

# Package ‘tidyprompt’

January 8, 2025

**Title** Prompt Large Language Models and Enhance Their Functionality

**Version** 0.0.1

**Description** Easily construct prompts and associated logic for interacting with large language models (LLMs).  
'tidyprompt' introduces the concept of prompt wraps, which are building blocks that you can use to quickly turn a simple prompt into a complex one. Prompt wraps do not just modify the prompt text, but also add extraction and validation functions that will be applied to the response of the LLM. This ensures that the user gets the desired output.  
'tidyprompt' can add various features to prompts and their evaluation by LLMs, such as structured output, automatic feedback, retries, reasoning modes, autonomous R function calling, and R code generation and evaluation. It is designed to be compatible with any LLM provider that offers chat completion.

**License** GPL (>= 3) | file LICENSE

**Encoding** UTF-8

**RoxygenNote** 7.3.2

**URL** <https://github.com/tjarkvandemerwe/tidyprompt>,  
<https://tjarkvandemerwe.github.io/tidyprompt/>

**BugReports** <https://github.com/tjarkvandemerwe/tidyprompt/issues>

**Suggests** knitr, rmarkdown, testthat (>= 3.0.0), here, callr, skimr,  
jsonvalidate, tidyjson, DBI

**VignetteBuilder** knitr

**Imports** dplyr, glue, httr2, jsonlite, stringr, utils, cli

**Config/testthat/edition** 3

**NeedsCompilation** no

**Author** Luka Koning [aut, cre, cph],  
Tjark Van de Merwe [aut, cph],  
Kennispunt Twente [fnd]

**Maintainer** Luka Koning <l.koning@kennispunttwente.nl>

**Repository** CRAN

**Date/Publication** 2025-01-08 15:30:02 UTC

## Contents

add_msg_to_chat_history . . . . .	3
add_text . . . . .	5
answer_as_boolean . . . . .	6
answer_as_integer . . . . .	7
answer_as_json . . . . .	8
answer_as_key_value . . . . .	15
answer_as_list . . . . .	17
answer_as_named_list . . . . .	19
answer_as_regex_match . . . . .	20
answer_as_text . . . . .	22
answer_by_chain_of_thought . . . . .	23
answer_by_react . . . . .	25
answer_using_r . . . . .	28
answer_using_sql . . . . .	31
answer_using_tools . . . . .	34
chat_history . . . . .	37
construct_prompt_text . . . . .	38
df_to_string . . . . .	40
extract_from_return_list . . . . .	41
get_chat_history . . . . .	42
get_prompt_wraps . . . . .	43
is_tidyprompt . . . . .	45
llm_break . . . . .	47
llm_feedback . . . . .	49
llm_provider-class . . . . .	50
llm_provider_google_gemini . . . . .	54
llm_provider_groq . . . . .	56
llm_provider_mistral . . . . .	57
llm_provider_ollama . . . . .	59
llm_provider_openai . . . . .	61
llm_provider_openrouter . . . . .	63
llm_provider_xai . . . . .	65
llm_verify . . . . .	67
persistent_chat-class . . . . .	69
prompt_wrap . . . . .	71
quit_if . . . . .	76
r_json_schema_to_example . . . . .	77
send_prompt . . . . .	83
set_chat_history . . . . .	86
set_system_prompt . . . . .	88
skim_with_labels_and_levels . . . . .	89
tidyprompt . . . . .	90
tidyprompt-class . . . . .	92
tools_add_docs . . . . .	96
tools_get_docs . . . . .	98
user_verify . . . . .	100

add\_msg\_to\_chat\_history

*Add a message to a chat history***Description**

This function appends a message to a `chat_history()` object. The function can automatically determine the role of the message to be added based on the last message in the chat history. The role can also be manually specified.

