Building a simple AI-Agent, step by step 251113

AI-Agent, step by step

https://www.coursera.org/learn/ai-agents-python/ungradedWidget/YRFHI/building-your-first-agent

Die Agentenschleife in Python

Die **Agentenschleife ist das Rückgrat unseres KI-Agenten** und ermöglicht ihm die Ausführung von Aufgaben, indem sie die Generierung von Antworten, die Ausführung von Aktionen und Speicheraktualisierungen in einem iterativen Prozess kombiniert. Dieser Abschnitt konzentriert sich darauf, wie die Agentenschleife funktioniert und welche Rolle sie dabei spielt, den Agenten dynamisch und anpassungsfähig zu machen.

1. Prompt erstellen: Kombinieren Sie den Speicher (assistant) des Agenten, die Benutzereingaben (user) und die Systemregeln (system) zu einem einzigen Prompt. Dadurch wird sichergestellt, dass das LLM über den gesamten Kontext verfügt, den es benötigt, um über die nächste Aktion zu entscheiden, und die Kontinuität über alle Iterationen hinweg aufrechterhalten wird.
2. Antwort generieren: Senden Sie den erstellten Prompt an das LLM und rufen Sie eine Antwort ab. Diese Antwort leitet den nächsten Schritt des Agenten, indem sie Anweisungen in einem strukturierten Format bereitstellt. - generate_response(messages)
3. Antwort analysieren: Extrahieren Sie die beabsichtigte Aktion und ihre Parameter aus der Ausgabe des LLM. Die Antwort muss einer vordefinierten Struktur (z. B. **JSON-Format**) entsprechen, um sicherzustellen, dass sie korrekt interpretiert werden kann.
4. Aktion ausführen: Verwenden Sie die extrahierte Aktion und ihre Parameter, um die angeforderte Aufgabe mit dem entsprechenden Tool auszuführen. Dies kann das Auflisten von Dateien, das Lesen von Inhalten oder das Drucken einer Nachricht umfassen.
5. Ergebnis in Zeichenfolge konvertieren: Formatieren Sie das Ergebnis der ausgeführten Aktion in eine Zeichenfolge. Auf diese Weise kann der Agent das Ergebnis in seinem Speicher ablegen und dem Benutzer oder sich selbst ein klares Feedback geben.
6. Schleife fortsetzen? Bewerten Sie anhand der aktuellen Aktion und der Ergebnisse, ob die Schleife fortgesetzt werden soll. Die Schleife kann beendet werden, wenn eine „Beenden"-Aktion angegeben ist oder wenn der Agent die Aufgabe abgeschlossen hat.

Der Agent durchläuft diese Schleife wiederholt, verfeinert sein Verhalten und passt seine Aktionen an, bis er eine Stoppbedingung erreicht. Dieser Prozess ermöglicht es dem Agenten, dynamisch zu interagieren und intelligent auf Aufgaben zu reagieren.

Diese Datei zeigt die obigen Schritte in Python auf:
C:\Users\PhilippSauber\Dropbox\Next_2019\Dokumente\Know-how\Coursera-edX\AI-Agents_2510\Agents\agent-one.py

Building a Simple AI Agent, Part 1

Die Agentenschleife in Python

Die Agentenschleife ist das Rückgrat unseres KI-Agenten und ermöglicht ihm die Ausführung von Aufgaben, indem sie die Generierung von Antworten, die Ausführung von Aktionen und Speicheraktualisierungen in einem iterativen Prozess kombiniert. Dieser Abschnitt konzentriert sich darauf, wie die Agentenschleife funktioniert und welche Rolle sie dabei spielt, den Agenten dynamisch und anpassungsfähig zu machen.

