1. Creating and importing a simple module

Step 1 – Make a module file

`my_print.py`

```
# my_print.py

MY_MESSAGE = "Hello!"

def my_print_func(text: str) -> None:
    print(MY_MESSAGE)
    print(text)
```

This file **is** a module called `my_print`.

Step 2 – Import the module in another file

`main.py`

```
# main.py
import my_print   # import the module (the whole file)

def main():
    my_print.my_print_func("Example.")
    print(my_print.MY_MESSAGE)

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

Key points:

- `import my_print` loads the **whole module** and runs its top-level code once.
- You use `module_name.thing` to access variables/functions defined inside.
- `if __name__ == "__main__":` makes the `main()` run **only** when you run `python main.py`, not when you import `main` from somewhere else.

Step 3 – Importing specific names

Instead of importing the whole module:

```
from my_print import MY_MESSAGE, my_print_func

def main():
    my_print_func("Example.")
    print(MY_MESSAGE)
```

- This pulls `MY_MESSAGE` and `my_print_func` **directly** into the current namespace.
- Then you don't need the `my_print.` prefix.

2. Packages, directory structure, and `__init__.py`

A **package** is a folder that Python treats as a "big module." Classic rule: it has an `__init__.py` inside.

2.1 Example directory structure

Example structure:

```
my_game/                # top-level folder (project)
├── main.py
├── constants.py
├── display/            # PACKAGE 'display'
│   ├── __init__.py
│   └── show_map.py     # MODULE 'show_map'
├── logic/              # PACKAGE 'logic'
│   ├── __init__.py
│   ├── computer/       # SUBPACKAGE 'computer'
│   │   ├── __init__.py
│   │   └── aimbot.py   # MODULE 'aimbot'
│   ├── game.py         # MODULE 'game'
│   └── win.py          # MODULE 'win'
```

- Every `.py` file = module.
- **Every folder with `__init__.py` = package.**
- Packages can contain:
  - modules (`game.py`)
  - subpackages (`computer/`).

2.2 Using `__init__.py` to re-export stuff

We use a middle man `__init__.py` to gather all functions / classes / variables we want to import to other files, keeps things clean.

Imagine this simplified structure:

```
logic/
├── __init__.py
├── constants/
│   ├── __init__.py
│   ├── player.py      # NUMBER_PLAYERS_Variable = 10
│   └── bot.py         # AIMBOT_PRECISION_Variable = 1.0
└── game.py
```

INSIDE `logic/__init__.py`

```
# logic/__init__.py
from .constants.player import NUMBER_PLAYERS
from .constants.bot import AIMBOT_PRECISION
```

Now in `main.py`: Much more simple to use with all imports in `__init__.py`

```
from logic import NUMBER_PLAYERS, AIMBOT_PRECISION

print(NUMBER_PLAYERS)
print(AIMBOT_PRECISION)
```

So:

- Even tho they are going into **init**.py we use `from logic import (var/class/function)`
- When you do `import logic`, Python runs `logic/__init__.py`.
- Whatever you import there becomes available as attributes on the `logic` package.

Absolute vs Relative Imports

```
my_app/
├── main.py
└── logic/
    ├── __init__.py
    ├── constants/
    │   ├── __init__.py
    │   └── player.py      # NUMBER_PLAYERS = 10
    └── computer/
        ├── __init__.py
        └── aimbot.py      # we are here

We are writing imports inside `logic/computer/aimbot.py`.
```

Absolute import (start from the top, “normal” way)

```
# logic/computer/aimbot.py
from logic.constants.player import NUMBER_PLAYERS
```

Read as: “From the top-level package `logic`, go into `constants.player`, import `NUMBER_PLAYERS`.”

- Always starts with the full package name (`logic`).
- Easy to read, doesn’t depend on where this file lives.

Relative import

```
(start from *this* package) .computer.module.func (go down one package), OR ..computer.module.func (go up
one package)
```

