agentchat.conversable_agent

ConversableAgent Objects

```
class ConversableAgent(Agent)
```

(In preview) A class for generic conversable agents which can be configured as assistant or user proxy.

After receiving each message, the agent will send a reply to the sender unless the msg is a termination msg. For example, AssistantAgent and UserProxyAgent are subclasses of this class, configured with different default settings.

To modify auto reply, override `generate_reply` method. To disable/enable human response in every turn, set `human_input_mode` to "NEVER" or "ALWAYS". To modify the way to get human input, override `get_human_input` method. To modify the way to execute code blocks, single code block, or function call, override `execute_code_blocks`, `run_code`, and `execute_function` methods respectively. To customize the initial message when a conversation starts, override `generate_init_message` method.

DEFAULT_CONFIG

An empty configuration

MAX_CONSECUTIVE_AUTO_REPLY

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

__init__

```
def __init__(name: str,
             system_message: Optional[Union[
                 str, List]] = "You are a helpful AI Assistant.",
             is_termination_msg: Optional[Callable[[Dict], bool]] = None,
             max_consecutive_auto_reply: Optional[int] = None,
             human_input_mode: Optional[str] = "TERMINATE",
             function_map: Optional[Dict[str, Callable]] = None,
             code_execution_config: Optional[Union[Dict,
                                                   Literal[False]]] = None,
             llm_config: Optional[Union[Dict, Literal[False]]] = None,
             default_auto_reply: Optional[Union[str, Dict, None]] = "",
             description: Optional[str] = None)
```

Arguments:

- `name` *str* - name of the agent.
- `system_message` *str or list* - system message for the ChatCompletion inference.
- `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".
- `max_consecutive_auto_reply` *int* - the maximum number of consecutive auto replies. default to None (no limit provided, class attribute MAX_CONSECUTIVE_AUTO_REPLY will be used as the limit in this case). When set to 0, no auto reply will be generated.
- `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) 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.
- `function_map` *dict[str, callable]* - Mapping function names (passed to openai) to callable functions, also used for tool calls.
- `code_execution_config` *dict or False* - config for the code execution. To disable code execution, set to False. Otherwise, set to a dictionary with the following keys:
  - work_dir (Optional, str): The working directory for the code execution. If None, a default working directory will be used. The default working directory is the "extensions" directory under "path_to_autogen".

    - use_docker (Optional, list, str or bool): The docker image to use for code execution. If a list or a str of image name(s) is provided, the code will be executed in a docker container with the first image successfully pulled. If None, False or empty, the code will be executed in the current environment. Default is True when the docker python package is installed. When set to True, a default list will be used. We strongly recommend using docker for code execution.
    - timeout (Optional, int): The maximum execution time in seconds.
    - last_n_messages (Experimental, Optional, int or str): The number of messages to look back for code execution. Default to 1. If set to 'auto', it will scan backwards through all messages arriving since the agent last spoke (typically this is the last time execution was attempted).
- `llm_config` *dict or False* - llm inference configuration. Please refer to OpenAIWrapper.create for available options. To disable llm-based auto reply, set to False.
- `default_auto_reply` *str or dict or None* - default auto reply when no code execution or llm-based reply is generated.
- `description` *str* - a short description of the agent. This description is used by other agents (e.g. the GroupChatManager) to decide when to call upon this agent. (Default: system_message)

register_reply

```
def register_reply(trigger: Union[Type[Agent], str, Agent,
                                  Callable[[Agent], bool], List],
                   reply_func: Callable,
                   position: int = 0,
                   config: Optional[Any] = None,
                   reset_config: Optional[Callable] = None,
                   *,
                   ignore_async_in_sync_chat: bool = False)
```

Register a reply function.

The reply function will be called when the trigger matches the sender. The function registered later will be checked earlier by default. To change the order, set the position to a positive integer.