1. Prompt erstellen: Kombinieren Sie den Speicher (assistant) des Agenten, die Benutzereingaben (user) und die Systemregeln (system) zu einem einzigen Prompt. Dadurch wird sichergestellt, dass das LLM über den gesamten Kontext verfügt, den es benötigt, um über die nächste Aktion zu entscheiden, und die Kontinuität über alle Iterationen hinweg aufrechterhalten wird.
2. Antwort generieren: Senden Sie den erstellten Prompt an das LLM und rufen Sie eine Antwort ab. Diese Antwort leitet den nächsten Schritt des Agenten, indem sie Anweisungen in einem strukturierten Format bereitstellt. - generate_response(messages)
3. Antwort analysieren: Extrahieren Sie die beabsichtigte Aktion und ihre Parameter aus der Ausgabe des LLM. Die Antwort muss einer vordefinierten Struktur (z. B. JSON-Format) entsprechen, um sicherzustellen, dass sie korrekt interpretiert werden kann.
4. Aktion ausführen: Verwenden Sie die extrahierte Aktion und ihre Parameter, um die angeforderte Aufgabe mit dem entsprechenden Tool auszuführen. Dies kann das Auflisten von Dateien, das Lesen von Inhalten oder das Drucken einer Nachricht umfassen.
5. Ergebnis in Zeichenfolge konvertieren: Formatieren Sie das Ergebnis der ausgeführten Aktion in eine Zeichenfolge. Auf diese Weise kann der Agent das Ergebnis in seinem Speicher ablegen und dem Benutzer oder sich selbst ein klares Feedback geben.
6. Schleife fortsetzen? Bewerten Sie anhand der aktuellen Aktion und der Ergebnisse, ob die Schleife fortgesetzt werden soll. Die Schleife kann beendet werden, wenn eine „Beenden"-Aktion angegeben ist oder wenn der Agent die Aufgabe abgeschlossen hat.

Der Agent durchläuft diese Schleife wiederholt, verfeinert sein Verhalten und passt seine Aktionen an, bis er eine Stoppbedingung erreicht. Dieser Prozess ermöglicht es dem Agenten, dynamisch zu interagieren und intelligent auf Aufgaben zu reagieren.

Diese Datei zeigt die obigen Schritte in Python auf:
C:\Users\PhilippSauber\Dropbox\Next_2019\Dokumente\Know-how\Coursera-edX\AI-Agents_2510\Agents\agent-one.py