**Usage**

```
add_msg_to_chat_history(
  chat_history,
  message,
  role = c("auto", "user", "assistant", "system", "tool"),
  tool_result = NULL
)
```

**Arguments**

chat_history	A single string, a data.frame which is a valid chat history (see <code>[chat_history()]</code> ), a list containing a valid chat history under key ' <code>\$chat_history</code> ', a <a href="#">tidyprompt-class</a> object, or NULL A <code>chat_history()</code> object
message	A character string representing the message to add
role	A character string representing the role of the message sender. One of: <ul style="list-style-type: none"> <li>"auto": The function automatically determines the role. If the last message was from the user, the role will be "assistant". If the last message was from anything else, the role will be "user"</li> <li>"user": The message is from the user</li> <li>"assistant": The message is from the assistant</li> <li>"system": The message is from the system</li> <li>"tool": The message is from a tool (e.g., indicating the result of a function call)</li> </ul>
tool_result	A logical indicating whether the message is a tool result (e.g., the result of a function call)

## Details

The `chat_history` object may be of different types:

- A single string: The function will create a new chat history object with the string as the first message; the role of that first message will be "user"
- A data.frame: The function will append the message to the data.frame. The data.frame must be a valid chat history; see `chat_history()`
- A list: The function will extract the chat history from the list. The list must contain a valid chat history under the key 'chat\_history'. This may typically be the result from `send_prompt()` when using 'return\_mode = "full"
- A Tidyprompt object (`tidyprompt`): The function will extract the chat history from the object. This will be done by concatenating the 'system\_prompt', 'chat\_history', and 'base\_prompt' into a chat history data.frame. Note that the other properties of the `tidyprompt` object will be lost
- NULL: The function will create a new chat history object with no messages; the message will be the first message

## Value

A `chat_history()` object with the message added as the last row

## Examples

```
chat <- "Hi there!" |>
  chat_history()
chat

chat_from_df <- data.frame(
  role = c("user", "assistant"),
  content = c("Hi there!", "Hello! How can I help you today?")
) |>
  chat_history()
chat_from_df

# `add_msg_to_chat_history()` may be used to add messages to a chat history
chat_from_df <- chat_from_df |>
  add_msg_to_chat_history("Calculate 2+2 for me, please!")
chat_from_df

# You can also continue conversations which originate from `send_prompt()`:
## Not run:
result <- "Hi there!" |>
  send_prompt(return_mode = "full")
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi there!
# --- Receiving response from LLM provider: ---
# It's nice to meet you. Is there something I can help you with, or would you
# like to chat?

# Access the chat history from the result:
```

```

chat_from_send_prompt <- result$chat_history

# Add a message to the chat history:
chat_history_with_new_message <- chat_from_send_prompt |>
  add_msg_to_chat_history("Let's chat!")

# The new chat history can be input for a new tidyprompt:
prompt <- tidyprompt(chat_history_with_new_message)

# You can also take an existing tidyprompt and add the new chat history to it;
# this way, you can continue a conversation using the same prompt wraps
prompt$set_chat_history(chat_history_with_new_message)

# send_prompt() also accepts a chat history as input:
new_result <- chat_history_with_new_message |>
  send_prompt(return_mode = "full")

# You can also create a persistent chat history object from
# a chat history data frame; see ?`persistent_chat-class`
chat <- `persistent_chat-class`$new(llm_provider_ollama(), chat_from_send_prompt)
chat$chat("Let's chat!")

## End(Not run)

```

---

add\_text

*Add text to a tidyprompt*


---

## Description

Add text to a prompt by adding a `prompt_wrap()` which will append the text to the before or after the current prompt text.

## Usage

```
add_text(prompt, text, position = c("after", "before"), sep = "\n\n")
```

## Arguments

prompt	A single string or a <code>tidyprompt()</code> object
text	Text to be added to the current prompt text
position	Where to add the text; either "after" or "before".
sep	Separator to be used between the current prompt text and the text to be added

## Value

A `tidyprompt()` with an added `prompt_wrap()` which will append the text to the end of the current prompt text

