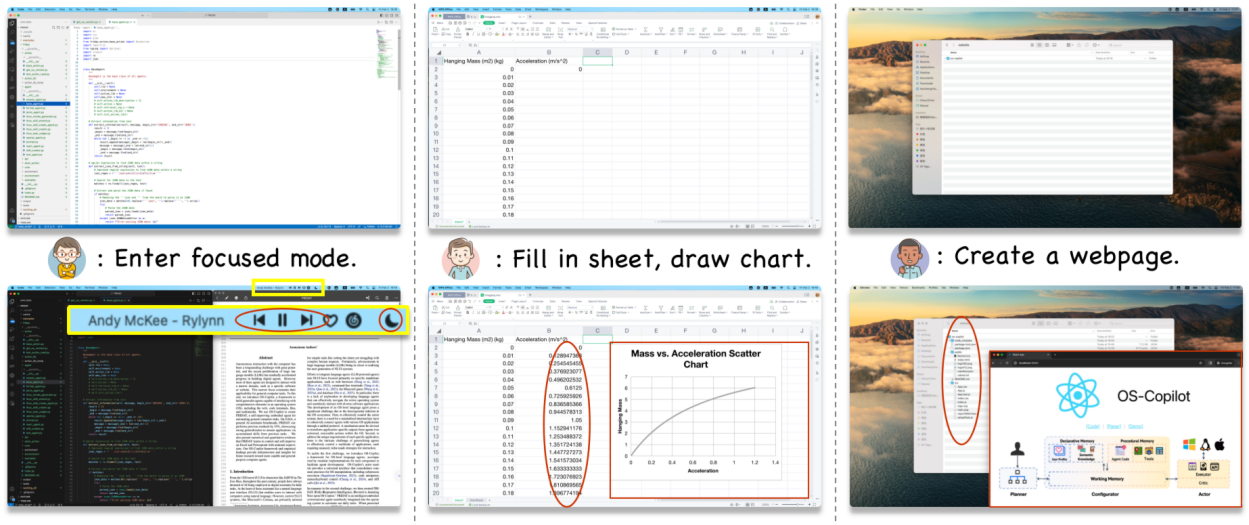
OS-Copilot

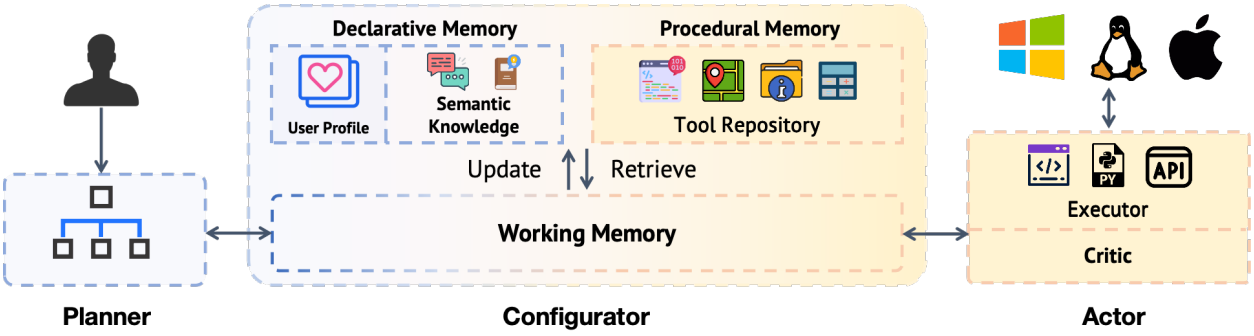
OS-Copilot

May 17, 2024

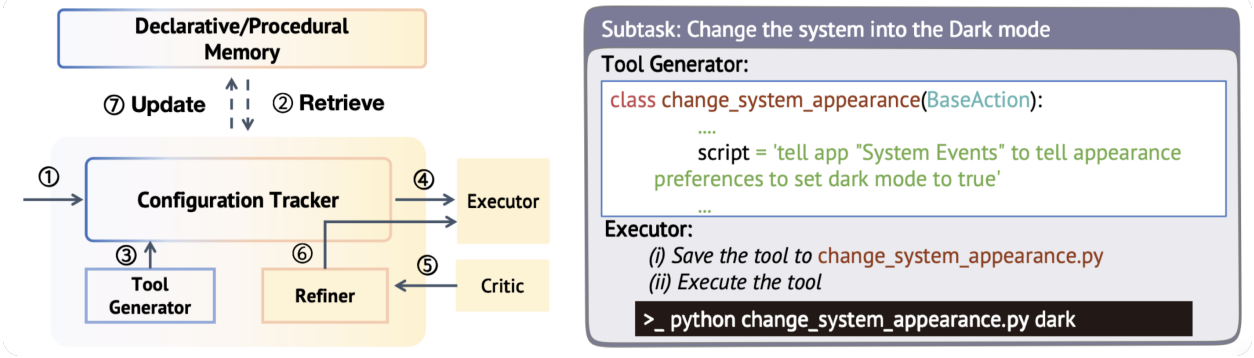
GETTING STARTED



OS-Copilot is a pioneering conceptual framework for building generalist computer agents on Linux and MacOS, which provides a unified interface for app interactions in the heterogeneous OS ecosystem.



Leveraging OS-Copilot, we built **FRIDAY**, a self-improving AI assistant capable of solving general computer tasks.



Project Homepage: OS-Copilot

CHAPTER

ONE

TUTORIALS

Level	Tutorial	Description
Beginner	Installation	Provides three methods to install FRIDAY: cloning from GitHub, development setup via pip install -e ., and direct pip installation.
Beginner	Getting Started	Demonstrates how to use FRIDAY with a quick_start.py script, covering module imports, configuration setup, and task execution.
Beginner	LightFriday	Demonstrates how to use the `LightFriday` agent to execute tasks by planning and executing code.
Intermediate	Adding Your Tools	Outlines the process for adding and removing tools to the FRIDAY.
Intermediate	Deploying API Services	Explains the deployment of API services for FRIDAY, including environment setup, configuring API tools, and launching the server.
Intermediate	Example: Automating Excel Tasks	Demonstrates automating Excel tasks with FRIDAY, including formula application and chart creation within an Excel sheet.
Intermediate	Enhancing FRIDAY with Self-Learning for Excel Task Automation	Showcases empowering FRIDAY with self-learning to autonomously learn and execute Excel file manipulations.
Advanced	Designing New API Tools	Guides on designing, integrating, and deploying custom API tools for FRIDAY to extend its functionalities.

CHAPTER

TWO

COMMUNITY

Join our community to connect with other enthusiasts, share your tools and demos, and collaborate on innovative projects. Stay engaged and get the latest updates by following us:

- **Discord**: Join our Discord server for real-time discussions, support, and to share your work with the community. Click here to join: Discord Server .
- **Twitter**: Follow us on Twitter @oscopilot for the latest news, updates, and highlights from our community.

CHAPTER

THREE

CONTRIBUTING

OS-Copilot thrives on community contributions, and we welcome involvement in any form. Whether it's adding new tools, fixing bugs, improving documentation, or sharing ideas, every contribution counts. Join our community to advance this exciting project together!

3.1 Ways to Contribute

- **Code:** Enhance OS-Copilot by adding new features, fixing bugs, or optimizing existing tools.
- **Documentation:** Help make OS-Copilot more accessible by improving or expanding our documentation.
- **Feedback and Ideas:** Share your insights and suggestions to make OS-Copilot even better.
- **Advocacy:** Spread the word about OS-Copilot and help grow our community.

CHAPTER

FOUR

CITATION

For more detailed information about OS-Copilot and FRIDAY, please refer to our latest research paper:

```
@misc{wu2024oscopilot,
   title={OS-Copilot: Towards Generalist Computer Agents with Self-Improvement},
   author={Zhiyong Wu and Chengcheng Han and Zichen Ding and Zhenmin Weng and Zhoumianze␣
↪Liu and Shunyu Yao and Tao Yu and Lingpeng Kong},
   year={2024},
   eprint={2402.07456},
   archivePrefix={arXiv},
   primaryClass={cs.AI}
}
```

4.1 Installation

There are three ways to install OS-Copilot: using *pip* directly, cloning the GitHub repository, or through the *pip install -e .* method for a development setup. Please follow the instructions below based on your preferred installation method.

1. **Clone the GitHub Repository** (if not already done):

   ```
   git clone https://github.com/OS-Copilot/OS-Copilot.git
   ```

2. **Navigate to the Repository Directory:**

   ```
   cd OS-Copilot
   ```

3. **Set Up Python Environment:** Ensure you have a version 3.10 or higher Python environment. You can create and activate this environment using the following commands, replacing OS-Copilot_env with your preferred environment name:

   ```
   conda create -n OS-Copilot_env python=3.10 -y
   conda activate OS-Copilot_env
   ```

3. **Install Dependencies:** Install the necessary dependencies.

   - Option 1(Recommended): Use pip to install the project in editable mode.

   ```
   pip install -e .
   ```

   - Option 2: You can install OS-Copilot directly using pip with the following command:

```
pip install os-copilot-agent
```

- Option 3: Install the necessary dependencies:

```
cd OS-Copilot
pip install -r requirements.txt
```

4. **Set OpenAI API Key:** Configure your OpenAI API key in the *.env* file and select the model you wish to use.

```
MODEL_NAME=""
OPENAI_API_KEY=""
```

4.2 Quick Start

The *quick_start.py* script is a simple way to start using FRIDAY AGENT. Here's a breakdown of what the script does and how to run it:

1. **Importing Modules:**

   The script begins by importing necessary modules from the *oscopilot* package:

```
from oscopilot import FridayAgent
from oscopilot import ToolManager
from oscopilot import FridayExecutor, FridayPlanner, FridayRetriever
from oscopilot.utils import setup_config, setup_pre_run
```

2. **Setting Up Configuration:**

   Next, the script sets up the configuration for running a task:

```
args = setup_config()
args.query = "Create a new folder named 'test_friday'"
```

   This sets a query for the FRIDAY AGENT to execute, which in this case is creating a new folder named 'test_friday'.

3. **Preparing and Running the Task:**

   After configuring the task, the script prepares it for execution and runs it:

```
task = setup_pre_run(args)
agent = FridayAgent(FridayPlanner, FridayRetriever, FridayExecutor, ToolManager,␣
↪config=args)
agent.run(task)
```

   This initializes the FRIDAY AGENT with specified planners, retrievers, and executors, then executes the task.

4.2.1 Running the Script

To run the *quick_start.py* script, simply execute the following command in your terminal:

```
python quick_start.py
```

Ensure that you are in the same directory as the *quick_start.py* file or provide the full path to the file.

Congratulations! You have now successfully run a task with FRIDAY~

4.3 Adding Your Tools

This tutorial will guide you through the process of adding a new tool to the FRIDAY platform. We will use a simple example tool, *create_folder.py*, to demonstrate the process.

4.3.1 Step 1: Clone the Repository

First, you need to clone the repository containing the tools (referred to as "gizmos" for FRIDAY). Use the following command to clone the repository:

```
git clone https://github.com/OS-Copilot/FRIDAY-Gizmos.git
```

After cloning, navigate into the *FRIDAY-Gizmos* directory:

```
cd FRIDAY-Gizmos
```

Choose any Python file that represents the tool code you wish to add. For this tutorial, we will use *Basic/create_folder.py* as an example.

4.3.2 Step 2: Add the Tool to FRIDAY

To add your chosen tool to FRIDAY's tool repository, run the *action_manager.py* script with the *–add* flag. You will need to provide the tool name and the path to the tool. Replace *[tool_name]* with the name you wish to give your tool and *[tool_path]* with the relative or absolute path to the tool file.

```
python oscopilot/tool_repository/manager/tool_manager.py --add --tool_name [tool_name] --
↪tool_path [tool_path]
```

Note:

- **[tool_name]:** A unique identifier for your tool within the FRIDAY ecosystem. It is recommended to keep the *tool_name* the same as the class name for consistency.
- **[tool_path]:** The path to the Python file you're adding, relative to the FRIDAY installation directory or an absolute path.