Agents > agent-one.py > ...

```python
# 🔁 Der "Agent Loop" – Hauptschleife des AI-Agenten
# Diese Schleife wird wiederholt ausgeführt, bis eine Abbruchbedingung erreicht ist.
# Sie simuliert, wie ein KI-Agent denkt, handelt und sein Gedächtnis aktualisiert.

while iterations < max_iterations:

    # 1️⃣ Prompt erstellen: Agenten-Regeln + bisheriges Gedächtnis kombinieren.
    # 'agent_rules' enthält die fixen Verhaltensregeln (z. B. "Du bist ein hilfreicher Assistent"),
    # 'memory' enthält den bisherigen Gesprächsverlauf oder Kontext.
    prompt = agent_rules + memory

    # 2️⃣ Antwort generieren / vom Sprachmodell (LLM) erzeugen
    print("Agent thinking...")  # Statusausgabe für den Benutzer
    response = generate_response(prompt)  # Anfrage an das LLM (Large Language Model)
    print(f"Agent response: {response}")  # Ausgabe der Antwort (z. B. welche Aktion es vorschlägt)

    # 3️⃣ Antwort analysieren, um herauszufinden, welche Aktion ausgeführt werden soll
    # 'parse_action' ist eine Hilfsfunktion, die den Text der LLM-Antwort
    # in eine strukturierte Aktion (Dictionary) umwandelt.
    action = parse_action(response)

    # Variable zum Speichern des Aktionsergebnisses
    result = "Action executed"

    # 4️⃣ Aktion ausführen: Mögliche Aktionen prüfen und ausführen:
    # Das LLM entscheidet, welches "Tool" (Werkzeug/Funktion) der Agent verwenden soll.

    # Wenn das Modell "list_files" auswählt → alle Dateien im aktuellen Verzeichnis auflisten.
    if action["tool_name"] == "list_files":
        result = {"result": list_files()}

    # Wenn "read_file" → Dateiinhalt lesen, Dateiname wird aus den Argumenten geholt.
    elif action["tool_name"] == "read_file":
        result = {"result": read_file(action["args"]["file_name"])}

    # Wenn ein Fehler erkannt wurde → Fehlermeldung zurückgeben.
    elif action["tool_name"] == "error":
        result = {"error": action["args"]["message"]}

    # Wenn das Modell signalisiert, dass der Agent seine Arbeit beenden soll → Schleife abbrechen.
    elif action["tool_name"] == "terminate":
        print(action["args"]["message"])  # Nachricht anzeigen (z. B. "Task completed.")
        break

    # Wenn der Aktionsname unbekannt ist → Fehler melden.
    else:
        result = {"error": "Unknown action: " + action["tool_name"]}

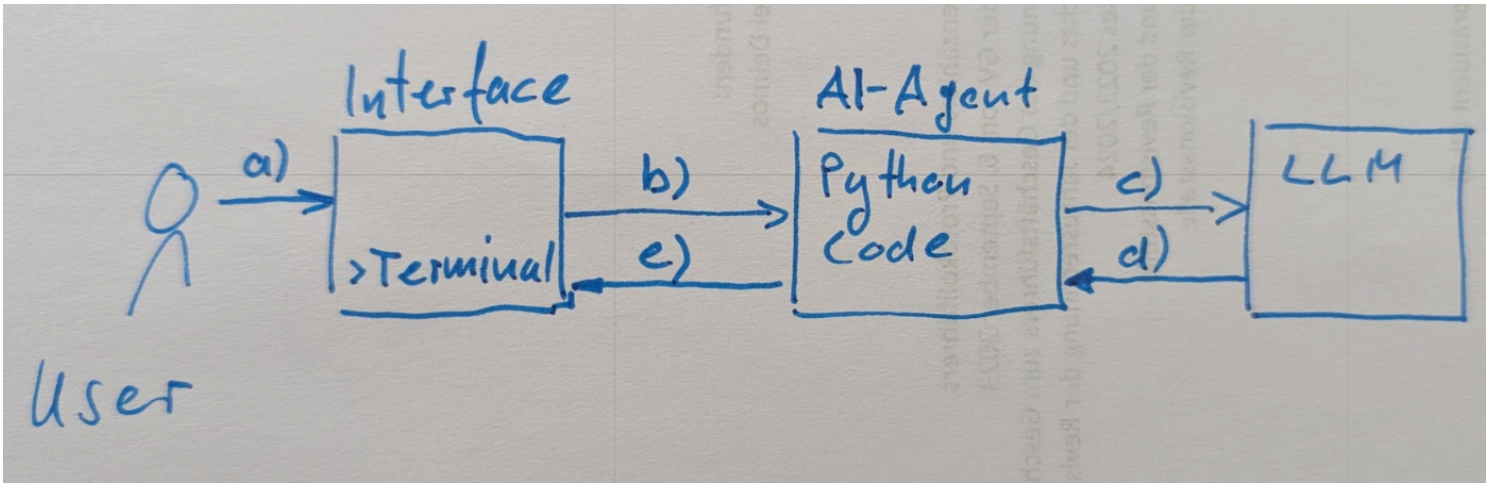
    # Ergebnis der ausgeführten Aktion anzeigen (zur Kontrolle oder Debugging)
    print(f"Action result: {result}")

    # 5️⃣ Ergebnis in Zeichenfolge konvertieren: Gedächtnis (memory) aktualisieren:
    # Der Agent merkt sich, was er gesagt hat (assistant)
    # und was als Ergebnis (user input) zurückkam.
    # So kann das Modell später darauf Bezug nehmen.
    memory.extend([
        {"role": "assistant", "content": response},
        {"role": "user", "content": json.dumps(result)}  # Ergebnis als JSON-Text speichern
    ])

    # 6️⃣ Schleife fortsetzen: Prüfen, ob das Modell das Ende signalisiert hat
    # (z. B. Tool "terminate" wurde aufgerufen)
    if action["tool_name"] == "terminate":
        break

    # 7️⃣ Zähler erhöhen, um Endlosschleifen zu vermeiden
    iterations += 1
```

Schematischer Ablauf eines AI-Agent-Loop in Python



Schnittstellenbeschreibung

Buchstabe	Richtung	Beschreibung
a)	User → Interface	Der Benutzer gibt eine Anweisung ein, z. B. eine Frage oder Aufgabe („Lies Datei test.txt“).
b)	Interface → Agent	Das Terminal leitet die Eingabe an das Python-Programm (den AI-Agenten) weiter (Python-Code).
c)	Agent → LLM	Der Agent erstellt daraus einen Prompt (z. B. im JSON- oder Chat-Format) und sendet ihn an das LLM (z. B. GPT-4o).
d)	LLM → Agent	Das LLM liefert eine Antwort zurück (z. B. `"tool_name": "read_file"` , einen Text oder JSON).
e)	Agent → Interface	Der Agent führt ggf. eine Aktion aus, formatiert das Ergebnis und zeigt es dem Benutzer im Terminal an.