```
# logic/computer/aimbot.py
from ..constants.player import NUMBER_PLAYERS
```

Read as: “From `logic.computer`, go up one package (`..` → `logic`), then into `constants.player`, import `NUMBER_PLAYERS`.” Reasons we use relative imports inside a package:

- They make imports shorter when you’re deep in subpackages.
- They keep imports working even if the top-level package gets renamed.
- They’re handy in `__init__.py` to pull internal stuff up into a clean public API.

3. Simple rules to remember for Import

```
# import (no leading dots)
import logic
import logic.player

# from-import (dots allowed)
from logic import player
from .player import <module/func/variable/class>       # same package
from ..constants import <module/func/variable/class>  # parent package
```

4. Extra: important “module gotchas”

4.1 Import runs the module code once

```
# my_mod.py
print("Top level running!")
```

```
X = 42
```

```
# main.py
import my_mod
import my_mod
```

- `"Top level running!"` prints **once**, because Python caches the module.

4.2 `__name__ == "__main__"`

```
# script.py
def main():
    print("hi")

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

- If you run `python script.py` → `__name__` is `"__main__"` → `main()` runs.
- If you `import script` → `__name__` is `"script"` → `main()` does **not** auto-run.

Virtual Environments Venv

MUST Use python to run code and pip (or uv add) to install packages (not py, python3, etc.) Purpose (1-liners):

```
venv = isolated Python env per project
use venv to avoid global package/version conflicts
after activation: only use `python` + `pip`
```

Create venv (classic):

```
# Windows
py -m venv venv
```

Create venv (uv):

```
uv venv venv
```

Activate venv:

```
# Windows cmd
.\venv\Scripts\activate.bat

# Windows PowerShell
.\venv\Scripts\activate.ps1
```

Use venv:

```
python my_app.py
pip install PACKAGE       # classic
uv add PACKAGE            # if using uv
deactivate                # optional
```

Requirements.txt / pyproject.toml:

```
requirements.txt
- plain list of deps + optional versions
- used with: pip install -r requirements.txt

pyproject.toml
- project metadata + deps
- used by modern tools (uv, build, etc.)
```

Example `requirements.txt` (important parts):

```
flask==3.0.0          # exact version
sqlalchemy>=2.0       # minimum version
pytest                # any version
```

Example `pyproject.toml` (important parts):

```
[project]
name = "my_app"
version = "0.1.0"

dependencies = [
  "flask==3.0.0",
  "sqlalchemy>=2.0",
  "pytest",
]
```

Pytest, Unit Test, @pytest.fixtures

Core rules box

```
pytest
- install: pip install pytest
- run: pytest
- test files: test_*.py or *_test.py
- test funcs: def test_something():
```

```
unittest vs pytest
- unittest: built-in, class-based (TestCase, setUp/tearDown)
- pytest: external package, simple function tests, still supports classes + fixtures
- this course: pytest is the main tool
```

Basic test skeleton (value check)

```
# code to test
def add_values(a, b):
    return a + b

# test file: test_add_values.py
def test_add_values():
    result = add_values(2, 3)  # Act
    assert result == 5         # Assert
```

Testing exceptions

```
import pytest

def test_add_values_invalid():
    with pytest.raises(TypeError):
        add_values([1], [2])
```

```
import pytest

def test_parse_int_invalid():
    with pytest.raises(ValueError):
        parse_int("abc")
```

```
common exception types for pytest.raises:
- ValueError: wrong *value* type/format (e.g., int("abc"), parse_int("cat"))
- TypeError: wrong *type* of argument (e.g., 1 + "2", func(expects_list="abc"))
- ZeroDivisionError: dividing by zero (e.g., 1 / 0)
- KeyError: missing key in dict (e.g., d["missing_key"])
- IndexError: index out of range in list/string (e.g., lst[100] on a short list)
```

