Function Registration with Decorators in Python

This document explains how **function registration using decorators** works in the `Agent` class of this repository.

🔹 What Problem Does This Solve?

When building an AI agent, we want to define multiple **tools** (functions like `search`, `add_numbers`, etc.). Each tool should be:

- Easy to add
- Automatically registered in a central place (`self.tools` dictionary)

Instead of manually writing:

self.tools["search"] = search
self.tools["add_numbers"] = add_numbers

the code uses a **decorator** to register tools automatically.

🔹 The `_tool` Method

```
def _tool(self, tools: Dict[str, Callable]) -> Callable:
    """Decorator to register functions as tools"""
    def decorator(func: Callable) -> Callable:
        tools[func.__name__] = func
        return func
    return decorator
```

How it Works

1. `_tool(tools)` is called → returns a `decorator` function.
2. That `decorator` is applied to a function.
3. The decorator:
    - Adds the function to the `tools` dictionary using its name.
    - Returns the function unchanged.

Result: Every decorated function gets stored in `tools` automatically.

🔹 Using the Decorator in `_initialize_tools`

```
def _initialize_tools(self) -> Dict[str, Callable]:
    tools = {}

    @self._tool(tools)
    def search(query: str) -> str:
        """Search for information online (mocked version)"""
        return f"Search results for: {query}"

    @self._tool(tools)
    def add_numbers(prompt: str) -> str:
        """Mathematical addition tool"""
        numbers = re.findall(r"\d+", prompt)
        if len(numbers) < 2:
            return "Error: Please provide at least two numbers to add"
        total = sum(float(num) for num in numbers)
        return f"Result: {total}"

    return tools
```

What Happens Here

- `@self._tool(tools)` wraps each function definition.

- When `search` is defined, the decorator adds it to the dictionary:
  tools["search"] = search
- When `add_numbers` is defined, it is also added:

At the end:

```
self.tools == {
  "search": <function search>,
  "add_numbers": <function add_numbers>
}
```

🔹 Flowchart of the Call Sequence

```
Agent() created
  │
  └── __init__() runs
        │
        └── calls _initialize_tools()
              │
              ├── defines @self._tool(tools) search()
              │       └── decorator adds "search" → tools dict
              │
              ├── defines @self._tool(tools) add_numbers()
              │       └── decorator adds "add_numbers" → tools dict
              │
              └── returns tools dict → stored in self.tools
```

🔹 Why This is Useful

- **Cleaner Code:** No need to manually maintain a dictionary of tools.
- **Extensible:** To add a new tool, just define a function and decorate it.
- **Scalable:** Similar to how frameworks like Flask/FastAPI auto-register routes.

This approach makes the agent code **modular, extensible, and professional**.