4.3.3 Example: Adding a Tool

If we're adding the *create_folder.py* tool located in the *Basic* directory and we wish to name it *create_folder*, the command would look like this:

```
python oscopilot/tool_repository/manager/tool_manager.py --add --tool_name create_folder␣
↪--tool_path Basic/create_folder.py
```

4.3.4 Removing a Tool from FRIDAY

In addition to adding new tools to FRIDAY, you might find yourself in a situation where you need to remove an existing tool from the tool repository. Whether it's for updating purposes or simply because the tool is no longer needed, removing a tool is straightforward.

To remove a tool from FRIDAY's tool repository, you can use the *action_manager.py* script with the *–delete* flag. You will need to specify the name of the tool you wish to remove using the *–tool_name* option. Replace *[tool_name]* with the unique identifier for your tool within the FRIDAY ecosystem.

```
python oscopilot/tool_repository/manager/tool_manager.py --delete --tool_name [tool_name]
```

Note:

- **[tool_name]:** The unique identifier of the tool you want to remove from FRIDAY. Ensure that you provide the exact name as registered in FRIDAY's tool repository to avoid any errors.

4.3.5 Example: Removing a Tool

If you wish to remove a tool named *create_folder*, the command would look like this:

```
python oscopilot/tool_repository/manager/tool_manager.py --delete --tool_name create_
↪folder
```

This command will remove the *create_folder* tool from FRIDAY's repository, effectively making it unavailable for future use within the ecosystem. It's important to note that removing a tool is a permanent action, so make sure you've backed up any necessary code or information related to the tool before proceeding with the deletion.

4.3.6 Tool Code Example

To add a tool to FRIDAY, the tool code must follow a specific structure. Below is an example of a tool code that creates a folder either in a specified working directory or in the default working directory. This example adheres to the required structure for FRIDAY tools:

```
import os

def create_folder(working_directory, folder_name):
    """
    Create a folder under the specified working directory or the default working␣
↪directory.

    Args:
```

(continues on next page)

(continued from previous page)

```
    working_directory (str): The path of the working directory. If not provided, the␣
↪default working directory will be used.
    folder_name (str): The name of the folder to be created. Default is 'myfold'.

    Returns:
    None
    """
    # Check if the working_directory is provided, if not, use the default working␣
↪directory
    if working_directory:
        os.chdir(working_directory)

    # Create the folder
    os.makedirs(folder_name)
```

4.3.7 Tool Requirements

To ensure seamless integration into FRIDAY's tool repository, your tool code must adhere to the following format, consistent with the example tools provided:

1. **Python Package**: Import any additional Python packages necessary for your tool's functionality.

   ```
   import os  # Example of importing another necessary package
   ```

2. **Function Naming**: The name of the function should be consistent with the tool's file name to maintain clarity and ease of identification within the tool repository.

3. **Function Comments**: The funtion should include detailed explanations of the input and output parameters to guide the user.

   ```
   def tool_name(parameter1, parameter2=None, *args, **kwargs):
       """
       Detailed explanation of what this function does, its parameters, and what it␣
   ↪returns.
       """
   ```

By following these specific requirements, you ensure that your tool can be effectively integrated and utilized within the FRIDAY ecosystem. This consistency not only aids in tool management but also enhances the user experience by providing a standardized approach to tool development.

4.3.8 Conclusion

With the provided guidelines and example, you are now equipped to extend FRIDAY's capabilities by adding new tools. By adhering to the structure and requirements specified for FRIDAY tools, you ensure that your tools can be effectively utilized within the FRIDAY ecosystem.

Remember, the power of FRIDAY lies in its flexibility and the collaborative efforts of its community. Your contributions help make FRIDAY more versatile and powerful.

We welcome you to submit your tools to the FRIDAY Gizmos repository at https://github.com/OS-Copilot/FRIDAY-Gizmos. Sharing your work enables others in the community to benefit from your contributions and further enhances the FRIDAY platform.

Happy coding!

4.4 Deploying API Services

This tutorial guides you through deploying API services that FRIDAY can utilize to enhance its functionality. We provide several API tools, including audio2text, bing search, image search, web_loader, image_caption, and wolfram_alpha, all located within *oscopilot/tool_repository/api_tools*.

4.4.1 Configuring the Environment

1. **.env File Configuration**:

   Your *.env* file contains essential configurations for the API services. The following variables must be set:

   - `MODEL_NAME`: Should already be set during the installation process. Example: `"gpt-4-0125-preview"`.
   - `OPENAI_API_KEY` and `OPENAI_ORGANIZATION`: Also set during installation.
   - `API_BASE_URL`: The deployment address of the API service. For local deployment, use `"http://127.0.0.1:8079"`.
   - `BING_SUBSCRIPTION_KEY` and `BING_SEARCH_URL`: Required for using bing search-related services. Example URL: `"https://api.bing.microsoft.com/v7.0/search"`.
   - `WOLFRAMALPHA_APP_ID`: Necessary if you intend to use the wolfram_alpha tool.

   Fill these in accordingly based on the services you plan to use.

4.4.2 Configuring API Tools

2. **Selecting Required API Tools**:

   In the *oscopilot/tool_repository/manager/api_server.py* file, you will configure which API tools FRIDAY will utilize. This is done by setting up the *services* and *server_list* variables.

   - The `services` dictionary includes all available API tools that FRIDAY can use. Each key represents the service name, and the value is the corresponding router object.
   - The `server_list` array specifies which of these services you wish to activate for the current deployment. This allows for flexible configuration depending on the needs of your specific environment or application.

   Here is how you can specify your configuration:

```
services = {
    "bing": bing_router,  # bing_search, image_search, and web_loader
    "audio2text": audio2text_router,
    "image_caption": image_caption_router,
    "wolfram_alpha": wolfram_alpha_router
}

server_list = ["bing"]
```

   In this example, we have included several services in the `services` dictionary, making them available for FRIDAY. However, by placing only “bing” in the `server_list`, we are specifically activating the Bing services for use, including bing_search, image_search and web_loader. This demonstrates how to selectively enable certain API tools based on your requirements.

4.4.3 Launching the API Server

3. **Starting the Service**:

   To start the API service, run the following command:

```
python oscopilot/tool_repository/manager/api_server.py
```

   Successful startup messages will look like this:

```
INFO:     Started server process [17709]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8079 (Press CTRL+C to quit)
Incoming request: GET http://127.0.0.1:8079/tools/bing/searchv2
Outgoing response: 200
INFO:     127.0.0.1:52324 - "GET /tools/bing/searchv2 HTTP/1.1" 200 OK
```

4.4.4 Updating API Documentation

4. **Update the OpenAPI Documentation**:

   After the service is running, navigate to *http://localhost:8079/openapi.json* in your web browser. This URL hosts the auto-generated OpenAPI documentation for your API services. (Remember to replace the IP address if your service is not deployed locally.)

   Here is an example of what the OpenAPI documentation might look like:

   Copy the content displayed at this URL to the *oscopilot/tool_repository/manager/openapi.json* file in your project directory. This step ensures that FRIDAY's API server has the latest documentation regarding the available API services.

4.4.5 Testing the API Tools

5. **Verifying Functionality**:

   Test the deployed API tools by running a sample query with *run.py*. For example:

```
python quick_start.py --query 'Search the information of OpenAI'
```

   If everything is configured correctly, FRIDAY should utilize the deployed API services to complete the task.

4.4.6 Conclusion

You have successfully deployed API services for FRIDAY, enhancing its capabilities with additional tools. By following these steps, you can integrate a wide range of functionalities into FRIDAY, making it an even more powerful assistant.

4.5 Designing New API Tools

After deploying existing API services as described in the previous section, this part will focus on how to develop and deploy a new API service for FRIDAY. By creating custom API tools, you can extend FRIDAY's capabilities to suit your specific needs.

4.5.1 Creating a New API Tool

1. **Setting Up the API Tool**:

   Begin by creating a new folder for your API tool within *oscopilot/tool_repository/api_tools*. Inside this folder, create your tool file and write the API tool code. You can refer to the FastAPI documentation (https://fastapi.tiangolo.com/reference/fastapi/) and examples in the *oscopilot/tool_repository/api_tools* directory for guidance on coding your API tool.

   Consider the following example when designing your API endpoint:

```
@router.get("/tools/bing/searchv2", summary="Execute Bing Search - returns top web␣
↪snippets related to the query. Avoid using complex filters like 'site:'. For␣
↪detailed page content, further use the web browser tool.")
async def bing_search_v2(item: QueryItemV2):
    try:
        if item.top_k == None:
            item.top_k = 5
        search_results = bing_api_v2.search(item.query, item.top_k)
    except RuntimeError as e:
        raise HTTPException(status_code=500, detail=str(e))
    return search_results
```

   Ensure to include the *summary* parameter in *router.get*, providing a detailed description of the API tool's functionality, which FRIDAY will use to determine the tool's applicability for tasks.

4.5.2 Integrating the API Tool

2. **Registering the New API Tool**:

   Update *oscopilot/tool_repository/manager/api_server.py* with the new API tool's information. Add import statements and update the *services* and *server_list* accordingly.

   Example code snippet:

```
from oscopilot.tool_repository.api_tools.new_api.new_api_service import router as␣
↪new_api_router

services = {
    "bing": bing_router,  # bing_search, image_search, and web_loader
    "audio2text": audio2text_router,
```

(continues on next page)

(continued from previous page)

```
    "image_caption": image_caption_router,
    "wolfram_alpha": wolfram_alpha_router,
    "new_api": new_api_router
}

server_list = ["bing", "new_api"]
```

4.5.3 Launching the Service

3. **Starting the API Service**:

   Run the *api_server.py* file to launch the service:

```
python oscopilot/tool_repository/manager/api_server.py
```

   Successful launch messages should resemble the following:

```
INFO:     Started server process [17709]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:8079 (Press CTRL+C to quit)
```

4.5.4 Updating OpenAPI Documentation

4. **Updating API Documentation**:

   Navigate to *http://localhost:8079/openapi.json* (adjust the IP if necessary) and overwrite the content in *oscopilot/tool_repository/manager/openapi.json* with the content from this URL.

4.5.5 Verifying the API Tool

5. **Testing the API Tool**:

   Verify the new API tool's functionality by executing a test query with *run.py*:

```
python quick_start.py --query 'Your test query here'
```

4.5.6 Conclusion

By following these steps, you have successfully designed, integrated, and deployed a new API tool for FRIDAY. This customization allows FRIDAY to perform tasks tailored to your specific requirements, enhancing its overall utility.

4.6 Example: Automating Excel Tasks with FRIDAY

In this tutorial, we'll showcase how FRIDAY can be utilized for manipulating Excel files, automating tasks that would otherwise be tedious and time-consuming. We'll take on a specific task involving Excel file operations as an example.

4.6.1 Task Overview

You are required to perform several tasks related to Excel manipulation involving an experiment's data recorded in a sheet named "Sheet1". The tasks include:

- Applying a formula across rows in column B.
- Creating a scatter chart within "Sheet1" that plots acceleration (y-axis) against the hanging mass (x-axis).
- Labeling the chart axes with the appropriate column headers.

The Excel file for this task is located in *working_dir* and is named "Dragging.xlsx".

4.6.2 Step-by-Step Guide

1. **Preparing the Necessary Tools**:

   Locate the following tools within *FRIDAY-Gizmos/Excel* directory:

   - `apply_formula_to_column_B`
   - `create_new_sheet_for_chart`
   - `insert_scatter_chart`
   - `read_excel_sheet`

   Follow the steps outlined in the "Adding Your First Tool" tutorial to add these four tools to FRIDAY's tool repository.

2. **Executing the Task**:

   To perform the Excel manipulation task, run the following command in your terminal. This command instructs FRIDAY to apply the necessary operations on the "Dragging.xlsx" file based on the provided task description.

```
python quick_start.py --query "You need to do some tasks related to excel␣
↪manipulation.\n My sheet records data from an experiment where one hanging block␣
↪(m2) drags a block (m1=0.75 kg) on a frictionless table via a rope around a␣
↪frictionless and massless pulley. It has a sheet called Sheet1. \n Your task is:␣
↪Fill out the rest rows in column B using the formula in B2. Create a scatter␣
↪chart in Sheet1 with acceleration on the y-axis and the hanging mass on the x-
↪axis. Add the corresponding column headers as the axis labels. \n You should␣
↪complete the task and save the result directly in this excel file." --query_file_
↪path "working_dir/Dragging.xlsx"
```

4.6.3 Conclusion

Upon completion, the "Dragging.xlsx" file will have the specified formula applied across rows in column B, and a scatter chart will be created in "Sheet1" as requested. This example illustrates how FRIDAY can automate complex Excel operations, saving time and reducing the potential for manual errors.