Both sync and async reply functions can be registered. The sync reply function will be triggered from both sync and async chats. However, an async reply function will only be triggered from async chats (initiated with `ConversableAgent.a_initiate_chat`). If an `async` reply function is registered and a chat is initialized with a sync function, `ignore_async_in_sync_chat` determines the behaviour as follows:

- if `ignore_async_in_sync_chat` is set to `False` (default value), an exception will be raised, and
- if `ignore_async_in_sync_chat` is set to `True`, the reply function will be ignored.

Arguments:

- `trigger` *Agent class, str, Agent instance, callable, or list* - the trigger.
    - If a class is provided, the reply function will be called when the sender is an instance of the class.
    - If a string is provided, the reply function will be called when the sender's name matches the string.
    - If an agent instance is provided, the reply function will be called when the sender is the agent instance.
    - If a callable is provided, the reply function will be called when the callable returns True.
    - If a list is provided, the reply function will be called when any of the triggers in the list is activated.
    - If None is provided, the reply function will be called only when the sender is None.
- `Note` - Be sure to register `None` as a trigger if you would like to trigger an auto-reply function with non-empty messages and `async`0.
- `async`1 *Callable* - the reply function. The function takes a recipient agent, a list of messages, a sender agent and a config as input and returns a reply message.
- `async`2 - the position of the reply function in the reply function list.
- `async`3 - the config to be passed to the reply function, see below.
- `async`4 - the function to reset the config, see below.
- `ignore_async_in_sync_chat` - whether to ignore the async reply function in sync chats. If `False`, an exception will be raised if an async reply function is registered and a chat is initialized with a sync function. `async`7
- `async`2 *int* - the position of the reply function in the reply function list. The function registered later will be checked earlier by default. To change the order, set the position to a positive integer.
- `async`3 *Any* - the config to be passed to the reply function. When an agent is reset, the config will be reset to the original value.
- `async`4 *Callable* - the function to reset the config. The function returns None. Signature: `ignore_async_in_sync_chat`1

system_message

```
@property
def system_message() -> Union[str, List]
```

Return the system message.

update_system_message

```
def update_system_message(system_message: Union[str, List])
```

Update the system message.

Arguments:

- `system_message` *str or List* - system message for the ChatCompletion inference.

update_max_consecutive_auto_reply

```
def update_max_consecutive_auto_reply(value: int,
                                      sender: Optional[Agent] = None)
```

Update the maximum number of consecutive auto replies.

Arguments:

- `value` *int* - the maximum number of consecutive auto replies.
- `sender` *Agent* - when the sender is provided, only update the max_consecutive_auto_reply for that sender.

max_consecutive_auto_reply

```
def max_consecutive_auto_reply(sender: Optional[Agent] = None) -> int
```

The maximum number of consecutive auto replies.

chat_messages

```
@property
def chat_messages() -> Dict[Agent, List[Dict]]
```

A dictionary of conversations from agent to list of messages.

last_message

```
def last_message(agent: Optional[Agent] = None) -> Optional[Dict]
```

The last message exchanged with the agent.

Arguments:

- `agent` *Agent* - The agent in the conversation. If None and more than one agent's conversations are found, an error will be raised. If None and only one conversation is found, the last message of the only conversation will be returned.

Returns:

The last message exchanged with the agent.

use_docker

```
@property
def use_docker() -> Union[bool, str, None]
```

Bool value of whether to use docker to execute the code, or str value of the docker image name to use, or None when code execution is disabled.

send

```
def send(message: Union[Dict, str],
         recipient: Agent,
         request_reply: Optional[bool] = None,
         silent: Optional[bool] = False)
```

Send a message to another agent.

Arguments:

- `message` *dict or str* - message to be sent. The message could contain the following fields:
  - content (str or List): Required, the content of the message. (Can be None)
  - function_call (str): the name of the function to be called.
  - name (str): the name of the function to be called.
  - role (str): the role of the message, any role that is not "function" will be modified to "assistant".
  - context (dict): the context of the message, which will be passed to OpenAIWrapper.create. For example, one agent can send a message A as:

```
{
    "content": lambda context: context["use_tool_msg"],
    "context": {
        "use_tool_msg": "Use tool X if they are relevant."
    }
}
```

Next time, one agent can send a message B with a different "use_tool_msg". Then the content of message A will be refreshed to the new "use_tool_msg". So effectively, this provides a way for an agent to send a "link" and modify the content of the "link" later.

