Python for developers with uv, pytest, and Flask

You learn modern Python with uv, type hints, and pytest, and build a tested HTTP API with Flask for the workshop shop.

Mike Bild

Exercises: github.com/MikeBild/introduction-python

Values, functions, collections

Names point to values, values have types · Clear function signatures instead of flag parameters · Which collection fits your data?

A name does not have a type, the value does

- A name points to a value and can point somewhere new
- type() shows the value's type at runtime
- Python never silently converts between types

```
price = 1299
type(price)              # <class 'int'>
price = "12,99 €"        # same name, a new value
"Preis: " + 1299         # TypeError: can only concatenate str
f"Preis: {1299}"         # 'Preis: 1299'
```

Calculate money in whole cents

```
0.1 + 0.2 == 0.3          # False
0.1 + 0.2                 # 0.30000000000000004
10 + 20 == 30             # True: cents as int
2 ** 100                  # int grows without overflow
divmod(1234, 100)         # (12, 34)
```

Equal or the same?

- == compares the contents of two values
- is checks whether two names point to the same object
- None is a single object, so use is None

```
a = [1299, 2499]
b = [1299, 2499]
a == b              # True: gleicher Inhalt
a is b              # False: zwei Listen
discount = None
discount is None    # True
```

What counts as false?

- 0, empty collections, "" and None are falsy
- if not price matches 0 and None alike
- Check for None explicitly when 0 is valid

```
def label(price_cents: int | None) -> str:
    if price_cents is None:
        return "Preis auf Anfrage"
    if price_cents == 0:
        return "gratis"
    return f"{price_cents / 100:.2f} €"
```

A function with a contract

- def sets the name, parameters, and return value
- Defaults make parameters optional
- The star makes everything after it keyword-only

```
def line_total(quantity: int, price_cents: int) -> int:
    return quantity * price_cents


def order_total(items, *, discount_percent=0):
    subtotal = 0
    for _product_id, quantity, price_cents in items:
        subtotal += line_total(quantity, price_cents)
    return subtotal - subtotal * discount_percent // 100

order_total(items, discount_percent=10)
```

The shared default

```
def add_item(item, items=[]):      # Default is created once
    items.append(item)
    return items

add_item("p-hammer")                # ['p-hammer']
add_item("p-saege")                 # ['p-hammer', 'p-saege']


def add_item(item, items=None):    # this is safe
    result = [] if items is None else list(items)
    result.append(item)
    return result
```

Functions are values

- You pass functions like any other value
- An inner function remembers its environment
- lambda fits short expressions

```
def make_discount(percent):
    def discount(total_cents):
        return total_cents - total_cents * percent // 100
    return discount

ten_percent_off = make_discount(10)
ten_percent_off(2000)                          # 1800
sorted(items, key=lambda item: item[2])        # by price
```

Four collections, four access patterns

- list orders, tuple groups fixed values
- dict finds by key, set checks membership
- You can loop over all four with for

```
products = [
    {"id": "p-hammer", "name": "Hammer", "priceCents": 1299, "stock": 12},
    {"id": "p-saege", "name": "Säge", "priceCents": 2499, "stock": 1},
]
item = ("p-hammer", 2)                    # tuple: product and quantity
index = {"p-hammer": products[0], "p-saege": products[1]}
vip = {"c-ada", "c-grace"}                # set: unique
index["p-saege"]["name"]                  # 'Saw'
"c-ada" in vip                            # True
```

Loop and unpack

```
for product in products:
    if product["stock"] < 3:
        print(product["name"])

for product_id, product in index.items():   # Key and value
    print(product_id, product["priceCents"])

for position, product in enumerate(products, start=1):
    print(position, product["name"])

price = index.get("p-bohrer", {}).get("priceCents")  # None instead of KeyError
```

Count, deduplicate, freeze

- Counter counts missing keys as zero
- set removes duplicates by itself
- frozendict protects data from changes

```
from collections import Counter

customer_ids = set()
counts = Counter()
for order in orders:
    customer_ids.add(order["customerId"])
    for item in order["items"]:
        counts[item["productId"]] += item["quantity"]
counts.most_common(1)             # [('p-saege', 4)]

prices = frozendict({"p-hammer": 1299})
prices["p-hammer"] = 1            # TypeError
```