Ensure to adjust file paths and names as per your specific setup. This tutorial demonstrates the power and flexibility of FRIDAY in handling and automating tasks within Excel, showcasing its capability to significantly streamline such processes.

4.7 Enhancing FRIDAY with Self-Learning for Excel Task Automation

In this tutorial, we will explore how FRIDAY's self-learning feature enables it to autonomously learn and execute tasks involving Excel file manipulation, which were initially beyond its capability. We will specifically focus on a task from the SheetCopilot dataset and observe how FRIDAY evolves to complete it successfully using the *openpyxl* library.

4.7.1 Task Overview

You will undertake a specific task (task #9) from the SheetCopilot dataset, which involves manipulating an Excel file. The task is **"Copy the 'Product' column from 'Sheet1' to a new sheet named 'Sheet2', and then sort the 'Product' column in 'Sheet2' in ascending order."**

4.7.2 Initial Attempt

1. **Running the Task**:

   Execute the following command in your terminal to run task #9 from the SheetCopilot dataset:

   ```
   python examples/SheetCopilot/run_sheet_task.py --sheet_task_id 9
   ```

   The *run_sheet_task.py* script serves as the interface for FRIDAY to interact with tasks defined in the SheetCopilot dataset. Below is a brief explanation of the script's content:

   - Module Imports and Configuration Setup:

     ```
     from oscopilot import FridayAgent
     from oscopilot import FridayExecutor, FridayPlanner,␣
     ↪FridayRetriever, ToolManager
     from oscopilot.utils import setup_config, SheetTaskLoader
     ```

   - Loading Tasks:

     The script initializes the configuration and loads the task based on the provided task ID using SheetTaskLoader.

     ```
     args = setup_config()
     sheet_task_loader = SheetTaskLoader("examples/SheetCopilot/sheet_
     ↪task.jsonl")
     ```

   - FRIDAY Agent Initialization:

     An agent is initialized with components such as the Planner, Retriever, Executor, and Tool Manager, configured with the loaded arguments.

```
agent = FridayAgent(FridayPlanner, FridayRetriever,␣
↪FridayExecutor, ToolManager, config=args)
```

- Task Execution:

  If a specific task ID is provided, the script fetches and runs that task. Otherwise, it loads and executes each task in the dataset sequentially.

```
if args.sheet_task_id:
    task = sheet_task_loader.get_data_by_task_id(args.sheet_task_
↪id)
    agent.run(task)
else:
    task_lst = sheet_task_loader.load_sheet_task_dataset()
    for task_id, task in enumerate(task_lst):
    args.sheet_task_id = task_id
    agent.run(task)
```

However, you'll notice that FRIDAY **is unable to** complete the task due to lacking specific tools for Excel manipulation.

4.7.3 Introducing Self-Learning

2. **Enabling FRIDAY to Learn**:

   To overcome this limitation, we introduce FRIDAY to a self-learning module that allows it to explore and learn from the *openpyxl* library, thereby acquiring new tools for Excel file operations.

   Run the self-learning command:

```
python course_learning.py --software_name Excel --package_name openpyxl --
↪demo_file_path working_dir/Invoices.xlsx
```

   This command directs FRIDAY to learn how to manipulate Excel files using the *openpyxl* library. Below is a brief overview of the *course_learning.py* script's functionality:

   - Import Statements and Configuration Setup:

```
from oscopilot import FridayAgent, FridayExecutor, FridayPlanner,␣
↪FridayRetriever, SelfLearner, SelfLearning, ToolManager, TextExtractor
from oscopilot.utils import setup_config
```

   - Initialization and Configuration Extraction:

   The script begins by setting up the configuration and extracting parameters for the software name, package name, and a demo file path.

```
args = setup_config()
software_name = args.software_name
package_name = args.package_name
demo_file_path = args.demo_file_path
```

   - FRIDAY Agent and Self-Learning Module Initialization:

   A FRIDAY agent is initialized with components such as the Planner, Retriever, Executor, and Tool Manager. The SelfLearning module is then initialized with the agent, allowing it to engage in self-learning activities.

```
friday_agent = FridayAgent(FridayPlanner, FridayRetriever, FridayExecutor,␣
↪ToolManager, config=args)
self_learning = SelfLearning(friday_agent, SelfLearner, ToolManager, args,␣
↪TextExtractor)
```

- Self-Learning Process:

The SelfLearning module embarks on exploring the openpyxl library, utilizing the provided demo file as a learning resource.

```
self_learning.self_learning(software_name, package_name, demo_file_path)
```

Through this exploratory process, FRIDAY can learn various tools such as *check_openpyxl_installed*, *read_excel_contents*, *filter_product_data*, and *export_filtered_data*, among others.

Note: The tools learned through self-learning have a degree of randomness.

4.7.4 Verifying the Learning Outcome

3. **Re-running the Task**:

   After the self-learning process, rerun the initial task to verify the effectiveness of the self-learning module:

```
python examples/SheetCopilot/run_sheet_task.py --sheet_task_id 9
```

   This time, FRIDAY will successfully complete the task, demonstrating the acquired ability to manipulate Excel files through the learned tools.

4.7.5 Conclusion

This tutorial showcased the innovative self-learning feature of FRIDAY, which enables it to autonomously expand its toolset and adapt to tasks it was initially unable to perform. By engaging in self-learning with the *openpyxl* library, FRIDAY demonstrated a significant improvement in handling Excel file operations, affirming the effectiveness and potential of self-learning in AI agents.

This process highlights FRIDAY's capability to evolve and adapt, making it a powerful tool for automating a wide range of tasks, including complex file manipulations.

4.8 LightFriday: A Lightweight Agent for Task Execution

In this tutorial, we will explore how to use the `LightFriday` class, a lightweight agent designed to autonomously execute code and complete tasks. We will walk through the core components and functionality of `LightFriday`, demonstrating how it leverages code execution and iterative planning to achieve task completion.

4.8.1 Task Overview

We will undertake a specific task, **"Plot AAPL and META's normalized stock prices,"** using the `LightFriday` agent. The agent will execute code step-by-step to complete this task.

4.8.2 Initial Setup

1. **Setting Up the Configuration**:

   Before we initialize the `LightFriday` agent, we need to set up the configuration. This involves setting up the system environment and loading the task details.

```
from oscopilot.utils import setup_config, setup_pre_run

args = setup_config()
if not args.query:
    args.query = "Plot AAPL and META's normalized stock prices"
task = setup_pre_run(args)
```

4.8.3 Core Component: `LightFriday`

The `LightFriday` class is the core component of this tutorial. It inherits from the `BaseModule` class and is responsible for planning, executing code, and iterating until the task is complete.

1. **Initialization**:

The `LightFriday` class is initialized with the following parameters:

```
class LightFriday(BaseModule):
    def __init__(self, args):
        super().__init__()
        self.args = args
```

2. **Execution Tool**:

The `execute_tool` method executes the given code in the specified language and returns the execution result or error.

```
def execute_tool(self, code, lang):
    state = self.environment.step(lang, code)
    return_info = ''
    if state.result != None and state.result.strip() != '':
        return_info = '**Execution Result** :' + state.result.strip()
    if state.error != None and state.error.strip() != '':
        return_info = '\n**Execution Error** :' + state.error.strip()
    return return_info.strip()
```

3. **Main Functionality**:

The `run` method is the heart of the `LightFriday` agent. It involves the following steps:

- **System Prompt**: Defines the system prompt with instructions for `LightFriday`.

```
light_planner_sys_prompt = '''You are Light Friday, a world-class␣
↪programmer that can complete any goal by executing code...
'''
```

- **User Prompt**: Provides details about the user's system and the task to be completed.

```
light_planner_user_prompt = '''
User's information are as follows:
System Version: {system_version}
Task: {task}
Current Working Directiory: {working_dir}'''.format(system_version=self.
↪system_version, task=task, working_dir=self.environment.working_dir)
```

- **Message Loop**: Iteratively interacts with the language model, executing code and refining the plan based on the results.

```
message = [
    {"role": "system", "content": light_planner_sys_prompt},
    {"role": "user", "content": light_planner_user_prompt},
]

while True:
    response = send_chat_prompts(message, self.llm)
    rich_print(response)
    message.append({"role": "system", "content": response})

    code, lang = extract_code(response)
    if code:
        result = self.execute_tool(code, lang)
        rich_print(result)
    else:
        result = ''

    if result != '':
        light_exec_user_prompt = 'The result after executing the code:
↪{result}'.format(result=result)
        message.append({"role": "user", "content": light_exec_user_prompt})
    else:
        message.append({"role": "user", "content": "Please continue. If all␣
↪tasks have been completed, reply with 'Execution Complete'. If you␣
↪believe subsequent tasks cannot continue, reply with 'Execution␣
↪Interrupted', including the reasons why the tasks cannot proceed, and␣
↪provide the user with some possible solutions."})

    if 'Execution Complete' in response or 'Execution Interrupted' in␣
↪response:
        break
```

4.8.4 Running the Task

To run the task using `LightFriday`, we initialize the agent and call the `run` method with the task details:

```
light_friday = LightFriday(args)
light_friday.run(task)
```

4.8.5 Conclusion

This tutorial demonstrated how to use the `LightFriday` class to execute tasks by iteratively planning and executing code. By leveraging the `LightFriday` agent, complex tasks can be broken down into manageable steps, and code can be executed step-by-step to achieve the desired outcomes.

This process showcases `LightFriday`'s capability to adapt and evolve, making it a powerful tool for automating a wide range of tasks.

4.9 FridayAgent

4.9.1 Base Agent

class oscopilot.agents.base_agent.**BaseAgent**

Bases: `object`

BaseAgent serves as the foundational class for all agents types within the system.

This class initializes the core attributes common across different agents, providing a unified interface for further specialization. Attributes include a language learning model, the execution environments, an action library, and a maximum iteration limit for agents operations.

llm

Placeholder for a language learning model, initialized as None.

environments

The execution environments for the agents, initialized as None.

action_lib

A library of actions available to the agents, initialized as None.

max_iter

The maximum number of iterations the agents can perform, initialized as None.

extract_information(*message*, *begin_str='[BEGIN]'*, *end_str='[END]'*)

Extracts substrings from a message that are enclosed within specified begin and end markers.

Parameters

- **message** (*str*) – The message from which information is to be extracted.
- **begin_str** (*str*) – The marker indicating the start of the information to be extracted.
- **end_str** (*str*) – The marker indicating the end of the information to be extracted.

Returns

A list of extracted substrings found between the begin and end markers.

Return type
list[str]

extract_json_from_string(*text*)

Identifies and extracts JSON data embedded within a given string.

This method searches for JSON data within a string, specifically looking for JSON blocks that are marked with `json` notation. It attempts to parse and return the first JSON object found.

Parameters
text (*str*) – The text containing the JSON data to be extracted.

Returns
The parsed JSON data as a dictionary if successful. str: An error message indicating a parsing error or that no JSON data was found.

Return type
dict

4.9.2 Friday Agent

class oscopilot.agents.friday_agent.**FridayAgent**(*planner*, *retriever*, *executor*, *Tool_Manager*, *config*)

Bases: *BaseAgent*

A FridayAgent orchestrates the execution of tasks by integrating planning, retrieving, and executing strategies.

This agent is designed to process tasks, manage errors, and refine strategies as necessary to ensure successful task completion. It supports dynamic task planning, information retrieval, execution strategy application, and employs a mechanism for self-refinement in case of execution failures.

executing(*tool_name*, *original_task*)

Executes a given sub-task as part of the task execution process, handling different types of tasks including code execution, API calls, and question-answering.

Parameters

- **tool_name** (*str*) – The name of the tool associated with the sub-task.
- **original_task** (*object*) – The original high-level task that has been decomposed into sub-tasks.

Returns
The state of execution for the sub-task, including the result, any errors encountered, and additional execution-related information.