- `recipient` *Agent* - the recipient of the message.
- `request_reply` *bool or None* - whether to request a reply from the recipient.
- `silent` *bool or None* - (Experimental) whether to print the message sent.

Raises:

- `ValueError` - if the message can't be converted into a valid ChatCompletion message.

a_send

```
async def a_send(message: Union[Dict, str],
                 recipient: Agent,
                 request_reply: Optional[bool] = None,
                 silent: Optional[bool] = False)
```

(async) Send a message to another agent.

Arguments:

- `message` *dict or str* - message to be sent. The message could contain the following fields:
    - content (str or List): Required, the content of the message. (Can be None)
    - function_call (str): the name of the function to be called.
    - name (str): the name of the function to be called.
    - role (str): the role of the message, any role that is not "function" will be modified to "assistant".
    - context (dict): the context of the message, which will be passed to OpenAIWrapper.create. For example, one agent can send a message A as:

```
{
    "content": lambda context: context["use_tool_msg"],
    "context": {
        "use_tool_msg": "Use tool X if they are relevant."
    }
}
```

Next time, one agent can send a message B with a different "use_tool_msg". Then the content of message A will be refreshed to the new "use_tool_msg". So effectively, this provides a way for an agent to send a "link" and modify the content of the "link" later.

- `recipient` *Agent* - the recipient of the message.
- `request_reply` *bool or None* - whether to request a reply from the recipient.
- `silent` *bool or None* - (Experimental) whether to print the message sent.

Raises:

- `ValueError` - if the message can't be converted into a valid ChatCompletion message.

receive

```
def receive(message: Union[Dict, str],
            sender: Agent,
            request_reply: Optional[bool] = None,
            silent: Optional[bool] = False)
```

Receive a message from another agent.

Once a message is received, this function sends a reply to the sender or stop. The reply can be generated automatically or entered manually by a human.

Arguments:

- `message` *dict or str* - message from the sender. If the type is dict, it may contain the following reserved fields (either content or function_call need to be provided).
    i. "content": content of the message, can be None.
    ii. "function_call": a dictionary containing the function name and arguments. (deprecated in favor of "tool_calls")
    iii. "tool_calls": a list of dictionaries containing the function name and arguments.
    iv. "role": role of the message, can be "assistant", "user", "function", "tool". This field is only needed to distinguish between "function" or "assistant"/"user".
    v. "name": In most cases, this field is not needed. When the role is "function", this field is needed to indicate the function name.
    vi. "context" (dict): the context of the message, which will be passed to OpenAIWrapper.create.
- `sender` - sender of an Agent instance.
- `request_reply` *bool or None* - whether a reply is requested from the sender. If None, the value is determined by `self.reply_at_receive[sender]`.
- `silent` *bool or None* - (Experimental) whether to print the message received.

Raises:

- `ValueError` - if the message can't be converted into a valid ChatCompletion message.

a_receive

```
async def a_receive(message: Union[Dict, str],
                    sender: Agent,
                    request_reply: Optional[bool] = None,
                    silent: Optional[bool] = False)
```

(async) Receive a message from another agent.

Once a message is received, this function sends a reply to the sender or stop. The reply can be generated automatically or entered manually by a human.

Arguments:

- `message` *dict or str* - message from the sender. If the type is dict, it may contain the following reserved fields (either content or function_call need to be provided).
    i. "content": content of the message, can be None.
    ii. "function_call": a dictionary containing the function name and arguments. (deprecated in favor of "tool_calls")
    iii. "tool_calls": a list of dictionaries containing the function name and arguments.
    iv. "role": role of the message, can be "assistant", "user", "function". This field is only needed to distinguish between "function" or "assistant"/"user".
    v. "name": In most cases, this field is not needed. When the role is "function", this field is needed to indicate the function name.
    vi. "context" (dict): the context of the message, which will be passed to OpenAIWrapper.create.
- `sender` - sender of an Agent instance.
- `request_reply` *bool or None* - whether a reply is requested from the sender. If None, the value is determined by `self.reply_at_receive[sender]`.
- `silent` *bool or None* - (Experimental) whether to print the message received.