**See Also**

Other pre\_built\_prompt\_wraps: [answer\\_as\\_boolean\(\)](#), [answer\\_as\\_integer\(\)](#), [answer\\_as\\_json\(\)](#), [answer\\_as\\_list\(\)](#), [answer\\_as\\_named\\_list\(\)](#), [answer\\_as\\_regex\\_match\(\)](#), [answer\\_as\\_text\(\)](#), [answer\\_by\\_chain\\_of\\_thought\(\)](#), [answer\\_by\\_react\(\)](#), [answer\\_using\\_r\(\)](#), [answer\\_using\\_sql\(\)](#), [answer\\_using\\_tools\(\)](#), [prompt\\_wrap\(\)](#), [quit\\_if\(\)](#), [set\\_system\\_prompt\(\)](#)

Other miscellaneous\_prompt\_wraps: [quit\\_if\(\)](#), [set\\_system\\_prompt\(\)](#)

**Examples**

```
prompt <- "Hi there!" |>
  add_text("How is your day?")
prompt
prompt |>
  construct_prompt_text()
```

---

answer_as_boolean	<i>Make LLM answer as a boolean (TRUE or FALSE)</i>
-------------------	---

---

**Description**

Make LLM answer as a boolean (TRUE or FALSE)

**Usage**

```
answer_as_boolean(
  prompt,
  true_definition = NULL,
  false_definition = NULL,
  add_instruction_to_prompt = TRUE
)
```

**Arguments**

prompt	A single string or a <a href="#">tidyprompt()</a> object
true_definition	(optional) Definition of what would constitute TRUE. This will be included in the instruction to the LLM. Should be a single string
false_definition	(optional) Definition of what would constitute FALSE. This will be included in the instruction to the LLM. Should be a single string
add_instruction_to_prompt	(optional) Add instruction for replying as a boolean to the prompt text. Set to FALSE for debugging if extractions/validations are working as expected (without instruction the answer should fail the validation function, initiating a retry)

**Value**

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM response is a boolean

**See Also**

Other pre\_built\_prompt\_wraps: `add_text()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other answer\_as\_prompt\_wraps: `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`

**Examples**

```
## Not run:
"Are you a large language model?" |>
  answer_as_boolean() |>
  send_prompt()
# --- Sending request to LLM provider (llama3.1:8b): ---
# Are you a large language model?
#
# You must answer with only TRUE or FALSE (use no other characters).
# --- Receiving response from LLM provider: ---
# TRUE
# [1] TRUE

## End(Not run)
```

---

answer\_as\_integer      *Make LLM answer as an integer (between min and max)*

---

**Description**

Make LLM answer as an integer (between min and max)

**Usage**

```
answer_as_integer(
  prompt,
  min = NULL,
  max = NULL,
  add_instruction_to_prompt = TRUE
)
```

**Arguments**

prompt	A single string or a <code>tidyprompt()</code> object
min	(optional) Minimum value for the integer
max	(optional) Maximum value for the integer
add_instruction_to_prompt	(optional) Add instruction for replying as an integer to the prompt text. Set to FALSE for debugging if extractions/validations are working as expected (without instruction the answer should fail the validation function, initiating a retry)

**Value**

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM response is an integer.

**See Also**

Other `pre_built_prompt_wraps`: `add_text()`, `answer_as_boolean()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other `answer_as_prompt_wraps`: `answer_as_boolean()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`

**Examples**

```
## Not run:
"\"What is 5 + 5?\" |>
  answer_as_integer() |>
  send_prompt()
# --- Sending request to LLM provider (llama3.1:8b): ---
#   What is 5 + 5?
#
#   You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
#    10
# [1] 10

## End(Not run)
```



## Description

This function wraps a prompt with settings that ensure the LLM response is a valid JSON object, optionally matching a given JSON schema.

The function can work with all models and providers through text-based handling, but also supports native settings for the OpenAI and Ollama API types. (See argument 'type'.) This means that it is possible to easily switch between providers with different levels of JSON support, while ensuring the results will be in the correct format.