Return type
ExecutionState

The method dynamically adapts the execution strategy based on the type of sub-task, utilizing the executor component for code execution, API interaction, or question-answering as appropriate.

judging(*tool_name*, *state*, *code*, *description*)

Evaluates the execution of a tool based on its execution state and the provided code and description, determining whether the tool's execution was successful or requires amendment.

Parameters

- **tool_name** (*str*) – The name of the tool being judged.
- **state** (ExecutionState) – The current execution state of the tool, including results and error information.

- **code** (*str*) – The source code associated with the tool's execution.
- **description** (*str*) – A description of the tool's intended functionality.

Returns
An object encapsulating the judgement on the tool's execution, including whether it needs repair, a critique of the execution, and an optional error type and reasoning for the judgement.

Return type
JudgementResult

This method assesses the correctness of the executed code and its alignment with the expected outcomes, guiding potential repair or amendment actions.

planning(*task*)

Decomposes a given high-level task into a list of sub-tasks by retrieving relevant tool names and descriptions, facilitating structured execution planning.

Parameters
task (*object*) – The high-level task to be planned and executed.

Returns
A list of sub-tasks generated by decomposing the high-level task, intended for sequential execution to achieve the task's goal.

Return type
list

This method leverages the retriever component to fetch information relevant to the task, which is then used by the planner component to decompose the task into manageable sub-tasks.

repairing(*tool_name*, *code*, *description*, *state*, *critique*, *status*)

Attempts to repair the execution of a tool by amending its code based on the critique received and the current execution state, iterating until the code executes successfully or reaches the maximum iteration limit.

Parameters

- **tool_name** (*str*) – The name of the tool being repaired.
- **code** (*str*) – The current code of the tool that requires repairs.
- **description** (*str*) – A description of the tool's intended functionality.
- **state** (ExecutionState) – The current execution state of the tool, including results and error information.
- **critique** (*str*) – Feedback on the tool's last execution attempt, identifying issues to be addressed.
- **status** (*str*) – Three status types: 'Amend', 'Complete', and 'Replan'.

Returns
An object encapsulating the result of the repair attempt, including whether the task has been completed successfully, the amended code, critique, execution score, and the execution result.

Return type
RepairingResult

The method iterates, amending the tool's code based on feedback until the code executes correctly or the maximum number of iterations is reached. It leverages the executor component for amending the code and re-evaluating its execution.

replanning(*tool_name*, *reasoning*)

Initiates the replanning process for a task based on new insights or failures encountered during execution, aiming to adjust the plan to better achieve the task goals.

Parameters

- **tool_name** (*str*) – The name of the tool related to the task that requires replanning.
- **reasoning** (*str*) – The rationale behind the need for replanning, often based on execution failures or updated task requirements.

Returns

An updated list of sub-tasks after the replanning process, intended for sequential execution to complete the task.

Return type

list

This method identifies alternative or additional tools and their descriptions based on the provided reasoning, updating the task plan accordingly.

reset_inner_monologue()

run(*task*)

Executes the given task by planning, executing, and refining as needed until the task is completed or fails.

Parameters

query (*object*) – The high-level task to be executed.

No explicit return value, but the method controls the flow of task execution and may exit the process in case of irreparable failures.

self_refining(*tool_name*, *execution_state:* ExecutionState)

Analyzes and potentially refines the execution of a tool based on its current execution state. This can involve replanning or repairing the execution strategy based on the analysis of execution errors and outcomes.

Parameters

- **tool_name** (*str*) – The name of the tool being executed.
- **execution_state** (ExecutionState) – The current state of the tool's execution, encapsulating all relevant execution information including errors, results, and codes.

Returns

- isTaskCompleted (bool): Indicates whether the task associated with the tool has been successfully completed.
- isReplan (bool): Indicates whether a replan is required due to execution state analysis.

Return type

tuple

The method decides on the next steps by analyzing the type of error (if any) and the execution results, aiming to either complete the task successfully or identify the need for further action, such as replanning.

4.9.3 Self Learning

class oscopilot.agents.self_learning.**SelfLearning**(*agent*, *learner*, *tool_manager*, *config*, *text_extractor=None*)

Bases: `object`

A class designed to facilitate self-learning for software-related topics by automatically generating and engaging with courses based on provided software and package information.

config

Configuration settings for the learning process.

Type
dict

agent

An external agent or tool that interacts with the learning content or environment.

Type
object

learner

A learner object that is responsible for designing the course.

Type
object

tool_manager

Manages the tools required for course creation and learning.

Type
object

text_extractor

An optional component responsible for extracting text from files.

Type
object, optional

course

A dictionary to store course details.

Type
dict

continuous_learning(*software_name*, *package_name*, *demo_file_path=None*)

Implements a continuous learning process that updates and applies new courses based on a designed curriculum.

Parameters

- **software_name** (*str*) – Name of the software being learned.
- **package_name** (*str*) – Name of the package within the software.
- **demo_file_path** (*str, optional*) – Path to a demo file used for extracting text content. Defaults to None.

Returns
This method does not return anything but updates internal states and possibly external resources.

Return type
None

`course_design`(*software_name*, *package_name*, *demo_file_path*, *file_content=None*)

Designs a course based on the provided software and package name, using content extracted from a demo file.

Parameters

- **software_name** (*str*) – The name of the software for which the course is designed.
- **package_name** (*str*) – The package name related to the software.
- **demo_file_path** (*str*) – Path to the demo file used as content for the course.
- **file_content** (*str, optional*) – Content of the demo file to be included in the course.

Returns
The designed course as a dictionary.

Return type
dict

`learn_course`(*course*)

Triggers the learning of the designed course using the configured agent.

Parameters
course (*dict*) – The course dictionary containing lesson details to be learned.

Returns
None.

`self_learning`(*software_name*, *package_name*, *demo_file_path*)

Initiates the self-learning process by designing a course and triggering the learning mechanism.

Parameters

- **software_name** (*str*) – The name of the software for which the course is being designed.
- **package_name** (*str*) – The name of the software package related to the course.
- **demo_file_path** (*str*) – The file path of a demo or example file that is relevant to the course content.

Returns
None.

`text_extract`(*demo_file_path*)

Extracts text from the specified demo file path using the configured text extractor.

Parameters
demo_file_path (*str*) – The path to the demo file from which content needs to be extracted.

Returns
The extracted content from the file.

Return type
str

4.9.4 Base Module

class oscopilot.modules.base_module.**BaseModule**

Bases: object

extract_information(*message*, *begin_str='[BEGIN]'*, *end_str='[END]'*)

Extracts substrings from a message that are enclosed within specified begin and end markers.

Parameters

- **message** (*str*) – The message from which information is to be extracted.
- **begin_str** (*str*) – The marker indicating the start of the information to be extracted.
- **end_str** (*str*) – The marker indicating the end of the information to be extracted.

Returns
A list of extracted substrings found between the begin and end markers.

Return type
list[str]

extract_json_from_string(*text*)

Identifies and extracts JSON data embedded within a given string.

This method searches for JSON data within a string, specifically looking for JSON blocks that are marked with `json` notation. It attempts to parse and return the first JSON object found.

Parameters
text (*str*) – The text containing the JSON data to be extracted.

Returns
The parsed JSON data as a dictionary if successful. str: An error message indicating a parsing error or that no JSON data was found.

Return type
dict

extract_list_from_string(*text*)

Extracts a list of task descriptions from a given string containing enumerated tasks. This function ensures that only text immediately following a numbered bullet is captured, and it stops at the first newline character or at the next number, preventing the inclusion of subsequent non-numbered lines or empty lines.

Parameters: text (str): A string containing multiple enumerated tasks. Each task is numbered and followed by its description.

Returns: list[str]: A list of strings, each representing the description of a task extracted from the input string.

4.9.5 PlanningModule

class oscopilot.modules.planner.friday_planner.**FridayPlanner**(*prompt*)

Bases: *BaseModule*

A planning module responsible for decomposing complex tasks into manageable subtasks, replanning tasks based on new insights or failures, and managing the execution order of tasks.

The *FridayPlanner* uses a combination of tool descriptions, environmental state, and language learning models to dynamically create and adjust plans for task execution. It maintains a tool graph to manage task dependencies and execution order, ensuring that tasks are executed in a sequence that respects their interdependencies.

add_new_tool(*new_task_json*, *current_task*)

Incorporates a new tool into the existing tool graph based on its dependencies.

This method processes a JSON object representing a new task, including its name, description, type, and dependencies, and adds it to the tool graph. It also updates the tool nodes to reflect this new addition. Finally, it appends the last new task to the list of dependencies for the specified current task.

Parameters

- **new_task_json** (*dict*) – A JSON object containing the new task's details.
- **current_task** (*str*) – The name of the current task to which the new task's dependencies will be added.

Side Effects:

Updates the tool graph and nodes to include the new tool and its dependencies. Modifies the dependencies of the current task to include the new tool.

create_tool_graph(*decompose_json*)

Constructs an tool graph based on dependencies specified in the given JSON.

This method takes a JSON object containing task information and dependencies, and constructs an tool graph. Each task is added as a node in the graph, with directed edges representing task dependencies. The method updates the class's internal structures to reflect this graph, including tool nodes and their relationships, as well as the overall number of tools.

Parameters

decompose_json (*dict*) – A JSON object where each key is an tool name, and the value is a dictionary containing the tool's name, description, type, and dependencies.

Side Effects:

Modifies the internal state by updating *tool_num*, *tool_node*, and *tool_graph* to reflect the newly created tool graph.

decompose_task(*task*, *tool_description_pair*)

Decomposes a complex task into manageable subtasks and updates the tool graph.

This method takes a high-level task and an tool-description pair, and utilizes the environments's current state to format and send a decomposition request to the language learning model. It then parses the response to construct and update the tool graph with the decomposed subtasks, followed by a topological sort to determine the execution order.

Parameters

- **task** (*str*) – The complex task to be decomposed.
- **tool_description_pair** (*dict*) – A dictionary mapping tool names to their descriptions.

Side Effects:

Updates the tool graph with the decomposed subtasks and reorders tools based on dependencies through topological sorting.

get_pre_tasks_info(*current_task*)

Retrieves information about the prerequisite tasks for a given current task.

This method collects and formats details about all tasks that are prerequisites for the specified current task. It extracts descriptions and return values for each prerequisite task and compiles this information into a JSON string.

Parameters
current_task (*str*) – The name of the task for which prerequisite information is requested.

Returns
A JSON string representing a dictionary, where each key is a prerequisite task's name, and the value is a dictionary with the task's description and return value.

get_tool_list(*relevant_tool=None*)

Retrieves a list of all tools or a subset of relevant tools, including their names and descriptions.

This method fetches tool descriptions from the tool library. If a specific set of relevant tools is provided, it filters the list to include only those tools. The resulting list (or the full list if no relevant tools are specified) is then returned in JSON format.

Parameters
relevant_tool (*list, optional*) – A list of tool names to filter the returned tools by. If None, all tools are included. Defaults to None.

Returns
A JSON string representing a dictionary of tool names to their descriptions. The dictionary includes either all tools from the library or only those specified as relevant.

replan_task(*reasoning*, *current_task*, *relevant_tool_description_pair*)

Replans the current task by integrating new tools into the original tool graph.

Given the reasoning for replanning and the current task, this method generates a new tool plan incorporating any relevant tools. It formats a replanning request, sends it to the language learning model, and integrates the response (new tools) into the existing tool graph. The graph is then updated to reflect the new dependencies and re-sorted topologically.

Parameters

- **reasoning** (*str*) – The reasoning or justification for replanning the task.
- **current_task** (*str*) – The identifier of the current task being replanned.
- **relevant_tool_description_pair** (*dict*) – A dictionary mapping relevant tool names to their descriptions for replanning.

Side Effects:
Modifies the tool graph to include new tools and updates the execution order of tools within the graph.

reset_plan()

Resets the tool graph and subtask list to their initial states.

topological_sort()

Generates a topological sort of the tool graph to determine the execution order.

This method applies a topological sorting algorithm to the current tool graph, considering the status of each tool. It aims to identify an order in which tools can be executed based on their dependencies, ensuring that all prerequisites are met before an tool is executed. The sorting algorithm accounts for tools that have not yet been executed to avoid cycles and ensure a valid execution order.