Comprehensions, project with uv, dataclasses

Comprehensions instead of loops with append · A project with uv instead of manual pip and venv · Dataclasses instead of boilerplate-heavy classes

What is created, not how

- First the expression, then for, then an optional if
- Square brackets build a list, braces build a dict or set
- The result is a new value, and the source stays unchanged

```
open_ids = []
for order in orders:
    if order["status"] == "placed":
        open_ids.append(order["id"])

open_ids = [order["id"] for order in orders if order["status"] == "placed"]
totals = {order["id"]: order_total(order) for order in orders}
customers = {order["customerId"] for order in orders}
```

Sums without an intermediate list

```
def order_total(order):
    return sum(item["quantity"] * item["priceCents"] for item in order["items"])

any(item["quantity"] > 10 for item in order["items"])    # stops early
max(orders, key=order_total)                              # largest order
sorted(totals.items(), key=lambda entry: entry[1], reverse=True)[:3]
```

Flatten and know the limits

- Since Python 3.15, * unpacks directly in the comprehension
- Two for clauses read from left to right
- With two or more conditions, a loop is clearer

```
all_items = [*order["items"] for order in orders]          # Python 3.15

products_sold = {
    item["productId"]
    for order in orders
    if order["status"] != "cancelled"
    for item in order["items"]
}
```

One tool for everything

- uv installs Python and creates the environment
- uv add writes dependencies to pyproject.toml
- uv.lock holds exact versions for everyone

```
uv python install 3.15
uv init --package werkstatt
cd werkstatt
uv add flask
uv add --dev pytest mypy
uv run pytest
```

pyproject.toml is the contract

```toml
[project]
name = "werkstatt"
version = "0.1.0"
requires-python = ">=3.15"
dependencies = []

[project.scripts]
werkstatt = "werkstatt.cli:main"

[dependency-groups]
dev = ["pytest>=9.2", "mypy>=2.4.1"]

[build-system]
requires = ["uv_build>=0.13.6,<0.14"]
build-backend = "uv_build"
```

Modules and packages

- Every .py file is a module
- A folder with **init**.py is a package
- Startup code goes under if **name** == "**main**"

```
# src/werkstatt/cli.py
import argparse

from werkstatt.money import format_cents
from werkstatt.orders import order_total, parse_item


def main(argv: list[str] | None = None) -> int:
    parser = argparse.ArgumentParser(prog="werkstatt")
    parser.add_argument("items", nargs="+")
    args = parser.parse_args(argv)
    print(format_cents(order_total(parse_item(i) for i in args.items)))
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

Load imports only when needed

- Each import runs once from start to finish at startup
- lazy import loads only on first access
- Errors then appear only when the code is used

```
lazy import json
lazy from pandas import DataFrame

def export(orders):
    return json.dumps(orders)        # json is loaded only here
```

A class by hand

- **init** sets the attributes during creation
- self is the object the method runs on
- **eq** and **repr** are missing without extra work

```
class OrderItem:
    def __init__(self, product_id: str, quantity: int, price_cents: int) -> None:
        self.product_id = product_id
        self.quantity = quantity
        self.price_cents = price_cents

    def total_cents(self) -> int:
        return self.quantity * self.price_cents


OrderItem("p-hammer", 2, 1299) == OrderItem("p-hammer", 2, 1299)  # False
```

The dataclass writes the rest

```
from dataclasses import dataclass


@dataclass(frozen=True, slots=True)
class OrderItem:
    product_id: str
    quantity: int
    price_cents: int

    def __post_init__(self) -> None:
        if self.quantity < 1:
            raise ValueError("quantity must be at least 1")

    @property
    def total_cents(self) -> int:
        return self.quantity * self.price_cents
```

Rules belong in the object

- StrEnum limits the status to allowed values
- default_factory creates a new list for each order
- Methods check the status sequence in one place

```
class Status(StrEnum):
    DRAFT = "draft"
    PLACED = "placed"
    CANCELLED = "cancelled"