Fixture example (consistent setup)

```
import pytest

class Cart:
    def __init__(self):
        self.items = []
    def add(self, x):
        self.items.append(x)

@pytest.fixture
def empty_cart():
    return Cart()  # setup shared object

def test_cart_starts_empty(empty_cart):
    assert empty_cart.items == []

def test_cart_adds_items(empty_cart):
    empty_cart.add("apple")
    assert empty_cart.items == ["apple"]
```

```
@fixture = reusable setup
- define with @pytest.fixture
- tests get it by listing name as parameter
- fixture sets up consistent test data/state for one or more tests
- keeps tests clean by avoiding repeated setup code
```

Common pytest patterns a prof will test

1) Simple assert

```
def test_uppercase():
    assert "abc".upper() == "ABC"
```

2) Multiple asserts

```
def test_len_and_membership():
    data = [1, 2, 3]
    assert len(data) == 3
    assert 2 in data
```

3) Testing function that raises

```
import pytest

def test_divide_by_zero():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

4) Class-based tests

```
class TestMath:
    def test_add(self):
        assert 1 + 1 == 2

    def test_minus(self):
        assert 5 - 3 == 2
```

Classes & Objects

```
OOP core terms:
- class = blueprint / custom type (e.g., Student)
- object / instance = concrete thing created from class
- attributes = data on the object (self.name, self.id)
- methods = functions inside class (behaviour)
- state = current values of attributes
```

```
# class + __init__ + method
class Student:
    def __init__(self, name, student_number):
        self.name = name              # instance attribute
        self.student_number = student_number
        self.program = "CIT"          # default value

    def display(self):
        print(f"{self.name}, {self.student_number} - {self.program}")
```

instantiation (creating objects)

```
john = Student("John Doe", "A01234567")
bob  = Student("Bob", "A07654321")

john.display()   # inside display: self == john
bob.display()    # inside display: self == bob
```

init (initializer):

```
- special method called right AFTER the instance is created
- receives the new object as self and sets up instance attributes (self.x = ...)
- does NOT create or return the object (constructor is __new__, handled by Python)
- you call it indirectly by doing: obj = ClassName(...)
```

self:

```
- first param of instance methods
- refers to the current instance (john, bob, etc.)
- used to read/write attributes: self.name, self.program
- NEVER used outside the class block
```

Instance vs Class – Attributes & Methods

```
Definitions

Instance attribute
- stored on the object itself (self.attr)
- usually created in __init__
- each object can have different values

Class attribute
- stored on the class object (ClassName.attr)
- defined once in the class body, outside methods
- shared by all instances

Instance method
- defined with: def method(self, ...)
- self = the instance that called the method
- can use instance attributes (self.x) and class attributes (ClassName.Y)

Class method
- defined with @classmethod + def method(cls, ...)
- cls = the class (Student, BankAccount, etc.)
- usually works with class attributes or used as an alternate constructor

Why we use them
- instance attribute: store data specific to each object (each student has their own name, grade, etc.)
- class attribute: store shared config/constants for all objects (school_name, TAX_RATE, MAX_SIZE)
- instance method: behavior that depends on that object's data (deposit, withdraw, introduce, etc.)
- class method: behavior for the class as a whole (change shared settings, or create objects in special
ways like from_string)
```

Key differences Method vs Instance Attribute:

- `name` is per instance → `s1.name` and `s2.name` can be different.
- `school_name` lives on the class → changing it once affects all students.

Class Method Vs Instance Method

```python
class Student:
    school_name = "BCIT"  # class attribute

    def __init__(self, name):
        self.name = name  # instance attribute

    # instance method (most common)
    def introduce(self):
        # self = specific student (s1, s2, etc.)
        print(f"Hi, I'm {self.name} from {self.school_name}")
```

```
    # class method
    @classmethod
    def set_school(cls, new_name):
        # cls = the class Student
        cls.school_name = new_name

    # another class method as "alternate constructor"
    @classmethod
    def from_string(cls, data: str):
        # "Ryan" -> Student("Ryan")
        name = data.strip()
        return cls(name)
```

Usage:

```
s1 = Student("Ryan")
s2 = Student.from_string("Marcus")   # uses classmethod as alt constructor

s1.introduce()   # self = s1
s2.introduce()   # self = s2

Student.set_school("BCIT CIT Program")  # change class-wide setting
s1.introduce()   # now prints new school name
s2.introduce()
```

Encapsulation, Public/Private, Properties

```
Encapsulation (concept)
- hide internal details, expose a clean public interface
- goal: control how attributes are read/changed, prevent invalid state
- in Python: done by naming conventions + @property, not true hard privacy
```

Public vs “Private” attributes

```
Public attribute
- normal name: balance
- meant to be used from outside the class
- no leading underscore

"Protected" attribute (by convention)
- single leading underscore: _balance
- "internal use", but still accessible (obj._balance)
- signals: "don’t touch this from outside unless you know what you're doing"

"Private" attribute (name-mangling)
- double leading underscore: __balance
- Python renames it internally to _ClassName__balance
- makes accidental access harder, but still not true security
```

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner      # public
        self._balance = balance # "protected" by convention
        self.__pin = "1234"     # "private" (name-mangled)
```

Properties & `@property` (property decoration)

```
Property (high-level)
- lets you access methods like attributes:
    acc.balance      # calls a getter
    acc.balance = x  # calls a setter (if defined)
- used to:
  - add validation when setting values
  - compute values on the fly
  - keep a stable attribute name even if internals change
```

Read-only (getter only, no setter) property example**

```python
class BankAccount:
    def __init__(self, owner, balance):
        self._owner = owner
        self._balance = balance

    @property
    def balance(self):
        return self._balance    # read-only: no setter
```

Usage:

```python
acc = BankAccount("Ryan", 100)
print(acc.balance)   # OK, calls getter
# acc.balance = 200  # ERROR: no setter defined
```

Read(getter) & Write(setter) property with validation

```python
class BankAccount:
    def __init__(self, owner, balance):
        self._owner = owner
        self._balance = balance

    @property
    def balance(self):          # getter
        return self._balance

    @balance.setter
    def balance(self, value):   # setter
        if value < 0:
            raise ValueError("Balance cannot be negative")
        self._balance = value
```

Usage:

```python
acc = BankAccount("Ryan", 100)
acc.balance = 200        # calls setter, stored in _balance
print(acc.balance)       # 200

# acc.balance = -50      # raises ValueError
```

```
Why use @property?
- keep attribute-style syntax (acc.balance) BUT add logic/validation
- hide internal storage name (_balance) from outside code
- you can change the internal implementation later without breaking callers
```

Inheritance – Parent/Child, `super()`, Overriding

```
Key ideas
- inheritance: child (subclass) IS-A parent (base class)
- parent/base class: common attributes + methods (Vehicle)
- child/subclass: reuses parent + can add/override behavior (Car)
- method overriding: child defines a method with SAME NAME as parent → replaces it
- super(): call the parent version of an overridden method from the child
- polymorphism: same method name, same goal, different behavior per class (Vehicle.start vs Car.start)
```

Basic inheritance + overriding + `super()`

```python
class Vehicle:                # parent class
    def start(self):
        print("Vehicle starting")

class Car(Vehicle):           # Car INHERITS from Vehicle  (Car IS-A Vehicle)
    def start(self):          # override Vehicle.start
        print("Car ignition on")
        super().start()       # call parent (Vehicle) start(), same as Vehicle.start(self)
        print("Car moving")
```

Usage:

```python
v = Vehicle()
v.start()
# Vehicle starting

c = Car()
c.start()
# Car ignition on
# Vehicle starting
# Car moving
```

Overriding and Polymorphism are linked

```
Overriding = the child provides its own version of the method.
Polymorphism = your code can treat everything as a Vehicle, call start(), and get different behavior
depending on whether it's a Vehicle, Car, Truck, etc.
```

SQLAlchemy 2.0 (ORM, Engine, Session, Base, Relationships)

Core ideas (super short): ORM = map DB rows ↔ Python objects (mapped classes). Engine = DB connection. Session = "unit of work" (tracks objects, commit/rollback). `Base` = parent class that knows all tables. `mapped_column`/`Mapped` = typed columns for SQLAlchemy 2.0. `ForeignKey` + `relationship` = links tables (one-to-many, many-to-many).