Raises:

- `ValueError` - if the message can't be converted into a valid ChatCompletion message.

initiate_chat

```
def initiate_chat(recipient: "ConversableAgent",
                  clear_history: Optional[bool] = True,
                  silent: Optional[bool] = False,
                  **context)
```

Initiate a chat with the recipient agent.

Reset the consecutive auto reply counter. If `clear_history` is True, the chat history with the recipient agent will be cleared. `generate_init_message` is called to generate the initial message for the agent.

Arguments:

- `recipient` - the recipient agent.
- `clear_history` *bool* - whether to clear the chat history with the agent.
- `silent` *bool or None* - (Experimental) whether to print the messages for this conversation.
- `**context` - any context information. "message" needs to be provided if the `generate_init_message` method is not overridden.

Raises:

- `RuntimeError` - if any async reply functions are registered and not ignored in sync chat.

a_initiate_chat

```
async def a_initiate_chat(recipient: "ConversableAgent",
                          clear_history: Optional[bool] = True,
                          silent: Optional[bool] = False,
                          **context)
```

(async) Initiate a chat with the recipient agent.

Reset the consecutive auto reply counter. If `clear_history` is True, the chat history with the recipient agent will be cleared. `generate_init_message` is called to generate the initial message for the agent.

Arguments:

- `recipient` - the recipient agent.

- `clear_history` *bool* - whether to clear the chat history with the agent.
- `silent` *bool or None* - (Experimental) whether to print the messages for this conversation.
- `**context` - any context information. "message" needs to be provided if the `generate_init_message` method is not overridden.

reset

```
def reset()
```

Reset the agent.

stop_reply_at_receive

```
def stop_reply_at_receive(sender: Optional[Agent] = None)
```

Reset the reply_at_receive of the sender.

reset_consecutive_auto_reply_counter

```
def reset_consecutive_auto_reply_counter(sender: Optional[Agent] = None)
```

Reset the consecutive_auto_reply_counter of the sender.

clear_history

```
def clear_history(agent: Optional[Agent] = None)
```

Clear the chat history of the agent.

Arguments:

- `agent` - the agent with whom the chat history to clear. If None, clear the chat history with all agents.

generate_oai_reply

```
def generate_oai_reply(
    messages: Optional[List[Dict]] = None,
    sender: Optional[Agent] = None,
    config: Optional[OpenAIWrapper] = None
) -> Tuple[bool, Union[str, Dict, None]]
```

Generate a reply using autogen.oai.

a_generate_oai_reply

```
async def a_generate_oai_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        config: Optional[Any] = None) -> Tuple[bool, Union[str, Dict, None]]
```

Generate a reply using autogen.oai asynchronously.

generate_code_execution_reply

```
def generate_code_execution_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        config: Optional[Union[Dict, Literal[False]]] = None)
```

Generate a reply using code execution.

generate_function_call_reply

```
def generate_function_call_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        config: Optional[Any] = None) -> Tuple[bool, Union[Dict, None]]
```

Generate a reply using function call.

"function_call" replaced by "tool_calls" as of [OpenAI API v1.1.0](https://platform.openai.com/docs/api-reference/chat/create#chat-create-functions) See https://platform.openai.com/docs/api-reference/chat/create#chat-create-functions

a_generate_function_call_reply

```
async def a_generate_function_call_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        config: Optional[Any] = None) -> Tuple[bool, Union[Dict, None]]
```

Generate a reply using async function call.