@dataclass(slots=True)
class Order:
    id: str
    customer_id: str
    items: list[OrderItem] = field(default_factory=list)
    status: Status = Status.DRAFT

    def place(self) -> None:
        if self.status is not Status.DRAFT or not self.items:
            raise ValueError("only a draft with items can be placed")
        self.status = Status.PLACED
```

Errors, files, type hints

Which exception do you catch, and which do you not? · Read and write JSON and CSV safely with pathlib · mypy checks type hints, not Python

Ask instead of checking

- Python tries first and handles the error afterward
- try contains only the line that can fail
- else runs only if nothing was raised

```
def price_of(product_id: str) -> tuple[dict[str, object], int]:
    try:
        product = catalog[product_id]
    except KeyError:
        return {"error": f"unknown product: {product_id}"}, 404
    else:
        return {"priceCents": product.price_cents}, 200
    finally:
        log("lookup", product_id)
```

Custom errors with meaning

```
class OrderError(Exception):
    """Basis für alle fachlichen Fehler beim Bestellen."""


class UnknownProduct(OrderError):
    def __init__(self, product_id: str) -> None:
        super().__init__(f"unknown product: {product_id}")
        self.product_id = product_id


class InsufficientStock(OrderError):
    def __init__(self, product_id: str, requested: int, available: int) -> None:
        super().__init__(f"only {available} of {product_id} in stock")
        self.requested, self.available = requested, available
```

Catch deliberately, keep the cause

- except catches only your own hierarchy
- raise from links the new error to the cause
- Everything else is a bug and keeps propagating

```
def parse_quantity(text: str) -> int:
    try:
        return int(text)
    except ValueError as error:
        raise InvalidQuantity(text) from error


def handle_place_order(items, catalog):
    try:
        total = place_order(items, catalog)
    except UnknownProduct as error:
        return 404, {"productId": error.product_id}
    except OrderError as error:
        return 409, {"error": str(error)}
    return 201, {"totalCents": total}
```

Paths are objects

- Path builds paths with / instead of strings
- Methods like exists, glob, and mkdir belong to the path
- with closes the file even after an error

```
import json
from pathlib import Path

data = Path(__file__).parent / "data"
with (data / "products.json").open(encoding="utf-8") as file:
    products = json.load(file)

orders_dir = data / "orders"
orders_dir.mkdir(parents=True, exist_ok=True)
sorted(orders_dir.glob("*.json"))
```

Write JSON without losing data

```
def save_order(directory: Path, order: dict) -> Path:
    directory.mkdir(parents=True, exist_ok=True)
    target = directory / f"{order['id']}.json"
    temporary = target.with_suffix(".tmp")
    temporary.write_text(
        json.dumps(order, ensure_ascii=False, indent=2), encoding="utf-8"
    )
    temporary.replace(target)
    return target
```

CSV for the spreadsheet

- DictWriter writes dicts as rows with a header row
- newline="" prevents blank lines on Windows
- DictReader returns all values as strings

```
import csv

fields = ["orderId", "productId", "quantity", "priceCents"]
with path.open("w", encoding="utf-8", newline="") as file:
    writer = csv.DictWriter(file, fieldnames=fields)
    writer.writeheader()
    for order in orders:
        for item in order["items"]:
            writer.writerow({"orderId": order["id"], **item})
```

Hints for tools, not for Python

- Python stores the hints, but does not check them
- mypy reads the code and reports contradictions
- –strict requires annotations at all boundaries

```
def find_order(orders: list[Order], order_id: OrderId) -> Order | None:
    for order in orders:
        if order["id"] == order_id:
            return order
    return None

order = find_order(orders, OrderId("o-1"))
order["items"]     # mypy: Value of type "Order | None" is not indexable
```

Types that describe the domain

```
from typing import Literal, NewType, TypedDict

OrderId = NewType("OrderId", str)
CustomerId = NewType("CustomerId", str)
type Status = Literal["draft", "placed", "paid", "shipped", "cancelled"]


class Order(TypedDict):
    id: OrderId
    customerId: CustomerId
    status: Status
    items: list[OrderItem]


discount_percent(order["id"], vip)   # mypy: expected "CustomerId"
```

Generic and structural

- Type parameters in square brackets preserve the type
- Protocol describes what an object can do, not where it comes from
- Any turns checking off and remains the exception

