MODULE 0 — Your Baseline: Python for Scalable, Readable AI Code

Even before CS concepts, agentic systems rely heavily on clean, modular Python.

🔑 Topics

- Python packaging basics (**init**.py, modules, imports)
- Virtual environments & dependency management
- Python dataclasses (hugely useful in agent & tool definitions)
- Type hints + Pydantic models (common in agent frameworks)
- Async programming fundamentals (async/await) — Agent frameworks often call tools asynchronously.

🧪 Exercises

- Rewrite a small script you've written using functions + modules.
- Convert a data-cleaning script to async I/O for file/database reads.
- Wrap an object using @dataclass to store parameters for a pipeline.

◇ Module 0.1 — Python Packaging Basics

(`__init__.py`, modules, imports) Agentic systems are built as collections of cooperating components, not single scripts. Packaging is what allows that.

☑ Concept

A Python project is usually organized like this:

```
my_agent/
    __init__.py
    tools/
        __init__.py
        file_tools.py
        math_tools.py
    agents/
        __init__.py
        planner.py
    config.py
main.py
```

Every folder that contains an `__init__.py` becomes a package.

📌 Example: turning a messy script into a small package

✕ Before — one giant script

```
# giant_script.py
import json
import math

def load_numbers(path):
    with open(path) as f:
        return json.load(f)

def compute_stats(numbers):
    return {
        "mean": sum(numbers)/len(numbers),
        "sq": [math.sqrt(n) for n in numbers]
    }

numbers = load_numbers("numbers.json")
print(compute_stats(numbers))
```

☑ **After — modular, scalable layout**

```
stats_app/
    main.py
    tools/
        __init__.py
        io_tools.py
        stats_tools.py
```

Takeaway Project has 1 high-level folder, in that is a main.py file & program subfolders (in this case being tools which computes the stats of interest.

tools/io_tools.py

```
import json

def load_numbers(path: str):
    with open(path) as f:
        return json.load(f)
```

tools/stats_tools.py

```
import math
from typing import List, Dict
```

```
def compute_stats(numbers: List[float]) -> Dict:
    return {
        "mean": sum(numbers) / len(numbers),
        "sq": [math.sqrt(n) for n in numbers],
    }
```

main.py

```
from tools.io_tools import load_numbers
from tools.stats_tools import compute_stats

numbers = load_numbers("numbers.json")
print(compute_stats(numbers))
```

Takeaway This is structured exactly how I do ad hoc scripting currently, but again written using more formal CS semantics.

💡 Why this matters for agentic AI

Agent frameworks (LangChain, Autogen, CrewAI, OpenAI Swarm, etc.) expect:

- tools in modules (e.g., tools.web_search:search_web)
- agents in modules (e.g., agents.planner:Planner)
- a main entry script to orchestrate everything You can't scale an agent system without package structure.

Q: What does it mean to have tools & agents in modules? Could you define what you mean by this word?

Appendix

- See Module 0, *What is a "module" in Python?*

A: A "module" is any Python file (.py) that Python can import.

- Example: `some_folder/some_process.py` -> module inside a package
- Packages vs. Modules: Package is a folder containing a `__init__.py` file and possible multiple modules (whereas a module could simply be a single .py file)

Example structure:

```
my_project/
    tools/              ← package
        __init__.py
        file_tools.py    ← module
        math_tools.py    ← module
    agents/             ← package
        __init__.py
        planner.py       ← module
    main.py              ← module
```

Takeaway

- Semantics here matter to make the job of AI easier, code is imported cleanly & logical flows are registered automatically
- Agents usually exist as classes or functions stored in modules
- In agentic AI
    - A tool = a function that lives inside a module
    - An agent = a class or object that lives inside a module
    - A workflow = something that imports these modules
- You're building structures, no scripts This is why you often see repos like:

```
/agents
/tools
/data_models
/pipelines
/app.py
```

🧪 Mini-Exercise

Organize this single-file script into a package with a `tools/` and `processing/` folder:

```
import csv

def read_csv(path):
    with open(path) as f:
        return list(csv.DictReader(f))

def filter_rows(rows):
    return [r for r in rows if float(r["value"]) > 10]

def main():
    rows = read_csv("data.csv")
    print(filter_rows(rows))

main()
```

If you want, I can walk you through the solution step-by-step.