Folder layout (typical bare-bones SQLAlchemy project)

```
sqlalchemy_demo/
  app/
    __init__.py
    database.py   # engine, Session, Base
    models.py     # mapped classes (tables + relationships)
    main.py       # create tables, add/query data
```

app/database.py – Engine, Session, DeclarativeBase

```
from sqlalchemy import create_engine              # Engine: DB connection + SQL executor
from sqlalchemy.orm import sessionmaker, DeclarativeBase  # Session factory + ORM base

engine = create_engine("sqlite:///demo.db", echo=True)    # echo=True logs SQL; set False to hide
Session = sessionmaker(bind=engine)                       # Session() = ORM work unit (rows ⇔ objects,
commit/rollback)

class Base(DeclarativeBase):                              # Base = parent for all ORM child classes
    pass                                                  # holds metadata + registry of mapped tables
```

app/models.py – Mapped classes + one-to-many relationship

```
from sqlalchemy import String, Integer, DECIMAL, ForeignKey   # SQL types + FK
from sqlalchemy.orm import Mapped, mapped_column, relationship
from .database import Base

class Category(Base):                                 # mapped class = table
    __tablename__ = "categories"                      # table name in DB

    id: Mapped[int] = mapped_column(primary_key=True)    # INTEGER PK AUTOINCREMENT
    name: Mapped[str] = mapped_column(String)            # TEXT/VARCHAR column

    books: Mapped[list["Book"]] = relationship(          # one Category → many Book
        back_populates="category"                        # matches Book.category
    )

class Book(Base):
    __tablename__ = "books"

    id = mapped_column(Integer, primary_key=True)          # PK
    title = mapped_column(String)                          # book title
    price = mapped_column(DECIMAL(10, 2))                  # money DECIMAL(precision, scale)
    available = mapped_column(Integer, default=0)          # stock
    category_id = mapped_column(ForeignKey("categories.id"))  # FK → categories.id
```

```
# Type hinting example
    id: Mapped[int] = mapped_column(primary_key=True)
```

app/main.py – create tables, add data, query with select()

```
from sqlalchemy import select                     # 2.0-style statement API
from .database import Base, engine, Session
from .models import Category, Book

# 1) Create tables from all Base child classes
Base.metadata.create_all(engine)                  # reads metadata from Base + mapped classes

# 2) Work with a Session (unit of work)
with Session() as session:                        # best practice: context manager

    # --- add rows ---
    sci = Category(name="Sci-Fi")
    b1 = Book(title="Dune", price=DECIMAL(10, 2)("29.99"), available=5, category=sci)
    b2 = Book(title="Neuromancer", price=DECIMAL(10, 2)("19.99"), available=3, category=sci)
```

```
    session.add_all([sci, b1, b2])                # stage objects for INSERT
    session.commit()                              # write changes to DB (objects → rows)

    # --- simple select ---
    stmt = select(Book).where(Book.available > 0)  # build SQL-like statement
    results = session.execute(stmt)               # run on engine via Session
    books = results.scalars().all()              # .scalars() → mapped objects, .all() → list

    # --- follow relationship attributes ---
    for book in books:
        print(book.title, "in category", book.category.name)  # uses Book.category relationship
```

`app/__init__.py` – (optional, but common)

```
# makes app/ a package so you can use `from app.models import Book`
```