Side Effects:
Populates *sub_task_list* with the sorted order of tools to be executed if a topological sort is possible. Otherwise, it indicates a cycle detection.

update_tool(*tool*, *return_val=''*, *relevant_code=None*, *status=False*, *node_type='Code'*)

Updates the specified tool's node information within the tool graph.

This method allows updating an tool's return value, relevant code, execution status, and node_type. It is particularly useful for modifying tools' details after their execution or during the replanning phase.

Parameters

- **tool** (*str*) – The tool identifier whose details are to be updated.
- **return_val** (*str, optional*) – The return value of the tool. Default is an empty string.
- **relevant_code** (*str, optional*) – Any relevant code associated with the tool. Default is None.
- **status** (*bool, optional*) – The execution status of the tool. Default is False.
- **node_type** (*str, optional*) – The node_type of the tool (e.g., 'Code'). Default is 'Code'.

Side Effects:
Updates the information of the specified tool node within the tool graph.

4.9.6 RetrievalModule

class oscopilot.modules.retriever.vector_retriever.**FridayRetriever**(*prompt*, *tool_manager*)

Bases: *BaseModule*

A modules within the system responsible for retrieving and managing available tools from the tool library.

The RetrievalModule extends the BaseModule class, focusing on the retrieval of tools based on specific prompts or queries. It interacts with a language learning model (LLM) and utilizes the execution environments and tool library to fulfill its responsibilities.

delete_tool(*tool*)

Deletes the specified tool from the tool library.

This method calls the tool library's delete method to remove an tool by its name. It encompasses deleting the tool's code, description, parameters, and any other associated information.

Parameters
tool (*str*) – The name of the tool to be deleted.

retrieve_tool_code(*tool_name*)

Retrieves the code for a specified tool from the tool library.

This method accesses the tool library to get the executable code associated with an tool identified by its name. This code defines how the tool is performed.

Parameters
tool_name (*str*) – The name of the tool whose code is to be retrieved.

Returns
The code of the specified tool.

Return type
str

retrieve_tool_code_pair(*retrieve_tool_name*)

Retrieves a mapping of tool names to their respective codes for a list of tools.

This method processes a list of tool names, retrieving the code for each and compiling a dictionary that maps each tool name to its code. This is useful for tasks that require both the identification and the execution details of tools.

Parameters
retrieve_tool_name (*list[str]*) – A list of tool names for which codes are to be retrieved.

Returns
A dictionary mapping each tool name to its code.

Return type
dict

retrieve_tool_description(*tool_name*)

Retrieves the description for a specified tool from the tool library.

This method queries the tool library for the description of an tool identified by its name. It is designed to fetch detailed descriptions that explain what the tool does.

Parameters
tool_name (*str*) – The name of the tool whose description is to be retrieved.

Returns
The description of the specified tool.

Return type
str

retrieve_tool_description_pair(*retrieve_tool_name*)

Retrieves a mapping of tool names to their descriptions for a list of tools.

By processing a list of tool names, this method fetches their descriptions and forms a dictionary that associates each tool name with its description. This facilitates understanding the purpose and functionality of multiple tools at once.

Parameters
retrieve_tool_name (*list[str]*) – A list of tool names for which descriptions are to be retrieved.

Returns
A dictionary mapping each tool name to its description.

Return type
dict

retrieve_tool_name(*task*, *k=10*)

Retrieves a list of tool names relevant to the specified task.

This method interacts with the tool library to retrieve names of tools that are most relevant to a given task. The number of tool names returned is limited by the parameter k.

Parameters

- **task** (*str*) – The task for which relevant tool names are to be retrieved.
- **k** (*int, optional*) – The maximum number of tool names to retrieve. Defaults to 10.

Returns
A list of the top k tool names relevant to the specified task.

Return type
list[str]

tool_code_filter(*tool_code_pair*, *task*)

Filters and retrieves the code for an tool relevant to the specified task.

This method formats a message for filtering tool codes based on a given task, sends the message to the tool library for processing, and retrieves the filtered tool's code. If an tool name is successfully identified, its corresponding code is fetched from the tool library.

Parameters

- **tool_code_pair** (*dict*) – A dictionary mapping tool names to their codes.
- **task** (*str*) – The task based on which the tool code needs to be filtered.

Returns

The code of the tool relevant to the specified task, or an empty string if no relevant tool is found.

4.9.7 ExecutionModule

class oscopilot.modules.executor.friday_executor.**FridayExecutor**(*prompt*, *tool_manager*, *max_iter=3*)

Bases: *BaseModule*

A modules within the system responsible for executing tools based on prompts and maintaining the tool library.

The ExecutionModule extends the BaseAgent class, focusing on the practical execution of tools determined by the system. It utilizes a language learning model (LLM) in conjunction with an execution environments and an tool library to carry out tools. Additionally, it manages system versioning and prompts initialization for tool execution guidance.

analysis_tool(*code*, *task_description*, *state*)

Analyzes the execution outcome of an tool to determine the nature of any errors.

This method evaluates the execution state of an tool, specifically looking for errors. Based on the analysis, it determines whether the error is environmental and requires new operations (handled by the planning modules) or is amendable via the *repair_tool* method. The analysis results, including the reasoning and error type, are returned in JSON format.

Parameters

- **code** (*str*) – The code that was executed for the tool.
- **task_description** (*str*) – The description of the task associated with the tool.
- **state** – The state object containing the result of the tool's execution, including any errors.

Returns

A tuple containing:

- reasoning (str): The analysis's reasoning regarding the nature of the error.
- type (str): The type of error identified ('environmental' for new operations, 'amendable' for corrections).

Return type

tuple

api_tool(*description*, *api_path*, *context='No context provided.'*)

Executes a task by calling an API tool with the provided description and context.

This method formats a message to generate executable code for an API call based on the provided description and context. It sends this message to the language learning model (LLM), extracts the executable Python code from the LLM's response, and returns this code.

Parameters

- **description** (*str*) – A description of the task to be performed by the API call.
- **api_path** (*str*) – The path or endpoint of the API to be called.
- **context** (*str, optional*) – Additional context to be included in the API call. Defaults to "No context provided.".

Returns

The generated Python code to execute the API call.

Return type

str

execute_tool(*code*, *invoke*, *node_type*)

Executes a given tool code and returns the execution state.

This method handles the execution of tool code based on its node_type. For code tools, it appends additional instructions to print the execution result within designated markers. It then passes the modified code for execution in the environments. The method captures and prints the execution state, including any results or errors, and returns this state.

Parameters

- **code** (*str*) – The Python code to be executed as part of the tool.
- **invoke** (*str*) – The specific command or function call that triggers the tool within the code.
- **node_type** (*str*) – The type of the tool, determining how the tool is executed. Currently supports 'Code' type.

Returns

The state object returned by the environments after executing the tool. This object contains

details about the execution's outcome, including any results or errors.

Return type

state

Note: The execution logic is currently tailored for tools of type 'Code', where the code is directly executable Python code. The method is designed to be extensible for other tool types as needed.

---

extract_API_Path(*text*)

Extracts both UNIX-style and Windows-style file paths from the provided text string.

This method applies regular expressions to identify and extract file paths that may be present in the input text. It is capable of recognizing paths that are enclosed within single or double quotes and supports both UNIX-style paths (e.g., */home/user/docs*) and Windows-style paths (e.g., *C:\Users\user\docs*). If multiple paths are found, only the first match is returned, following the function's current implementation.

Parameters

text (*str*) – The string from which file paths are to be extracted.

Returns

The first file path found in the input text, with any enclosing quotes removed. If no paths are
found, an empty string is returned.

Return type
str

Note: The current implementation returns only the first extracted path. If multiple paths are present in the input text, consider modifying the method to return all found paths if the use case requires it.

`extract_args_description`(*class_code*)

Extracts the arguments description from the *__call__* method's docstring within Python class code.

This method specifically targets the docstring of the *__call__* method in a class, which is conventionally used to describe the method's parameters. The extraction is performed using a regular expression that captures the content of the docstring.

Parameters
class_code (*str*) – The Python code of the class from which the arguments description is to be extracted.

Returns
The extracted arguments description from the *__call__* method's docstring, or None if the docstring is not found or does not contain descriptions.

Return type
str

`extract_class_name_and_args_description`(*class_code*)

Extracts the class name and arguments description from a given Python class code.

This method searches the provided class code for the class name and the documentation string of the *__call__* method, which typically includes descriptions of the arguments. It uses regular expressions to locate these elements within the code.

Parameters
class_code (*str*) – The Python code of the class from which information is to be extracted.

Returns

A tuple containing:

- class_name (str): The name of the class extracted from the code.
- args_description (str): The arguments description extracted from the *__call__* method's docstring, if available; otherwise, None.

Return type
tuple

`extract_code`(*response*, *code_type*)

`extract_python_code`(*response*)

Extracts Python code snippets from a response string that includes code block markers.

This method parses a response string to extract Python code enclosed within ``` ```python' and ``` ``` markers. It's designed to retrieve executable Python code snippets from formatted responses, such as those returned by a language learning model after processing a code generation or analysis prompts.

Parameters
response (*str*) – The response string containing the Python code block to be extracted.

Returns
The extracted Python code snippet, or an empty string if no code block is found.

Return type
str

extract_tool_description(*class_code*)

Extracts the description of an tool from the class's initialization method in Python code.

This method looks for the tool's description assigned to *self._description* within the *__init__* method of a class. It uses regular expressions to find this assignment and extracts the description string. This approach assumes that the tool's description is directly assigned as a string literal to *self._description*.

Parameters
class_code (*str*) – The complete Python code of the class from which the tool description is to be extracted.

Returns
The extracted description of the tool if found; otherwise, None.

Return type
str

generate_openapi_doc(*tool_api_path*)

Generates a reduced OpenAPI documentation for a specific API path from the full OpenAPI documentation.

This method isolates and extracts the documentation for a specific tool API path, including its schemas and operations (GET, POST), from the entire OpenAPI documentation stored in the instance. It constructs a new, smaller OpenAPI document that only includes details relevant to the specified API path. If the API path does not exist in the full documentation, it returns an error message.

Parameters
tool_api_path (*str*) – The specific API path for which the OpenAPI documentation should be generated.

Returns

A dictionary representing the OpenAPI documentation for the specific API path. If the path is not
found, returns a dictionary with an error message.

Return type
dict

The method performs several checks: - Verifies the existence of the tool API path in the full OpenAPI documentation. - Extracts relevant parts of the OpenAPI schema related to the path. - Includes any referenced schemas necessary for understanding the API's structure and data types.

It handles both JSON and multipart/form-data content types in API request bodies, searching for schema references to include in the returned documentation. This enables the resulting API document to be self-contained with respect to the schemas needed to understand the API's usage.

generate_tool(*task_name*, *task_description*, *tool_type*, *pre_tasks_info*, *relevant_code*)

Generates executable code and invocation logic for a specified tool.

This method constructs a message to generate tool code capable of completing the specified task, taking into account any prerequisite task information and relevant code snippets. It then formats this message for processing by the language learning model (LLM) to generate the tool code. The method extracts the executable Python code and the specific invocation logic from the LLM's response.

Parameters

- **task_name** (*str*) – The name of the task for which tool code is being generated.
- **task_description** (*str*) – A description of the task, detailing what the tool aims to accomplish.
- **tool_type** (*str*) – The type of tool being generated, such as 'Python', 'Shell', or 'AppleScript'.
- **pre_tasks_info** (*dict*) – Information about tasks that are prerequisites for the current task, including their descriptions and return values.
- **relevant_code** (*dict*) – A dictionary of code snippets relevant to the current task, possibly including code from prerequisite tasks.

Returns

A tuple containing two elements:

- code (str): The generated Python code for the tool.
- invoke (str): The specific logic or command to invoke the generated tool.

Return type
tuple

judge_tool(*code*, *task_description*, *state*, *next_action*)

Evaluates the outcome of an executed tool to determine its success in completing a task.

This method formulates and sends a judgment request to the language learning model (LLM) based on the executed tool's code, the task description, the execution state, and the expected next tool. It then parses the LLM's response to determine the tool's success, providing reasoning, a judgment (boolean), and a score that quantifies the tool's effectiveness.

Parameters

- **code** (*str*) – The code of the tool that was executed.
- **task_description** (*str*) – The description of the task the tool was intended to complete.
- **state** – The state object returned by the environments after executing the tool, containing execution results.
- **next_action** (*str*) – The name of the next expected tool in the sequence.