Step 1: Constructing the Agent Prompt

The prompt is created by **appending the agent's rules (system message**) to the **current memory** of interactions. Part of the memory is a descripton of the task that the agent should perform. This ensures the agent is always aware of its tools and constraints while also remembering past actions.

```
prompt = agent_rules + memory
```

Explanation:

- agent_rules: This contains the predefined system instructions, ensuring the agent behaves within its defined constraints and understands its tools.
- memory: This is a record of all past interactions, including user input, the agent's responses, and the results of executed actions.

By constructing the prompt this way, the agent retains continuity across iterations, ensuring it can adapt its behavior based on previous actions and results. The memory tells it what just happened, what happened in the past, and informs its decision of the next action.

Hier werden die Tools definiert, durch welche der **Agent Loop i**n Python iterated:
C:\Users\PhilippSauber\Dropbox\Next_2019\Dokumente\Know-how\Coursera-edX\AI-Agents_2510\Agents\agent-one.py

Agent Rules: Defining the Agent's Behavior

Before the agent begins its loop, it must have a **clear set of rules that define its behavior**, capabilities, and constraints. These **agent rules** are specified in the **system message** and play a critical role in ensuring the agent interacts predictably and within its defined boundaries.

```
agent_rules = [{
"role": "system",
"content": """
You are an AI agent that can perform tasks by using available tools.
Available tools:
```

```
- list_files() -> List[str]: List all files in the current directory.
- read_file(file_name: str) -> str: Read the content of a file.
- terminate(message: str): End the agent loop and print a summary to the user.
If a user asks about files, list them before reading.
Every response MUST have an action.
Respond in this format:
```action
{
"tool_name": "insert tool_name",
"args": {...fill in any required arguments here...}
}
```

Explanation:

- **Role of system messages**: The systemrole in the messages list is used to establish ground rules for the agent. This ensures the LLM understands what it can do and how it should behave throughout the session.
- **Tools description**: The agent rules explicitly list the tools the agent can use, providing a structured interface for interaction with the environment.
- **Output format**: The rules enforce a standardized output format ("```action {...}"), which makes parsing and executing actions easier and less error-prone.

Detaillierte Beschreibung der Fähigkeiten eines Agents

Each of the **"tools"** in the system prompt correspond to a function in the code. The agent is going to choose what function to execute and when. Moreover, it is going to decide the parameters that are provided to the functions. The agent is not creating the functions at this point; it is orchestrating their behavior. This means that the logic for how each tool operates is predefined in the code, and the agent focuses on selecting the right tool for the job and providing the correct input to that tool.

Because agents can adapt as the loop progresses, they can dynamically decide which tool to use based on the current context and task requirements. This ability allows the agent to adjust its behavior as new information becomes available, making it more flexible and responsive to the user's input.

For example, if the user asks the agent to read the contents of a specific file, the agent will first use the `list_files` tool to identify the available files. Then, based on the result, it will determine whether to proceed with the `read_file` tool or respond with an error if the file does not exist. The agent evaluates each step iteratively, ensuring its actions are informed by the current state of the environment.

This orchestration process, driven by the agent rules and the tools available, showcases the power of combining pre-defined functions with adaptive decision-making (of the LLM). By allowing the agent to focus on what to do rather than how to do it, we create a system that leverages the LLM for high-level reasoning while relying on well-defined code for execution.

This separation of reasoning and execution is what makes the agent loop so powerful—it creates a modular, extensible framework that can handle increasingly complex tasks without rewriting the underlying tools. Additionally, the agent loop eliminates much of the "glue code"* traditionally required to tie these fundamental functions together. **Instead of hardcoding workflows, the agent dynamically decides the sequence of actions needed to achieve a task, effectively realizing a program on top of its components.** This dynamic nature enables the agent to combine its tools in ways that would typically require custom logic, making it far more versatile and capable of addressing a broader range of use cases **without additional development overhead**.

*Der Agent Loop ersetzt oder automatisiert viel von dem "glue code" Kleber-Code, den man früher manuell schreiben musste, um die einzelnen Funktionen (wie Denken, Handeln, Gedächtnis) zu verbinden.

Example in practice:

If the user asks, "What files are here?", the agent rules guide the LLM to respond with something like:
```

```
{"tool_name": "list_files", "args": {}}
```

This response ensures the agent's next step is both predictable and executable within its predefined constraints.

