agentchat.contrib.math_user_proxy_agent

MathUserProxyAgent Objects

```
class MathUserProxyAgent(UserProxyAgent)
```

(Experimental) A MathChat agent that can handle math problems.

MAX_CONSECUTIVE_AUTO_REPLY

maximum number of consecutive auto replies (subject to future change)

__init__

```
def __init__(name: Optional[str] = "MathChatAgent",
             is_termination_msg: Optional[Callable[
                 [Dict], bool]] = _is_termination_msg_mathchat,
             human_input_mode: Optional[str] = "NEVER",
             default_auto_reply: Optional[Union[str, Dict,
                                                None]] = DEFAULT_REPLY,
             max_invalid_q_per_step=3,
             **kwargs)
```

Arguments:

- `name` *str* - name of the agent
- `is_termination_msg` *function* - a function that takes a message in the form of a dictionary and returns a boolean value indicating if this received message is a termination message. The dict can contain the following keys: "content", "role", "name", "function_call".
- `human_input_mode` *str* - whether to ask for human inputs every time a message is received. Possible values are "ALWAYS", "TERMINATE", "NEVER". (1) When "ALWAYS", the agent prompts for human input every time a message is received. Under this mode, the conversation stops when the human input is "exit", or when is_termination_msg is True and there is no human input. (2) When "TERMINATE", the agent only prompts for human input only when a termination message is received or the number of auto reply reaches the max_consecutive_auto_reply. (3) (Default) When "NEVER", the agent will never prompt for human input. Under this mode, the conversation stops when the number of auto reply reaches the max_consecutive_auto_reply or when is_termination_msg is True.
- `default_auto_reply` *str or dict or None* - the default auto reply message when no code execution or llm based reply is generated.
- `max_invalid_q_per_step` *int* - (ADDED) the maximum number of invalid queries per step.
- `**kwargs` *dict* - other kwargs in UserProxyAgent.

generate_init_message

```
def generate_init_message(problem,
                          prompt_type="default",
                          customized_prompt=None)
```

Generate a prompt for the assistant agent with the given problem and prompt.

Arguments:

- `problem` *str* - the problem to be solved.
- `prompt_type` *str* - the type of the prompt. Possible values are "default", "python", "wolfram". (1) "default": the prompt that allows the agent to choose between 3 ways to solve a problem:
    i. write a python program to solve it directly.
    ii. solve it directly without python.
    iii. solve it step by step with python. (2) "python": a simplified prompt from the third way of the "default" prompt, that asks the assistant to solve the problem step by step with python. (3) "two_tools": a simplified prompt similar to the "python" prompt, but allows the model to choose between Python and Wolfram Alpha to solve the problem.
- `customized_prompt` *str* - a customized prompt to be used. If it is not None, the prompt_type will be ignored.

Returns:

- `str` - the generated prompt ready to be sent to the assistant agent.

execute_one_python_code

```
def execute_one_python_code(pycode)
```

Execute python code blocks.

Previous python code will be saved and executed together with the new code. the "print" function will also be added to the last line of the code if needed

execute_one_wolfram_query

```
def execute_one_wolfram_query(query: str)
```

Run one wolfram query and return the output.

Arguments:

- `query` - string of the query.

Returns:

- `output` - string with the output of the query.
- `is_success` - boolean indicating whether the query was successful.

get_from_dict_or_env

```
def get_from_dict_or_env(data: Dict[str, Any],
                         key: str,
                         env_key: str,
                         default: Optional[str] = None) -> str
```

Get a value from a dictionary or an environment variable.

WolframAlphaAPIWrapper Objects

```
class WolframAlphaAPIWrapper(BaseModel)
```

Wrapper for Wolfram Alpha.

Docs for using:

1. Go to wolfram alpha and sign up for a developer account
2. Create an app and get your APP ID
3. Save your APP ID into WOLFRAM_ALPHA_APPID env variable
4. pip install wolframalpha

wolfram_client

:meta private:

Config Objects

```
class Config()
```

Configuration for this pydantic object.

validate_environment

```
@root_validator(skip_on_failure=True)
def validate_environment(cls, values: Dict) -> Dict
```

Validate that api key and python package exists in environment.

run

```
def run(query: str) -> Tuple[str, bool]
```

Run query through WolframAlpha and parse result.

Edit this page

Previous
« llava_agent

Next
multimodal_conversable_agent »

Community

Discord

Twitter