Mate': 3.5, 'Spezi': 3.4,
'Kombucha': 4.2}
zones = {"Hafen": {"fee": 2.50, "eta": 20}}   # nested: dict inside dict
print(zones["Hafen"]["fee"])          # 2.5  - outer key, then inner key
```

Sets

Each value once, no order: `set(list)` drops duplicates, `len()` counts the distinct ones.

```
print(len(set(["nina", "tom", "nina", "ada", "tom"])))   # 3
```

Comprehensions

A loop that builds a list (or dict) in one line: `[expr for x in things]`, `{k: v for k, v in pairs}`. `.items()` hands you each key, value pair.

```
counts = [2, 1, 3]
print([c * 2 for c in counts])        # [4, 2, 6]
print([round(p * 0.9, 2) for p in [3.50, 3.20, 2.80]])   # [3.15, 2.88, 2.52]
prices = {"Mate": 3.50, "Spezi": 3.20}
print({k: round(v * 0.9, 2) for k, v in prices.items()})   # {'Mate': 3.15,
'Spezi': 2.88}
```

Part I has no files, so data ships inside the code: `text.splitlines()` gives one string per line, `line.split(";")` splits a line at the separator.

Session V: Errors & Debugging

Reading a traceback, and the big five

Read a traceback bottom-up: the last line names *what* went wrong, the lines above show *where* (file and line number). Five exception types cover almost everything:

- `ValueError`: right type, senseless value: `int("lots")`
- `TypeError`: wrong type entirely: `"Bowl " + 9`
- `KeyError`: a dictionary key that is not there: `prices["Cola"]`
- `IndexError`: a list position past the end: `seats[9]` on a 3-item list

- `ZeroDivisionError`: `88.00 / 0`

`try` / `except`, then fall back

Catch the specific type; a bare `except:` also hides your own typos. `as e` keeps the error message. After the `except`, the program carries on.

```
try:
    label = "Party of " + 3
except TypeError as e:
    print("TypeError:", e)            # TypeError: can only concatenate str
(not "int") to str

def seats(text):
    try:
        return int(text)
    except ValueError:
        return 2                      # a sane default

print(seats("6"))                     # 6
print(seats("full"))                  # 2
```

`raise`, `assert`, `isinstance`

`raise` makes your own code refuse bad input; `assert` guards a fact that must always hold; `isinstance` asks "is this a number?" before it explodes.

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

try:
    charge(-4.50)
except ValueError as e:
    print("refused:", e)              # refused: price cannot be negative

subtotal = 9.60
assert subtotal >= 0, "subtotal went negative"   # silent when True
print(isinstance(9.60, (int, float)))            # True
print(isinstance("9.60", (int, float)))          # False
```

Debugging loop: read the traceback, reproduce, isolate, fix one thing. A well-placed `print("DEBUG", value)` is the flashlight; remove it afterwards.

Session VI: Modules & the Standard Library

Three ways to import

`dir(math)` lists what a module contains; `help(math.ceil)` explains one tool.

```
import math                          # use with a prefix
from statistics import mean, median  # only these names, no prefix
import statistics as stats           # a nickname

print(math.ceil(130 / 48))           # 3   - round UP
print(math.floor(-2.5))              # -3  - round DOWN the number line
print(mean([4.5, 4.8, 1.0, 5.0, 4.2]))     # 3.9
print(median([4.5, 4.8, 1.0, 5.0, 4.2]))   # 4.5  - sorts for you
print(stats.median([9, 2, 5]))       # 5
```

`random`, and `seed` for repeatable luck

One seed fixes the whole stream: a second batch continues where the first stopped. Re-seed to rewind.

```
import random

random.seed(7)                       # same seed, same sequence, every run
print([random.randint(1, 20) for _ in range(5)])   # [11, 5, 13, 2, 3]
print(random.random())               # a float in [0.0, 1.0)
print(random.choice(["latte", "mocha", "tea"]))    # one item
random.shuffle([1, 2, 3, 4, 5])      # reorders a list in place
```

Session VII: NumPy

Arrays and vectorised arithmetic

One `dtype` for the whole array; one operation hits every element at once.

```
import numpy as np

prices = np.array([12.0, 9.0, 15.0])
print(prices.shape, prices.dtype, prices.size)   # (3,) float64 3
print(prices * 1.19)                 # [14.28 10.71 17.85]  - a list would
repeat instead
print(np.arange(0, 10, 2))              # [0 2 4 6 8]       - start, stop
(excluded), step
print(np.linspace(0, 1, 5))          # [0.   0.25 0.5  0.75 1.  ] - start,
stop, count
```

Masks

A comparison gives a True/False array. `arr[mask]` filters, `mask.sum()` counts the Trues, `mask.mean()` gives their share.

```
times = np.array([25, 41, 18, 33, 52, 29, 44, 12])
late = times > 30
print(late)                          # [False  True False  True  True False
```

```
True False]
print(times[late])                    # [41 33 52 44]
print(int(late.sum()))                # 4     - how many late
print(float(times[late].mean()))      # 42.5  - average of the late ones
print(float(late.mean()))             # 0.5   - the share that were late
```

2-D arrays, `axis`, `argmax`

`axis=0` collapses down the rows (one number per column); `axis=1` collapses across the columns (one per row). `argmax` gives the position of the maximum, `max` its value.

```
deliveries = np.array([[ 9, 14, 11,  6],      # rows: days
                       [15, 12,  8,  9],      # columns: zones
                       [13, 20, 16, 11]])
print(deliveries.shape, deliveries[0, 2])   # (3, 4) 11  - row 0, column 2
print(deliveries.sum(axis=0))         # [37 46 35 26]  - per zone
print(deliveries.sum(axis=1))         # [40 44 60]     - per day
zone_totals = deliveries.sum(axis=0)
print(int(zone_totals.max()))         # 46  - the value
print(int(zone_totals.argmax()))      # 1   - its position
```