Done. Make sure to add """Short description""" to each function.

◇ Module 0.2 — Virtual Environments & Dependency Management

(`venv`, `pip`, `pyproject.toml`) This is about isolating your project, especially tools that call APIs.

📌 Example: Create and use a virtual environment

1. Create a venv

```
python3 -m venv .venv
```

2. Activate it

Mac/Linux:

```
source .venv/bin/activate
```

Windows:

```
.venv\Scripts\activate
```

3. Install dependencies

```
pip install pydantic aiohttp numpy
```

4. Freeze dependencies

```
pip freeze > requirements.txt
```

5. Re-create environment later

```
pip install -r requirements.txt
```

Q: Typically I use Anaconda Navigator to manage python environments in projects, what are the differences and when is it appropriate to use one over the other?

A: When you're doing heavy scientific computing (NumPy, SciPy, GPU, etc.) -> `conda` is good here A: Building modern Python apps / agent frameworks / web backends that use pip , uv , etc. use `venv` .

Q: If I've created a data science solution while using a `conda` environment and I'd like to automate this solution through the use of AI, can I create requirements.txt files from `conda` environments?

A: Yes but make sure `conda` environments used pip install for everything!

- `conda` -> for exploratory, scientific, messy, "anything goes" development
- `venv` + `requirements.txt` -> for reproducible, lightweight, production-ready environments
- Key is understand what can and cannot be directly exported from `conda` to `venv` .
  - `pip freeze > requirements.txt` -> inside `conda` environment terminal window (e.g., same as you'd do when creating & freezing a `venv` environment.
  - Caveat: this will export only packages that were installed via pip

Appendix

- See Module 0, *The Right Way: pip-compatible export*

Setting Up Python Natively On Machine

- See Section, *Should you use `venv` instead of `conda`?*

Takeaways

- `conda` is good for managing non-automated (e.g., ad hoc, experimental, etc.) code environments because it manages a lot of details regarding Python environments for you, easy to share & install with others
  - `conda` is not as AI-friendly as `venv` though, especially when it comes to using AI to interpret & build code
- Action: Use pre-established `conda` environments for exploratory development work, but back into `venv` if exploratory work needs to be scaled for automation

Q: What is `pyproject.toml` and how is it used in the project settings of AI for data science?

A: modern Python project configuration file -> Agent systems need modular components, `pyproject.toml` defines the package structure cleanly (precise version locking)

- E.g., ensures analysts & AI agents are running the same code -> most AI environments rely on tools which generate these!

Exercise 2 Notes

- Make sure main folder name of package matches name attribute in `pyproject.toml` file + have a `__init__.py` file directly under it as well otherwise you won't be able to use the package.
  - Both main project folder & `pyproject.toml` files need to be stored in the same hierarchy level as well!

- Once `venv` environment is activated, type & enter python to activate a Python REPL instance (which allows you to run single line commands for testing the use of the package created.

Project folder should be structured to look something like this

```
C:\...\module_0\exercise_2\
    pyproject.toml
    my_project.egg-info
    my_project/
        __init__.py
        main.py
        tools/
            __init__.py
            io_tools.py
        processing/
            __init__.py
            data_tools.py
```

Q: What is the purpose of the `__init__.py` file & what kind of contents are supposed to go into it?

Appendix

- See Module 0, What Is The Purpose of The `__init__.py` File & What Kind of Contents Go Into It?

A: First & foremost, it's what Python uses to differentiate a folder from a Python package.

- Not technically needed with modern Python, but having it is considered a best practice

A: An empty `__init__.py` file is still considered valid, it's just content Python checks for when import calls are made. There common pattern uses for this file

1. Leave it blank
2. Exposing submodules / public API (see Use Case 2 example below)
3. Package-level constraints, configurations, metadata
    1. Can expose program's app name, version, etc.
    2. Agent frameworks use it for agent configuration details, global settings, & tool registries
4. Pack initialization logic (rare but powerful case)
    1. Often used in plug-in systems, agent tool auto-discovery, framework that load modules by side effect
    2. Seems to take place of having to print certain actions occurring within a program

In short, the `__init__.py` file comes into play as part of the transition to viewing tools & agents as modules in code development, which is a big part of AI-driven code development.

