AI AGENTS AND AGENTIC AI WITH PYTHON & GENERATIVE AI COURSE

Module 2 _ Lecture Notes

INSTRUCTOR: DR. JULES WHITE
WRITTEN BY: HOUSHYAR JAFARI ASL

Abstract

Course: *AI Agents and Agentic AI with Python & Generative AI* Module 1 (Presented by Dr. Jules White)

This module dives into practical implementation of agent tools and function calling, covering:

Core Topics:

- **GAIL Framework**: Structuring agent prompts with Goals, Actions, Information, and Language
- **Tool Design Principles**: Effective naming conventions and description templates for custom tools
- **Feedback Loops**: Implementing actionable result reporting and error handling
- **Function Calling**: Leveraging LLM capabilities to interface with Python functions

Key Features:
• **Hands-on Python implementations** of tool-using agents (CPU-optimized for Google Colab)
• **Real-world case studies** including alien spaceship escape and microwave control systems
• **Best practices** for error message design and state management
• **Function calling workflows** from basic to advanced agent loops

The materials include video walkthroughs, executable code samples, and practical exercises focused on building production-ready agent systems without GPU/API dependencies.

Developed as a companion to Dr. Jules White's course, with added optimizations for broader accessibility.

Link of the Course

Link of the Colab Notebook (Github)

GAIL- Goals, Actions, Information, Language

Why Prompt Design Matters

Just like giving vague instructions to a new intern guarantees failure, poor prompt design sets up AI agents to fail. The GAIL framework provides a structured approach to avoid these pitfalls:

1. **The Intern Analogy**
    - Bad instructions → Failed tasks
    - Unstructured prompts → Agent errors
    - Solution: Treat agent prompts like training manuals for new hires
2. **Common Pitfalls**
    - "Wall of text" prompts
    - Missing process steps
    - Unbounded actions

The GAIL Framework Explained

1. Goals/Instructions (The Rulebook)

- **Persona**: "You are Action Agent, a helpful AI assistant"
- **Rules**:
    - "Never modify financial records without approval"
    - "Always verify data sources"
- **Process Flows**:

1. Check for existing expense entries
2. If duplicate exists, ask user for confirmation
3. Only then proceed with entry

2. Actions (The Toolbelt)

- Bounded set of permitted operations:

- query_database()
- generate_report()
- send_approval_request()

3. Information (The Context)

- **Task-Specific**: User queries, API responses
- **Session State**: Action history, intermediate results
- **Environment**: System status, external data

4. Language (The Protocol)

- Strict output formatting:

```
{
  "thoughts": "<reasoning>",
  "action": {
    "tool": "expense_checker",
    "args": {"date": "2025-07-31"}
  }
}
```

Building an Agent Prompt

System Message Template:

```
ROLE: Action Agent (Customer Service Specialist)

GOAL: Complete tasks while preventing duplicates

RULES:

- Always verify before writing data

- Never store PII


TOOLS:

• expense_checker(date): Returns matching entries

• receipt_parser(file): Extracts expense details


RESPONSE FORMAT:

THOUGHTS: <step-by-step reasoning>

ACTION: {"tool": "...", "args": {...}}
```

User Message Components:

- Task: "Add $58.70 lunch expense for 2025-07-30"
- Feedback: "expense_checker found 0 duplicates"

Implementation Architecture

Layer	Content	Example
System	GAIL rules	Persona, Actions, Output Format
User	Task Information	Expense Details, API Results
Assistant	Action Decisions	Chosen Tool + Parameters

Giving Agents Tools

Agents interact with the world through **constrained tools/actions** rather than unlimited possibilities. This mirrors real-world constraints where:

- Humans work with limited resources (e.g., only a skillet and wood fire for cooking)
- Computer systems have fixed APIs/operations

Key Analogy:

"Just as you wouldn't expect an intern to use equipment that doesn't exist, agents must work within defined toolkits."

Implementing Tools in Agent Prompts

(Example: Cooking Agent with Limited Utensils)

1. Tool Declaration

```
Available Tools:
- 1-quart sauté pan
- Cast iron skillet
- Wood fire (oak-burning)
```

2. Constrained Problem-Solving Flow:

1. User requests: *"Cook pizza on wood fire"*
2. Agent adapts to tools:
    - Skillet → Pizza stone substitute
    - Wood fire → High-heat oven alternative
3. Outputs tool-specific instructions:

```
ACTION: Preheat skillet on wood fire
FEEDBACK: Skillet at 400°F
ACTION: Cook dough with olive oil
```

3. Why Constraints Matter:

- Prevents unrealistic solutions (e.g., suggesting microwave when unavailable)
- Forces creative adaptation to available resources
- Mirrors API limitations in software systems

Tools vs. Actions: When to Use Each

Approach	Best For	Example
Tools	Human Collaborators	*"Use skillet to sear steak"*
Actions	Rigid Computer Systems	*"Call CRM API (endpoint: /update_contact)"*