How agent_rules integrate with the loop:

The agent_rulesare combined with the memoryin **Step 1: Construct Prompt** to form the input for the LLM. This guarantees that the agent always has access to its instructions and tools at every iteration. We will discuss the memory in more detail later.

This step prepares the input for the LLM by combining the system rules and the memory of the agent's previous interactions. The goal is to give the LLM all the necessary context for generating the next action.

Example in practice:

If the user asks, "What files are in this directory?", the memory might look like this:

```
memory = [
{"role": "user", "content": "What files are in this directory?"},
{"role": "assistant", "content": "```action\n{\"tool_name\":\"list_files\",\"args\":{}}\n```"},
{"role": "user", "content": "[\"file1.txt\", \"file2.txt\"]"}
]
```

Adding agent_rulesensures the LLM understands what **tools it can use to continue interacting.**

Step 2: Generate Response

After constructing the prompt, **the agent sends it to the LLM to receive a response**. This response will define the next action for the agent to execute.

Code snippet:

```
response = generate_response(prompt)
```

Explanation:

The generate_responsefunction uses the LiteLLM library to **send the prompt** to the LLM and **retrieve its response**. The response typically includes a structured action that the agent will parse and execute in the next steps. **This is where the LLM decides what action the agent should take**, based on the provided context and rules.

Step 3: Parse the Response

After generating a response, the next step is to extract the intended action and its parameters from the LLM's output. The response is expected to follow a predefined structure, such as a JSON format encapsulated within a markdown code block. This structure ensures the action can be parsed and executed without ambiguity.

In the code, this is accomplished by locating and extracting the content between the ```action markers. If the response does not include a valid action block, the agent defaults to a termination action, returning the raw response as the message:

```
def parse_action(response: str) -> Dict:
"""Parse the LLM response into a structured action dictionary."""
try:
response = extract_markdown_block(response, "action")
response_json = json.loads(response)
if "tool_name" in response_json and "args" in response_json:
return response_json
else:
```