💡 Why this matters for agentic AI

Each agent might depend on:

- different API clients

- different versions of libraries
- local toolchains (OCR, embeddings, audio models) A clean venv ensures isolation + reproducibility.

◇ Module 0.3 — Dataclasses

Super useful for agent configurations, messages, and tool definitions.

📌 Example: without dataclass

```
agent_config = {
    "temperature": 0.2,
    "model": "gpt-4.1",
    "max_tokens": 1024
}
```

This is error-prone — any key typo breaks things.

☑ With dataclass

```
from dataclasses import dataclass

@dataclass
class AgentConfig:
    temperature: float
    model: str
    max_tokens: int = 1024
```

Usage:

```
cfg = AgentConfig(temperature=0.2, model="gpt-4.1")
print(cfg)
```

Output:

```
AgentConfig(temperature=0.2, model='gpt-4.1', max_tokens=1024)
```

Q: What is the purpose of the `@dataclass` decorator?

A: In the backend, Python handles certain boilerplate code automatically with the use of it. -> makes the class itself more lightweight

- By including this, replaces the need to write `__init__`, `__repr__`, & `__eq__` methods within class (e.g., methods that should exist in every place basically
- Basically instead of predefining the `__init__` method (e.g., what the class is going to expect to receive as values & parameters upon initializing an object instance) the parameters & values are just passed & an `__init__` method is interpreted based on how the class was called!
- Can also make frozen (immutable) data classes by calling `@dataclass(frozen=True)` -> if a script tries to overwrite a parameter value in an object instance an error is thrown.

Note: Pydantic's `.model_dump()` converts a class into a dictionary, while `.model_dump_json()` serializes it to a JSON-encoded string.

💡 Why this matters

Agent frameworks often require:

- structured definitions
- tools with well-defined parameters
- type-safe configs Dataclasses give you clean, validated containers.

🧪 Mini-Exercise

Create a @dataclass called DatasetInfo that stores:

- name (str)
- num_rows (int)
- source_url (optional str, default None)

◇ Module 0.4 — Type Hints + Pydantic Models

Type hints = clarity + IDE help Pydantic = validation for agent messages & tool inputs

---

📌 Example: Type hints for functions

```
from typing import List

def normalize(values: List[float]) -> List[float]:
    total = sum(values)
    return [v / total for v in values]
```

📌 Example: Pydantic model for a tool request

```
from pydantic import BaseModel

class WeatherRequest(BaseModel):
    city: str
    units: str = "metric"
```

Pydantic ensures:

```
WeatherRequest(city=123)
```

→ raises error, because city must be a string.

✔ Type hints

Type hints declare what types your variables, function arguments, and class attributes are supposed to be.

✔ Pydantic models (BaseModel)

Pydantic validates and enforces types at runtime.

🔥 Why this matters specifically for agentic & multimodal AI

Agent frameworks (OpenAI function calling, LangChain, CrewAI, Autogen, etc.) rely on structured data schemas to:

- Validate tool inputs
- Validate tool outputs
- Prevent LLM hallucination
- Interpret user requests into typed objects
- Ensure the agent's memory and state are type-safe
- Convert between Python objects ↔ JSON

Pydantic is the backbone of this in modern AI systems.

Think of Pydantic models as:

"Guaranteed-correct containers for data passing between tools, agents, and models." Type hints alone cannot do this.

🌉 Dataclasses vs. Pydantic: When to use which?

✔ Use @dataclass when:

- You want a lightweight container
- No validation needed

- Data is guaranteed to be internal/controlled
- You want immutability or automatic methods Examples:
- Agent configuration
- Internal state objects
- Parameters passed between internal functions

✔ **Use Pydantic models when:**

- Data might be invalid
- Data comes from users, APIs, agents, LLMs, or files
- You want guaranteed structure
- You're building tools for agents Examples:
- Tool input schemas
- API request/response schemas
- Agent output schemas
- Any structured data passed between systems

Takeaway

- typing allows for more specific constraints on data types & `BaseModel` takes class definitions a step further by actually throwing an error if there's typing mismatches
  - Typically without this the only thing explicitly defining parameter data types does is provide developers guidance in the hover tooltip
- This constraints are important when it comes to debugging AI-generated code, errors occur at the boundary, not deep inside logic (issues occurring are less abstract)

Q: How does `BaseModel` compare to `@dataclass` mentioned previous? They seem to cover the same bases?