"function_call" replaced by "tool_calls" as of [OpenAI API v1.1.0](https://platform.openai.com/docs/api-reference/chat/create#chat-create-functions) See https://platform.openai.com/docs/api-reference/chat/create#chat-create-functions

generate_tool_calls_reply

```
def generate_tool_calls_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        config: Optional[Any] = None) -> Tuple[bool, Union[Dict, None]]
```

Generate a reply using tool call.

a_generate_tool_calls_reply

```
async def a_generate_tool_calls_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        config: Optional[Any] = None) -> Tuple[bool, Union[Dict, None]]
```

Generate a reply using async function call.

check_termination_and_human_reply

```
def check_termination_and_human_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        config: Optional[Any] = None) -> Tuple[bool, Union[str, None]]
```

Check if the conversation should be terminated, and if human reply is provided.

This method checks for conditions that require the conversation to be terminated, such as reaching a maximum number of consecutive auto-replies or encountering a termination message. Additionally, it prompts for and processes human input based on the configured human input mode, which can be 'ALWAYS', 'NEVER', or 'TERMINATE'. The method also manages the consecutive auto-reply counter for the conversation and prints relevant messages based on the human input received.

Arguments:

- messages (Optional[List[Dict]]): A list of message dictionaries, representing the conversation history.
- sender (Optional[Agent]): The agent object representing the sender of the message.
- config (Optional[Any]): Configuration object, defaults to the current instance if not provided.

Returns:

- Tuple[bool, Union[str, Dict, None]]: A tuple containing a boolean indicating if the conversation should be terminated, and a human reply which can be a string, a dictionary, or None.

a_check_termination_and_human_reply

```
async def a_check_termination_and_human_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        config: Optional[Any] = None) -> Tuple[bool, Union[str, None]]
```

(async) Check if the conversation should be terminated, and if human reply is provided.

This method checks for conditions that require the conversation to be terminated, such as reaching a maximum number of consecutive auto-replies or encountering a termination message. Additionally, it prompts for and processes human input based on the configured human input mode, which can be 'ALWAYS', 'NEVER', or 'TERMINATE'. The method also manages the consecutive auto-reply counter for the conversation and prints relevant messages based on the human input received.

Arguments:

- messages (Optional[List[Dict]]): A list of message dictionaries, representing the conversation history.
- sender (Optional[Agent]): The agent object representing the sender of the message.
- config (Optional[Any]): Configuration object, defaults to the current instance if not provided.

Returns:

- Tuple[bool, Union[str, Dict, None]]: A tuple containing a boolean indicating if the conversation should be terminated, and a human

reply which can be a string, a dictionary, or None.

generate_reply

```
def generate_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        exclude: Optional[List[Callable]] = None) -> Union[str, Dict, None]
```

Reply based on the conversation history and the sender.

Either messages or sender must be provided. Register a reply_func with `None` as one trigger for it to be activated when `messages` is non-empty and `sender` is `None`. Use registered auto reply functions to generate replies. By default, the following functions are checked in order:

1. check_termination_and_human_reply
2. generate_function_call_reply (deprecated in favor of tool_calls)
3. generate_tool_calls_reply
4. generate_code_execution_reply
5. generate_oai_reply Every function returns a tuple (final, reply). When a function returns final=False, the next function will be checked. So by default, termination and human reply will be checked first. If not terminating and human reply is skipped, execute function or code and return the result. AI replies are generated only when no code execution is performed.

Arguments:

- `messages` - a list of messages in the conversation history.
- `default_reply` *str or dict* - default reply.
- `sender` - sender of an Agent instance.
- `exclude` - a list of functions to exclude.

Returns:

str or dict or None: reply. None if no reply is generated.

a_generate_reply

```
async def a_generate_reply(
        messages: Optional[List[Dict]] = None,
        sender: Optional[Agent] = None,
        exclude: Optional[List[Callable]] = None) -> Union[str, Dict, None]
```

(async) Reply based on the conversation history and the sender.

