Cheat Sheet: CrewAI Fundamentals and Advanced Applications

Estimated time: 15 minutes

1. What is CrewAI?

CrewAI is a framework for orchestrating autonomous AI agents, enabling them to collaborate on complex tasks. It allows you to build a "crew" of agents, each with a specific `role`, `goal`, and `tools`, who work together to achieve an objective, much like a human team. This approach is ideal for automating multi-step workflows, from research and analysis to content creation and customer service.

---

2. Core Components of a Crew

A CrewAI system is built from three main components: **Agents**, **Tasks**, and the **Crew** itself.

AI Agents

Agents are the individual workers in your crew. Each one is an autonomous AI with a defined purpose.

Attribute	Description
Role	The agent's job title or function (e.g., "Nutrition Analyst").
Goal	The specific objective the agent is meant to achieve.
Backstory	Context or personality that shapes the agent's behavior.
Tools	A list of functions or capabilities the agent can use (e.g., web search, PDF reader).
llm	The language model that powers the agent's reasoning.
verbose	Is a boolean parameter; if `True`, it enables detailed logging of the agent's thought process.

Example: Creating an Agent

```
from crewai import Agent
from crewai_tools import SerperDevTool
# Initialize a search tool
search_tool = SerperDevTool()
# Define an agent and provide it with the search tool
research_agent = Agent(
  role='Senior Research Analyst',
  goal='Uncover cutting-edge information on a given topic',
  backstory='An expert researcher skilled at synthesizing data.',
  tools=[search_tool],
  llm=my_llm, # An initialized LLM
  verbose=True
)
```

Tasks

Tasks are the specific assignments given to agents. They define what needs to be done and what the expected outcome is.

Attribute	Description
Description	A clear explanation of the assignment, often with placeholders like {topic} for dynamic inputs.
Expected Output	A description of what a successful result should look like.
Agent	The agent assigned to complete the task.
Context	A list of other tasks whose output this task depends on.
Tools	A list of tools specifically provided for this task (see Section 3).
output_pydantic	A Pydantic model to structure the task's output (see Section 4).

Example: Creating a Task

```
from crewai import Task
# Define a task for the research_agent
research_task = Task(
  description='Analyze the major trends in {topic}.',
  expected_output='A detailed report on key trends and technologies.',
  agent=research_agent
)
```

Crews

The Crew brings everything together, managing the agents and tasks according to a defined process.

Attribute	Description
Agents	A list of all agents in the crew.
Tasks	A list of all tasks to be executed.
Process	The execution strategy. `Process.sequential` runs tasks one after another. `Process.hierarchical` allows for more complex, delegation-based workflows.

Example: Assembling and Running a Crew

```
from crewai import Crew, Process
# Create the crew with a sequential process
content_crew = Crew(
    agents=[research_agent, writer_agent],
    tasks=[research_task, writer_task],
    process=Process.sequential,
    verbose=True
)
# Start the crew's work with a specific input
result = content_crew.kickoff(inputs={'topic': 'AI in Healthcare'})
print(result)
```

---

3. Structuring Workflows: Agent-Centric vs. Task-Centric Tools

A key design choice in CrewAI is *how* you provide tools to your agents. This choice impacts the efficiency and reliability of your system.

The Generalist (Agent-Centric Approach)

This is the standard method where you give an agent a full "toolbox" of all the tools it might need. The agent uses its own reasoning to decide which tool to use for a given task.

- **Pro:** Simple to set up
- **Con:** Can be inefficient and unpredictable, as the agent may choose the wrong tool or take longer to decide

Agent-Centric Code Snippet

```
# The agent is given a toolbox with multiple tools
generalist_agent = Agent(
    role='Inquiry Specialist',
    goal='Answer all customer questions.',
    tools=[pdf_search_tool, web_search_tool] # Agent must choose
)
```

The Specialist (Task-Centric Approach)

In this more advanced and robust approach, you assign tools directly to the tasks that require them. When you assign tools to a task, CrewAI completely overrides the agent's tools for that specific task so only the tools it needs for that job are used.

- **Pro:** More efficient, predictable, and secure. It eliminates ambiguity and focuses the agent's efforts
- **Con:** Requires a more structured, multi-task workflow

Task-Centric Code Snippet

```
# The agent is not given tools directly
specialist_agent = Agent(
    role='Customer Service Specialist',
    goal='Follow a step-by-step process to answer questions.',
    tools=[] # No tools assigned to the agent; however, if tools are provided they are overridden
)
# Each task gets its own specific tool
faq_search_task = Task(
    description="Search the FAQ PDF for the query: '{customer_query}'",
    expected_output="Relevant info from the PDF.",
    tools=[pdf_search_tool], # Tool is specific to THIS task
    agent=specialist_agent
)
```

---

4. Structuring Output Data with Pydantic

To ensure your agents produce consistent, reliable, and structured data (like JSON), you can use Pydantic models. By defining a `BaseModel`, you create a template for the agent's output.

Example: Defining and Using a Pydantic Model

```
from pydantic import BaseModel, Field
from typing import List #Used to ensure that the AI's output is a list of strings
# 1. Define the data structure
class GroceryShoppingPlan(BaseModel):
    """Complete simplified shopping plan"""
    total_budget: str = Field(description="Total planned budget")
    meal_plans: List[str] = Field(description="Planned meals")
    shopping_tips: List[str] = Field(description="Money-saving tips")
# 2. Assign the model to a task's output
shopping_task = Task(
    description="Organize a shopping list for {meal_name}.",
    expected_output="An organized shopping plan.",
    agent=shopping_organizer,
    output_pydantic=GroceryShoppingPlan
)
```

5. Advanced Crew Management with @CrewBase and YAML

For larger, production-level projects, CrewAI offers a powerful way to organize your agents and tasks using the `@CrewBase` decorator and YAML configuration files.

- **`@CrewBase` Decorator:** A Python class decorator that automates the setup of your crew by discovering methods marked with `@agent` and `@task`.
- **`@crew` Decorator:** An optional decorator used inside a `@CrewBase` class that marks the specific method responsible for assembling the agents and tasks into a final `Crew` object. This method defines which agents and tasks are part of the crew and sets the execution process (e.g., `Process.sequential`).
- **YAML Configuration:** Allows you to define your agents' and tasks' properties (like `role`, `goal`, `description`) in separate `agents.yaml` and `tasks.yaml` files. This separates configuration from code, making your project cleaner and easier to manage.

  Crucial Note: The `@CrewBase` class relies on Python's `inspect` module to find file paths, which **does not work correctly inside a Jupyter Notebook**. For this reason, you should always define your `@CrewBase` classes in separate **`.py` files** and import them into your main application.

Example: Conceptual Structure in a `.py` file

```
# In a file named 'my_crew_defs.py'
from crewai.project import CrewBase, agent, task, crew
from crewai import Agent, Task, Crew, Process
@CrewBase
class FinancialCrew:
    """A class to manage the financial analysis crew."""
    agents_config = 'config/agents.yaml'
    tasks_config = 'config/tasks.yaml'
    @agent
    def market_analyst(self) -> Agent:
        return Agent(config=self.agents_config['market_analyst'])
    @task
    def analysis_task(self) -> Task:
        return Task(config=self.tasks_config['analysis_task'])
    @crew
    def crew(self) -> Crew:
        """Assembles the agents and tasks into a crew."""
        return Crew(
            agents=[self.market_analyst()],
            tasks=[self.analysis_task()],
            process=Process.sequential,
            verbose=True
        )
```