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
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)

```python
# 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)
```

```python
# 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

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

# relative 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
```

Gotcha: Import runs the module code once

```python
# 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

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:

```
activate venv  # `python` now points to venv's Python
python my_app.py       # MUST Use python to run code, not py or python3, will run app using venv's
interpreter + packages
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()
```

Abstract Base Classes

An Abstract Base Class (often referred to as an ABC) is a mechanism used to define "generic classes." These classes define a specific public interface (a set of methods) without actually implementing the logic for them. Like a strict blueprint, tells subclasses what methods they need without telling them how those methods should work.

Why:

- Enforcing interfaces(insures child classes follow a guideline to prevent issues down the line)
- Polymorphism(Allows the use of different objects the same way) --- Example below

```
from abc import ABC, abstractmethod

# 1. Define the Abstract Base Class
class Animal(ABC):

    def __init__(self, name):
        self.name = name

    # 2. Define the abstract method (The Interface)
    # This acts as a rule: "All animals must make a sound"
    @abstractmethod
    def sound(self):
        pass

# 3. Define Child Classes (Concrete Classes)

class Dog(Animal):
    # We MUST implement sound(), or this class will error
    def sound(self):
        return "Woof"

class Cow(Animal):
    def sound(self):
        return "Moo"

# Usage
# my_animal = Animal("Generic") # This would RAISE AN ERROR because you cannot instantiate an ABC

dog = Dog("Buddy")
cow = Cow("Daisy")

print(f"{dog.name} says {dog.sound()}") # Output: Buddy says Woof
print(f"{cow.name} says {cow.sound()}") # Output: Daisy says Moo

# Polymorphism check
print(isinstance(cow, Animal)) # Output: True [2]
```

```
# This is BAD (Not Polymorphic)
if type(my_animal) == Cow:
    my_animal.moo()
elif type(my_animal) == Dog:
    my_animal.bark()

# This below is GOOD (Polymorphic)
# You don't care if it is a Cow or a Dog, you just know it is an "Animal", so it MUST have a .sound()
method.
my_animal.sound()
```

```
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**

```
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:

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

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

```
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:

```
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 ONLY

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

```python
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
```

NEED TO FINISH SQL ALCHEMY ONLY

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.)
```

Application structure

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
```

Flask Class Instantiation

```
from flask import Flask
app = Flask(__name__)   # instantiate the Flask class
```

- `Flask` is a **class** provided by the framework.
- `Flask(__name__)` **creates the application object** your project uses.
- `__name__` helps Flask locate templates, static files, etc.
- **All routes, config, and extensions are handed off to this `app` instance** (everything attaches to it).

Flask Application

Flask application is the Central object that represents your web application. Its an instance of the `Flask` class and is entry point for handling http requests

Flask application responsibilities:

```
Initialize the application with configuration
Register blueprints and routes
Handle the request/response cycle
Manage application context and configuration
Provide access to app-wide resources
```

Simple Flask Project Folder Example

```
# STRUCTURE
# my_flask_app/
# ├─ run.py
# ├─ requirements.txt
# └─ app/
#     ├─ __init__.py
#     ├─ routes.py
#     ├─ models.py
#     ├─ templates/
#     │   └─ index.html
#     └─ static/
#         └─ style.css
```

```python
# ========= run.py =========
from app import create_app          # import factory from package
app = create_app()                  # create Flask app instance
if __name__ == "__main__":          # only if file run directly
    app.run(debug=True)             # start dev server

# ======== app/__init__.py ======== APPLICATION FACTORY
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()                   # global db object shared by models

def create_app():                   # app factory (returns configured app)
    app = Flask(__name__)           # __name__ = this module path
    app.config["SECRET_KEY"] = "dev-secret-key"         # for sessions/forms
    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///demo.db"  # DB URL
    app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False         # disable extra overhead
    db.init_app(app)                # bind db to this app
    from .routes import main_bp     # import blueprint AFTER app exists
    app.register_blueprint(main_bp) # attach routes to app
    return app