Either messages or sender must be provided. Register a reply_func with `None` as one trigger for it to be activated when `messages` is non-empty and `sender` is `None`. Use registered auto reply functions to generate replies. By default, the following functions are checked in order:

1. check_termination_and_human_reply
2. generate_function_call_reply
3. generate_tool_calls_reply
4. generate_code_execution_reply
5. generate_oai_reply Every function returns a tuple (final, reply). When a function returns final=False, the next function will be checked. So by default, termination and human reply will be checked first. If not terminating and human reply is skipped, execute function or code and return the result. AI replies are generated only when no code execution is performed.

Arguments:

- `messages` - a list of messages in the conversation history.
- `default_reply` *str or dict* - default reply.
- `sender` - sender of an Agent instance.
- `exclude` - a list of functions to exclude.

Returns:

str or dict or None: reply. None if no reply is generated.

get_human_input

```
def get_human_input(prompt: str) -> str
```

Get human input.

Override this method to customize the way to get human input.

Arguments:

- `prompt` *str* - prompt for the human input.

Returns:

- `str` - human input.

a_get_human_input

```
async def a_get_human_input(prompt: str) -> str
```

(Async) Get human input.

Override this method to customize the way to get human input.

Arguments:

- `prompt` *str* - prompt for the human input.

Returns:

- `str` - human input.

run_code

```
def run_code(code, **kwargs)
```

Run the code and return the result.

Override this function to modify the way to run the code.

Arguments:

- `code` *str* - the code to be executed.
- `**kwargs` - other keyword arguments.

Returns:

A tuple of (exitcode, logs, image).

- `exitcode` *int* - the exit code of the code execution.
- `logs` *str* - the logs of the code execution.
- `image` *str or None* - the docker image used for the code execution.

execute_code_blocks

```
def execute_code_blocks(code_blocks)
```

Execute the code blocks and return the result.

execute_function

```
def execute_function(func_call,
                     verbose: bool = False) -> Tuple[bool, Dict[str, str]]
```

Execute a function call and return the result.

Override this function to modify the way to execute function and tool calls.

Arguments:

- `func_call` - a dictionary extracted from openai message at "function_call" or "tool_calls" with keys "name" and "arguments".

Returns:

A tuple of (is_exec_success, result_dict).

- `is_exec_success` *boolean* - whether the execution is successful.
- `result_dict` - a dictionary with keys "name", "role", and "content". Value of "role" is "function".

  "function_call" deprecated as of OpenAI API v1.1.0 See https://platform.openai.com/docs/api-reference/chat/create#chat-create-function_call

a_execute_function

```
async def a_execute_function(func_call)
```

Execute an async function call and return the result.

Override this function to modify the way async functions and tools are executed.

Arguments:

- `func_call` - a dictionary extracted from openai message at key "function_call" or "tool_calls" with keys "name" and "arguments".

Returns:

A tuple of (is_exec_success, result_dict).

- `is_exec_success` *boolean* - whether the execution is successful.
- `result_dict` - a dictionary with keys "name", "role", and "content". Value of "role" is "function".

  "function_call" deprecated as of [OpenAI API v1.1.0](https://platform.openai.com/docs/api-reference/chat/create#chat-create-function_call) See https://platform.openai.com/docs/api-reference/chat/create#chat-create-function_call

generate_init_message

```
def generate_init_message(**context) -> Union[str, Dict]
```

Generate the initial message for the agent.

Override this function to customize the initial message based on user's request. If not overridden, "message" needs to be provided in the context.

Arguments:

- `**context` - any context information, and "message" parameter needs to be provided.

register_function

```
def register_function(function_map: Dict[str, Callable])
```

Register functions to the agent.

Arguments:

- `function_map` - a dictionary mapping function names to functions.

update_function_signature

```
def update_function_signature(func_sig: Union[str, Dict], is_remove: None)
```

update a function_signature in the LLM configuration for function_call.

