Functional Pipeline Basics

Scripts

- pipeline1.py
- pipeline2.py

Self-Exploration (≈ 5 min)

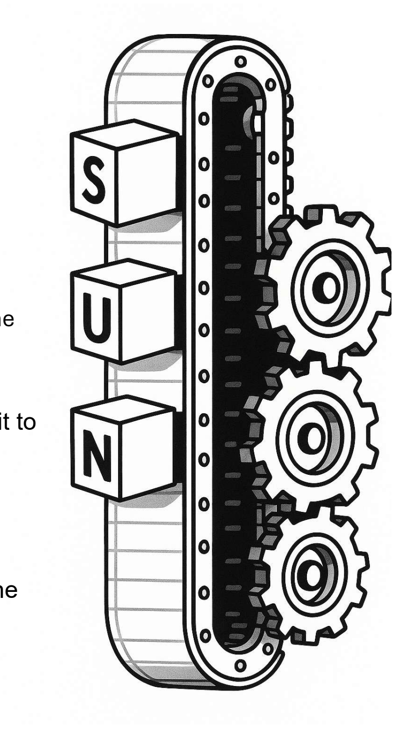
1. Skim each function and add a short **docstring** that states purpose, parameters, and return value.
2. In pipeline2.py, locate run_independent and run_chained. Add two inline comments that clarify *how the flow of data differs* between them.

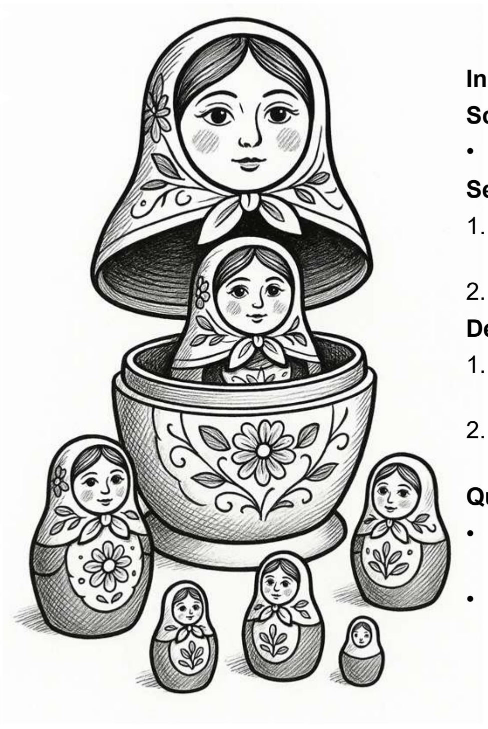
Deepening Tasks

1. Adapt the script so the pipeline **reverses** the string *before* converting it to upper-case. Verify the result with an assert.
2. Explain in one sentence why the existing assert at the bottom of pipeline2.py is a good practice when sharing code.

Quick Notes / Hints

- *Independent* runs each function on the **original** input; *chained* feeds the **output** of each step into the next.
- A minimal test prevents silent logic drift when you refactor.





Inner and Outer Methods & Auto-Tests

Scripts

- pipeline3.py

Self-Exploration (≈ 5 min)

1. Identify the function that **returns** another function. Add a docstring explaining what goes in and what comes out.
2. Mark each place where that returned function is *invoked*.

Deepening Tasks

1. Write a new closure make_vowel_doubler(vowel:str) that doubles any given vowel inside the input string.
2. Insert an assert that checks the transformed string contains **exactly twice** the number of that vowel.

Quick Notes / Hints

- A closure "remembers" the variable values that were in scope when it was created.
- Small asserts serve as living documentation and regression tests.

Decorators & Debugging

Scripts

- pipeline4.py

Self-Exploration (≈ 5 min)

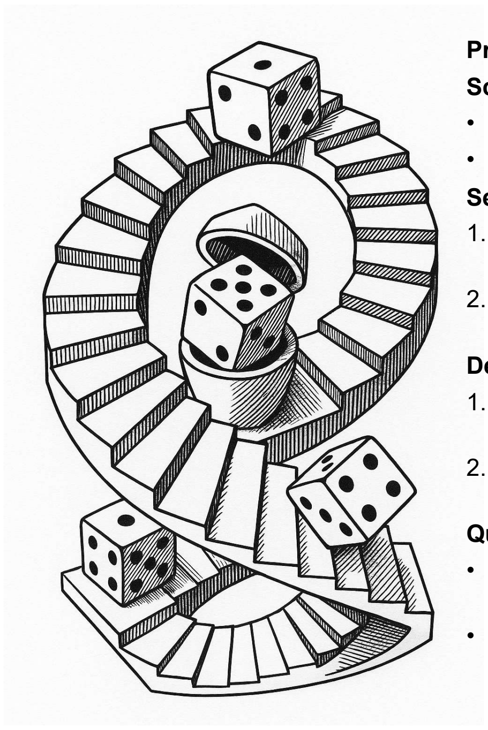
1. Locate make_double_run_method. Add a docstring explaining *why* the wrapper executes its argument twice.
2. Comment the **print** statements to clarify what the log shows.