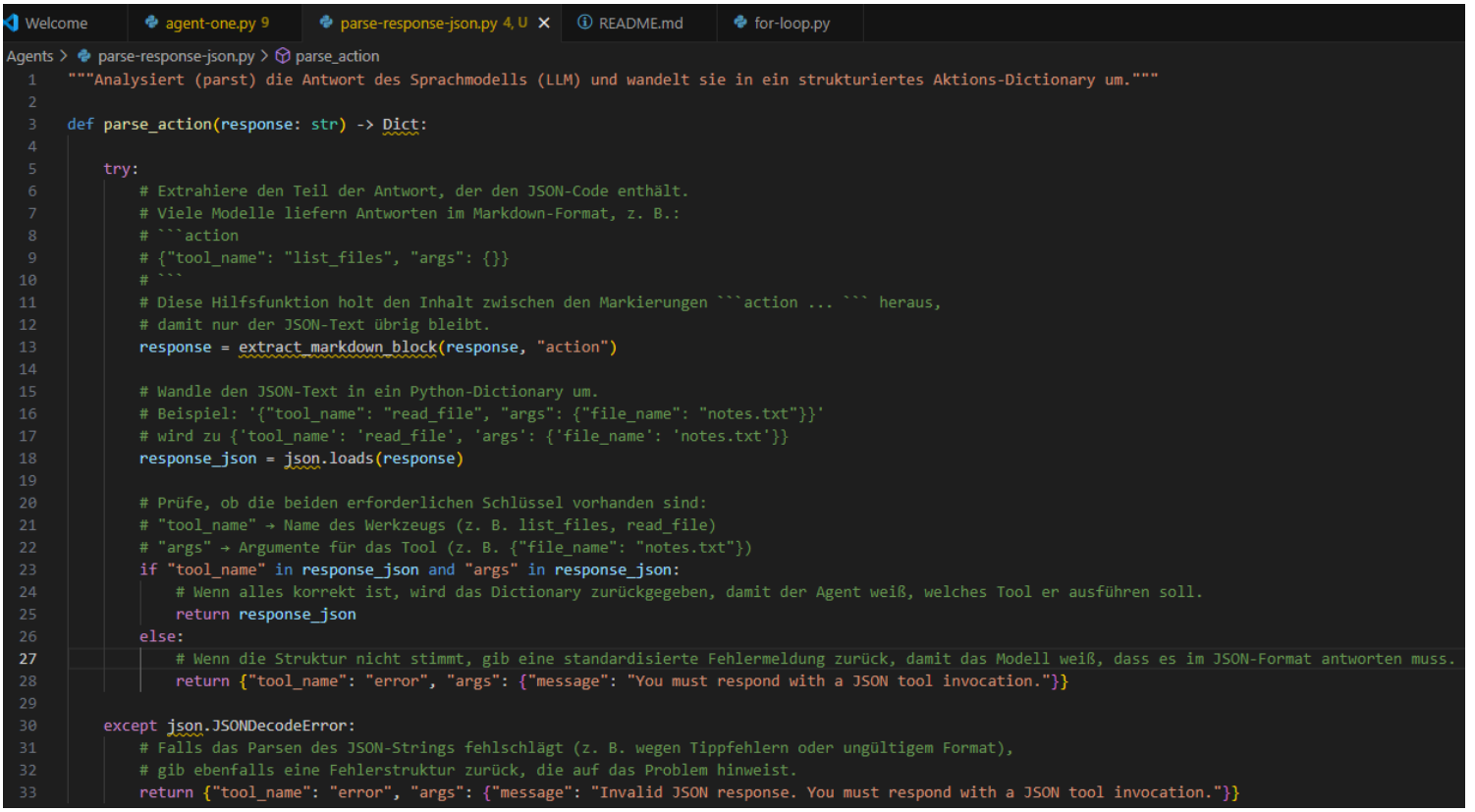
```
return {"tool_name": "error", "args": {"message": "You must respond with a JSON tool
invocation."}}
except json.JSONDecodeError:
return {"tool_name": "error", "args": {"message": "Invalid JSON response. You must respond with
a JSON tool invocation."}}
```

This parsing step is critical to ensuring the response is actionable. It provides a structured output, such as:

```
{
"tool_name": "list_files",
"args": {}
}
```

By breaking down the LLM's output into **tool_name and args**, the agent can precisely determine the next action and its inputs.
If the LLM response does not contain a valid action block, the agent defaults to an error message, prompting the LLM to provide a valid JSON tool invocation. The error message appears to have come from the "user". This fallback mechanism ensures the agent can recover if it starts outputting invalid responses that aren't in the desired format.

Meine Interpretation: C:\Users\PhilippSauber\Dropbox\Next_2019\Dokumente\Know-how\Coursera-edX\AI-Agents_2510\Agents\parse-response-json.py



Diese Funktion prüft, ob die Modellantwort gültiges JSON ist und die erwarteten Schlüssel "tool_name" und "args" enthält.
Wenn ja, wird das Dictionary zurückgegeben, das z. B. { "tool_name": "read_file", "args": {"file_name": "notes.txt"} } enthält.
Wenn nein, wird stattdessen eine Fehleraktion zurückgegeben, die das Modell dazu auffordert, korrektes JSON zu liefern.
args -> steht nicht für den Text der LLM-Antwort, sondern für die Argumente, die das Sprachmodell dem „Tool" (also der Funktion) mitgeben will.

Schlüssel	Bedeutung
`tool_name`	Der Name des Werkzeugs oder der Funktion, die ausgeführt werden soll
`args`	Die Argumente (Parameter), die diese Funktion benötigt

Step 4: Execute the Action

Once the response is parsed, the agent uses the extracted **tool_name and args** to execute the corresponding function. **Each predefined tool** in the system instructions corresponds to a specific function in the code, **enabling the agent to interact with its environment**.

The execution logic involves mapping the tool_nameto the appropriate function and passing the provided arguments:

```
if action["tool_name"] == "list_files":
result = {"result": list_files()}
elif action["tool_name"] == "read_file":
result = {"result": read_file(action["args"]["file_name"])}
elif action["tool_name"] == "error":
result = {"error": action["args"]["message"]}
elif action["tool_name"] == "terminate":
print(action["args"]["message"])
break
else:
result = {"error":"Unknown action: "+action["tool_name"]}
```

For example, if the action specifies **tool_name as list_files with empty args**, the list_files() function is called, and the **agent returns the list of files in the directory**. Similarly, a read_fileaction extracts the filename from the arguments and retrieves its content.