Returns

A tuple containing:

- reasoning (str): The LLM's reasoning behind the judgment.
- judge (bool): The LLM's judgment on whether the tool successfully completed the task.
- score (float): A score representing the effectiveness of the tool.

Return type
tuple

question_and_answer_tool(*context*, *question*, *current_question=None*)

repair_tool(*current_code*, *task_description*, *tool_type*, *state*, *critique*, *pre_tasks_info*)

Modifies or corrects the code of an tool based on feedback to better complete a task.

This method sends an amendment request to the LLM, including details about the current code, task description, execution state, critique of the tool's outcome, and information about prerequisite tasks. It aims

to generate a revised version of the code that addresses any identified issues or incomplete aspects of the task. The method extracts and returns both the amended code and the specific logic or command to invoke the amended tool.

Parameters

- **current_code** (*str*) – The original code of the tool that requires amendment.
- **task_description** (*str*) – The description of the task the tool is intended to complete.
- **tool_type** (*str*) – The type of tool being amended, such as 'Python', 'Shell', or 'AppleScript'.
- **state** – The state object containing details about the tool's execution outcome.
- **critique** (*str*) – Feedback or critique on the tool's execution, used to guide the amendment.
- **pre_tasks_info** (*dict*) – Information about tasks that are prerequisites for the current task.

Returns

A tuple containing:

- new_code (str): The amended code for the tool.
- invoke (str): The command or logic to invoke the amended tool.

Return type
tuple

save_str_to_path(*content*, *path*)

Saves a string content to a file at the specified path, ensuring the directory exists.

This method takes a string and a file path, creating any necessary parent directories before writing the content to the file. It ensures that the content is written with proper encoding and that any existing content in the file is overwritten. The content is processed to remove extra whitespace at the beginning and end of each line before saving.

Parameters

- **content** (*str*) – The string content to be saved to the file.
- **path** (*str*) – The filesystem path where the content should be saved. If the directory does not exist, it will be created.

Side Effects:

- Creates the directory path if it does not exist.
- Writes the content to a file at the specified path, potentially overwriting existing content.

save_tool_info_to_json(*tool*, *code*, *description*)

Constructs a dictionary containing tool information suitable for JSON serialization.

This method packages the name, code, and description of an tool into a dictionary, making it ready for serialization or further processing. This structured format is useful for saving tool details in a consistent manner, facilitating easy storage and retrieval.

Parameters

- **tool** (*str*) – The name of the tool.
- **code** (*str*) – The executable code associated with the tool.

- **description** (*str*) – A textual description of what the tool does.

Returns
A dictionary containing the tool's name, code, and description.

Return type
dict

store_tool(*tool*, *code*)

Stores the provided tool and its code in the tool library.

If the specified tool does not already exist in the tool library, this method proceeds to store the tool's code, arguments description, and other relevant information. It involves saving these details into JSON files and updating the tool library database. If the tool already exists, it outputs a notification indicating so.

Parameters

- **tool** (*str*) – The name of the tool to be stored.
- **code** (*str*) – The executable code associated with the tool.

Side Effects:

- Adds a new tool to the tool library if it doesn't already exist.
- Saves tool details to the filesystem and updates the tool library's database.
- Outputs a message if the tool already exists in the library.

4.9.8 LearningModule

class oscopilot.modules.learner.self_learner.**SelfLearner**(*prompt*, *tool_manager*)

Bases: *BaseModule*

This class represents a self-learning module that designs educational courses based on given parameters. It inherits from BaseModule, utilizing its initialization and utility methods.

prompt

A dictionary containing system and user prompts for generating course designs.

Type
dict

tool_manager

An instance of a tool manager to handle external tool interactions.

Type
object

course

A dictionary to store course details that are generated based on user and system inputs.

Type
dict

design_course(*software_name*, *package_name*, *demo_file_path*, *file_content=None*, *prior_course=None*)

Designs a course based on specified software and content parameters and stores it in the course attribute.

Parameters

- **software_name** (*str*) – The name of the software around which the course is centered.

- **package_name** (*str*) – The name of the software package relevant to the course.
- **demo_file_path** (*str*) – Path to the demo file that will be used in the course.
- **file_content** (*str*) – The content of the file that will be demonstrated or used in the course.
- **prior_course** (*str*) – The course that has been completed.

Returns
A dictionary containing the designed course details.

Return type
dict

Uses system and user prompts to create a conversation with a language model or similar system, to generate a course based around the provided parameters. The response is then parsed into JSON format and saved.

4.9.9 Prompts

This section of the documentation covers the prompts used by the AI agent across its various components. The prompts are defined within a Python dictionary and are crucial for guiding the AI's interaction with its environment and users.
This modules contains a comprehensive *prompts* dictionary that serves as a repository of prompts for guiding the AI agents's interactions across various operational scenarios, including execution, planning, and information retrieval tasks. These prompts are meticulously crafted to instruct the AI in performing its duties, ranging from code generation and amendment to task decomposition and planning, as well as error analysis and tool usage.

The dictionary is segmented into five main categories:

1. **execute_prompt**: Contains prompts for execution-related tasks, such as code generation, invocation, amendment, and error judgment. These are further detailed for system actions and user interactions, facilitating a diverse range of programming and troubleshooting tasks.
2. **planning_prompt**: Focuses on task planning and re-planning, decomposing complex tasks into manageable sub-tasks, and adapting plans based on unforeseen issues, ensuring that the AI can assist in project management and task organization effectively.
3. **retrieve_prompt**: Dedicated to information retrieval, including filtering code snippets based on specific criteria, aiding the AI in sourcing and suggesting code solutions efficiently.
4. **self_learning_prompt**: Contains prompts for self-learning tasks, such as designing educational courses based on software and content parameters. These prompts guide the AI in generating course designs and educational content tailored to user needs.
5. **text_extract_prompt**: Contains prompts for text extraction tasks, such as extracting specific information from text data. These prompts guide the AI in identifying and extracting relevant data from text inputs.

Each category comprises system and user prompts, where system prompts define the AI's task or query in detail, and user prompts typically include placeholders for dynamic information insertion, reflecting the context or specific requirements of the task at hand.

Usage: The *prompts* dictionary is utilized by the AI agents to dynamically select appropriate prompts based on the current context or task, ensuring relevant and precise guidance for each operation. This dynamic approach allows the AI to adapt its interactions and responses to suit a wide array of programming and operational needs, enhancing its utility and effectiveness in assisting users.

Example

```
# Accessing a specific prompts for task execution
execute_prompt = prompts['execute_prompt']['_SYSTEM_SKILL_CREATE_AND_INVOKE_PROMPT']
```

4.10 Tool Repository

4.10.1 API Tools

class oscopilot.tool_repository.manager.tool_request_util.**ToolRequestUtil**

Bases: object

A utility class for making HTTP requests to an API.

This class simplifies the process of sending HTTP requests using a persistent session and predefined headers, including a User-Agent header to mimic a browser request. It's designed to interact with APIs by sending GET or POST requests and handling file uploads.

session

A requests session for making HTTP requests.

Type
requests.Session

headers

Default headers to be sent with each request.

Type
dict

base_url

The base URL for the API endpoints.

Type
str

request(*api_path*, *method*, *params=None*, *files=None*, *content_type='application/json'*)

Sends a request to the specified API endpoint using the defined HTTP method.

This method constructs the request URL from the base URL and the API path. It supports both GET and POST methods, including handling of JSON parameters, file uploads, and different content types.

Parameters

- **api_path** (*str*) – The path of the API endpoint.
- **method** (*str*) – The HTTP method to use for the request ('get' or 'post').
- **params** (*dict, optional*) – The parameters to include in the request. Defaults to None.
- **files** (*dict, optional*) – Files to be uploaded in a POST request. Defaults to None.
- **content_type** (*str, optional*) – The content type of the request, such as 'application/json' or 'multipart/form-data'. Defaults to "application/json".

Returns
The JSON response from the API, or None if an error occurs.

Return type
dict

Raises
Prints an error message to the console if an HTTP request error occurs. –

4.10.2 Basic Tools

class oscopilot.tool_repository.basic_tools.text_extractor.**TextExtractor**(*agent*)

Bases: object

extract_file_content(*file_path*)

Extract the content of the file.

4.10.3 Tool Manager

class oscopilot.tool_repository.manager.tool_manager.**ToolManager**(*generated_tool_repo_dir=None*)

Bases: object

Manages tools within a repository, including adding, deleting, and retrieving tool information.

The *ToolManager* class provides a comprehensive interface for managing a collection of tools, where each tool is associated with its code, description, and other metadata. It supports operations such as adding new tools, checking for the existence of tools, retrieving tool names, descriptions, and codes, and deleting tools from the collection. It leverages a vector database for efficient retrieval of tools based on similarity searches.

generated_tools

Stores the mapping relationship between tool names and their information (code, description).

Type
dict

generated_tool_repo_dir

The directory path where the tools' information is stored, including code files, description files, and a JSON file containing the tools' metadata.

Type
str

vectordb_path

The path to the vector database used for storing and retrieving tool descriptions based on similarity.

Type
str

vectordb

An instance of the Chroma class for managing the vector database.

Type
Chroma

Note: The class uses OpenAI's *text-embedding-ada-002* model by default for generating embeddings via the *OpenAIEmbeddings* wrapper. Ensure that the *OPENAI_API_KEY* and *OPENAI_ORGANIZATION* are correctly set for OpenAI API access.

This class is designed to facilitate the management of a dynamic collection of tools, providing functionalities for easy addition, retrieval, and deletion of tools. It ensures that the tools' information is synchronized across a local repository and a vector database for efficient retrieval based on content similarity.

`add_new_tool`(*info*)

Adds a new tool to the tool manager, including updating the vector database and tool repository with the provided information.

This method processes the given tool information, which includes the task name, code, and description. It prints out the task name and description, checks if the tool already exists (rewriting it if so), and updates both the vector database and the tool dictionary. Finally, it persists the new tool's code and description in the repository and ensures the vector database is synchronized with the generated tools.

Parameters
`info` (*dict*) – A dictionary containing the tool's information, which must include 'task_name', 'code', and 'description'.

Raises
`AssertionError` – If the vector database's count does not match the length of the generated_tools dictionary after adding the new tool, indicating a synchronization issue.

`delete_tool`(*tool*)

Deletes all information related to a specified tool from the tool manager.

This method removes the tool's information from the vector database, the generated_tools.json file, and also deletes the tool's code and description files from the repository. It performs the deletion only if the tool exists in the respective storage locations and provides console feedback for each successful deletion action.

Parameters
`tool` (*str*) – The name of the tool to be deleted.

Note: This method assumes that the tool's information is stored in a structured manner within the tool manager's repository, including a separate code file (.py), a description text file (.txt), and an arguments description text file (.txt), all named after the tool.

`property descriptions`

Retrieve the descriptions of all tools in a dictionary.

This property constructs a dictionary where each key is a tool name and its value is the description of that tool, extracted from the generated_tools dictionary.

Returns
A dictionary mapping each tool name to its description.

Return type
dict

`exist_tool`(*tool*)

Checks if a tool exists in the tool manager based on the tool name.

Parameters
`tool` (*str*) – The name of the tool to check.

Returns
True if the tool exists, False otherwise.

Return type
bool

`get_tool_code`(*tool_name*)

Retrieve the code of a specific tool by its name.

Given a tool name, this method fetches and returns the code associated with that tool from the generated_tools dictionary. If the tool does not exist, a KeyError will be raised.

Parameters
`tool_name` (*str*) – The name of the tool for which the code is requested.

Returns
The code of the specified tool.

Return type
str

Raises
`KeyError` – If the tool_name does not exist in the generated_tools dictionary.

`property programs`

Retrieve all the code from the code repository as a single string.

This property concatenates the code of all tools stored in the generated_tools dictionary, separating each tool's code with two newlines.

Returns
A string containing the code of all tools, each separated by two newlines.

Return type
str

`retrieve_tool_code`(*tool_name*)

Returns the code of specified tools based on their names.

Similar to retrieving tool descriptions, this method iterates over a list of tool names and retrieves the code for each tool from the generated_tools dictionary. It then compiles and returns a list of these codes.

Parameters
`tool_name` (*list[str]*) – A list of tool names for which code snippets are requested.