Flask

```
Why Flask?
  * Lightweight, simple, and easy to use for web apps / APIs
  * Lots of libraries and extensions available
  * Great official docs, including a full step-by-step tutorial

What is Flask?
  * A **microframework** that handles most of the HTTP request/response work
  * Implements **WSGI** (Web Server Gateway Interface) → standard way for Python apps to talk to web
servers
  * Built-in **JSON** support (easy serialize/deserialize)
  * Comes with a **development server** so you can run and test locally
  * Built on top of **Werkzeug** (powerful underlying library for WSGI, routing, etc.)
```

Flask applications are built on these core concepts:

```
Application: Central object managing the entire web application

Views: Functions that handle requests and generate responses

Routes: URL patterns that map to view functions

Templates: Jinja2-powered HTML files for dynamic content generation

Blueprints: Modular components for organizing application functionality
```

url_for

```
from flask import url_for

@app.route("/users/<user_id>")
def profile(user_id):
    ...

# Somewhere else in code/template:
url_for("profile", user_id=5)   # -> "/users/5"
```

`url_for` is useful because it dynamically generates URLs based on the view function name, so your links never break if routes change. It keeps navigation consistent, avoids hard-coding paths, and makes your app easier to maintain.

1) Basic app structure – single file

```
# app.py
from flask import Flask, render_template, request, redirect, url_for  # core Flask

app = Flask(__name__)  # Flask instance = web app (loads config, finds templates/static)

@app.route("/")  # route: GET /
def home():      # view function: handles request, returns response
    return "<h1>Hello Flask</h1>"  # simple HTML response string

@app.route("/hello/<name>")        # dynamic URL: /hello/Ryan → name="Ryan"
def hello(name):
    return f"Hi {name}!"

@app.route("/submit", methods=["GET","POST"])  # allow both GET+POST
def submit():
    if request.method == "POST":              # form submit
        data = request.form["username"]       # POST form data
        return redirect(url_for("hello",name=data))  # build URL from endpoint name
    return render_template("form.html")       # GET: render template file

if __name__ == "__main__":                    # run dev server
    app.run(debug=True)                       # debug=True = auto reload + error page
```

---

2) Application Factory Pattern + Blueprints

Folder layout (tiny):

```
flask_app/
  app/
    __init__.py      # create_app
    pages.py         # blueprint
    templates/
      pages/
        home.html
```

`app/pages.py` **– blueprint module**

```
from flask import Blueprint, render_template, request

pages_bp = Blueprint("pages",__name__)  # name used in url_for("pages.home")

@pages_bp.route("/")                    # route now lives on blueprint, not app
def home():
    return render_template("pages/home.html")  # looks in templates/pages/home.html

@pages_bp.route("/search")
def search():
    q = request.args.get("q","")          # query string: ?q=python
    return f"Results for {q}"
```

`app/__init__.py` **– application factory**

```
from flask import Flask
from .pages import pages_bp

def create_app():
    app = Flask(__name__)               # create new app instance
    app.config["SECRET_KEY"] = "dev"    # example config
    app.register_blueprint(pages_bp)    # attach blueprint routes to app
    # app.register_blueprint(pages_bp,url_prefix="/pages")  # optional prefix
    return app
```

Running with factory (example `run.py`):

```
from app import create_app
app = create_app()      # app built by factory

if __name__=="__main__":
    app.run(debug=True)
```

3) Routing + url_for (static, dynamic, methods)

```
from flask import Flask, request, render_template, redirect, url_for
app = Flask(__name__)

@app.route("/about")               # static URL
def about():
    return render_template("about.html")

@app.route("/user/<int:user_id>")  # dynamic int converter
def user_profile(user_id):
    return f"User id = {user_id}"

@app.route("/post/<slug>")        # dynamic string by default
def show_post(slug):
    return f"Post: {slug}"

@app.route("/login",methods=["GET","POST"])
def login():
    if request.method=="POST":
        username = request.form.get("username")  # POST form field
        # auth logic...
        return redirect(url_for("dashboard"))    # build URL from endpoint name
    return render_template("login.html")         # GET: show form

@app.route("/dashboard")
def dashboard():
    return "Welcome!"
```

