```
from __future__ import annotations


from abc import abstractmethod

from typing import Sequence



class Message:
    """
    The base abstract Message class.
    Messages are the inputs and outputs of ChatModels.
    """


    def __init__(
        self, content: str, role: str, additional_kwargs: dict = None
    ):
        self.content = content
        self.role = role
        self.additional_kwargs = (
            additional_kwargs if additional_kwargs else {}
        )


    @abstractmethod
    def get_type(self) -> str:
        pass
```

```
class HumanMessage(Message):
    """
    A Message from a human.
    """

    def __init__(
        self,
        content: str,
        role: str = "Human",
        additional_kwargs: dict = None,
        example: bool = False,
    ):
        super().__init__(content, role, additional_kwargs)
        self.example = example

    def get_type(self) -> str:
        return "human"


class AIMessage(Message):
    """
    A Message from an AI.
    """

    def __init__(
        self,
```

```
        content: str,
        role: str = "AI",
        additional_kwargs: dict = None,
        example: bool = False,
    ):
        super().__init__(content, role, additional_kwargs)
        self.example = example

    def get_type(self) -> str:
        return "ai"


class SystemMessage(Message):
    """
    A Message for priming AI behavior, usually passed in as the first of a sequence
    of input messages.
    """

    def __init__(
        self,
        content: str,
        role: str = "System",
        additional_kwargs: dict = None,
    ):
        super().__init__(content, role, additional_kwargs)
```

```
    def get_type(self) -> str:
        return "system"


class FunctionMessage(Message):
    """
    A Message for passing the result of executing a function back to a model.
    """

    def __init__(
        self,
        content: str,
        role: str = "Function",
        name: str = None,
        additional_kwargs: dict = None,
    ):
        super().__init__(content, role, additional_kwargs)
        self.name = name

    def get_type(self) -> str:
        return "function"


class ChatMessage(Message):
    """
    A Message that can be assigned an arbitrary speaker (i.e. role).
```

```
    """

    def __init__(
        self, content: str, role: str, additional_kwargs: dict = None
    ):
        super().__init__(content, role, additional_kwargs)

    def get_type(self) -> str:
        return "chat"



def get_buffer_string(
    messages: Sequence[Message],
    human_prefix: str = "Human",
    ai_prefix: str = "AI",
) -> str:
    string_messages = []
    for m in messages:
        message = f"{m.role}: {m.content}"
        if (
            isinstance(m, AIMessage)
            and "function_call" in m.additional_kwargs
        ):
            message += f"{m.additional_kwargs['function_call']}"
        string_messages.append(message)
```

```
    return "\n".join(string_messages)


def message_to_dict(message: Message) -> dict:
    return {"type": message.get_type(), "data": message.__dict__}


def messages_to_dict(messages: Sequence[Message]) -> list[dict]:
    return [message_to_dict(m) for m in messages]


def message_from_dict(message: dict) -> Message:
    _type = message["type"]
    if _type == "human":
        return HumanMessage(**message["data"])
    elif _type == "ai":
        return AIMessage(**message["data"])
    elif _type == "system":
        return SystemMessage(**message["data"])
    elif _type == "chat":
        return ChatMessage(**message["data"])
    elif _type == "function":
        return FunctionMessage(**message["data"])
    else:
        raise ValueError(f"Got unexpected message type: {_type}")
```

```
def messages_from_dict(messages: list[dict]) -> list[Message]:
    return [message_from_dict(m) for m in messages]
```