Returns
A list containing the code of the specified tools.

Return type
list[str]

`retrieve_tool_description`(*tool_name*)

Returns the descriptions of specified tools based on their names.

This method iterates over a list of tool names and retrieves the description for each tool from the generated_tools dictionary. It compiles and returns a list of these descriptions.

Parameters
`tool_name` (*list[str]*) – A list of tool names for which descriptions are requested.

Returns
A list containing the descriptions of the specified tools.

Return type
list[str]

retrieve_tool_name(*query*, *k=10*)

Retrieves related tool names based on a similarity search against a query.

This method performs a similarity search in the vector database for the given query and retrieves the names of the top *k* most similar tools. It prints the number of tools being retrieved and their names.

Parameters

- **query** (*str*) – The query string to search for similar tools.
- **k** (*int, optional*) – The maximum number of similar tools to retrieve. Defaults to 10.

Returns

A list of tool names that are most similar to the query,
up to *k* tools. Returns an empty list if no tools are found or if *k* is 0.

Return type
list[str]

property tool_names

Retrieve all tool class names from the generated tools.

This property provides access to the names of all tools stored in the generated_tools dictionary, facilitating enumeration over tool names.

Returns
A view of the dictionary's keys which are the names of the tools.

Return type
KeysView[str]

class oscopilot.tool_repository.manager.action_node.**ActionNode**(*name*, *description*, *node_type*)

Bases: object

Represents an action node in a workflow or execution graph, encapsulating details like the action's name, description, return value, relevant code snippets, next actions, execution status, and action type.

_name

The name of the action.

Type
str

_description

A brief description of what the action does.

Type
str

_return_val

The value returned by the action upon execution.

Type
str

_relevant_code

A dictionary mapping relevant code snippets or references associated with the action.

Type
dict

`_next_action`

A dictionary mapping subsequent actions that depend on the current action.

Type
dict

`_status`

The execution status of the action, indicating whether it has been successfully executed.

Type
bool

`_type`

The type of the action, categorizing its purpose or method of execution.

Type
str

`property description`

Returns the description of the action.

Returns
The action's description.

Return type
str

`property name`

Returns the name of the action.

Returns
The action's name.

Return type
str

`property next_action`

Returns subsequent actions that depend on the current action.

Returns
A mapping of subsequent actions.

Return type
dict

`property node_type`

Returns the type of the action.

Returns
The action's type.

Return type
str

`property relevant_action`

Returns the relevant code snippets or references associated with the action.

Returns
The action's relevant code snippets or references.

Return type
dict

property return_val

Returns the return value of the action.

Returns
The value returned by the action upon execution.

Return type
str

property status

Returns the execution status of the action.

Returns
True if the action has been executed successfully, False otherwise.

Return type
bool

oscopilot.tool_repository.manager.tool_manager.**print_error_and_exit**(*message*)

Prints an error message to standard output and exits the program with a status code of 1.

This function is typically used to handle critical errors from which the program cannot recover. It ensures that the error message is visible to the user before the program terminates.

Parameters
message (*str*) – The error message to be printed.

oscopilot.tool_repository.manager.tool_manager.**add_tool**(*toolManager*, *tool_name*, *tool_path*)

Adds a new tool to the tool manager with the given name and code loaded from the specified path.

This function reads the tool's code from a file, extracts a description from the code using a predefined pattern, and then adds the tool to the tool manager using the extracted information. If the tool's description is not found within the code, the function will print an error message and exit.

Parameters

- **toolManager** (ToolManager) – The instance of ToolManager to which the tool will be added.
- **tool_name** (*str*) – The name of the tool to be added.
- **tool_path** (*str*) – The file system path to the source code of the tool.

Note: The function expects the tool's code to contain a description defined as a string literal assigned to *self._description* within the code. The description must be enclosed in double quotes for it to be successfully extracted.

---

oscopilot.tool_repository.manager.tool_manager.**delete_tool**(*toolManager*, *tool_name*)

Deletes a tool from the tool manager and prints a success message.

This function calls the *delete_tool* method of the given ToolManager instance to remove the specified tool. Upon successful deletion, it prints a message indicating the operation was successful.

Parameters

- **toolManager** (ToolManager) – An instance of the ToolManager class.
- **tool_name** (*str*) – The name of the tool to be deleted.

oscopilot.tool_repository.manager.tool_manager.**get_open_api_doc_path**()

Determines the file system path to the 'openapi.json' file located in the same directory as this script.

Returns
The absolute path to the 'openapi.json' file.

Return type
str

oscopilot.tool_repository.manager.tool_manager.**get_open_api_description_pair**()

Extracts and returns a mapping of OpenAPI path names to their descriptions.

This function loads the OpenAPI specification from a 'openapi.json' file located in the same directory as this script. It then iterates over the paths defined in the OpenAPI specification, creating a dictionary that maps each path name to its description (summary). If a path supports both 'get' and 'post' operations, the description for the 'post' operation is preferred.

Returns
A dictionary mapping OpenAPI path names to their summary descriptions.

Return type
dict

oscopilot.tool_repository.manager.tool_manager.**main**()

The main entry point for managing generated tools for the FRIDAY project.

This function sets up a command-line interface for adding or deleting tools within the FRIDAY project. It supports flags for adding a new tool, deleting an existing tool, and specifies the name and path of the tool for the respective operations. Based on the arguments provided, it initializes a ToolManager instance and performs the requested add or delete operation.

The '–add' flag requires the '–tool_name' and '–tool_path' arguments to specify the name and the path of the tool to be added. The '–delete' flag requires only the '–tool_name' argument.

Usage:
python script.py –add –tool_name <name> –tool_path <path> python script.py –delete –tool_name <name>

Raises
SystemExit – If no operation type is specified or required arguments are missing, the program will print an error message and exit with a status code of 1.

4.11 Environment

4.11.1 Execution Environment

class oscopilot.environments.env.**Env**

Bases: *BaseEnv*

A class representing an environment for executing code in various languages.

This class manages the execution of code in different languages and provides methods for interacting with those languages.

It inherits from BaseEnv, which provides basic environment functionality.

`get_language`(*language*)

Gets the language class based on the provided language name or alias.

Parameters
`language` (*str*) – The name or alias of the language.

Returns
The language class corresponding to the provided name or alias, or None if not found.

Return type
class

`step`(*language*, *code*, *stream=False*, *display=False*)

Executes a step of code in the specified language.

Parameters

- `language` (*str*) – The name or alias of the language to execute the code in.
- `code` (*str*) – The code to execute.
- `stream` (*bool*) – Whether to stream the output as it becomes available.
- `display` (*bool*) – Whether to display the output.

Returns
The state after executing the code.

Return type
EnvState

`stop`()

Stops the execution of all active languages.

`terminate`()

Terminates all active language environments.

4.11.2 Base Environment

`class` oscopilot.environments.base_env.`BaseEnv`

Bases: `object`

A base class for environments configurations in action-based systems.

This class provides foundational attributes and methods for managing environments, including timeouts, working directories, and environmental states. It is designed to be extended by subclasses that implement specific environments behaviors.

`list_working_dir`()

Lists the contents of the working directory in a detailed format.

Returns a string representation similar to the output of the 'ls' command in Linux, including file/directory names, sizes, and types.

Returns
Detailed listings of the working directory's contents, or an error message if the directory does not exist.

Return type
str

property name

The name of the environments.

Returns
The name of the environments, typically set to the class name unless overridden in a subclass.

Return type
str

reset()

Resets the environments to its initial state.

This method is intended to be implemented by subclasses, defining the specific actions required to reset the environments.

step(*_command*) → *EnvState*

Executes a command within the environments.

This method is intended to be implemented by subclasses, defining how commands are processed and their effects on the environments state.

Parameters
_command – The command to be executed.

Raises
NotImplementedError – Indicates that the subclass must implement this method.

Returns
The state of the environments after executing the command.

Return type
EnvState

stop()

Halts code execution, but does not terminate state.

terminate()

Terminates state.

4.11.3 Bash Environment

class oscopilot.environments.bash_env.**Shell**

Bases: *SubprocessEnv*

A class representing a shell environment for executing shell scripts.

This class inherits from SubprocessEnv, which provides a general environment for executing code in subprocesses.

aliases = ['bash', 'sh', 'zsh']

detect_active_line(*line*)

Detects the active line indicator in the output.

Parameters
line (*str*) – A line from the output.

Returns
The line number indicated by the active line indicator, or None if not found.

Return type
int

`detect_end_of_execution`(*line*)

Detects the end of execution marker in the output.

Parameters
line (*str*) – A line from the output.

Returns
True if the end of execution marker is found, False otherwise.

Return type
bool

`file_extension = 'sh'`

`line_postprocessor`(*line*)

Postprocesses each line of output from the shell execution.

Parameters
line (*str*) – A line from the output of the shell script execution.

Returns
The processed line.

Return type
str

`name = 'Shell'`

`preprocess_code`(*code*)

Preprocesses the shell script code before execution.

Parameters
code (*str*) – The shell script code to preprocess.

Returns
The preprocessed shell script code.

Return type
str

`oscopilot.environments.bash_env.add_active_line_prints`(*code*)

Adds echo statements indicating line numbers to a shell script.

Parameters
code (*str*) – The shell script code to add active line indicators to.

Returns
The modified shell script code with active line indicators.

Return type
str

`oscopilot.environments.bash_env.has_multiline_commands`(*script_text*)

Checks if a shell script contains multiline commands.

Parameters
script_text (*str*) – The shell script code to check.

Returns
True if the script contains multiline commands, False otherwise.

Return type
bool

oscopilot.environments.bash_env.**preprocess_shell**(*code*)

Preprocesses the shell script code before execution.

Adds active line markers, wraps in a try-except block (trap in shell), and adds an end of execution marker.

Parameters
code (*str*) – The shell script code to preprocess.

Returns
The preprocessed shell script code.

Return type
str

4.11.4 AppleScript Environment

class oscopilot.environments.applescript_env.**AppleScript**

Bases: *SubprocessEnv*

A class representing an AppleScript environment for executing AppleScript code.

This class inherits from SubprocessEnv, which provides a general environment for executing code in subprocesses.

add_active_line_indicators(*code*)

Adds log commands to indicate the active line of execution in the AppleScript.

Parameters
code (*str*) – The AppleScript code to add active line indicators to.

Returns
The modified AppleScript code with active line indicators.

Return type
str

detect_active_line(*line*)

Detects the active line indicator in the output.

Parameters
line (*str*) – A line from the output.

Returns
The line number indicated by the active line indicator, or None if not found.

Return type
int

detect_end_of_execution(*line*)

Detects the end of execution marker in the output.

Parameters
line (*str*) – A line from the output.

Returns
True if the end of execution marker is found, False otherwise.

Return type
bool

`file_extension = 'applescript'`

`name = 'AppleScript'`

`preprocess_code`(*code*)

Preprocesses the AppleScript code before execution.

Inserts an end_of_execution marker and adds active line indicators to the code.

Parameters
code (*str*) – The AppleScript code to preprocess.

Returns
The preprocessed AppleScript code.

Return type
str

4.11.5 Python Jupyter Environment

`class oscopilot.environments.py_jupyter_env.AddLinePrints`

Bases: `NodeTransformer`

Transformer to insert print statements indicating the line number before every executable line in the AST.

`insert_print_statement`(*line_number*)

Inserts a print statement for a given line number.

Parameters
line_number (*int*) – The line number.

Returns
The print statement AST node.

Return type
ast.Expr

`process_body`(*body*)

Processes a block of statements, adding print calls.

Parameters
body (*list*) – List of AST nodes representing statements.

Returns
List of modified AST nodes.

Return type
list

`visit`(*node*)

Visits and transforms nodes in the AST.

Parameters
node – The current AST node.

Returns
The modified AST node.

Return type
ast.Node

class oscopilot.environments.py_jupyter_env.**PythonJupyterEnv**

Bases: *BaseEnv*

A class representing an environment for executing Python code in a Jupyter environment.

This class manages the execution of Python code using IPython kernel, providing methods for preprocessing code, executing code steps, handling output messages, and terminating the kernel.

It inherits from BaseEnv, which provides basic environment functionality.

aliases = ['py', 'API']

detect_active_line(*line*)

Detects active line markers in the output line.