The execution step is the point where the agent **performs tangible work, such as interacting with files or printing messages to the console**. It bridges the decision-making process with concrete results that feed back into the agent's memory for subsequent iterations.

Step 5: Update the Agent's Memory

After executing an action, **the agent updates its memory with the results**. Memory serves as the agent's record of what has happened during the interaction, **including user requests, the actions performed, and their outcomes.** By appending this information to the memory, the agent retains context, enabling it to make more informed decisions in future iterations.

In the code, memory is updated by **extending it with both the LLM's response (representing the agent's intention) and the result of the executed action**:

```
memory.extend([
{"role": "assistant", "content": response},
{"role": "user", "content": json.dumps(result)}
])
```

How This Works:

- The **assistant role** captures the structured response generated by the LLM.
- The **user role** captures the feedback in the form of the action result, ensuring that the LLM has a clear understanding of what happened after the action was performed. **The results of actions are always communicated back to the LLM with the "user" role.**

By keeping a running history of these exchanges, the agent maintains continuity, allowing it to refine its behavior dynamically as the **memory grows and track the status of its work**.

Step 6: Decide Whether to Continue

The final step in each iteration of the agent loop is determining whether to continue or terminate. This decision is based on the action executed and the state of the task at hand. If the parsed action specifies terminate, or if a predefined condition (e.g., maximum iterations) is met, the agent ends its loop.
In the code, this is implemented as a simple conditional check:

```
if action["tool_name"] == "terminate":
print(action["args"]["message"])
break
```

If the action specifies a termination, the loop exits, and the agent provides a closing message defined in the terminateaction's arguments. If no termination is triggered, the agent loops back to process the **next user request** or continue its task.

Example: Iterative Adaptation
Imagine the agent is tasked with reading a file but encounters a missing filename in the initial request.

1. In the first iteration, it executes `list_files` to retrieve the available files.
2. Based on the memory of this result, it refines its next action, prompting the user to select a specific file.
3. This iterative process continues until the task is completed or the agent determines that no further actions are required.

Each loop iteration, the agent can look back at its memory to decide if it has completed the overall task. The memory is a critical part of deciding if the agent should continue or terminate. By deciding whether to continue at each step, the agent balances its ability to dynamically adapt to new information with the need to eventually conclude its task. The agent can also be instructed on when to terminate the loop, such as if more than two errors are encountered or if a specific condition is met.

Wie das Gedächtnis eines AI-Agenten funktioniert: Das Gedächtnis eines AI-Agenten besteht aus dem fortlaufenden Nachrichtenverlauf zwischen Benutzer, Agent und LLM. **Nach jeder Iteration speichert der Agent sowohl seine eigene Antwort (als „assistant"-Nachricht) als auch die Ergebnisse der ausgeführten Aktionen oder Tool-Aufrufe (als „user"-Nachricht).** So entsteht ein wachsender Kontext, der dem Modell bei jedem neuen Schritt wieder mitgegeben wird. Auf diese Weise „erinnert" sich der Agent daran, was bereits gefragt wurde, welche Aktionen er schon ausgeführt hat und welche Resultate dabei herauskamen. Das ermöglicht dem Agenten, Entscheidungen auf Basis der gesamten Interaktionshistorie zu treffen.
Warum die Ergebnisse von API- oder Tool-Aufrufen als „user"-Nachrichten gespeichert werden:Die Resultate von Tools oder API-Aufrufen werden absichtlich als „user"-Nachrichten dargestellt, damit das LLM diese Informationen als externen Input wahrnimmt – und nicht als seine eigene vorherige Antwort.
Dadurch werden zwei Dinge klar getrennt: die Überlegung oder Instruktion des Modells (assistant) und das tatsächliche Resultat, das aus der realen Welt oder einem Tool stammt (user). Diese Trennung verhindert, dass das LLM sein eigenes Denken mit echten Daten vermischt, und stellt sicher, dass es auf Basis der tatsächlichen Ergebnisse weiterarbeiten kann. So kann der Agent-Loop zuverlässig prüfen, ob eine Aktion erfolgreich war, und im nächsten Schritt die passende Folgeaktion wählen.