Arguments:

- `func_sig` *str or dict* - description/name of the function to update/remove to the model. See: https://platform.openai.com/docs/api-reference/chat/create#chat/create-functions
- `is_remove` - whether removing the function from llm_config with name 'func_sig'

  Deprecated as of [OpenAI API v1.1.0](https://platform.openai.com/docs/api-reference/chat/create#chat-create-function_call) See https://platform.openai.com/docs/api-reference/chat/create#chat-create-function_call

update_tool_signature

```
def update_tool_signature(tool_sig: Union[str, Dict], is_remove: None)
```

update a tool_signature in the LLM configuration for tool_call.

Arguments:

- `tool_sig` *str or dict* - description/name of the tool to update/remove to the model. See: https://platform.openai.com/docs/api-reference/chat/create#chat-create-tools
- `is_remove` - whether removing the tool from llm_config with name 'tool_sig'

can_execute_function

```
def can_execute_function(name: Union[List[str], str]) -> bool
```

Whether the agent can execute the function.

function_map

```
@property
def function_map() -> Dict[str, Callable]
```

Return the function map.

register_for_llm

```
def register_for_llm(
        *,
        name: Optional[str] = None,
        description: Optional[str] = None,
        api_style: Literal["function", "tool"] = "tool") -> Callable[[F], F]
```

Decorator factory for registering a function to be used by an agent.

It's return value is used to decorate a function to be registered to the agent. The function uses type hints to specify the arguments and return type. The function name is used as the default name for the function, but a custom name can be provided. The function description is used to describe the function in the agent's configuration.

Arguments:

name (optional(str)): name of the function. If None, the function name will be used (default: None). description (optional(str)): description of the function (default: None). It is mandatory for the initial decorator, but the following ones can omit it.

- `api_style` - (literal): the API style for function call. For Azure OpenAI API, use version 2023-12-01-preview or later. `"function"` style will be deprecated. For earlier version use `"function"` if `"tool"` doesn't work. See Azure OpenAI documentation for details.

Returns:

The decorator for registering a function to be used by an agent.

Examples:

```
```
@user_proxy.register_for_execution()
@agent2.register_for_llm()
@agent1.register_for_llm(description="This is a very useful function")
def my_function(a: Annotated[str, "description of a parameter"] = "a", b: int, c=3.14) -> str:
    return a + str(b * c)
```
```

For Azure OpenAI versions prior to 2023-12-01-preview, set `api_style` to `"function"` if `"tool"` doesn't work:
`@agent2.register_for_llm(api_style="function") def my_function(a: Annotated[str, "description of a parameter"] = "a", b: int, c=3.14) -> str: return a + str(b * c)`

register_for_execution

```
def register_for_execution(name: Optional[str] = None) -> Callable[[F], F]
```

Decorator factory for registering a function to be executed by an agent.

It's return value is used to decorate a function to be registered to the agent.

Arguments:

name (optional(str)): name of the function. If None, the function name will be used (default: None).

Returns:

The decorator for registering a function to be used by an agent.

Examples:

```
```
@user_proxy.register_for_execution()
@agent2.register_for_llm()
@agent1.register_for_llm(description="This is a very useful function")
def my_function(a: Annotated[str, "description of a parameter"] = "a", b: int, c=3.14):
    return a + str(b * c)
```
```

register_hook

```
def register_hook(hookable_method: Callable, hook: Callable)
```

Registers a hook to be called by a hookable method, in order to add a capability to the agent. Registered hooks are kept in lists (one per hookable method), and are called in their order of registration.

Arguments:

- `hookable_method` - A hookable method implemented by ConversableAgent.
- `hook` - A method implemented by a subclass of AgentCapability.

process_last_message

```
def process_last_message(messages)
```

Calls any registered capability hooks to use and potentially modify the text of the last message, as long as the last message is not a function call or exit command.

print_usage_summary

```
def print_usage_summary(
        mode: Union[str, List[str]] = ["actual", "total"]) -> None
```

Print the usage summary.

get_actual_usage

```
def get_actual_usage() -> Union[None, Dict[str, int]]
```

Get the actual usage summary.

get_total_usage

```
def get_total_usage() -> Union[None, Dict[str, int]]
```

Get the total usage summary.

Edit this page

Previous
« assistant_agent

Next
groupchat »