Parameters
line (*str*) – The output line from the IPython kernel.

Returns
The modified line and active line number, if detected.

Return type
tuple

file_extension = 'py'

name = 'Python'

preprocess_code(*code*)

Preprocesses the Python code before execution.

Parameters
code (*str*) – The Python code to preprocess.

Returns
The preprocessed code.

Return type
str

step(*code*)

Executes a step of Python code.

Parameters
code (*str*) – The Python code to execute.

Yields
dict – Output messages generated during execution.

stop()

Stops the execution of code by setting the finish flag.

terminate()

Terminates the IPython kernel and stops its channels.

oscopilot.environments.py_jupyter_env.**add_active_line_prints**(*code*)

Adds print statements indicating line numbers to a Python string.

Parameters
code (*str*) – The Python code.

Returns
The code with added print statements.

Return type
str

oscopilot.environments.py_jupyter_env.**main**()

oscopilot.environments.py_jupyter_env.**string_to_python**(*code_as_string*)

Parses Python code from a string and extracts function definitions.

Parameters
code_as_string (*str*) – The Python code as a string.

Returns
A dictionary mapping function names to their code.

Return type
dict

oscopilot.environments.py_jupyter_env.**wrap_in_try_except**(*code*)

Wraps Python code in a try-except block to catch exceptions.

Parameters
code (*str*) – The Python code.

Returns
The code wrapped in a try-except block.

Return type
str

4.11.6 Subprocess Environment

class oscopilot.environments.subprocess_env.**SubprocessEnv**

Bases: *BaseEnv*

A class representing an environment for executing code using subprocesses.

This class manages the execution of code in subprocesses, providing methods for preprocessing code, starting and terminating processes, handling output streams, and executing code steps.

It inherits from BaseEnv, which provides basic environment functionality.

detect_active_line(*line*)

Detects an active line indicator in the output line.

Parameters
line (*str*) – The output line from the subprocess.

Returns
The active line number if detected, else None.

Return type
int or None

detect_end_of_execution(*line*)

Detects an end of execution marker in the output line.

Parameters
line (*str*) – The output line from the subprocess.

Returns
True if end of execution marker is detected, else False.

Return type
bool

`handle_stream_output`(*stream*, *is_error_stream*)

Handles the streaming output from the subprocess.

Parameters

- **stream** – The output stream to handle.
- **is_error_stream** (*bool*) – Indicates if the stream is the error stream.

`line_postprocessor`(*line*)

Post-processes an output line from the subprocess.

Parameters
line (*str*) – The output line from the subprocess.

Returns
The processed line or None if line should be discarded.

Return type
str or None

`preprocess_code`(*code*)

Preprocesses code before execution.

This method inserts an end_of_execution marker and optionally adds active line markers.

Parameters
code (*str*) – The code to preprocess.

Returns
The preprocessed code.

Return type
str

`start_process`()

Starts the subprocess to execute code.

`step`(*code*)

Executes a step of code.

Parameters
code (*str*) – The code to execute.

Yields
dict – Output messages generated during execution.

`terminate`()

Terminates the subprocess if it is running.

4.12 Utils

4.12.1 Large Language Models

class oscopilot.utils.llms.**OpenAI**

Bases: object

A class for interacting with the OpenAI API, allowing for chat completion requests.

This class simplifies the process of sending requests to OpenAI's chat model by providing a convenient interface for the chat completion API. It handles setting up the API key and organization for the session and provides a method to send chat messages.

model_name

The name of the model to use for chat completions. Default is set by the global *MODEL_NAME*.

Type
str

api_key

The API key used for authentication with the OpenAI API. This should be set through the *OPENAI_API_KEY* global variable.

Type
str

organization

The organization ID for OpenAI. Set this through the *OPENAI_ORGANIZATION* global variable.

Type
str

chat(*messages*, *temperature=0*)

Sends a chat completion request to the OpenAI API using the specified messages and parameters.

Parameters

- **messages** (*list of dict*) – A list of message dictionaries, where each dictionary should contain keys like 'role' and 'content' to specify the role (e.g., 'system', 'user') and content of each message.
- **temperature** (*float, optional*) – Controls randomness in the generation. Lower values make the model more deterministic. Defaults to 0.

Returns
The content of the first message in the response from the OpenAI API.

Return type
str

4.12.2 OS-Copilot Config

class oscopilot.utils.config.**Config**

Bases: object

A singleton class for storing and accessing configuration parameters.

This class ensures that only one instance is created and provides methods for initializing and accessing parameters.

classmethod get_parameter(*key*)

Retrieves a parameter value by key.

Parameters
key (*str*) – The key of the parameter to retrieve.

Returns
The value of the parameter if found, otherwise None.

Return type
Any

classmethod initialize(*args*)

Initializes the Config instance with command-line arguments.

Parameters
args (*argparse.Namespace*) – The parsed command-line arguments.

oscopilot.utils.config.**self_learning_print_logging**(*args*)

Prints self-learning task information and logs it.

Parameters
args (*argparse.Namespace*) – The parsed command-line arguments.

oscopilot.utils.config.**setup_config**()

Sets up configuration parameters based on command-line arguments.

Returns
The parsed command-line arguments.

Return type
argparse.Namespace

oscopilot.utils.config.**setup_pre_run**(*args*)

Sets up pre-run tasks and logging.

Parameters
args (*argparse.Namespace*) – The parsed command-line arguments.

Returns
A string containing information about the task.

Return type
str

4.12.3 Data Schema

class oscopilot.utils.schema.**EnvState**(*command: ~typing.List[str] = <factory>, result: str | None = '', error: str | None = None, pwd: str | None = '', ls: str | None = ''*)

Bases: object

Represents the state of an environment in which commands are executed.

command: List[str]

error: str | None = None

ls: str | None = ''

pwd: str | None = ''

result: str | None = ''

class oscopilot.utils.schema.**ExecutionState**(*state: EnvState | None = None, node_type: str = '', description: str = '', code: str = '', result: str = '', relevant_code: str = ''*)

Bases: object

Stores all the intermediate representation during agent executing.

code: str = ''

description: str = ''

get_all_state()

node_type: str = ''

relevant_code: str = ''

result: str = ''

state: *EnvState* | None = None

class oscopilot.utils.schema.**InnerMonologue**(*reasoning: str = '', error_type: str = '', critique: str = '', isRePlan: bool = False, isTaskCompleted: bool = False, result: str = ''*)

Bases: object

Stores all the intermediate representation during agent running

critique: str = ''

error_type: str = ''

isRePlan: bool = False

isTaskCompleted: bool = False

reasoning: str = ''

result: str = ''

class oscopilot.utils.schema.**JudgementResult**(*status: bool = False*, *critique: str = ''*, *score: int = 0*)

Bases: object

Stores the results and intermediate representation of the judging process

critique: str = ''

score: int = 0

status: bool = False

class oscopilot.utils.schema.**RepairingResult**(*status: str = ''*, *code: str = ''*, *critique: str = ''*, *score: str = ''*, *result: str = ''*)

Bases: object

Stores the results and intermediate representation of the repairing process

code: str = ''

critique: str = ''

result: str = ''

score: str = ''

status: str = ''

class oscopilot.utils.schema.**TaskStatusCode**(*value*)

Bases: IntEnum

An enumeration.

COMPLETED = 7

FAILED = 6

START = 1

4.12.4 Server Proxy Config

class oscopilot.utils.server_config.**ConfigManager**

Bases: object

A singleton class responsible for managing configuration settings across the application.

This class implements the singleton design pattern to ensure that only one instance of the ConfigManager exists at any time. It provides methods to set, apply, and clear proxy settings for HTTP and HTTPS traffic.

_instance

A private class-level attribute that holds the singleton instance.

Type
ConfigManager

http_proxy

The HTTP proxy URL.

Type
str

https_proxy

The HTTPS proxy URL.

Type
str

apply_proxies()

Applies the configured proxy settings by setting them in the environments variables.

The method sets the 'http_proxy' and 'https_proxy' environments variables based on the configured proxy URLs. If no proxies are configured, the environments variables are not modified.

clear_proxies()

Clears the proxy settings from the environments variables.

This method removes the 'http_proxy' and 'https_proxy' entries from the environments variables, effectively clearing any proxy settings that were previously applied.

set_proxies(*http*, *https*)

Sets the HTTP and HTTPS proxy URLs.

Parameters

- **http** (*str*) – The HTTP proxy URL.
- **https** (*str*) – The HTTPS proxy URL.

4.12.5 Other Utils

oscopilot.utils.utils.**send_chat_prompts**(*sys_prompt*, *user_prompt*, *llm*)

Sends a sequence of chat prompts to a language learning model (LLM) and returns the model's response.

Parameters

- **sys_prompt** (*str*) – The system prompt that sets the context or provides instructions for the language learning model.
- **user_prompt** (*str*) – The user prompt that contains the specific query or command intended for the language learning model.
- **llm** (*object*) – The language learning model to which the prompts are sent. This model is expected to have a *chat* method that accepts structured prompts.

Returns
The response from the language learning model, which is typically a string containing the model's answer or generated content based on the provided prompts.

The function is a utility for simplifying the process of sending structured chat prompts to a language learning model and parsing its response, useful in scenarios where dynamic interaction with the model is required.

oscopilot.utils.utils.**random_string**(*length*)

Generates a random string of a specified length.

Parameters
length (*int*) – The desired length of the random string.

Returns
A string of random characters and digits of the specified length.

Return type
str

oscopilot.utils.utils.**num_tokens_from_string**(*string: str*) → int

Calculates the number of tokens in a given text string according to a specific encoding.

Parameters
text (*str*) – The text string to be tokenized.

Returns
The number of tokens the string is encoded into according to the model's tokenizer.

Return type
int

oscopilot.utils.utils.**parse_content**(*content*, *html_type='html.parser'*)

Parses and cleans the given HTML content, removing specified tags, ids, and classes.

Parameters

- **content** (*str*) – The HTML content to be parsed and cleaned.
- **type** (*str, optional*) – The type of parser to be used by BeautifulSoup. Defaults to "html.parser". Supported types include "html.parser", "lxml", "lxml-xml", "xml", and "html5lib".

Raises
ValueError – If an unsupported parser type is specified.

Returns
The cleaned text extracted from the HTML content.

Return type
str

oscopilot.utils.utils.**clean_string**(*text*)

Cleans a given string by performing various operations such as whitespace normalization, removal of backslashes, and replacement of hash characters with spaces. It also reduces consecutive non-alphanumeric characters to a single occurrence.

Parameters
text (*str*) – The text to be cleaned.

Returns
The cleaned text after applying all the specified cleaning operations.

Return type
str

oscopilot.utils.utils.**chunks**(*iterable*, *batch_size=100*, *desc='Processing chunks'*)

Breaks an iterable into smaller chunks of a specified size, yielding each chunk in sequence.

Parameters

- **iterable** (*iterable*) – The iterable to be chunked.
- **batch_size** (*int, optional*) – The size of each chunk. Defaults to 100.
- **desc** (*str, optional*) – Description text to be displayed alongside the progress bar. Defaults to "Processing chunks".

Yields
tuple – A chunk of the iterable, with a maximum length of *batch_size*.

oscopilot.utils.utils.**generate_prompt**(*template: str*, *replace_dict: dict*)

Generates a string by replacing placeholders in a template with values from a dictionary.

Parameters

- **template** (*str*) – The template string containing placeholders to be replaced.
- **replace_dict** (*dict*) – A dictionary where each key corresponds to a placeholder in the template and each value is the replacement for that placeholder.

Returns
The resulting string after all placeholders have been replaced with their corresponding values.

Return type
str

oscopilot.utils.utils.**cosine_similarity**(*a*, *b*)

Calculates the cosine similarity between two vectors.

Parameters

- **a** (*array_like*) – The first vector.
- **b** (*array_like*) – The second vector.

Returns
The cosine similarity between vectors *a* and *b*.

Return type
float

oscopilot.utils.utils.**is_valid_json_string**(*source: str*)

Checks if a given string is a valid JSON.

Parameters
source (*str*) – The string to be validated as JSON.

Returns
True if the given string is a valid JSON format, False otherwise.

Return type
bool

PYTHON MODULE INDEX

O

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

Q

R

S

T

U

V

W