## Usage

```
answer_as_json(
  prompt,
  schema = NULL,
  schema_strict = FALSE,
  schema_in_prompt_as = c("example", "schema"),
  type = c("text-based", "auto", "openai", "ollama", "openai_oo", "ollama_oo")
)
```

## Arguments

prompt	A single string or a <code>tidyprompt()</code> object
schema	A list which represents a JSON schema that the response should match. See example and your API's documentation for more information on defining JSON schemas. Note that the schema should be a list (R object) representing a JSON schema, not a JSON string (use <code>jsonlite::fromJSON()</code> and <code>jsonlite::toJSON()</code> to convert between the two)
schema_strict	If TRUE, the provided schema will be strictly enforced. This option is passed as part of the schema when using type "openai" or "ollama", and when using the other types it is passed to <code>jsonvalidate::json_validate()</code>
schema_in_prompt_as	If providing a schema and when using type "text-based", "openai_oo", or "ollama_oo", this argument specifies how the schema should be included in the prompt: <ul style="list-style-type: none"> <li>"example" (default): The schema will be included as an example JSON object (tends to work best). <code>r_json_schema_to_example()</code> is used to generate the example object from the schema</li> <li>"schema": The schema will be included as a JSON schema</li> </ul>
type	The way that JSON response should be enforced: <ul style="list-style-type: none"> <li>"text-based": Instruction will be added to the prompt asking for JSON; when a schema is provided, this will also be included in the prompt (see argument 'schema_in_prompt_as'). JSON will be parsed from the LLM response and, when a schema is provided, it will be validated against the schema with <code>jsonvalidate::json_validate()</code>. Feedback is sent to the LLM when the response is not valid. This option always works, but may in some cases may be less powerful than the other native JSON options</li> </ul>

- "auto": Automatically determine the type based on 'llm\_provider\$api\_type'. This does not consider model compatibility and could lead to errors; set 'type' manually if errors occur; use 'text-based' if unsure
- "openai" and "ollama": The response format will be set via the relevant API parameters, making the API enforce a valid JSON response. If a schema is provided, it will also be included in the API parameters and also be enforced by the API. When no schema is provided, a request for JSON is added to the prompt (as required by the APIs). Note that these JSON options may not be available for all models of your provider; consult their documentation for more information. If you are unsure or encounter errors, use "text-based"
- "openai\_oo" and "ollama\_oo": Similar to "openai" and "ollama", but if a schema is provided it is not included in the API parameters. Schema validation will be done in R with `jsonvalidate::json_validate()`. This can be useful if you want to use the API's JSON support, but their schema support is limited

Note that the "openai" and "ollama" types may also work for other APIs with a similar structure

### Value

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM response is a valid JSON object

### See Also

Other `pre_built_prompt_wraps`: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other `answer_as_prompt_wraps`: `answer_as_boolean()`, `answer_as_integer()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`

Other json: `r_json_schema_to_example()`

### Examples

```
base_prompt <- "How can I solve 8x + 7 = -23?"

# This example will show how to enforce JSON format in the response,
# with and without a schema, using the 'answer_as_json()' prompt wrap.
# If you use type = 'auto', the function will automatically detect the
# best way to enforce JSON based on the LLM provider you are using.
# Note that the default type is 'text-based', which will work for any provider/model

#### Enforcing JSON without a schema: ####

## Not run:
## Text-based (works for any provider/model):
# Adds request to prompt for a JSON object
# Extracts JSON from textual response (feedback for retry if no JSON received)
```

```

# Parses JSON to R object
json_1 <- base_prompt |>
  answer_as_json() |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# How can I solve  $8x + 7 = -23$ ?
#
# Your must format your response as a JSON object.
# --- Receiving response from LLM provider: ---
# Here is the solution to the equation formatted as a JSON object:
#
# ```
# {
#   "equation": " $8x + 7 = -23$ ",
#   "steps": [
#     {
#       "step": "Subtract 7 from both sides of the equation",
#       "expression": " $-23 - 7$ "
#     },
#     {
#       "step": "Simplify the expression on the left side",
#       "result": " $-30$ "
#     },
#     {
#       "step": "Divide both sides by -8 to solve for x",
#       "expression": " $-30 / -8$ "
#     },
#     {
#       "step": "Simplify the expression on the right side",
#       "result": " $3.75$ "
#     }
#   ],
#   "solution": {
#     "x": 3.75
#   }
# }
# ```

## Ollama:
# - Sets 'format' parameter to 'json', enforcing JSON
# - Adds request to prompt for a JSON object, as is recommended by the docs
# - Parses JSON to R object
json_2 <- base_prompt |>
  answer_as_json(type = "auto") |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# How can I solve  $8x + 7 = -23$ ?
#
# Your must format your response as a JSON object.
# --- Receiving response from LLM provider: ---
# {"steps": [
#   "Subtract 7 from both sides to get  $8x = -30$ ",