```
from typing import Protocol


def first[T](values: list[T]) -> T | None:
    return values[0] if values else None


class OrderStore(Protocol):
    def get(self, order_id: OrderId) -> Order | None: ...
    def save(self, order: Order) -> None: ...
```

Testing and an HTTP API for the workshop shop

Check behavior with pytest, not details · An HTTP API with Flask for the workshop shop

A test is a function with assert

- pytest finds test_.*py and test_* itself
- assert compares; pytest explains the difference
- Code receives time and IDs from the outside

```
from datetime import UTC, datetime

from shop import Product, place_order

FIXED_NOW = datetime(2026, 10, 1, 9, 30, tzinfo=UTC)
catalog = {"p-hammer": Product("p-hammer", "Hammer", 1299, 5)}


def test_place_order_sums_all_items() -> None:
    order = place_order("c-ada", {"p-hammer": 2}, catalog,
                        now=lambda: FIXED_NOW, next_id=lambda: "o-1")
    assert order.total_cents == 2598
    assert order.created_at == FIXED_NOW
```

Fixtures provide fresh data

```
# conftest.py
import pytest


@pytest.fixture
def catalog() -> dict[str, Product]:
    return {
        "p-hammer": Product("p-hammer", "Hammer", 1299, 5),
        "p-saege": Product("p-saege", "Säge", 2499, 1),
    }


# test_shop.py
def test_receipt(catalog: dict[str, Product], tmp_path: Path) -> None:
    order = place_order("c-ada", {"p-hammer": 2}, catalog, now=now, next_id=ids)
    path = write_receipt(order, tmp_path)
    assert "Summe: 25,98 €" in path.read_text(encoding="utf-8")
```

Many cases, one test

- parametrize runs the same test with many inputs
- pytest.raises checks the type and text of an error
- ids make the output readable

```
@pytest.mark.parametrize(
    ("items", "error"),
    [
        ({}, OrderError),
        ({"p-bohrer": 1}, UnknownProduct),
        ({"p-saege": 2}, InsufficientStock),
    ],
    ids=["empty", "unknown-product", "insufficient-stock"],
)
def test_rejects_invalid_orders(place, items, error) -> None:
    with pytest.raises(error):
        place(items)
```

Why Flask?

- Small core, routing and JSON built in
- Test client without a server and without an extra package
- Synchronous handlers, easy to read and test

```
from flask import Flask


def create_app(store: Store | None = None) -> Flask:
    app = Flask(__name__)
    shop = store if store is not None else Store.with_demo_data()

    @app.get("/products")
    def list_products():
        return [product_to_json(p) for p in shop.products.values()]

    return app
```

Validate the body, then place the order

```
@app.post("/orders")
def post_order():
    customer_id, items = parse_place_order(request.get_json(silent=True))
    order = shop.place_order(customer_id, items)
    return order_to_json(order), 201, {"Location": f"/orders/{order.id}"}


def parse_place_order(payload: object) -> tuple[str, list[tuple[str, int]]]:
    if not isinstance(payload, dict):
        raise ValidationError(["body must be a JSON object"])
    issues = [f"unexpected field: {key}" for key in payload
              if key not in {"customerId", "items"}]
    ...
```

Translate errors centrally

- ValidationError becomes 400 with all issues
- Domain errors become 409 or 422
- Unknown routes respond with JSON instead of HTML

```
@app.errorhandler(ValidationError)
def invalid_input(error: ValidationError):
    return {"error": "invalid input", "issues": error.issues}, 400

@app.errorhandler(InsufficientStock)
def insufficient_stock(error: InsufficientStock):
    return {"error": str(error)}, 409

@app.errorhandler(HTTPException)
def http_error(error: HTTPException):
    return {"error": error.name.lower()}, error.code or 500
```

Test with the test client

```
@pytest.fixture
def client() -> FlaskClient:
    app = create_app(Store.with_demo_data())
    app.config["TESTING"] = True
    return app.test_client()


def test_post_order_creates_the_order(client: FlaskClient) -> None:
    body = {"customerId": "c-ada", "items": [{"productId": "p-hammer", "quantity": 2}]}
    response = client.post("/orders", json=body)
    assert response.status_code == 201
    assert response.get_json()["totalCents"] == 2598
```

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen