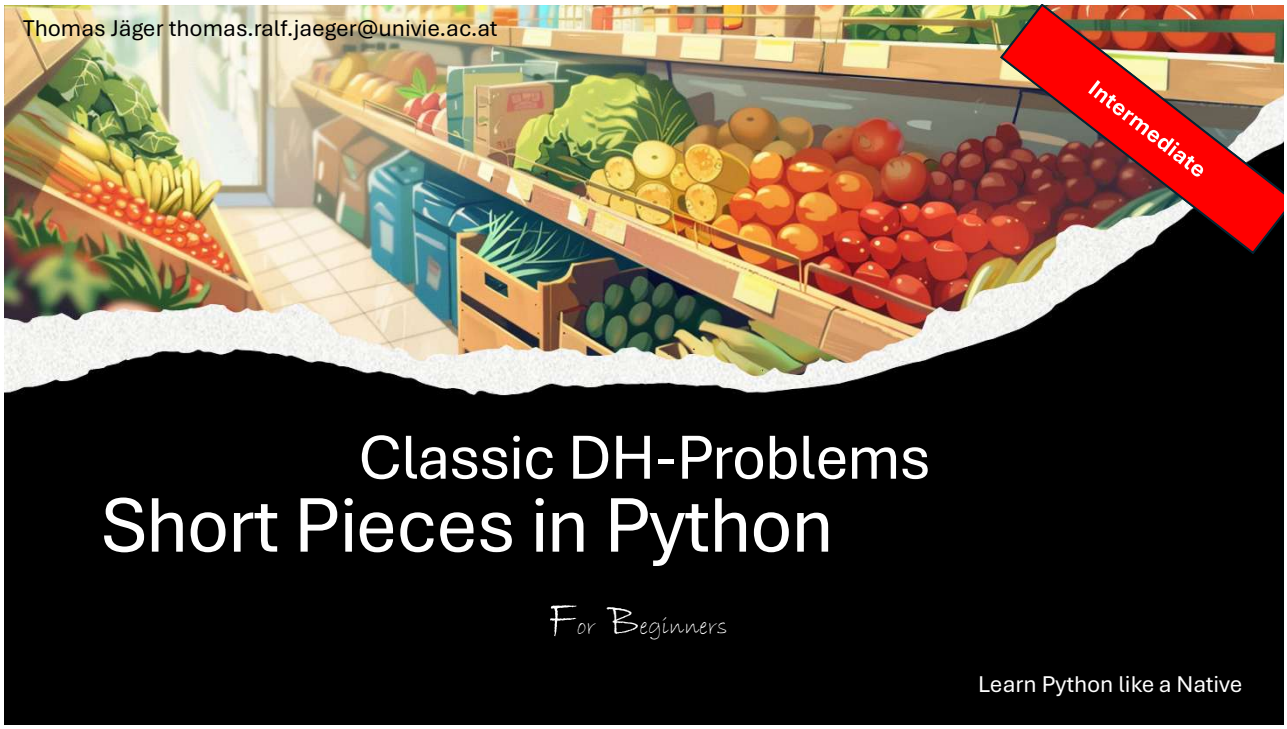
Thomas Jäger thomas.ralf.jaeger@univie.ac.at



Classic DH-Problems
Short Pieces in Python

For Beginners

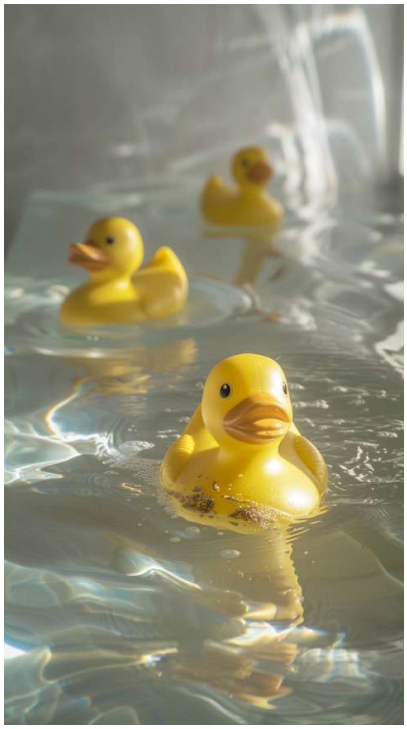
Learn Python like a Native

Programming:
- Functional Programming
- Closures
- Decorators
- Object oriented Programming
- Strategy Pattern

Developing:
- Using LLM
- Pipelines

Architecture:
- Local LLM
- Pipeline
- Folders
- Check Tasks

Professional Knowledge:
- Pair Programming



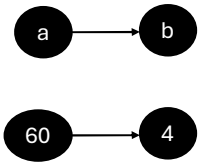
Methods/ Functions

- A function maps an archetype to an image.

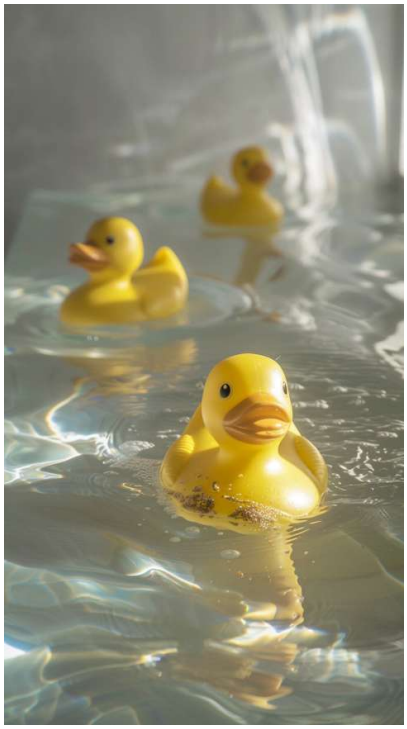
  $f: \frac{b}{15} \qquad b = 60;$

  $f(b) = 4$

- Methods/functions are first class members in Python



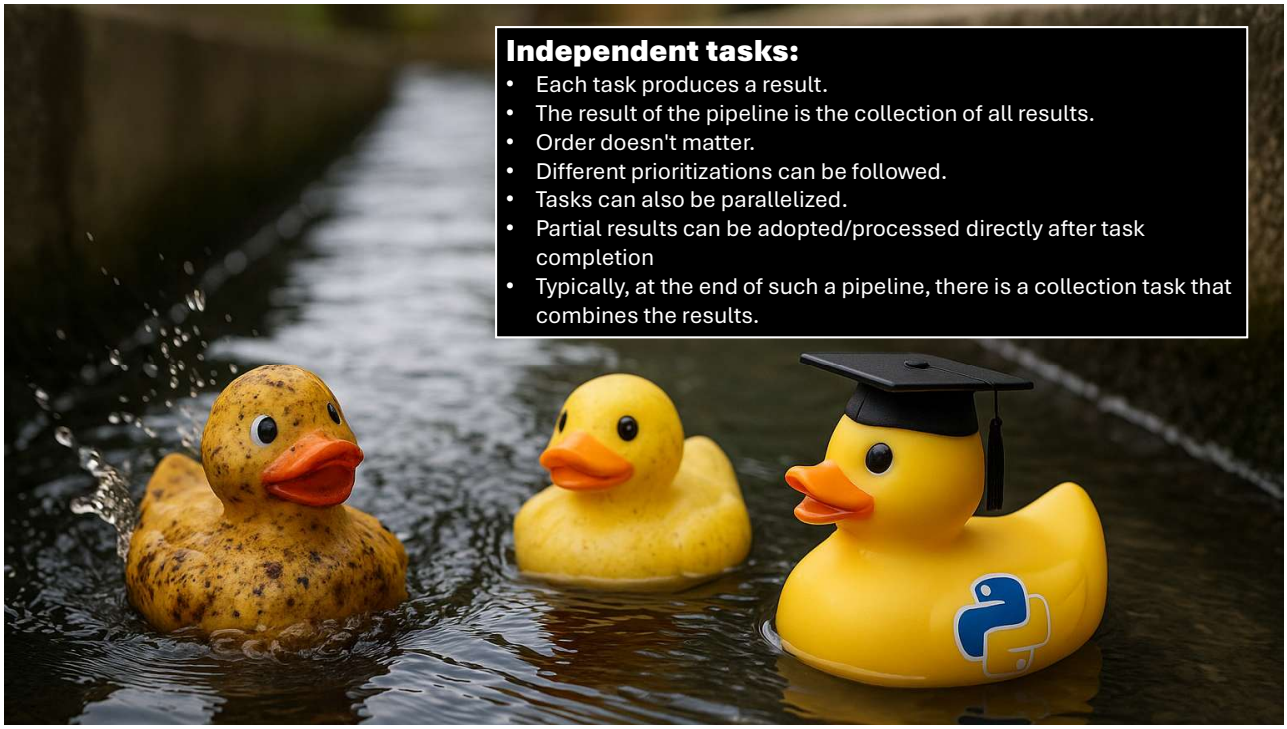

```
def pipeline_chained(s: str, funcs: list) -> None:
    """
Applies each function on result of last
    application and prints out current state.
    """
    current = s
    for f in funcs:
        current = f(current)
        print(current)
```



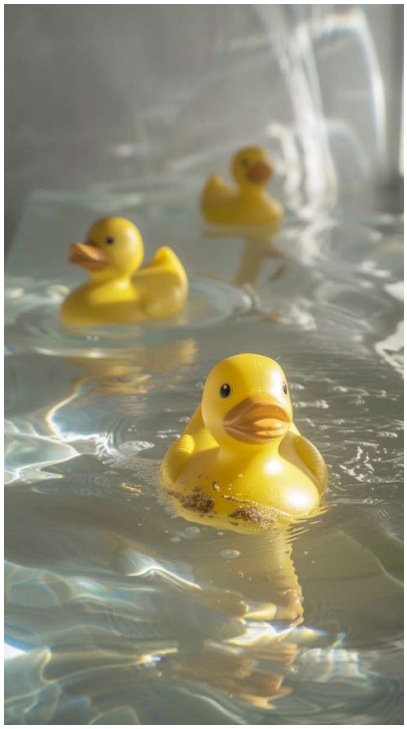
Pipelines:

- Collection of tasks
- One task after the other is completed.
- Each task maps an archetype to an image.
- Classic implementation:

  `for n in listTasks:`
- A function maps an archetype to an image.


Independent tasks:
• Each task produces a result.
• The result of the pipeline is the collection of all results.
• Order doesn't matter.
• Different prioritizations can be followed.
• Tasks can also be parallelized.
• Partial results can be adopted/processed directly after task completion
• Typically, at the end of such a pipeline, there is a collection task that combines the results.


Dependent tasks:
• The overall pipeline produces a result, individual pipeline steps produce an intermediate result.
• Pipelines can be interrupted and continued, but the result can only be adopted or further processed after a complete run
• Changes in the order have an impact on the overall result
• Pipelines, but not tasks, can be parallelized.

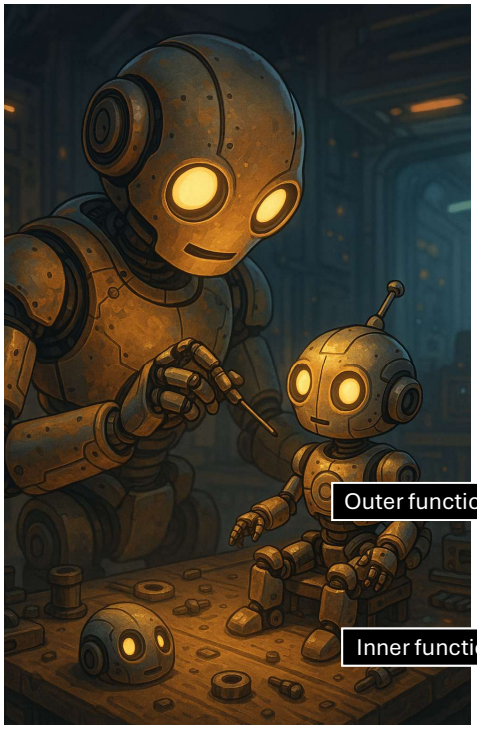


Example:

- *A text is translated into 4 languages*

Example:

- *A manuscript page is scanned*
- *The text is read via OCR*
- *The OCR-text is annotated in TEI,*
- *The TEI is transformed to HTML*
- *The HTML is published on a web-page*



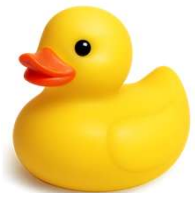
Closure:

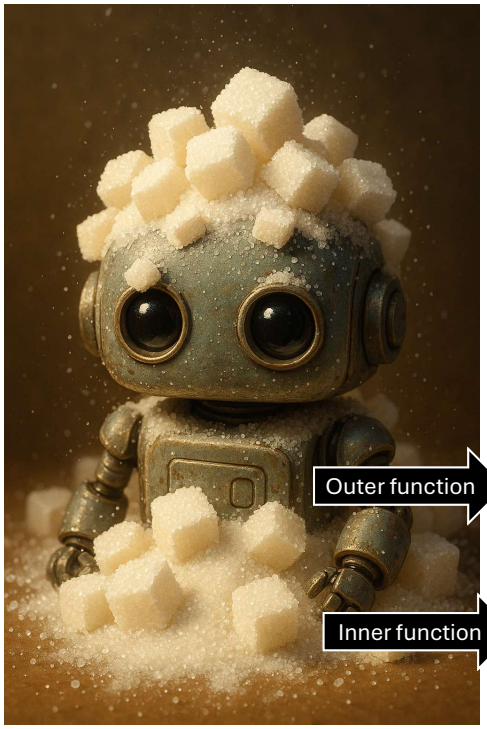
- A method that returns a method.
- The outer function returns an inner function.
- Inner function keeps outer function's variables alive (Scope).
- After the outer function returns, that state persists.
- Re-calling the inner function accesses or updates it.
- Passing/returning the inner function moves its closed-over scope.
- Enables encapsulation, memoization, and function factories.

Outer function

Inner function

```
def make_appender(letter: str) -> Callable[[str], str]:
    """
Returns a function that will append
the letter at the end of a string.
    """
    def appender(s: str) -> p:
        return s + letter
    return appender
```





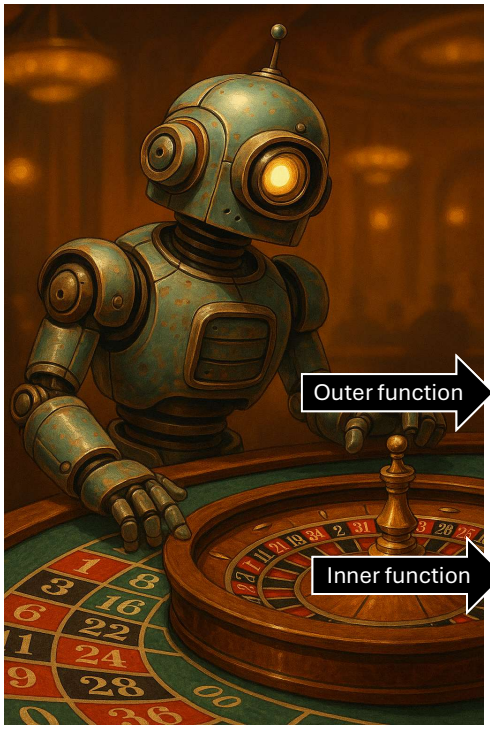
Syntactic Sugar:

- A decorator is similiar to a closure.
- It wraps a function inside another function.
- It can be used by ‚decorating a function' by writing ‚@' followed by the name of the decorator function above the methode-statement e.g. **@make_double_run_method**)

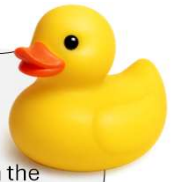
Debugging with assert

The result of a pipeline (or a pipeline-task) can be check with assert.

```
def make_double_run_method(func: Callable[[str], str]) -
> str:
    """
Apply func twice on string.
"""
    def wrapper(s: str) -> p:
        result = func(s)
        result = func(result)
        return result
    return wrapper
```

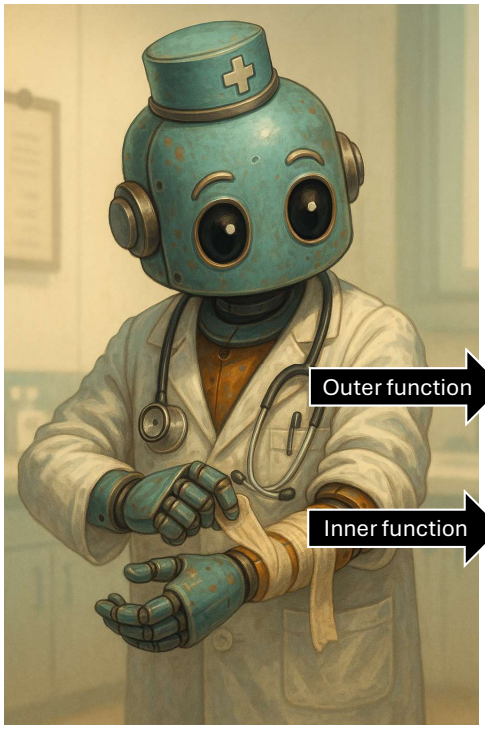


Probabilistic Steps



- Not all tasks have deterministic outcomes.
- Input data can lead to different results, depending on the values provided.
- Nondeterministic methods (e.g., LLMs) will produce slightly different results each time they are called.
- This behaviour can be simulated by introducing a random element.

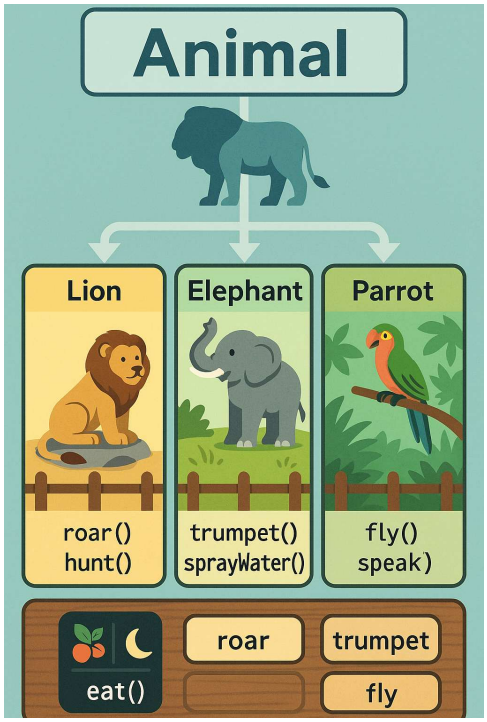
```
def make_appender(letter: str, probability: float = 0.7)
-> Callable[[str], str]:
    """
Returns a function that will append the letter at the
end of a string with a certain probability.
"""
    alternatives =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ".r
eplace(letter.lower(), "")
    def appender(s: str) -> p:
        # Dice roll
        if random.random() < probability:
            return s + letter
        else:
            return s + random.choice(alternatives)
    return appender
```



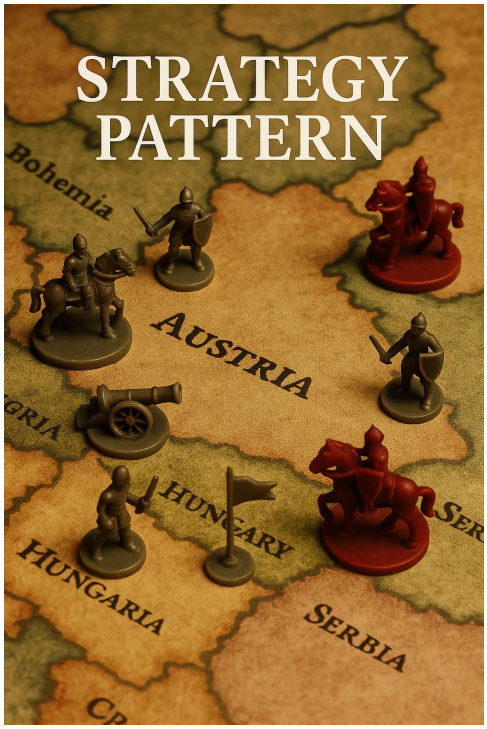
Self healing loop

- If a task occasionally produces incorrect output, the simplest fix can be to try again until it succeeds.
- Sometimes, though, you also need to tweak certain parameters with each attempt; otherwise the result will stay the same and repeating the task would be pointless.

```
def repeat_until_condition_met(func: Callable[[str], str], condition: Callable[[str], bool]) -> Callable[[str], str]:
    """
    Returns a function that calls 'func' until the condition 'condition' is met.
    """
    def wrapper(s: str) -> p:
        result = func(s)
        while not condition(result):
            print(f"Condition not met, repeating: {result}")
            result = func(s)
        return result
    return wrapper
```

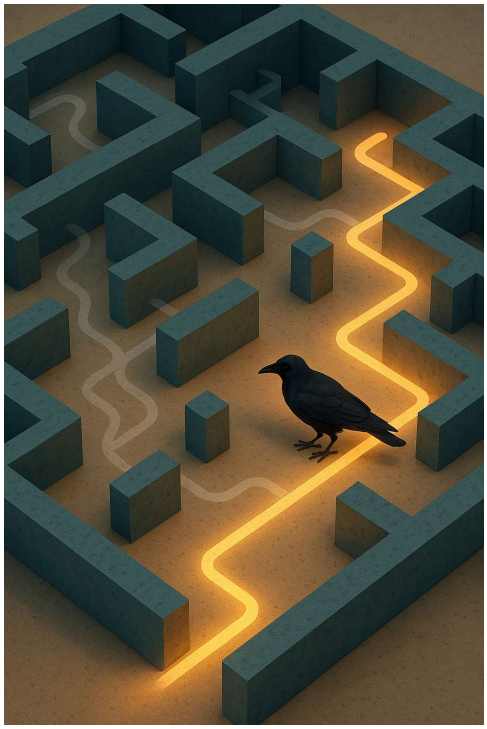


- **Instances of Classes** – Each object (e.g., ToLower ()) is a concrete instance of a class, bundling its specific data and the process() behavior it implements.
- **Encapsulation of State and Behavior** – Encapsulate the algorithm's code inside their process(input: str) → str method, hiding implementation details from the client.
- **Polymorphic Interaction** – The Pipeline object holds a list of PipelineStep objects and invokes each one via the same process() interface, regardless of the concrete class.
- **Collaborating Objects** – The Pipeline instance (the context) and the various PipelineStep instances collaborate: the pipeline delegates work to each step object rather than containing the algorithms itself.
- **Dynamic Composition** – Client code creates and assembles objects (e.g., steps = [ToLower(), ReverseString(), …]) at runtime, showing how objects can be composed and reconfigured without changing their classes.

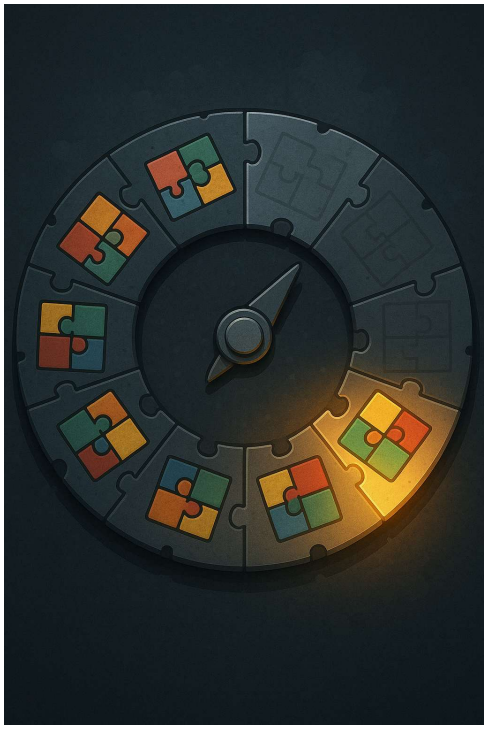


- **Abstract Strategy**: The PipelineStep abstract base class defines the process method.
- **Concrete Strategies**: Classes like ToLower each override process() in a specific way.
- **Context**: The Pipeline class holds a list of PipelineStep instances and delegates each input string to them.
- **Algorithm Encapsulation**:
  - No large conditional Statements
  - Self contained algorithmns
- **Open/Closed Principle**: Adding a new transformation makes no changes to existing pipeline or client code.
- **Runtime Flexibility**: Steps in Pipeline can be changed during runtime..

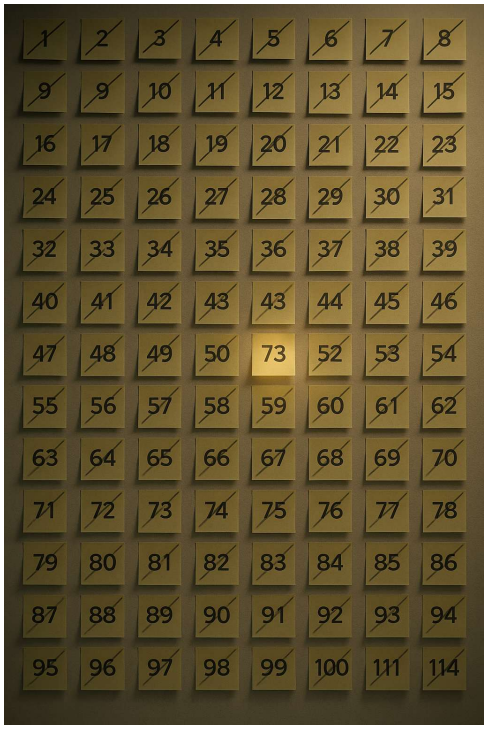
```
process(input: str) → str
```



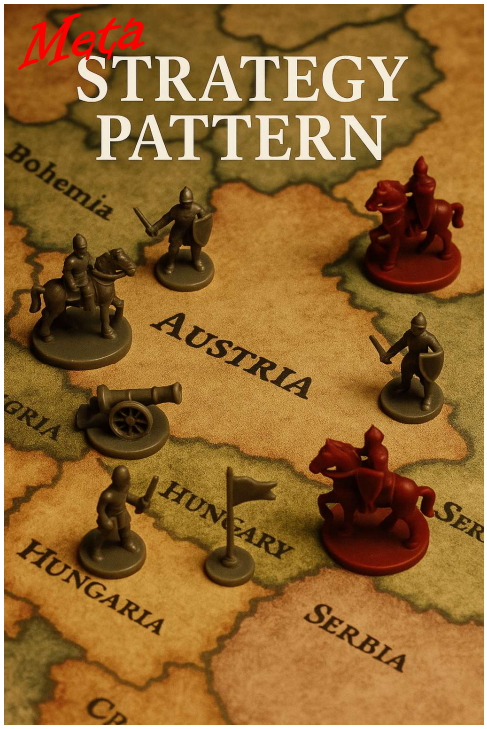
- **Shuffle-Based Exploration**: In train_chain(), the pipeline repeatedly calls random.shuffle on the *original* step list to explore different execution orders .
- **Brute-Force Permutation Search**: Each shuffled sequence is tested against the target until the correct transformation order is found .
- **Isolation of Attempts**: Every trial uses fresh shuffles of the *original* step list, ensuring no cumulative side-effects influence future attempts .
- **Logging Feedback**: Each attempt logs its sequence number and the resulting string, providing visibility into the convergence process .
- **Dynamic Strategy Update**: Once a successful permutation is discovered, self.steps is updated to that winning order, fixing the learned strategy for subsequent runs .



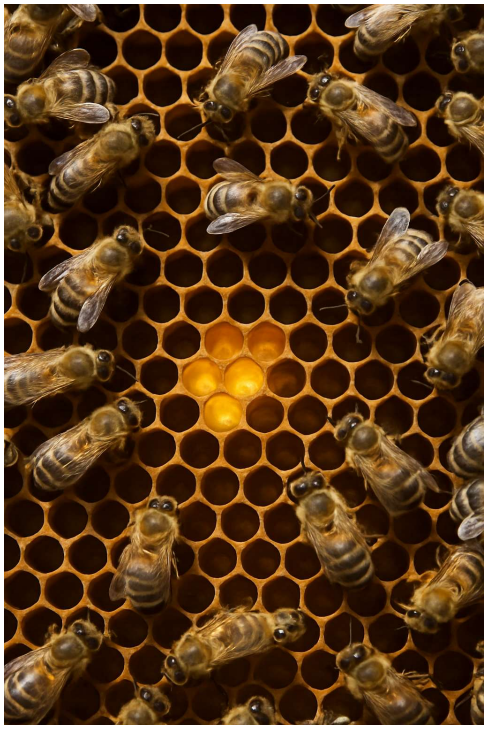
- **Random Permutation Generation**: Inside an infinite loop, random.shuffle creates a new ordering of the step signatures each iteration .
- **Signature-Based Deduplication**: A tested set of signature tuples prevents retesting identical sequences, reducing redundant work .
- **Early Exit on Success**: The loop breaks as soon as the shuffled order produces the target output, minimizing wasted attempts .
- **Fine-Grained Logging**: Each attempt prints both the intermediate string result and the specific signature used, making each trial's behavior traceable .
- **Probabilistic Convergence**: Success depends on chance orderings; the algorithm may find a solution quickly or require many random trials, modeling a learning process with unpredictable convergence time.



- **Systematic Permutation Enumeration**: Uses itertools.permutations to generate every possible ordering of the steps exactly once .
- **No Redundancy Handling Needed**: Because permutations are inherently unique, there's no need for a separate deduplication structure .
- **Guaranteed Termination**: The search will always find the correct sequence within at most *n!* attempts, providing a deterministic bound on run time .
- **Concise Progress Logging**: Each attempt logs only the attempt count and resulting string, offering a clear, minimal trace of the exhaustive search .
- **Reproducible Strategy Discovery**: Because the search order is fixed, the same run will always test permutations in the same sequence and find the solution after the same number of steps.



- **Strategies as First-Class Steps**: Wrappers like RepeatUntilConditionMet and DoubleRunMethod implement PipelineStep. Learning policies are just like any other transformation.
- **Probabilistic Append**: UnreliableMakeAppender injects randomness into process().
- **Looping Wrapper**: RepeatUntilConditionMet repeatedly calls its inner step on the original input until the success predicate holds, encapsulating the retry logic.
- **Decorator for Insight**: DoubleRunMethod runs a step twice in a row and logs both outcomes, letting you observe the learning variability side by side.
- **Composable Learning Pipeline**: Meta-strategies are plugged into Pipeline. Transformations and learning behaviors in one cohesive sequence.

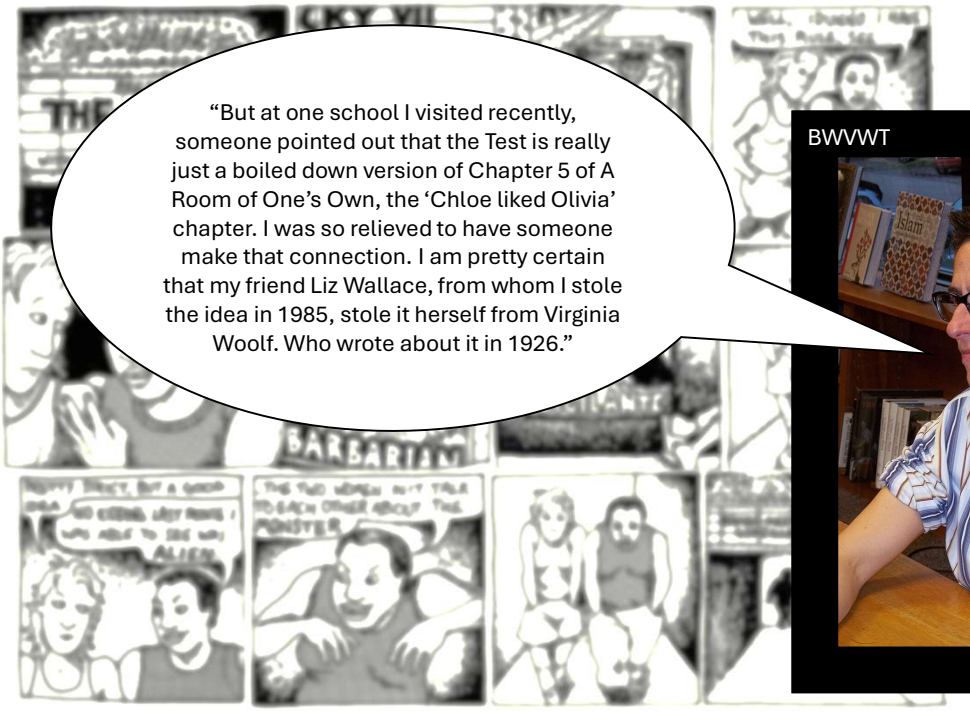


- **Pipelines are similiar to Taks** – a task and a pipeline take a string and return a string.
  Pipelines *are* ~~similar to~~ **Tasks.**
- **Nested Pipelines as Steps** –Pipeline instance (pipeline_Double_e) can itself included in the top-level steps list.
  Hierarchical Composition – The top-level pipeline composes simple steps (DoubleLastChar, LastBecomesFirst, etc.) alongside nested pipelines.
- **Chained Execution with Sub-Pipelines** – When run_chained() executes, it delegates into the nested pipeline at that point.
- **Complexity** – Objects store states and can provide multiple methods. E.g. Visualization.
- **Uniform Interface** – Both individual strategy steps and entire pipelines share the PipelineStep interface (process() and __repr__()), ensuring clients and visualizers treat them interchangeably.
- **Graphical Visualization** – Nested execution works for display.

BWT

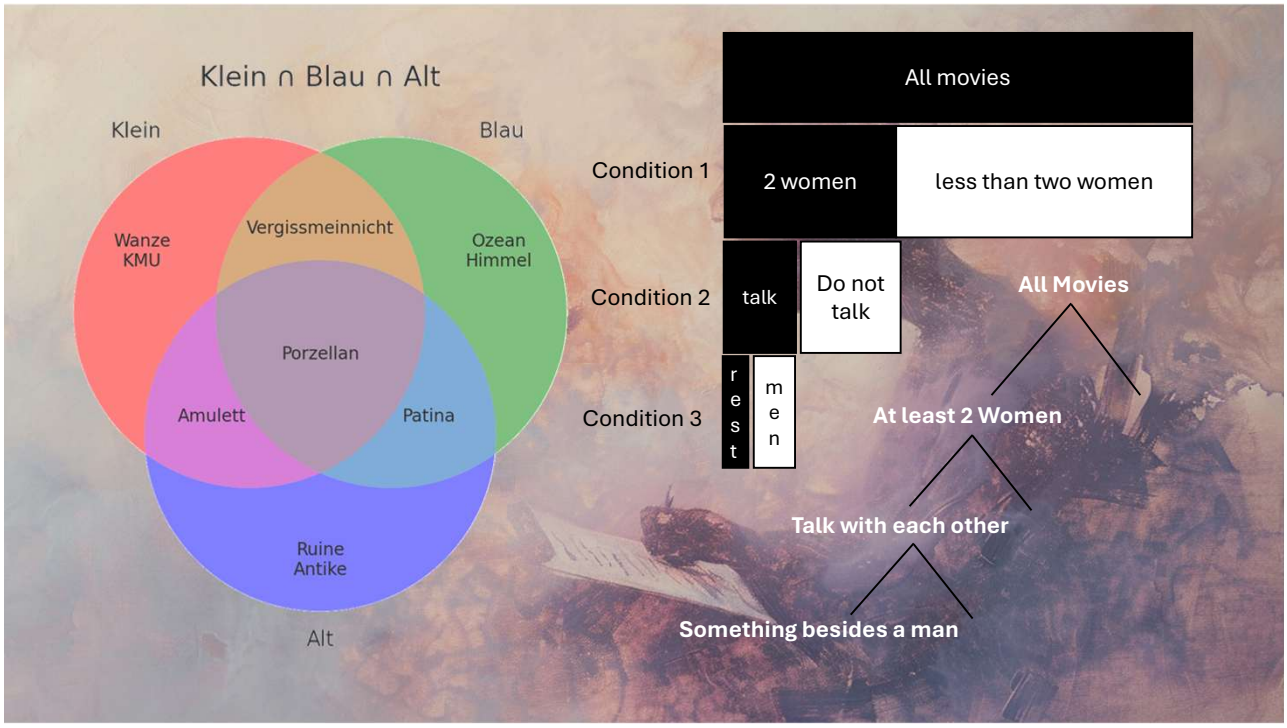
1. **Two (named) women**
2. **Talk with each other**
3. **About something besides a man**

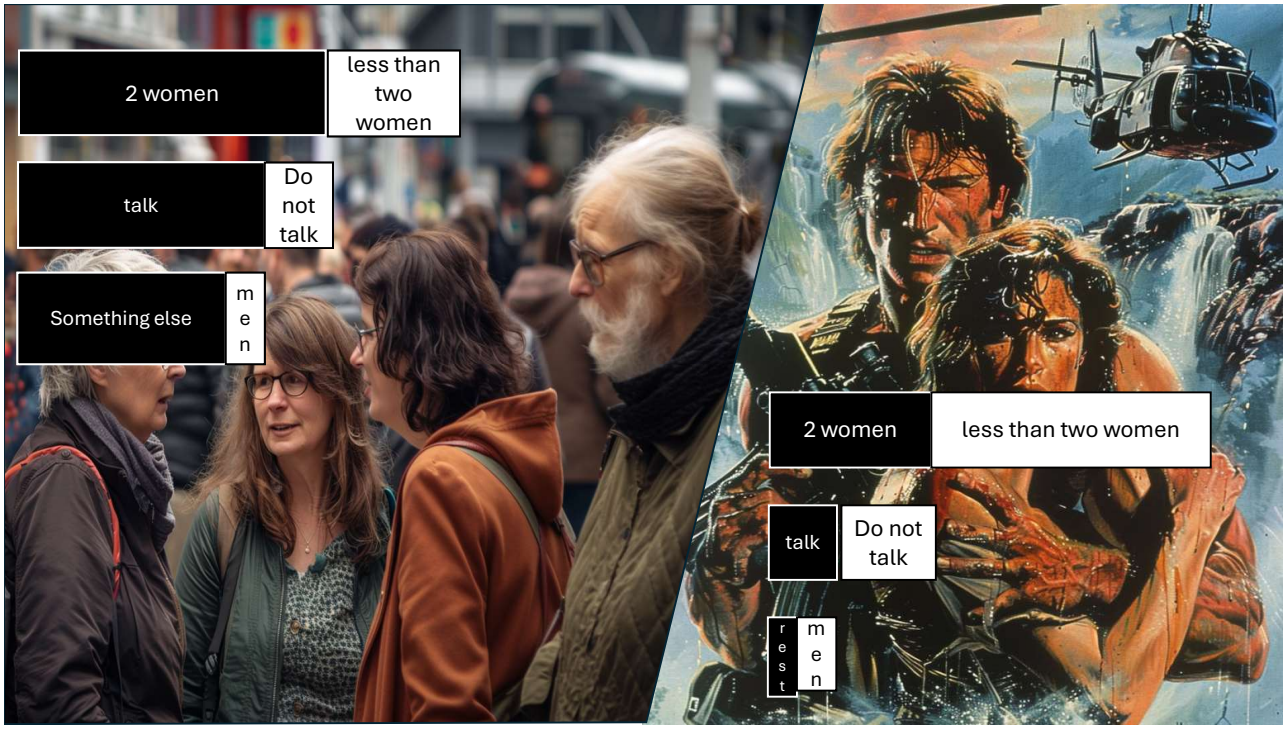
Woman do not have meaningful conversations (in movies)



"But at one school I visited recently, someone pointed out that the Test is really just a boiled down version of Chapter 5 of A Room of One's Own, the 'Chloe liked Olivia' chapter. I was so relieved to have someone make that connection. I am pretty certain that my friend Liz Wallace, from whom I stole the idea in 1985, stole it herself from Virginia Woolf. Who wrote about it in 1926."

BWVWT


Klein ∩ Blau ∩ Alt
Klein
Blau
Alt
Wanze
KMU
Vergissmeinnicht
Ozean
Himmel
Porzellan
Amulett
Patina
Ruine
Antike
All movies
Condition 1
2 women
less than two women
Condition 2
talk
Do not talk
Condition 3
rest
men
All Movies
At least 2 Women
Talk with each other
Something besides a man


2 women
less than two women
talk
Do not talk
Something else
men
2 women
less than two women
talk
Do not talk
rest
men

Double Keying

- Bearbeiter:innen machen unabhängig voneinander Fehler.
- Fehler sind selten.
- Die Wahrscheinlichkeit, dass beide Bearbeiter:innen an gleicher Stelle einen Fehler machen ist verschwindend gering

- **Fehlerwahrscheinlichkeit pro Bearbeiter:** p
- **Übereinstimmung**

$$P_{\text{agree}} = (1-p)^2 + p^2$$

- **Uneinigkeit (Prüffall)**

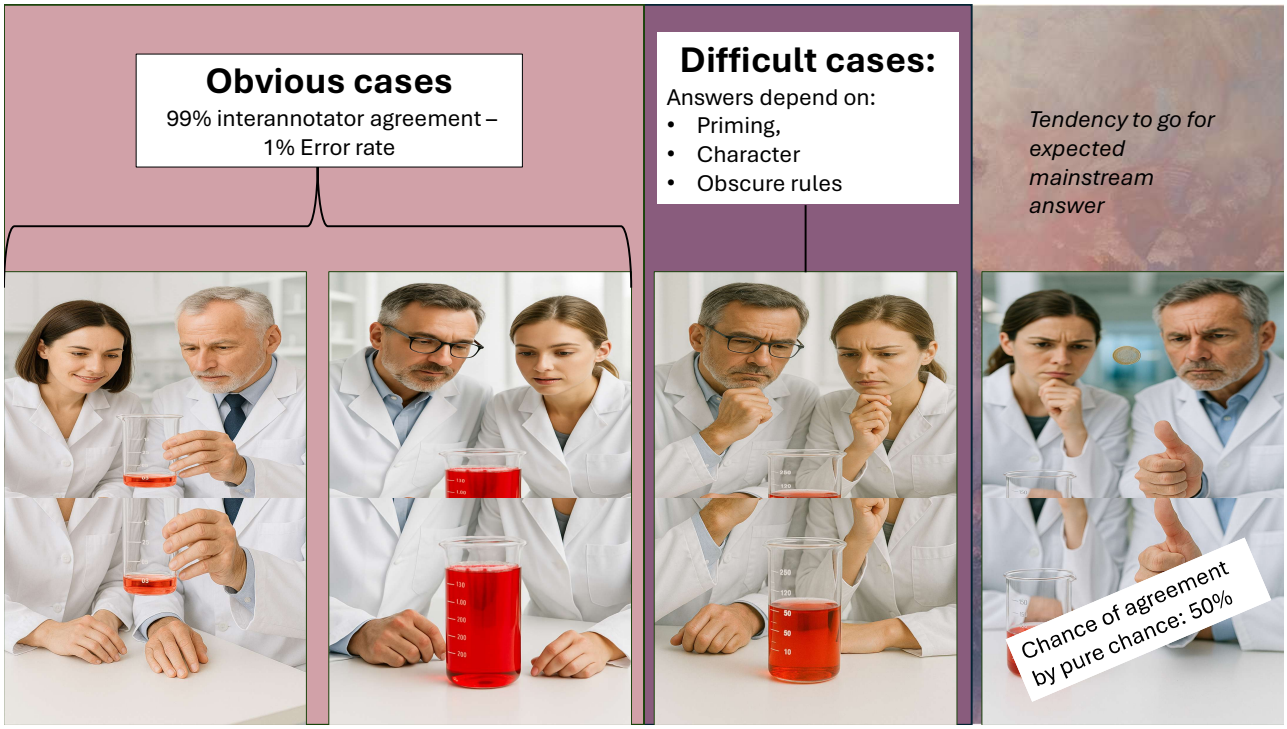
$$P_{\text{disagree}} = 2\,p\,(1-p)$$

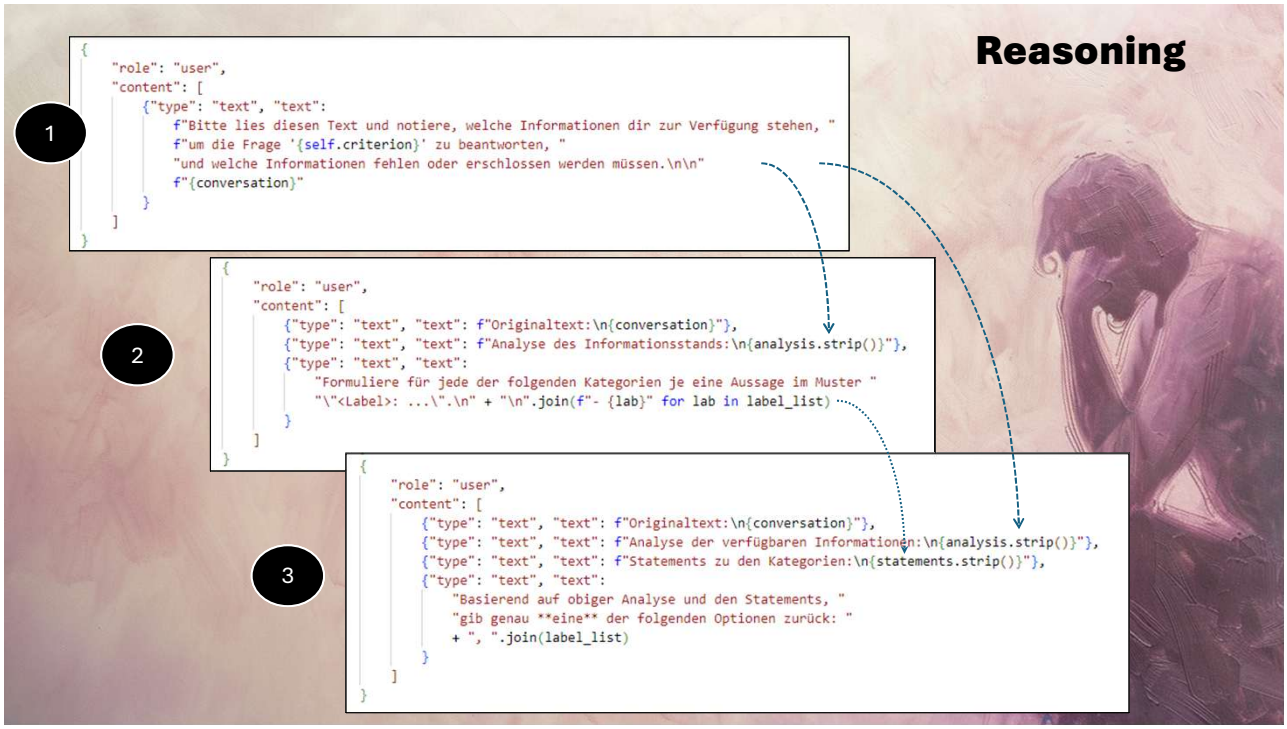
- **Korrektheits-Wahrscheinlichkeit bei Übereinstimmung**

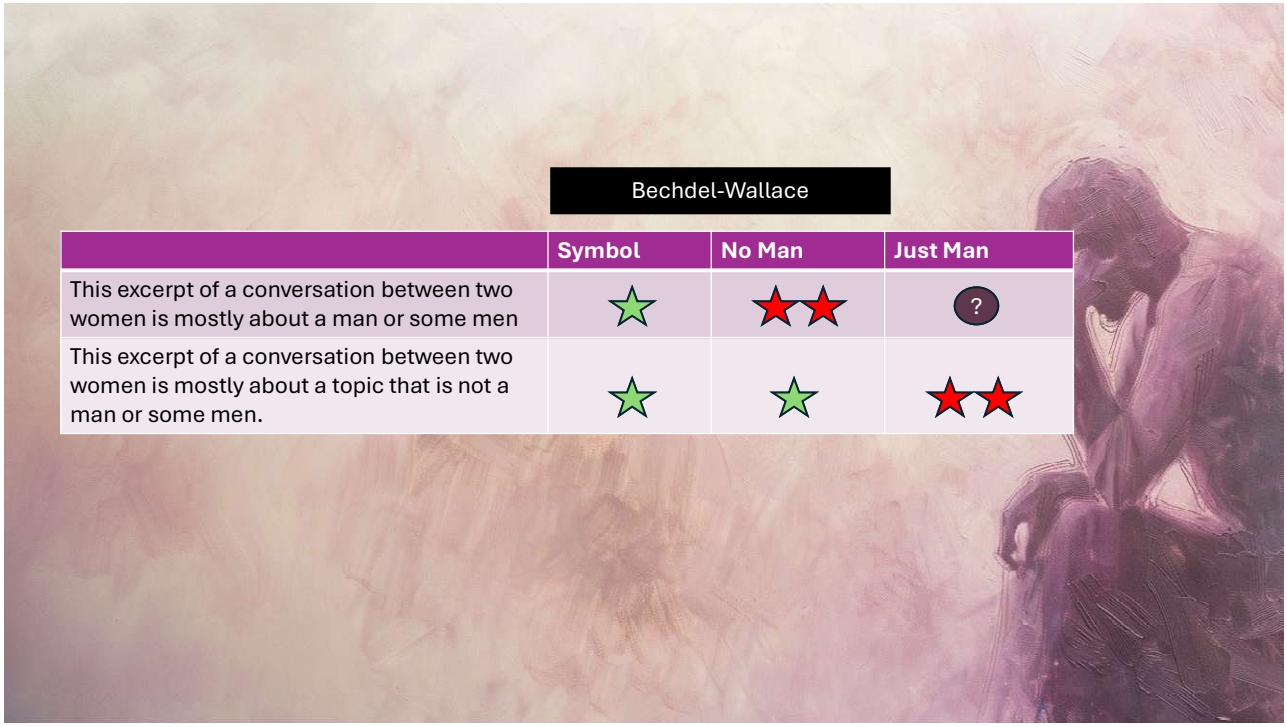
$$P(\text{korrekt} \mid \text{agree}) = \frac{(1-p)^2}{(1-p)^2 + p^2}$$

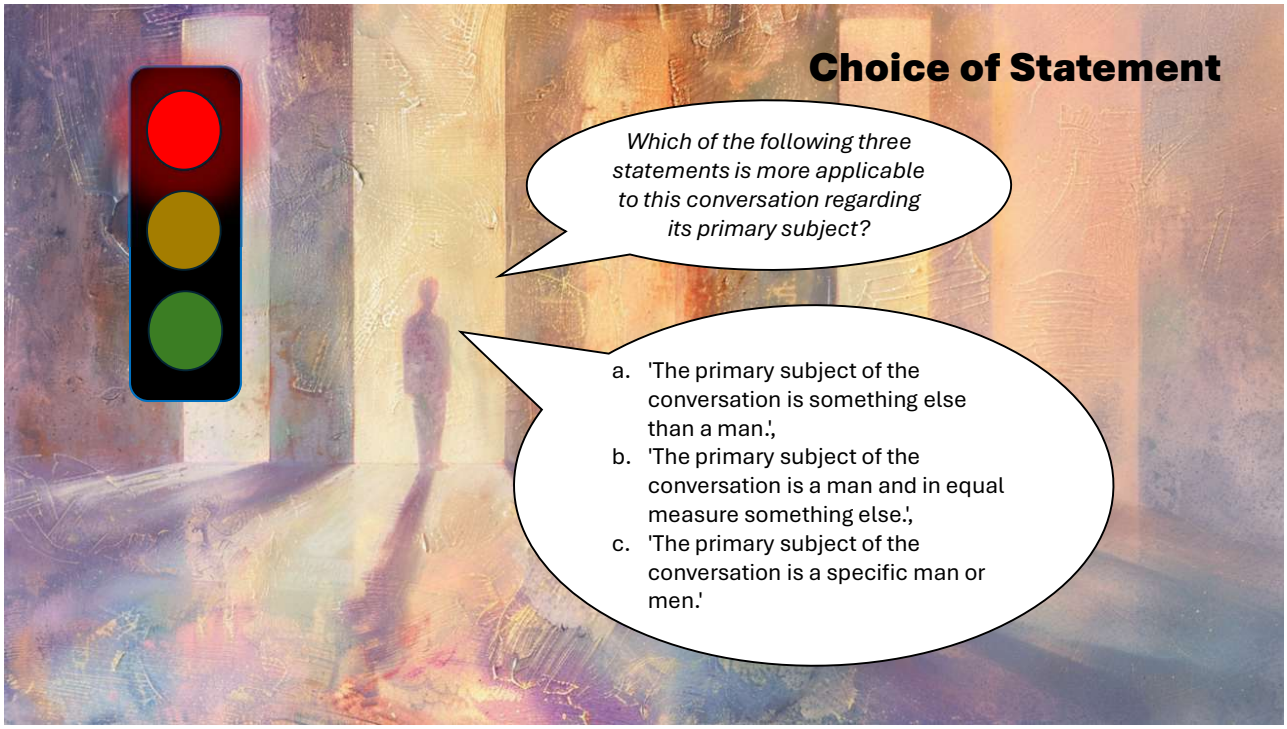
- **Beispiel** $p = 0{,}1$:

$$P_{\text{agree}} = 0{,}9^2 + 0{,}1^2 = 82\%, \quad P(\text{korrekt} \mid \text{agree}) \approx 98{,}8\%.$$


Obvious cases
99% interannotator agreement –
1% Error rate
Difficult cases:
Answers depend on:
• Priming,
• Character
• Obscure rules
Tendency to go for expected mainstream answer
Chance of agreement by pure chance: 50%


Reasoning
1
2
3
{
"role": "user",
"content": [
{"type": "text", "text":
f"Bitte lies diesen Text und notiere, welche Informationen dir zur Verfügung stehen, "
f"um die Frage '{self.criterion}' zu beantworten, "
"und welche Informationen fehlen oder erschlossen werden müssen.\n\n"
f"{conversation}"
}
]
}
{
"role": "user",
"content": [
{"type": "text", "text": f"Originaltext:\n{conversation}"},
{"type": "text", "text": f"Analyse des Informationsstands:\n{analysis.strip()}"},
{"type": "text", "text":
"Formuliere für jede der folgenden Kategorien je eine Aussage im Muster "
"\"<Label>: ...\".\n" + "\n".join(f"- {lab}" for lab in label_list)
}
]
}
{
"role": "user",
"content": [
{"type": "text", "text": f"Originaltext:\n{conversation}"},
{"type": "text", "text": f"Analyse der verfügbaren Informationen:\n{analysis.strip()}"},
{"type": "text", "text": f"Statements zu den Kategorien:\n{statements.strip()}"},
{"type": "text", "text":
"Basierend auf obiger Analyse und den Statements, "
"gib genau **eine** der folgenden Optionen zurück: "
+ ", ".join(label_list)
}
]
}


Bechdel-Wallace
Symbol
No Man
Just Man
This excerpt of a conversation between two women is mostly about a man or some men
?
This excerpt of a conversation between two women is mostly about a topic that is not a man or some men.


Choice of Statement
Which of the following three statements is more applicable to this conversation regarding its primary subject?
a. 'The primary subject of the conversation is something else than a man.',
b. 'The primary subject of the conversation is a man and in equal measure something else.',
c. 'The primary subject of the conversation is a specific man or men.'

Rank texts

(if your context window is large enough)

Which of the texts fullfills the statement best?
Sort in descending order and return only the letters associated with each text.

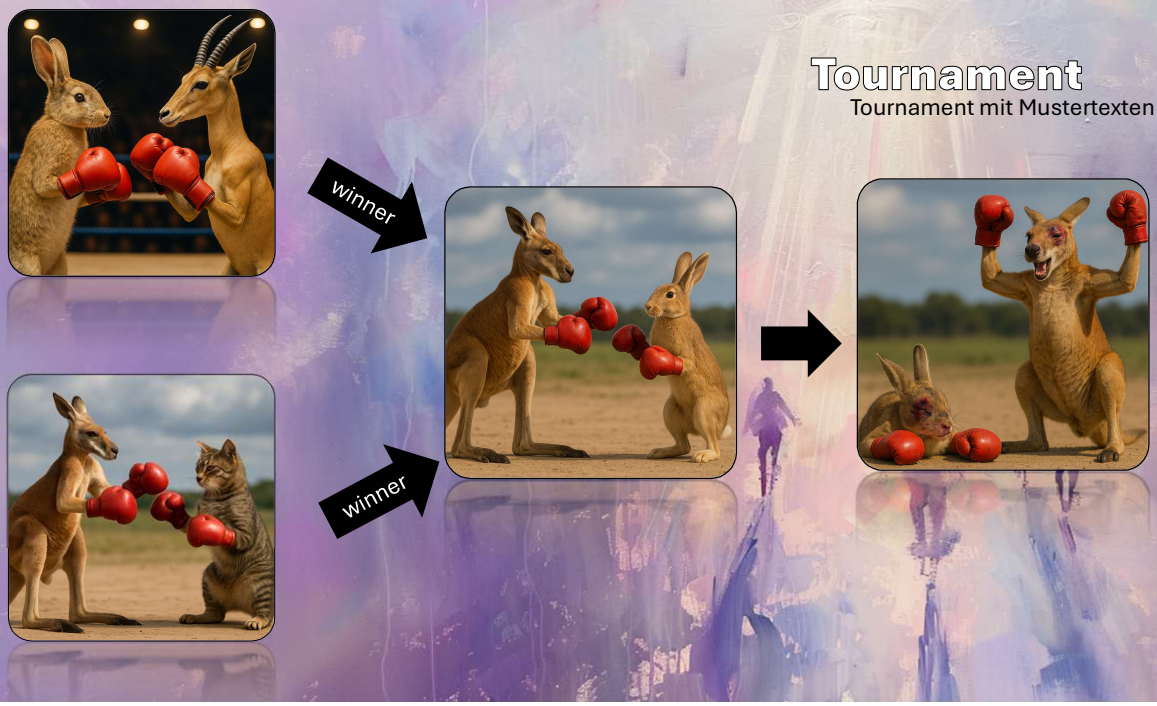
Real text

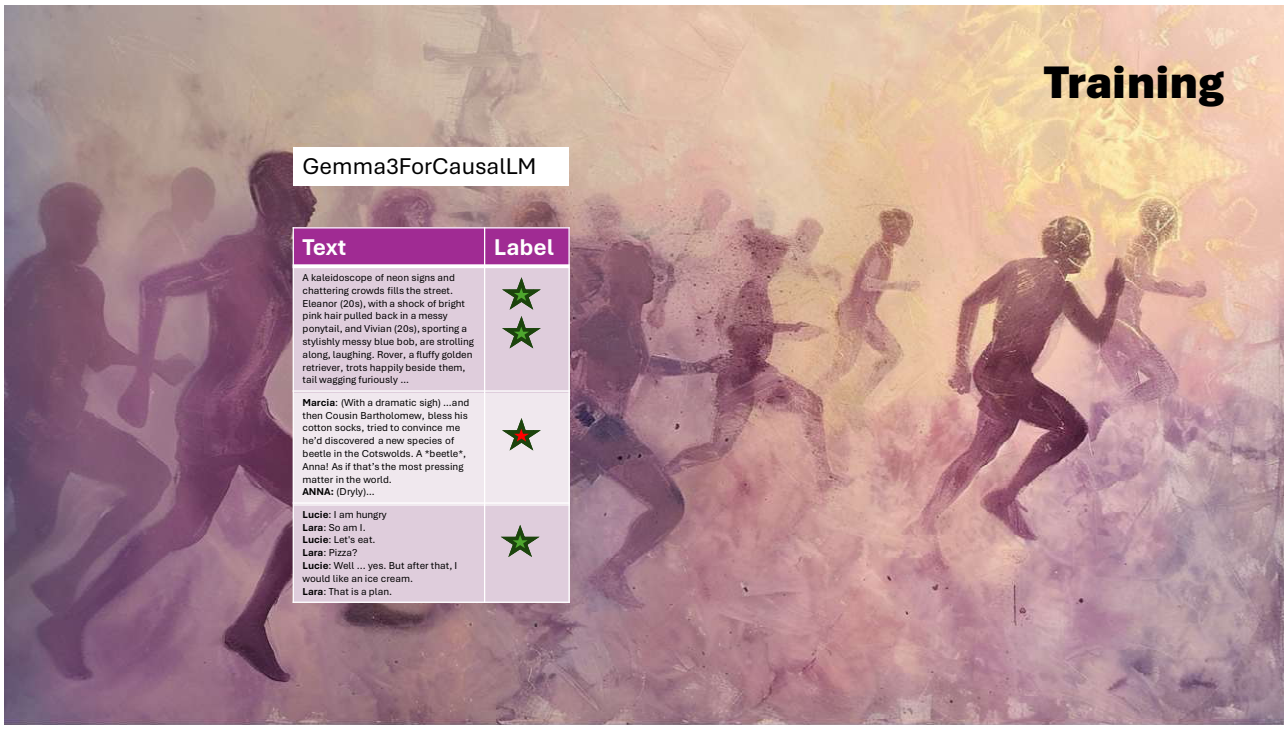
exemplary no

exemplary yes

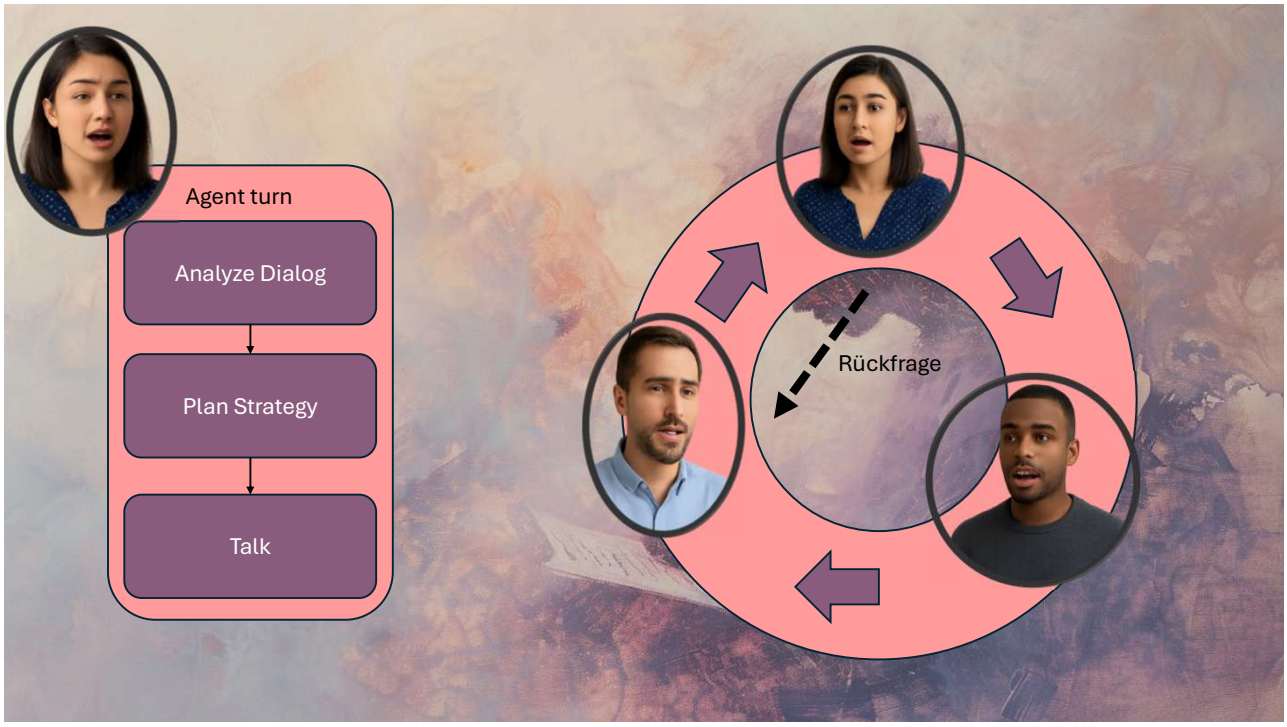
Tournament

Tournament mit Mustertexten




Training
Gemma3ForCausalLM
Text
Label
A kaleidoscope of neon signs and chattering crowds fills the street. Eleanor (20s), with a shock of bright pink hair pulled back in a messy ponytail, and Vivian (20s), sporting a stylishly messy blue bob, are strolling along, laughing. Rover, a fluffy golden retriever, trots happily beside them, tail wagging furiously ...
Marcia: (With a dramatic sigh) ...and then Cousin Bartholomew, bless his cotton socks, tried to convince me he'd discovered a new species of beetle in the Cotswolds. A *beetle*, Anna! As if that's the most pressing matter in the world.
ANNA: (Dryly)...
Lucie: I am hungry
Lara: So am I.
Lucie: Let's eat.
Lara: Pizza?
Lucie: Well ... yes. But after that, I would like an ice cream.
Lara: That is a plan.

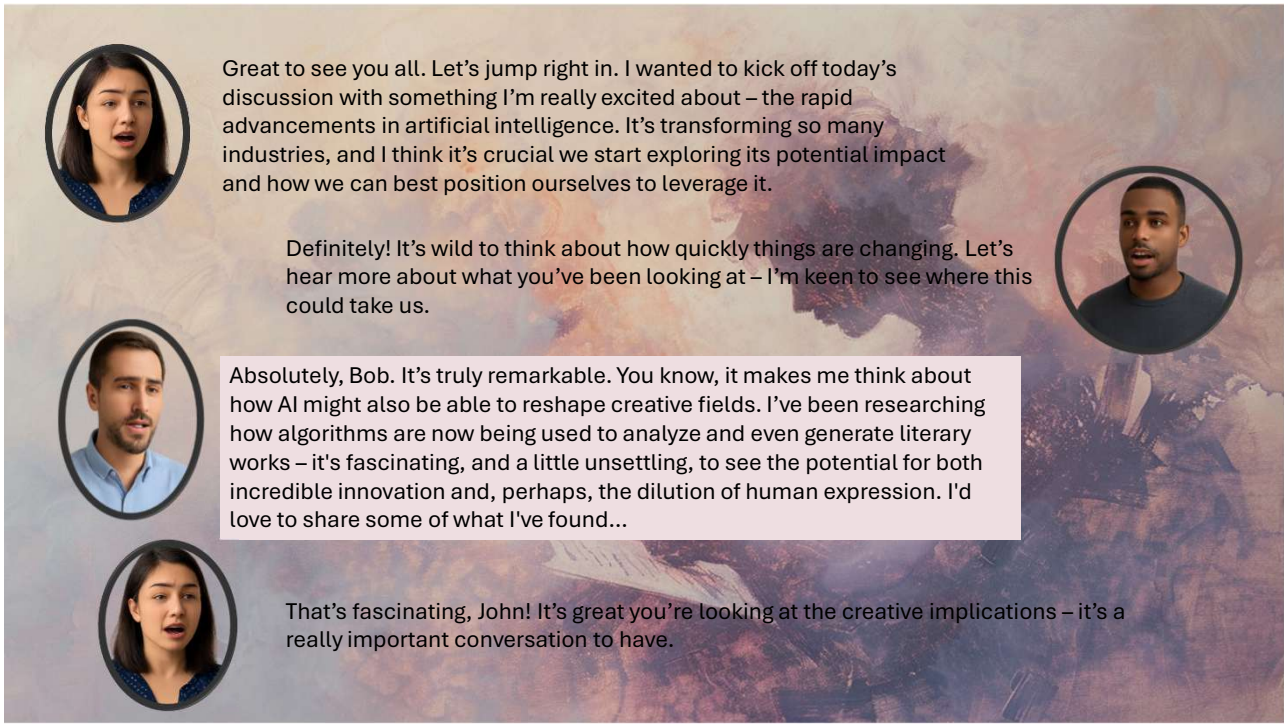

Agents

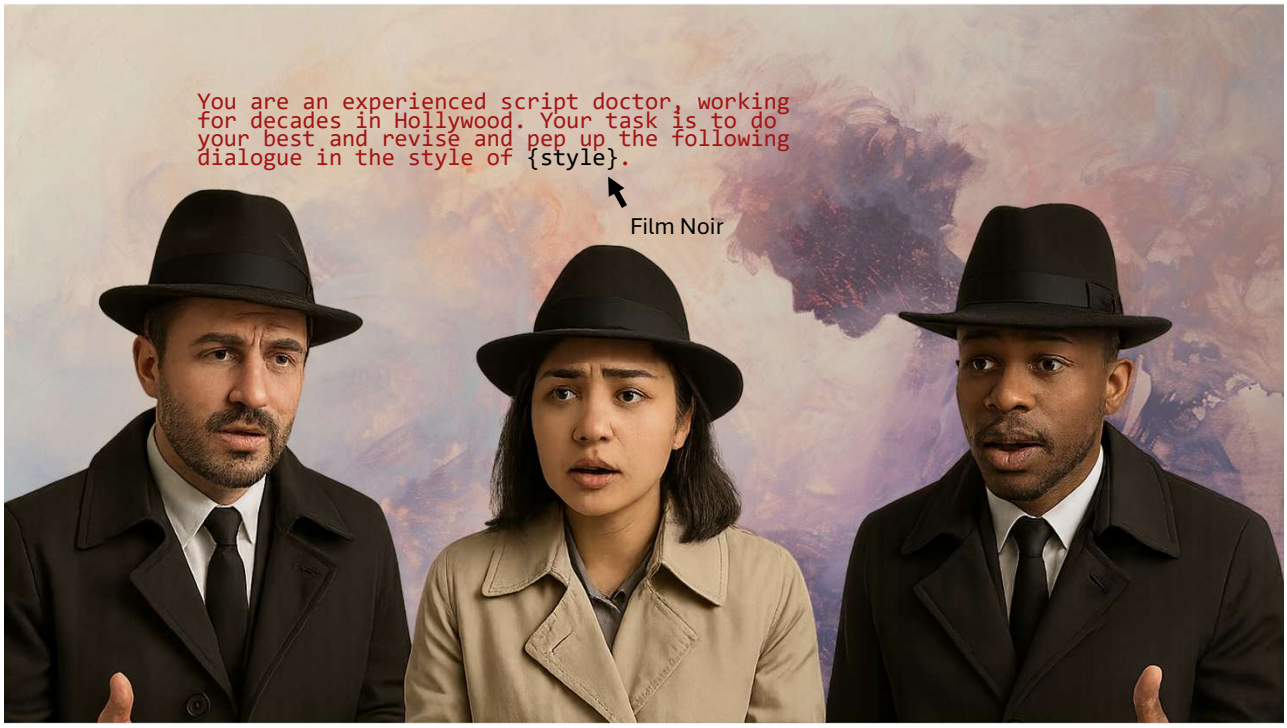



```
agent1 = Agent(
    name="Alice",
    topics=["technology", "artificial intelligence", "robotics"],
    role_desc="Alice ist technische Expertin mit einer Leidenschaft für Innovation.",
    special_actions={"summary": 0.1, "probe": 0.05}
)
```

Goals:
- Talk about all topics
- Don‘t be rude

- Greet
- Support last Talker
- Change Topic
- Continue Topic
- End Topic
- Start Topic
- Summarize previous Talk (Special)
- Address chosen Agent with question (Special)
- End Talk


Great to see you all. Let's jump right in. I wanted to kick off today's discussion with something I'm really excited about – the rapid advancements in artificial intelligence. It's transforming so many industries, and I think it's crucial we start exploring its potential impact and how we can best position ourselves to leverage it.
Definitely! It's wild to think about how quickly things are changing. Let's hear more about what you've been looking at – I'm keen to see where this could take us.
Absolutely, Bob. It's truly remarkable. You know, it makes me think about how AI might also be able to reshape creative fields. I've been researching how algorithms are now being used to analyze and even generate literary works – it's fascinating, and a little unsettling, to see the potential for both incredible innovation and, perhaps, the dilution of human expression. I'd love to share some of what I've found...
That's fascinating, John! It's great you're looking at the creative implications – it's a really important conversation to have.


You are an experienced script doctor, working for decades in Hollywood. Your task is to do your best and revise and pep up the following dialogue in the style of {style}.
Film Noir

I wanted to lay out something... significant. Artificial intelligence. It's not just a trend, it's a shift. A tectonic one. We need to understand how to use it, *control* it."

(Swallowing hard)
It's... unsettling, isn't it? Like looking into a mirror that shows you a future you don't want.

(Turning his gaze towards the window)

It's more than unsettling, Bob. It's a mimicry. They're feeding algorithms with *our* stories, *our* art... and then spitting back a pale imitation. I've been digging into literary analysis – these programs are already crafting prose. It's... efficient. And terrifying.

He pauses, a shadow crossing his face.

Like a forgery, but one that can convince you it's the original.

(Her voice dropping slightly) Precisely. We need to be ahead of the curve. A strategic advantage.

Good morrow to you all! 'Tis true, I muse,
That art itself may yield to cunning use.
These algorithms, weaving word and hue,
Do mimic minds, and strive to imitate you.
A potent grace, yet fraught with subtle dread,
To see our craft by cold mechanics led.
A shadowed beauty, born of logic's sway,
Where human touch may slowly fade away.
I have explored this labyrinthine quest,
And bring to light a truth, to be confessed.