In template (Jinja) – link building with `url_for`:

```
<a href="{{ url_for('about') }}">About</a>       <!-- /about -->
<a href="{{ url_for('user_profile',user_id=5) }}">User 5</a>
<a href="{{ url_for('login') }}">Login</a>
```

---

4) Views & Request Data + Responses (render_template / redirect)

```
from flask import Flask, request, render_template, redirect, url_for
app = Flask(__name__)
```

```
@app.route("/search")
def search():
    # /search?term=flask&page=2
    term = request.args.get("term","")    # query string (?term=)
    page = request.args.get("page",1,type=int)
    context = {"term":term,"page":page}
    return render_template("search.html",**context)  # HTML response

@app.route("/contact",methods=["GET","POST"])
def contact():
    if request.method=="POST":
        name = request.form.get("name")     # POST form body
        msg  = request.form.get("message")
        # process/store...
        return redirect(url_for("thanks",name=name))  # HTTP 302 → /thanks/<name>
    return render_template("contact.html")  # GET: show form

@app.route("/thanks/<name>")
def thanks(name):
    return render_template("thanks.html",name=name)
```

Key mini-reminders:

- `request.args` → query string (`GET /path?x=1`)
- `request.form` → form body from POST
- `render_template("file.html",x=1)` → returns HTML response
- `redirect(url_for("endpoint",**params))` → HTTP redirect to another route

Flask-SQLAlchemy (ORM) – compact cheat sheet

```
# --- CONFIG + DB SETUP ---
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import select

db = SQLAlchemy()                                   # db = engine + session + Model base

def create_app():
    app = Flask(__name__)
    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///taskmanager.db"  # DB URL (driver://path)
    app.config["SQLALCHEMY_ECHO"] = True            # log SQL in console (dev only)
    db.init_app(app)                                # bind db to app
    with app.app_context(): db.create_all()         # create tables for all db.Model subclasses
    return app
```

MODELS + ONE-TO-MANY

```
from datetime import datetime

class User(db.Model):                               # inherit from db.Model → table
    __tablename__ = "user"                          # explicit table name
    id = db.Column(db.Integer, primary_key=True)    # INTEGER PK AUTOINCREMENT
    username = db.Column(db.String(80), unique=True, nullable=False)  # UNIQUE, NOT NULL
    email = db.Column(db.String(120), unique=True, nullable=False)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)      # default timestamp
    tasks = db.relationship(                        # one User → many Task (ORM link, no column)
        "Task", back_populates="assignee",
        cascade="all, delete-orphan"                # delete user → delete all their tasks
    )

class Task(db.Model):
```

```
    __tablename__ = "task"
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(200), nullable=False)
    is_done = db.Column(db.Boolean, default=False, nullable=False)
    assignee_id = db.Column(                          # actual FK column in “many” side
        db.Integer, db.ForeignKey("user.id"), nullable=False
    )
    assignee = db.relationship(                       # many Task → one User (Python side)
        "User", back_populates="tasks"
    )

    def __repr__(self): return f"<Task {self.title!r}>"  # debug string; !r = repr(self.title)
```

MANY-TO-MANY (ASSOCIATION TABLE)

```
task_tag = db.Table(                                  # pure association TABLE (no extra cols)
    "task_tag",
    db.Column("task_id", db.Integer, db.ForeignKey("task.id"), primary_key=True),
    db.Column("tag_id", db.Integer, db.ForeignKey("tag.id"), primary_key=True)
)

class Tag(db.Model):
    __tablename__ = "tag"
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), unique=True, nullable=False)
    tasks = db.relationship(                          # Tag ⇔ Task many-to-many
        "Task", secondary=task_tag, back_populates="tags"
    )

class Task(db.Model):
    __tablename__ = "task"
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(200), nullable=False)
    tags = db.relationship(                           # uses same association table
        "Tag", secondary=task_tag, back_populates="tasks"
    )
```