# ======== app/models.py ========
from . import db                    # use same db as in __init__

class User(db.Model):               # ORM mapped class → users table
    id = db.Column(db.Integer, primary_key=True)                 # PK
    username = db.Column(db.String(80), unique=True, nullable=False)  # NOT NULL + UNIQUE
    def __repr__(self):
        return f"<User {self.username!r}>"   # nice debug display

# ======== app/routes.py ========
from flask import Blueprint, render_template
from .models import User
from . import db

main_bp = Blueprint("main", __name__)   # blueprint = mini app module

@main_bp.route("/")                     # route for "/"
def index():
    # SQLAlchemy 2.0 style query: select(User) → execute → scalars().all()
    users = db.session.execute(db.select(User)).scalars().all()
    return render_template("index.html", users=users)  # pass data to template
```

Application Factory Pattern

App Factory Pattern: rather than have `app = Flask(__name__)` Global, its inside a function

```python
from flask import Flask

def create_app(config=None):
    app = Flask(__name__)              # create a new App Factory

    # 1) load config here if needed
    # 2) register blueprints here
    # 3) init extensions (db, login_manager, etc.)

    return app                         # give the caller the ready-to-use app
```

Why use create_app() instead of global `app = Flask(__name__)`?

- Can create **multiple app instances** with different configs (dev/prod/tests).
- Avoids **circular imports** (routes/blueprints live in separate files).
- **Easier testing**: tests just call `create_app(test_config)`.
- Without factory: one global app made at import time = less flexible.
- With factory: call a function that **builds + configures + wires + returns** a fresh app.

Flask Routing

@app.route decorator

`@app.route("/something")` tells Flask: **"When a browser asks for this URL, call this function."**

HTTP Methods

- "GET" → browser is asking for the page from you (server) (load the form, show HTML).
- "POST" → browser is sending data made by users to you (server) (submitting a form).
- `request.method` = "GET" or "POST"; use it in `if` to choose between "show form" and "handle submitted form".

```python
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/feedback", methods=["GET", "POST"])
def feedback():
    if request.method == "GET":              # browser is asking for the page
        return render_template("feedback.html")  # show the empty form

    if request.method == "POST":             # browser is sending form data
        msg = request.form.get("message")    # read <input name="message">
        # here you would save msg to a DB, send email, etc.
        return "Thanks for your feedback!"   # simple confirmation text
```

Flask Blueprints (Alternate Routing / Modularization)

blueprint is mainly about routing + organizing chunks of your app.

- **What**: A *Blueprint* is a mini Flask module that groups related routes, templates, and logic, but is **not** a full app. It must be plugged into a real app with `app.register_blueprint(...)`.
- **Why**
  - **Organization**: split big apps into chunks, BETTER FOR BIG APPS
    - `pages` → home/about/contact, `auth` → login/logout/register, `api` → JSON endpoints
  - **Reusability**: reuse the same blueprint in different projects.
  - **Scalability**: add features by adding blueprints, not editing one giant `app.py`.
  - **Team-friendly**: each dev owns a blueprint/module.

  - **Namespaces**: avoids name clashes; use `url_for("pages.home")` vs `url_for("auth.login")`.

- **How blueprints modularize the app**
  - Each feature lives in its own file/package (routes, templates, static).
  - The main app (`create_app`) only needs to **register** the pieces.
  - You can enable/disable sections by registering/unregistering blueprints.

Define + Register a Blueprint Example

Define

```
# pages.py  (feature module)
from flask import Blueprint, render_template

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

@pages_bp.route("/")        # "/" route on this blueprint
def home():
    return render_template("home.html")

@pages_bp.route("/about")   # "/about" route on this blueprint
def about():
    return render_template("about.html")
```

Register

```
# __init__.py  (application factory)
from flask import Flask
from .pages import pages_bp

def create_app():
    app = Flask(__name__)
    app.register_blueprint(pages_bp)              # URLs: "/", "/about"
    # app.register_blueprint(pages_bp, url_prefix="/pages")
    # → URLs: "/pages/", "/pages/about"
    return app
```

URL building with `url_for`

- builds URL from **view/endpoint name**, not hardcoded path.