Session VIII: pandas & Working with AI

The DataFrame

A table with named columns of mixed types. `pd.DataFrame` from a dict, `pd.read_csv("file.csv")` from a file (one line, also in the browser). Look before you leap: `.head()`, `.shape`, `.info()`, `.describe()`.

```
import pandas as pd

df = pd.DataFrame({
    "zone":      ["Nord", "Sued", "Nord", "Hafen", "Sued"],
    "items":     [2, 1, 3, 1, 4],
    "total_eur": [18.50, 7.20, 24.00, 6.80, 31.40],
})
print(df.head(2))                     # first rows
print(df.shape, len(df))              # (5, 3) 5  - (rows, columns), rows
print(df.describe())                 # count, mean, std, min, quartiles, max
```

Select, filter, count, sum

`df["col"]` is one column (a Series). A mask keeps the `True` rows; text comparisons are case-sensitive, and a wrong spelling silently returns zero rows. Wrap each mask in parentheses when joining with `&`.

```
nord = df[df["zone"] == "Nord"]
print(len(nord), nord["total_eur"].sum())   # 2 42.5
```

```
print(len(df[(df["zone"] == "Sued") & (df["items"] >= 2)]))   # 1
print(int((df["total_eur"] > 10).sum()))                      # 3  - True
counts as 1
```

New column on a copy, sort, `groupby`

`groupby` splits the rows by a category and computes once per group. `.idxmax()` returns the label of the largest value, `.to_dict()` turns the result into a plain dictionary.

```
priced = df.copy()                    # never mutate the original
priced["eur_per_item"] = priced["total_eur"] / priced["items"]
print(priced["eur_per_item"].max())   # 9.25
print(priced.sort_values("total_eur", ascending=False).head(2))

by_zone = df.groupby("zone")["total_eur"].sum()
print(by_zone.round(2).to_dict())     # {'Hafen': 6.8, 'Nord': 42.5, 'Sued':
38.6}
print(by_zone.idxmax())               # Nord  - the label, not the value
print(df.groupby("zone")["total_eur"].mean().round(2).to_dict())          #
{'Hafen': 6.8, 'Nord': 21.25, 'Sued': 19.3}
```

Working with AI (Part II)

- **Prompt with context and constraints**: name the DataFrame, its columns and their types, and say exactly what you want back ("one line that returns the total `total_eur` for zone `Nord`").
- **Verify: read it, run it, test it** on a tiny case whose answer you already know. An AI that sounds sure is not an API that exists (`.summarize()` does not; `.describe()` does).
- **Disclose in one line** on every submission that used AI, e.g. "Used the chatbot to draft the `groupby` line; I checked the totals by hand."

Session IX: Plotting with matplotlib

The frame around every chart

Open with `plt.figure()` (a fresh canvas, so lines do not pile onto the last chart) and end the cell with `plt.gca()` (marimo shows the last expression). A script run from the terminal uses `plt.show()` instead.

```
import matplotlib.pyplot as plt

days = [1, 2, 3, 4, 5]
revenue = [120, 90, 140, 160, 130]

plt.figure()
plt.plot(days, revenue, label="Revenue", color="crimson", linestyle="--",
marker="o")
plt.xlabel("Day")
```

```
plt.ylabel("Revenue (EUR)")
plt.title("This week's revenue")
plt.legend()                          # one entry per label
plt.gca()
```

Which chart for which question

The question	The chart	The call
Compare categories?	bar	`plt.bar(labels, heights)`
See a distribution?	histogram	`plt.hist(values, bins=5)`
Two numbers per order?	scatter	`plt.scatter(x, y)`
Change over time?	line	`plt.plot(x, y)`

No pie charts: the eye cannot compare slice sizes. A bar counts categories; a histogram counts ranges. Same `plt.figure()` … `plt.gca()` frame around each.

The honest axis

A growth claim starts the y-axis at 0: `plt.ylim(0, 100)`. A tight `plt.ylim(96, 99)` turns a wobble into a rocket. Before you believe an AI-drafted chart, check three things: do the columns it used exist, does the y-axis start where you claim, does the chart type fit the question?

Best practices

1. Short, meaningful names in `snake_case`; classes in `PascalCase`; `round(x, 2)` and `:.2f` for money
2. Order `if`/`elif` ladders from strictest to loosest; make every `while` move toward its end
3. Write a calculation once as a function and catch its return value; work on a copy (`dict(d)`, `df.copy()`) when the original must survive
4. Catch specific exceptions, `raise` on bad input, read tracebacks bottom-up
5. Don't build it, import it; `random.seed(n)` before anything that must be reproducible
6. Prefer one vectorised expression or one `groupby` over a loop; every chart gets `plt.figure()`, labels, a title, `plt.gca()`, and a y-axis from 0 for growth
7. Test AI output on a tiny case you can check by hand and add the one-line disclosure

Recap

You can now look up:

1. Types, arithmetic and f-string formatting (I)
2. Conditionals, loops and string methods (II)
3. Functions, scope and a small class (III)
4. Lists, tuples, dictionaries, sets and comprehensions (IV)
5. Tracebacks, `try`/`except`, `raise` and `assert` (V)
6. `import`, `math`, `statistics` and seeded `random` (VI)

7. NumPy arrays, masks and `axis` (VII)
8. pandas selection, filtering, `groupby` and the AI verify workflow (VIII)
9. The four matplotlib charts and the honest axis (IX)