Deepening Tasks

1. Modify the wrapper so it prints the *length* of the string after each run.
2. In one sentence, contrast a *decorator* with a *plain higher-order function*.

Quick Notes / Hints

- Decorators are syntactic sugar around higher-order functions that *wrap* another function.
- Debug output can be toggled off later but speeds up reasoning during development.



Probabilistic Steps & Self-Healing Loops

Scripts

- pipeline4a.py
- pipeline4b.py

Self-Exploration (≈ 5 min)

1. Add a docstring to UnreliableMakeAppender describing its failure chance.
2. In repeat_until_condition_met, annotate the loop condition with a comment.

Deepening Tasks

1. Change the condition so the loop stops when the string **ends with a period (.)**.
2. Briefly discuss: *Why might repeated application be preferable to raising an exception in DH text processing?*

Quick Notes / Hints

- Probabilistic steps model noisy processes such as OCR, language detection or LLMs.
- A retry loop trades runtime for robustness when exact success timing is uncertain.

Object-Oriented Pipelines & Strategy Pattern

Scripts

- pipeline5.py

Self-Exploration (≈ 5 min)

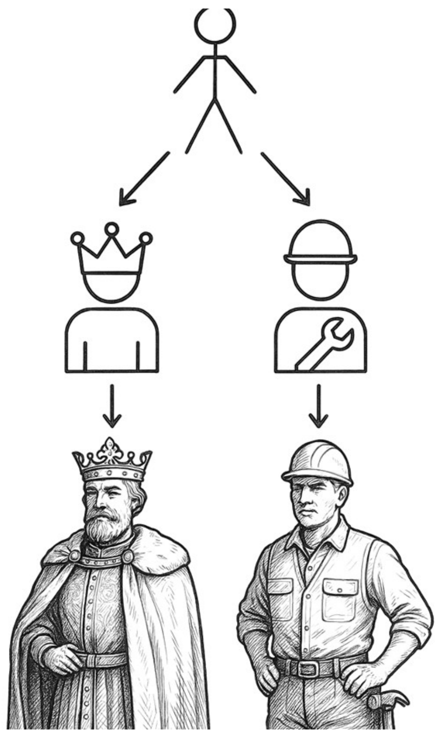
1. Provide docstrings for PipelineStep and Pipeline.
2. Comment the difference between run_independent and run_chained methods.

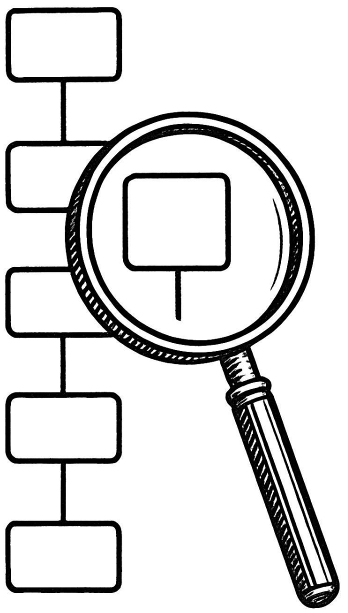
Deepening Tasks

1. Implement a new class RemovePunctuationStep that deletes commas and periods from the input.
2. Reflect: *One advantage* and *one drawback* of an OO approach compared to pure functions.

Quick Notes / Hints

- Each step is a *strategy* object holding the algorithm for a single transformation.
- Method polymorphism allows easy swapping without touching the orchestration code.





Optimisation & Visualisation

Scripts

- pipeline6.py,
- pipeline7.py,
- pipeline7b.py,
- pipeline8.py,
- pipeline9.py

Self-Exploration (≈ 5 min)

1. Add a one-line docstring to train_chain describing its search strategy.
2. In pipeline9.py, mark the spot where a Pipeline instance is **re-used** as a PipelineStep.

Deepening Tasks

1. Time how long train_chain takes in pipeline6.py vs pipeline7b.py on the same input. Note the difference.
2. Use visualize() to export a pipeline9.gv file. In two bullet points, describe what the diagram reveals.

Quick Notes / Hints

- Hash-based skip lists cut down duplicate work in random search.
- Turning a pipeline into its own step enables **nesting** and hierarchical workflows.