A: `BaseModel` feels like a `@dataclass` , but it's not literally using `@dataclass` under the hood.

- `@dataclass` differs because
  - No validation by default
  - No automatic coercion
  - No .model_dump() or JSON tools
- `BaseModel` also offers
  - Nested models being validated recursively (see bonus exercise under Exercise 4)
  - I/O boundaries (APIs, tools, agents)
- `BaseModel` is more like `@dataclass` + validation + serialization + nesting behaviors"

💡 Why this matters

Agents exchange structured data, e.g.:

- user messages
- tool invocation schemas
- agent plans In modern agent frameworks, Pydantic is a first-class citizen.

◇ Module 0.5 — Async Programming

Agentic systems frequently:

- call APIs
- run tools
- fetch documents
- process events You need async to avoid blocking an entire workflow.

Core idea

Synchronous code:

- Does one thing at a time
- If a network call or file read is slow, everything waits Async code:
- Lets you start several tasks (like API calls)
- While one waits on I/O, others can run
- Perfect for agents calling multiple tools / APIs

🔬 Where this matters in agent systems

Agent frameworks make heavy use of:

Async

For:

- fetching web pages
- running 3 tools at once
- streaming responses
- hitting multiple APIs in parallel Example:

```
await asyncio.gather(
    search_google(query),
    search_bing(query),
    search_pubmed(query)
)
```

🔬 How AI Systems Use This Combo (Real Examples)

✔ LLM Serving

Parallel processes → handle multiple users Async inside each process → handle streaming, tokenization, external API calls

✔ Agent Systems

Parallel processes → multiple agents running simultaneously Async inside each agent → multiple tool calls at once

✔ Scientific Workflows

Parallel processes → batch simulation runs Async inside each simulation → log streaming, file checkpoints, API calls

✔ Web Services

Parallel processes → gunicorn workers Async inside each worker → asyncio-based request handling

---

Tiny concrete example

Synchronous version

```
import time

def fetch_data(name: str):
    print(f"Starting {name}")
    time.sleep(2)  # pretend this is a slow API call
    print(f"Finished {name}")

def main():
    fetch_data("task 1")
    fetch_data("task 2")
    fetch_data("task 3")

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

This always runs 1 → 2 → 3, ~6 seconds total.

Async version

```
import asyncio

async def fetch_data(name: str):
    print(f"Starting {name}")
    await asyncio.sleep(2)  # pretend this is a slow API call
    print(f"Finished {name}")

async def main():
    tasks = [
        fetch_data("task 1"),
        fetch_data("task 2"),
        fetch_data("task 3"),
    ]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

All three tasks “sleep” at the same time → finishes in ~2 seconds instead of ~6. This is exactly what you want when an agent:

- hits 3 APIs
- or queries 3 tools
- or downloads 3 documents in parallel.

Q: Is what's happening in the async example the same as spawning parallel processes? I know parallelism will be covered in a later module but it's another term I've heard about and was curious how it related to this.

Appendix

- See Module 0, *Question: Is Asynchronous Code The Same Idea As Parallel Processing?*

A: Async is not the same as parallel processes

- Async -> lets you overlap waiting time, not computation -> basically in the list of tasks everything waits then starts at the same time as opposed to going through each list item one at a time, but from a hardware perspective everything is still happening within a single CPU thread
  - Async is cooperative concurrency, not true parallel execution
- Parallelism -> actually runs multiple tasks at the same time, across multiple CPU cores or even machines & is used for
  - Heavy numerics (computation)
  - ML training
  - CPU-bound simulations
  - vectorized data transformations
  - physics or materials modeling kernels

Takeaway

- Async, everything starts at once, allows tasks to run concurrently which helps in cases where 1 task takes much longer than others to complete
- Agent frameworks rely on async for -> I/O logic
  - Fetching web pages
  - running 3 tools at once
  - streaming responses
  - hitting multiple APIs in parallel

Q: Is it fair to say that asynchronous and parallelism can be used together in a single software solution then?

A: Most modern software & agentic AI systems run this way.