```

```

# "Simplify the right side of the equation to get  $8x = -30$ ",
# "Divide both sides by 8 to solve for x, resulting in  $x = -30/8$ ",
# "Simplify the fraction to find the value of x"
# ],
# "value_of_x": "-3.75"}

## OpenAI-type API without schema:
# - Sets 'response_format' parameter to 'json_object', enforcing JSON
# - Adds request to prompt for a JSON object, as is required by the API
# - Parses JSON to R object
json_3 <- base_prompt |>
  answer_as_json(type = "auto") |>
  send_prompt(llm_provider_openai())
# --- Sending request to LLM provider (gpt-4o-mini): ---
# How can I solve  $8x + 7 = -23$ ?
#
# Your must format your response as a JSON object.
# --- Receiving response from LLM provider: ---
# {
#   "solution_steps": [
#     {
#       "step": 1,
#       "operation": "Subtract 7 from both sides",
#       "equation": " $8x + 7 - 7 = -23 - 7$ ",
#       "result": " $8x = -30$ "
#     },
#     {
#       "step": 2,
#       "operation": "Divide both sides by 8",
#       "equation": " $8x / 8 = -30 / 8$ ",
#       "result": " $x = -3.75$ "
#     }
#   ],
#   "solution": {
#     "x": -3.75
#   }
# }

## End(Not run)

#### Enforcing JSON with a schema: ####

# Make a list representing a JSON schema,
# which the LLM response must adhere to:
json_schema <- list(
  name = "steps_to_solve", # Required for OpenAI API
  description = NULL, # Optional for OpenAI API
  schema = list(
    type = "object",
    properties = list(

```

```

    steps = list(
      type = "array",
      items = list(
        type = "object",
        properties = list(
          explanation = list(type = "string"),
          output = list(type = "string")
        ),
        required = c("explanation", "output"),
        additionalProperties = FALSE
      )
    ),
    final_answer = list(type = "string")
  ),
  required = c("steps", "final_answer"),
  additionalProperties = FALSE
)
# 'strict' parameter is set as argument 'answer_as_json()'
)
# Note: when you are not using an OpenAI API, you can also pass just the
# internal 'schema' list object to 'answer_as_json()' instead of the full
# 'json_schema' list object

# Generate example R object based on schema:
r_json_schema_to_example(json_schema)

## Not run:
## Text-based with schema (works for any provider/model):
# - Adds request to prompt for a JSON object
# - Adds schema to prompt
# - Extracts JSON from textual response (feedback for retry if no JSON received)
# - Validates JSON against schema with 'jsonvalidate' package (feedback for retry if invalid)
# - Parses JSON to R object
json_4 <- base_prompt |>
  answer_as_json(schema = json_schema) |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# How can I solve  $8x + 7 = -23$ ?
#
# Your must format your response as a JSON object.
#
# Your JSON object should match this example JSON object:
# {
#   "steps": [
#     {
#       "explanation": "...",
#       "output": "..."
#     }
#   ],
#   "final_answer": "..."
# }
# --- Receiving response from LLM provider: ---
# Here is the solution to the equation:

```

```

#
# ---
# {
#   "steps": [
#     {
#       "explanation": "First, we want to isolate the term with 'x' by
#       subtracting 7 from both sides of the equation.",
#       "output": "8x + 7 - 7 = -23 - 7"
#     },
#     {
#       "explanation": "This simplifies to: 8x = -30",
#       "output": "8x = -30"
#     },
#     {
#       "explanation": "Next, we want to get rid of the coefficient '8' by
#       dividing both sides of the equation by 8.",
#       "output": "(8x) / 8 = (-30) / 8"
#     },
#     {
#       "explanation": "This simplifies to: x = -3.75",
#       "output": "x = -3.75"
#     }
#   ],
#   "final_answer": "-3.75"
# }
# ---

## Ollama with schema:
# - Sets 'format' parameter to 'json', enforcing JSON
# - Adds request to prompt for a JSON object, as is recommended by the docs
# - Adds schema to prompt
# - Validates JSON against schema with 'jsonvalidate' package (feedback for retry if invalid)
json_5 <- base_prompt |>
  answer_as_json(json_schema, type = "auto") |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# How can I solve 8x + 7 = -23?
#
# Your must format your response as a JSON object.
#
# Your JSON object should match this example JSON object:
# {
#   "steps": [
#     {
#       "explanation": "...",
#       "output": "..."
#     }
#   ],
#   "final_answer": "..."
# }
# --- Receiving response from LLM provider: ---
# {

```

```

# "steps": [
#   {
#     "explanation": "First, subtract 7 from both sides of the equation to
#       isolate the term with x.",
#     "output": "8x = -23 - 7"
#   },
#   {
#     "explanation": "Simplify the right-hand side of the equation.",
#     "output": "8x = -30"
#   },
#   {
#     "explanation": "Next, divide both sides of the equation by 8 to solve for x.",
#     "output": "x = -30 / 8"
#   },
#   {
#     "explanation": "Simplify the right-hand side of the equation.",
#     "output": "x = -3.75"
#   }
# ],
# "final_answer": "-3.75"
# }

## OpenAI with schema:
# - Sets 'response_format' parameter to 'json_object', enforcing JSON
# - Adds json_schema to the API request, API enforces JSON adhering schema
# - Parses JSON to R object
json_6 <- base_prompt |>
  answer_as_json(json_schema, type = "auto") |>
  send_prompt(llm_provider_openai())
# --- Sending request to LLM provider (gpt-4o-mini): ---
# How can I solve  $8x + 7 = -23$ ?
# --- Receiving response from LLM provider: ---
# {"steps":[
# {"explanation":"Start with the original equation.",
# "output":"8x + 7 = -23"},
# {"explanation":"Subtract 7 from both sides to isolate the term with x.",
# "output":"8x + 7 - 7 = -23 - 7"},
# {"explanation":"Simplify the left side and the right side of the equation.",
# "output":"8x = -30"},
# {"explanation":"Now, divide both sides by 8 to solve for x.",
# "output":"x = -30 / 8"},
# {"explanation":"Simplify the fraction by dividing both the numerator and the
# denominator by 2.",
# "output":"x = -15 / 4"}
# ], "final_answer":"x = -15/4"}

## End(Not run)

```

---

answer\_as\_key\_value     *Make LLM answer as a list of key-value pairs*

---

**Description**

This function is similar to `answer_as_list()` but instead of returning a list of items, it instructs the LLM to return a list of key-value pairs.

**Usage**

```
answer_as_key_value(
  prompt,
  key_name = "key",
  value_name = "value",
  pair_explanation = NULL,
  n_unique_items = NULL,
  list_mode = c("bullet", "comma")
)
```

**Arguments**

<code>prompt</code>	A single string or a <code>tidyprompt()</code> object
<code>key_name</code>	(optional) A name or placeholder describing the "key" part of each pair
<code>value_name</code>	(optional) A name or placeholder describing the "value" part of each pair
<code>pair_explanation</code>	(optional) Additional explanation of what a pair should be. It should be a single string. It will be appended after the list instruction.
<code>n_unique_items</code>	(optional) Number of unique key-value pairs required in the list
<code>list_mode</code>	(optional) Mode of the list: "bullet" or "comma". <ul style="list-style-type: none"> <li>• "bullet" mode expects pairs like: <pre>-- key1: value1 -- key2: value2</pre> </li> <li>• "comma" mode expects pairs like: <pre>1. key: value, 2. key: value, etc.</pre> </li> </ul>

**Value**

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM response is a list of key-value pairs.