Key Differences:

- **Tools** imply flexibility (human interprets how to use)
- **Actions** require precision (exact API calls/parameters)

Technical Implementations

1. **For Human-in-the-Loop Systems**:

```
tools = [
    "screenshot: Capture visual system state",
    "cli_query: Run terminal command"
]
```

2. **For API-Driven System:**

```
actions = {
    "query_database": {"params": ["date", "id"]},
    "send_email": {"params": ["recipient", "body"]}
}
```

Tool Descriptions and Naming

When working with agentic AI systems, we face two distinct scenarios:

1. **Intuitive Tools** (Common knowledge)
    - Example: "skillet", "pot" - AI understands basic functionality
    - Requires minimal description
2. **Custom Tools** (System-specific)
    - Example: "XR155_processor", "Q63_portal"
    - Requires explicit documentation
    - "We've built the equivalent of alien tools in our computer systems"

Case Study: Alien Spaceship Escape

1. Naming Tools with Descriptions

Tools: - X155 - prepares alien pizza - Q63 - opens dimensional portal - L199 - plays Beatles music on loop

Agent Behavior:

- Correctly uses X155 to create distraction
- Strategically chains tools (distract → escape)
- Maintains logical flow despite cryptic names

2. Well-Named Tools (No Descriptions Needed)

Tools: - makeAlienPizza - openDimensionalPortal - playBeatlesMusic

Agent Behavior:

- Same effective strategy
- More reliable understanding
- Reduced prompt complexity

3. Cryptic Abbreviations (Failure Case)

```
Tools:
- mkpz
- odprtl
- pbm
```

Agent Errors:

- Misinterprets mkpz (makes pizza → creates map)
- Underutilizes portal capability
- Loses strategic advantage

Key Principles from the Transcript

1. **The Name-Description Partnership**
    - Names provide quick recognition
    - Descriptions ensure precise understanding
    - "The description of the tool was critical"
2. **Contextual Awareness**
    - Cultural assumptions emerge (aliens as threatening)
    - System-specific knowledge must be explicit
    - "It will not have institutional knowledge"
3. **Tool Presentation Matters**
    - Ordering implies priority
    - Must specify optional vs. mandatory tools

- Should indicate typical use cases

Description Template

{name}: {function}. {Inputs}. {Outputs}. {Constraints}.

Example:

"thermalScanner: Detects lifeforms within 20m. Input: coordinates. Output: heat signatures. 5-second recharge between scans."

Implementation Example

```
tools = {
    "makeAlienPizza": {
        "purpose": "Creates food-based distraction",
        "inputs": None,
        "outputs": "Alien attention shift",
        "risks": "May attract additional aliens"
    },
    # Anti-pattern example:
    "mkpz": {}  # Never leave tools undocumented
}
```

Key Takeaways

1. **Names Are Contracts**
    - Test with: "Would an intern understand this immediately?"
2. **Descriptions Are Requirements**
    - Must specify: purpose, inputs, outputs, constraints
3. **Context Guides Usage**
    - Clarify tool relationships and priorities
    - Provide usage examples

Tool Results and Agent Feedback

Core Concept:
Agentic AI relies on a continuous cycle of:

1. Action specification
2. Execution (by human/system)
3. Result feedback
4. Adaptive next-step selection

Key Insight:

"The result of applying that tool is the follow-up prompt - this is how the agent 'sees' the world"

Microwave Control Case Study

1. Action Definitions

```
tools = {
    "microwave_get_current_time": None,
    "microwave_increase_time": None,  # No parameter specified
    "microwave_start": {"depends_on": "time_set"},
    "microwave_open_door": None
}
```

2. Interaction Flow

Step	Agent Action	System Feedback	Agent Adaptation
1	insert_food_in_microwave	"Food in microwave"	Proceeds to timing
2	microwave_increase_time	"Time increased by 5 seconds"	Requests more time
3	microwave_start	“ERROR: DOOR OPEN”	Corrects with close_door
4	microwave_close_door	“Door Closed”	Retries start

Critical Observations:

- The 5-second increment revelation forced strategy adjustment
- The "ERROR: Door open" message contained actionable context
- Missing error codes ("Error 32") would break the flow

Key Takeaways

1. **Feedback is Perception**:
    - Agents only "know" what you tell them occurred
2. **Error Design is UX Design**:
    - Write messages for the agent, not just developers
3. **State Tracking is Essential**:
    - Always include resultant system state
4. **Dependencies Must Be Explicit**:
    - Document preconditions and requirements

These sections are written in a Colab Notebook:
*** Agent Tools in Python**
*** Try Out an Agent that Calls Python Functions**
*** Using LLM Function Calling for AI-Agent Interaction**
*** Simplifying the AI Agent Loop with Function Calling**
*** Tool Design and Naming Best Practices for AI Agents**

Link of the Colab Notebook (Github)