- Each parallel process gets its own async event loop, and each event loop manages its own asynchronous tasks.
- It's common for handling processes such as

- Model-serving systems
- Data pipelines
- Agent tool orchestration
- Scientific computing workflows
- microservices

- Visual comparison of how async & parallelism fit together

```
Process A:
    async task1
    async task2
    async task3
Process B:
    async task4
    async task5
    async task6
```

- In this approach, the individual processes are not replicating async tasks
  - Spawning multiple processes (example: 1 per CPU core)
  - Each process runs its own async tasks -> not every process would be handling exactly the same series of steps
  - The operating system (OS) schedules those processes in parallel

---

🧪 Mini-Exercise 0.5

Let's do a quick async practice tailored to your context. Goal: simulate running 3 model evaluations in parallel.

Step 1 — Write an async function

Create:

```
import asyncio

async def run_experiment(name: str, seconds: int):
    print(f"[{name}] starting...")
    await asyncio.sleep(seconds)
    print(f"[{name}] done after {seconds} seconds")
    return name, seconds
```

Step 2 — Run 3 of them concurrently

```
async def main():
    tasks = [
        run_experiment("exp_small", 1),
        run_experiment("exp_medium", 2),
        run_experiment("exp_large", 3),
    ]
    results = await asyncio.gather(*tasks)
    print("All done:", results)

asyncio.run(main())
```

What you should see conceptually

- exp_small, exp_medium, exp_large all “start” almost at the same time
- They complete in order of their sleep times
- Total runtime ≈ 3 seconds, not 1 + 2 + 3 = 6

If you’d like, you can:

- Paste your version of that async exercise (like you’ve done for previous ones), and I’ll review
- Or we can take this and then connect it directly to “how async is used in real agent frameworks” next (e.g., concurrent tool calls, streaming, etc.) Either way, you’re doing really solid work here.

📌 Example: synchronous vs async API call

❌ Synchronous

```
import requests

def get_page(url):
    return requests.get(url).text
```

Blocks the whole program.

☑ Async version

```
import aiohttp
import asyncio

async def get_page(url: str) -> str:
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as resp:
            return await resp.text()

asyncio.run(get_page("https://example.com"))
```

Now you can run 100 requests concurrently:

```
async def main():
    urls = ["https://example.com"] * 100
    pages = await asyncio.gather(*(get_page(u) for u in urls))

asyncio.run(main())
```

💡 Why this matters for agentic AI

Most agent frameworks use async patterns:

- multiple tools invoked at once
- parallel function calls
- background reasoning loops
- streaming responses If you know async, you can build much more powerful agents.

🧠 Module 0 Capstone: The Async EDA Micro-Agent

Let's build your Module 0 Micro-Agent Project — a small but meaningful agent that uses every concept you learned:

- Python packaging & modules
- virtual environment
- dataclasses
- Pydantic models
- async functions & concurrency
- clean folder structure and imports This will feel like your first "real" agentic system, just simplified.

🎯 Goal

Build a tiny agent that:

1. Loads a dataset (CSV)
2. Validates a user-provided configuration (Pydantic)
3. Runs multiple analysis tasks concurrently (async)
4. Uses a dataclass to store agent config
5. Outputs results as a Pydantic model This mirrors the core flow of many agent pipelines used in AI-for-science.

🗂 Folder Structure (Recommended)

```
project_root/
    pyproject.toml
    my_agent/
        __init__.py
        config.py           ← dataclass-based agent configuration
        models.py           ← Pydantic schemas for results
        tools/
            __init__.py
            io_tools.py     ← load CSV
            analysis_tools.py← async analysis functions
        agent.py            ← orchestrates async tasks
        main.py             ← runs the micro-agent
```

This mirrors the layout of real agent frameworks.

Copy each file's code into your project:

- `config.py` → dataclass
- `models.py` → Pydantic schemas
- `tools/io_tools.py` → CSV loader
- `tools/analysis_tools.py` → async tasks
- `agent.py` → async orchestrator
- `main.py` → entrypoint runner

This is your agent's "brain."

🔧 1. Define a Config Dataclass

my_agent/config.py

```
from dataclasses import dataclass

@dataclass
class AgentConfig:
    dataset_path: str
    run_summary: bool = True
    run_row_count: bool = True
    run_column_count: bool = True
```

This matches how agent configurations are stored in LangChain, Autogen, Swarm, etc.