**Examples**

```
## Not run:
"What are a few capital cities around the world?" |>
  answer_as_key_value(
    key_name = "country",
    value_name = "capital"
  ) |>
  send_prompt()
# --- Sending request to LLM provider (llama3.1:8b): ---
# What are a few capital cities around the world?
```



```

#
# Respond with a list of key-value pairs, like so:
# -- <<country 1>>: <<capital 1>>
# -- <<country 2>>: <<capital 2>>
# etc.
# --- Receiving response from LLM provider: ---
# Here are a few:
# -- Australia: Canberra
# -- France: Paris
# -- United States: Washington D.C.
# -- Japan: Tokyo
# -- China: Beijing
# $Australia
# [1] "Canberra"
#
# $France
# [1] "Paris"
#
# $`United States`
# [1] "Washington D.C."
#
# $Japan
# [1] "Tokyo"
#
# $China
# [1] "Beijing"

## End(Not run)

```

---

answer\_as\_list

*Make LLM answer as a list of items*

---

## Description

Make LLM answer as a list of items

## Usage

```

answer_as_list(
  prompt,
  item_name = "item",
  item_explanation = NULL,
  n_unique_items = NULL,
  list_mode = c("bullet", "comma")
)

```

## Arguments

prompt            A single string or a `tidyprompt()` object

**item\_name** (optional) Name of the items in the list  
**item\_explanation** (optional) Additional explanation of what an item should be. Item explanation should be a single string. It will be appended after the list instruction  
**n\_unique\_items** (optional) Number of unique items required in the list  
**list\_mode** (optional) Mode of the list. Either "bullet" or "comma". "bullet mode expects items to be listed with "-" before each item, with a new line for each item (e.g., "- item1\n- item2\n- item3"). "comma" mode expects items to be listed with a number and a period before (e.g., "1. item1, 2. item2, 3. item3"). "comma" mode may be easier for smaller LLMs to use

### Value

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM response is a list of items

### See Also

[answer\\_as\\_named\\_list\(\)](#)

Other `pre_built_prompt_wraps`: [add\\_text\(\)](#), [answer\\_as\\_boolean\(\)](#), [answer\\_as\\_integer\(\)](#), [answer\\_as\\_json\(\)](#), [answer\\_as\\_named\\_list\(\)](#), [answer\\_as\\_regex\\_match\(\)](#), [answer\\_as\\_text\(\)](#), [answer\\_by\\_chain\\_of\\_thought\(\)](#), [answer\\_by\\_react\(\)](#), [answer\\_using\\_r\(\)](#), [answer\\_using\\_sql\(\)](#), [answer\\_using\\_tools\(\)](#), [prompt\\_wrap\(\)](#), [quit\\_if\(\)](#), [set\\_system\\_prompt\(\)](#)

Other `answer_as_prompt_wraps`: [answer\\_as\\_boolean\(\)](#), [answer\\_as\\_integer\(\)](#), [answer\\_as\\_json\(\)](#), [answer\\_as\\_named\\_list\(\)](#), [answer\\_as\\_regex\\_match\(\)](#), [answer\\_as\\_text\(\)](#)

### Examples

```

## Not run:
"What are some delicious fruits?" |>
  answer_as_list(item_name = "fruit", n_unique_items = 5) |>
  send_prompt()
# --- Sending request to LLM provider (llama3.1:8b): ---
# What are some delicious fruits?
#
# Respond with a list, like so:
# -- <<fruit 1>>
# -- <<fruit 2>>
# etc.
# The list should contain 5 unique items.
# --- Receiving response from LLM provider: ---
# Here's the list of delicious fruits:
# -- Strawberries
# -- Pineapples
# -- Mangoes
# -- Blueberries
# -- Pomegranates
# [[1]]
# [1] "Strawberries"
#

```

```

# [[2]]
# [1] "Pineapples"
#
# [[3]]
# [1] "Mangoes"
#
# [[4]]
# [1] "Blueberries"
#
# [[5]]
# [1] "Pomegranates"

## End(Not run)

```

---

answer\_as\_named\_list *Make LLM answer as a named list*

---

## Description

Get a named list from LLM response with optional item instructions and validations.