```
from flask import url_for, redirect  # import in Python files

@app.route("/about")
def about(): ...                     # endpoint name = "about"
url_for("about")                     # "/about"

@app.route("/user/<int:id>")
def user_profile(id): ...            # endpoint name = "user_profile"
url_for("user_profile", id=3)        # "/user/3" (fills <int:id>)

# with blueprint "pages" and def home(): ...
url_for("pages.home")                # uses blueprint namespace

redirect(url_for("about"))           # build URL then redirect
```

Dynamic URL Paremeter

```
"/user/id" → # static path.
Only matches exactly /user/id.
id here is just plain text in the URL.

"/user/<id>" → # DYNAMIC URL PARAMETER.
Matches /user/ryan, /user/123, /user/anything.
The part in < > becomes a function argument.

"/user/<int:id>" → # DYNAMIC URL PARAMETER WITH TYPE CONVERTER.
#FLASK WILL CONVERT "5" --> 5
```

```
@app.route("/user/id")
def static_example():
    return "This only matches /user/id"

@app.route("/user/<id>")
def dynamic_example(id):
    return f"User ID is {id}"
```

Flask: Views and Requests

Views: The functions under Routes (`@app.route`, `@pages_bp.route`)

The same view(function) handles GET and POST, and chooses logic based on request.method

```
from flask import request, render_template

@app.route("/feedback", methods=["GET", "POST"])
def feedback():                # ← this is the *view function*
    if request.method == "GET":
        return render_template("feedback.html")  # show form
    if request.method == "POST":
        msg = request.form.get("message")        # handle form submit
        return "Thanks!"
```

The `request` object

- Represents the **current HTTP request** (everything the browser sent).
- Import in views: `from flask import request`
- Important:
  - `request.method` → "GET", "POST", etc. (which HTTP method was used)
  - `request.args` → access query parameters in URL (usually with **GET**).
  - `request.form` → form fields sent in request body (usually with **POST**).

Accessing Query parameters – `request.args`

- Query Parameters: Data in the **URL after** `?` (query string), typically on **GET** requests.
- Example URL: `/search?term=cat&limit=10`
- In view:

```
term = request.args.get("term")     # "cat"
limit = request.args.get("limit")   # "10" (string)
```

Accessing form data - `request.form`

- Form data: Key–value data sent in the HTTP **request body** (not the URL) from an HTML `<form>`.
- Typically on **POST** requests (`<form method="post">`), but any method with form-encoded body can use it.

- Example flow:
  - HTML: `<form method="post" action="/login">`
  - Browser sends: `POST /login` with body `username=ryan&password=secret`
  - In view: use `request.form` to read `"username"` / `"password"`.

```
- request.form["field"]          # strict, KeyError if missing
- request.form.get("field")      # safer, returns None if missing
- request.form.get("field", "")  # safer with default
- request.form.getlist("field")  # checkbox / multi-select
```

```
@pages_bp.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        # ◇ access form data sent from browser
        username = request.form.get("username")  # safe, returns None if missing
        password = request.form["password"]      # raises KeyError if missing
```

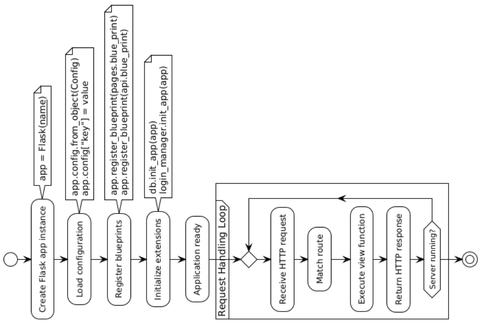
Responses: `redirect`, `render_template`.

In Flask, your view function must return a response. Two super-common helpers for this are `render_template` and `redirect`

```
from flask import Blueprint, render_template, request, redirect, url_for

bp = Blueprint("pages", __name__)

@bp.route("/")
def home():
    # render_template -> load templates/home.html, return HTML response
    return render_template("home.html", title="Home")