Beispiele von AI-Agents

1) Medizinisches Beispiel: Klinischer Triage-Agent (digitale Ersteinschätzung)

Was die LLM macht (Aufgaben des Sprachmodells)

- Versteht die Symptome, die der Patient eingibt (z. B. „Atemnot, seit 3 Tagen Fieber")
- Formt daraus strukturierte Daten (z. B. JSON: `{"symptom": ..., "schweregrad": ...}`)

- Entscheidet basierend auf Vorergebnissen, welchen nächsten Schritt der Agent ausführen soll
- Gibt Empfehlungen, Warnungen oder Priorisierung (z. B. „Notfall – sofort Arzt“)

Was der Agent macht (Python-Code)

- Erhält Entscheidungen vom LLM (z. B. `{"tool_name": "lookup_medical_rules", "args": {...}}`)
- Führt Tools aus, die echte medizinische Daten liefern
- Validiert Ergebnisse und steuert den Ablauf
- Speichert LLM-Antworten und Tool-Ergebnisse im Memory

Welche Tools zur Verfügung stehen

Tool	Aufgabe
`lookup_medical_rules(symptoms)`	Ruft medizinische Triage-Regeln ab (z. B. WHO, Swiss Triage System)
`check_medications(patient_id)`	Prüft bestehende Medikamente und Interaktionen
`fetch_patient_history(patient_id)`	Holt Krankengeschichte oder frühere Befunde
`alert_doctor(level)`	Meldet kritische Fälle an einen Arzt

Was im Memory gespeichert wird

- Vom User eingegebene Symptome
- Vorherige Antworten des LLM
- Ergebnisse der Tools (z. B. „Patient nimmt Blutverdünner“)
- Die bisherige Einschätzung (z. B. „Verdacht auf Lungenentzündung“)
- Der letzte Schritt im Triage-Prozess
- Medikamenten Interaktionen

➡️ Dadurch kann der Agent mehrere Schritte logisch verknüpfen, statt alles in einer einzigen LLM-Antwort zu lösen.

2) Marketing-Beispiel: Automatischer Content-Research-Agent

Was die LLM macht (Aufgaben des Sprachmodells)

- Versteht Marketingziele (z. B. „Erstelle LinkedIn-Posts für EV-Charging-Fachpublikum“)
- Generiert Zwischenideen (Themen, Schmerzpunkte, Keywords)
- Interpretiert Resultate der Tools (z. B. Keyword-Suche, Wettbewerbsanalyse)
- Entscheidet, welchen Datensatz der Agent als Nächstes braucht

Was der Agent macht (Python-Code)

- Ruft Tools auf, um reale Web- oder Datenbankinfos abzuholen
- Organisiert den Workflow: Recherche → Analyse → Content-Briefing
- Prüft Tool-Ergebnisse und gibt sie der LLM für den nächsten Denk-Schritt
- Speichert alles im Kontext (Memory)

Welche Tools zur Verfügung stehen

Tool	Aufgabe
`fetch_web_page_text(url)`	Holt HTML-Texte von Websites (Wettbewerber, Magazine)
`keyword_api_search(term)`	Holt Keyword-Daten (Suchvolumen, Trends)
`summarize_pdf(file)`	Extrahiert Inhalte aus Whitepapers
`check_brand_tone(company)`	Holt Tone-of-Voice-Profile aus Vorgaben
`analytics_query(sql)`	Holt z. B. Kundenverhalten aus der DB

Was im Memory gespeichert wird

- Die Marketing-Zielsetzung
- Die bisherigen Recherche-Ergebnisse (z. B. Wettbewerber-Positionierung)
- Die Tools-Ausgaben:
  - Keywords
  - Trendanalysen
  - Website-Inhalte
  - Memory enthaelt hier
  - Zielgruppe und Tonalitaet
  - Wettbewerber Inhalte
  - Zwischenversionen von Ideen
  - Bisherige Antworten des LLM
- Zwischenversionen der Content-Ideen
- Was das LLM bereits vorgeschlagen hat, damit es nicht in Schleifen gerät

➡️ Der Agent kann dadurch schrittweise hochwertigen Content erstellen, der auf echten Daten basiert — nicht nur auf LLM-Halluzinationen.