MANY-TO-MANY (ASSOCIATION OBJECT WITH EXTRA DATA)

```
class TaskTag(db.Model):                              # association OBJECT = extra columns allowed
    __tablename__ = "task_tag"
    task_id = db.Column(db.Integer, db.ForeignKey("task.id"), primary_key=True)
    tag_id  = db.Column(db.Integer, db.ForeignKey("tag.id"), primary_key=True)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
    task = db.relationship("Task", back_populates="task_tags")
    tag  = db.relationship("Tag", back_populates="task_tags")

class Task(db.Model):
    __tablename__ = "task"
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(200), nullable=False)
    task_tags = db.relationship(                      # one Task → many TaskTag rows
        "TaskTag", back_populates="task",
        cascade="all, delete-orphan"                  # delete task → delete link rows
    )

class Tag(db.Model):
    __tablename__ = "tag"
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), unique=True, nullable=False)
    task_tags = db.relationship(                      # one Tag → many TaskTag rows
        "TaskTag", back_populates="tag",
        cascade="all, delete-orphan"
    )
```

CREATE (C in CRUD)

```
u = User(username="ryan", email="ryan@example.com")   # create Python object (not in DB yet)
db.session.add(u)                                     # stage INSERT in current transaction
db.session.commit()                                   # flush changes → DB + commit transaction

t = Task(title="Study", assignee=u)                   # can set relationship instead of assignee_id
db.session.add(t); db.session.commit()
```

READ / QUERY (2.0 STYLE)

```
# all users ordered by username
stmt = select(User).order_by(User.username)           # build SELECT * FROM user ORDER BY username
users = db.session.execute(stmt).scalars().all()      # execute → get list[User]

# filter with .where()
stmt = select(Task).where(Task.is_done == False)      # WHERE is_done = 0
open_tasks = db.session.execute(stmt).scalars().all()

# filter with multiple conditions
stmt = select(Task).where(
    Task.assignee_id == u.id,
    Task.is_done == False
).order_by(Task.title)
user_tasks = db.session.execute(stmt).scalars().all()

# single row helpers
one_user = db.session.execute(
    select(User).where(User.username == "ryan")
).scalar_one_or_none()                                # 0/1 results → User or None

user = db.get_or_404(User, user_id)                   # SELECT by PK, abort(404) if not found
```

UPDATE (U in CRUD)

```
user = db.get_or_404(User, user_id)                   # load existing row as object
user.email = "new@example.com"                        # modify fields (session marks as dirty)
user.username = "newname"
db.session.commit()                                   # on commit → UPDATE user SET ... WHERE id=?

task = db.get_or_404(Task, task_id)
task.is_done = True                                   # flip boolean
db.session.commit()
```

DELETE (D in CRUD)

```
user = db.get_or_404(User, user_id)
db.session.delete(user)                               # mark object (and cascades) for DELETE
db.session.commit()                                   # DELETE FROM user WHERE id=?; cascades run

task = db.get_or_404(Task, task_id)
db.session.delete(task); db.session.commit()
# if relationship had cascade="all, delete-orphan", children/association rows are removed too
```

QUICK RECAP OF KEY SYNTAX

```
# config: app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///file.db"
# model base: class User(db.Model): id = db.Column(db.Integer, primary_key=True)
# column types: db.Integer, db.String(80), db.Boolean, db.DateTime, db.Text, db.Float
# constraints: primary_key=True, unique=True, nullable=False, default=value, db.ForeignKey("table.col")
# one-to-many: db.relationship("Child", back_populates="parent", cascade="all, delete-orphan")
# many-to-many: db.relationship("Other", secondary=assoc_table, back_populates="others")
# session ops: db.session.add(obj); db.session.delete(obj); db.session.commit()
# querying: stmt = select(Model).where(...).order_by(...); result = db.session.execute(stmt)
# results: result.scalars().all(), .scalar_one(), .scalar_one_or_none()
# helper: db.get_or_404(Model, pk)
```