🔧 2. Define Pydantic Result Models

my_agent/models.py

```
from pydantic import BaseModel
from typing import Optional

class SummaryResult(BaseModel):
    mean: float
    std: float

class AnalysisResult(BaseModel):
    row_count: Optional[int] = None
    column_count: Optional[int] = None
    summary: Optional[SummaryResult] = None
```

Pydantic gives us well-typed, validated outputs.

🔧 3. Build Tools (I/O + async analysis)

I/O Tool

my_agent/tools/io_tools.py

```
import csv

def load_csv(path: str):
    with open(path, "r") as f:
        return list(csv.DictReader(f))
```

Async Analysis Tools

my_agent/tools/analysis_tools.py

```
import asyncio
import statistics

async def count_rows(rows):
    await asyncio.sleep(0)  # pretend async work
    return len(rows)

async def count_columns(rows):
    await asyncio.sleep(0)
    return len(rows[0]) if rows else 0

async def compute_summary(rows):
    await asyncio.sleep(0)
    # Assume dataset has a column named "value"
    values = [float(r["value"]) for r in rows]
    return {
        "mean": statistics.mean(values),
        "std": statistics.stdev(values) if len(values) > 1 else 0.0
    }
```

Note: the fake sleeps simulate async; real agents would be calling APIs or tools.

🤖 4. Create the Agent (Async Orchestrator)

my_agent/agent.py

```
import asyncio
from .config import AgentConfig
from .tools.io_tools import load_csv
from .tools.analysis_tools import (
    count_rows,
    count_columns,
    compute_summary,
)
from .models import AnalysisResult, SummaryResult

async def run_agent(cfg: AgentConfig) -> AnalysisResult:
    rows = load_csv(cfg.dataset_path)

    tasks = []

    if cfg.run_row_count:
        tasks.append(count_rows(rows))
    if cfg.run_column_count:
        tasks.append(count_columns(rows))
    if cfg.run_summary:
        tasks.append(compute_summary(rows))

    results = await asyncio.gather(*tasks)

    # Map results back to named fields
    output = AnalysisResult()

    idx = 0
    if cfg.run_row_count:
        output.row_count = results[idx]; idx += 1
    if cfg.run_column_count:
        output.column_count = results[idx]; idx += 1
    if cfg.run_summary:
        summary_values = results[idx]
        output.summary = SummaryResult(**summary_values)

    return output
```

This block demonstrates:

- async orchestration
- dynamic task scheduling

- mapping results to typed models
- clean layering of I/O, compute, models, agent logic

🎬 5. Entry Point

my_agent/main.py

```
import asyncio
from .config import AgentConfig
from .agent import run_agent

if __name__ == "__main__":
    cfg = AgentConfig(dataset_path="data.csv")
    result = asyncio.run(run_agent(cfg))
    print(result.model_dump())
```

🔧 6. Create your pyproject.toml

Something like:

```
[project]
name = "my_agent"
version = "0.1.0"
description = "Module 0 Micro-Agent Project"
requires-python = ">=3.10"
dependencies = [
    "pydantic>=2.0",
]

[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"
```

Keep it simple — Module 0 is about foundations.

🔧 7. Install your project in editable mode

From the folder that contains `pyproject.toml`:

```
pip install -e .
```

This is where most packaging errors show up, and if they do, we'll fix them together.

🔧 8. Create a simple data.csv

Example:

```
value
1
2
3
4
5
```

Place it in the project root or specify a full path in your config.

🔧 9. Run the micro-agent

In your venv:

```
python -m my_agent.main
```

or just:

```
python my_agent/main.py
```

Expected output:

```
{'row_count': 5, 'column_count': 1, 'summary': {'mean': 3.0, 'std': 1.5811}}
```

🧩 When you're done...

Reply here with:

✔ Your folder layout

✔ Your code (you can paste each file as you did before)

✔ The console output

✔ Any errors you hit (if something breaks)

I will:

- review your structure
- validate your implementation
- help debug anything
- tune the design like a software engineer reviewing a junior engineer’s PR
- optionally help you extend the agent with additional tools or behaviors