@bp.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        username = request.form.get("username")  # form data from POST body
        password = request.form.get("password")
        # (check creds, maybe flash error, etc.)
        return redirect(url_for("pages.home"))   # redirect -> new GET to "/"
    # first visit or failed login: show form
    return render_template("login.html")
```

```
Responses (Flask)
- render_template("file.html", x=1)
    -> uses Jinja2 template in templates/ dir
    -> injects vars as {{ x }} and returns HTML response
- redirect("/path") / redirect(url_for("pages.home"))
    -> returns 3xx response with Location header
    -> browser does NEW GET to that URL
    -> used after POST (Post/Redirect/Get)
```

Flask Application Lifecycle



Flask-SQLAlchemy

In a typical Flask + SQLAlchemy app, they “meet” in 3 places:

1) app/init.py → configure DB + init SQLAlchemy with Flask app

2) app/models.py → define models that inherit from db.Model

3) app/views.py or app/routes.py → use models + db.session to query/save

1.) app/init.py: CONFIG + DB SETUP

```
Plain SQLAlchemy init.py
 - You must create everything manually:
    engine = create_engine("sqlite:///db.db")
    Session = sessionmaker(bind=engine)
    class Base(DeclarativeBase): pass
    class Product(Base): ...
        - Models inherit from Base (Product(Base)).

Flask-SQLAlchemy init.py
    - The SQLAlchemy() extension builds all of this for you:
    - Instead You just do:
    db = SQLAlchemy()
    db.init_app(app)
    class Product(db.Model): ...
```

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import select

db = SQLAlchemy()                                    # db = engine + session + Model base. Flask allows
us to skip

def create_app():   # APPLICATION FACTORY
    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
```

2.) app/models.py: Where we define our DB tables as Python classes.`

SQLalchemy only: models inherit from `Base`, use bare `Column`, `Integer`, etc.

Flask + SQLalchemy: models inherit `db.Model` and use `db.Column`, `db.Integer`,

```python
# This file = where we define our DB tables as Python classes.

from . import db  # db = SQLAlchemy() created in app/__init__.py

class Product(db.Model):
    """Single table example: products in a store."""

    __tablename__ = "product"  # actual table name in the database

    id = db.Column(db.Integer, primary_key=True)           # PK: unique row id
    name = db.Column(db.String(100), nullable=False)       # NOT NULL name
    price = db.Column(db.Float, nullable=False)            # NOT NULL price
    in_stock = db.Column(db.Boolean, default=True)         # default True

    def __repr__(self):
        # nice string when you print(Product(...))
        return f"<Product id={self.id} name={self.name!r}>"
```

3.) app/views.py (or app/routes.py): Use models + db.session to query/save data.

SQLalchemy only: you instantiate Session() yourself and pass it around. No HTTP, just Python code.

Flask + SQLalchemy: you use the global db.session provided by the extension inside views/routes, and also handle HTTP (request, render_template, redirect).

```python
from flask import Blueprint, render_template, request, redirect, url_for
from . import db                  # db = SQLAlchemy() from __init__.py
from .models import Product       # our model class

store_bp = Blueprint("store", __name__)

@store_bp.route("/products")
def list_products():
    # READ: query all products from the DB
    products = db.session.query(Product).all()
    return render_template("products/list.html", products=products)

@store_bp.route("/products/new", methods=["GET", "POST"])
def create_product():
    if request.method == "POST":
        # READ form data from POST body
        name = request.form.get("name")
        price = float(request.form.get("price", 0))

        # CREATE: make a Product object (not in DB yet)
        product = Product(name=name, price=price)

        # ADD + COMMIT = insert row into DB
        db.session.add(product)
        db.session.commit()

        # go back to list page after saving
        return redirect(url_for("store.list_products"))

    # GET: show the form
    return render_template("products/new.html")
```

Relationship patterns

One-to-Many:

- Example: User -> Task
- One User has many Tasks
- Each Task has exactly ONE User

- Implemented with FK on the "many" side: Task.user_id -> User.id

Many-to-Many:

- Example: Task <-> Tag
- One Task: many Tags
- One Tag: many Tasks
- Implemented with association table: task_tag(task_id FK -> task.id, tag_id FK -> tag.id)