## Usage

```

answer_as_named_list(
  prompt,
  item_names,
  item_instructions = NULL,
  item_validations = NULL
)

```

## Arguments

prompt	A single string or a <code>tidyprompt()</code> object
item_names	A character vector specifying the expected item names
item_instructions	An optional named list of additional instructions for each item
item_validations	An optional named list of validation functions for each item. Like validation functions for a <code>prompt_wrap()</code> , these functions should return <code>llm_feedback()</code> if the validation fails. If the validation is successful, the function should return TRUE

## Value

A `tidyprompt()` with an added `prompt_wrap()` that ensures the LLM response is a named list with the specified item names, optional instructions, and validations.

**See Also**

[answer\\_as\\_list\(\)](#) [llm\\_feedback\(\)](#)

Other pre\_built\_prompt\_wraps: [add\\_text\(\)](#), [answer\\_as\\_boolean\(\)](#), [answer\\_as\\_integer\(\)](#), [answer\\_as\\_json\(\)](#), [answer\\_as\\_list\(\)](#), [answer\\_as\\_regex\\_match\(\)](#), [answer\\_as\\_text\(\)](#), [answer\\_by\\_chain\\_of\\_thought\(\)](#), [answer\\_by\\_react\(\)](#), [answer\\_using\\_r\(\)](#), [answer\\_using\\_sql\(\)](#), [answer\\_using\\_tools\(\)](#), [prompt\\_wrap\(\)](#), [quit\\_if\(\)](#), [set\\_system\\_prompt\(\)](#)

Other answer\_as\_prompt\_wraps: [answer\\_as\\_boolean\(\)](#), [answer\\_as\\_integer\(\)](#), [answer\\_as\\_json\(\)](#), [answer\\_as\\_list\(\)](#), [answer\\_as\\_regex\\_match\(\)](#), [answer\\_as\\_text\(\)](#)

**Examples**

```
## Not run:
persona <- "Create a persona for me, please." |>
  answer_as_named_list(
    item_names = c("name", "age", "occupation"),
    item_instructions = list(
      name = "The name of the persona",
      age = "The age of the persona",
      occupation = "The occupation of the persona"
    )
  ) |> send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# Create a persona for me, please.
#
# Respond with a named list like so:
# -- name: <<value>> (The name of the persona)
# -- age: <<value>> (The age of the persona)
# -- occupation: <<value>> (The occupation of the persona)
# Each name must correspond to: name, age, occupation
# --- Receiving response from LLM provider: ---
# Here is your persona:
#
# -- name: Astrid Welles
# -- age: 32
# -- occupation: Museum Curator
persona$name
# [1] "Astrid Welles"
persona$age
# [1] "32"
persona$occupation
# [1] "Museum Curator"

## End(Not run)
```

**Description**

Make LLM answer match a specific regex

**Usage**

```
answer_as_regex_match(prompt, regex, mode = c("full_match", "extract_matches"))
```

**Arguments**

prompt	A single string or a <code>tidyprompt()</code> object
regex	A character string specifying the regular expression the response must match
mode	A character string specifying the mode of the regex match. Options are "exact_match" (default) and "extract_all_matches". Under "full_match", the full LLM response must match the regex. If it does not, the LLM will be sent feedback to retry. The full LLM response will be returned if the regex is matched. Under "extract_matches", all matches of the regex in the LLM response will be returned (if present). If the regex is not matched at all, the LLM will be sent feedback to retry. If there is at least one match, the matches will be returned as a character vector

**Value**

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM response matches the specified regex

**See Also**

Other pre\_built\_prompt\_wraps: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other answer\_as\_prompt\_wraps: `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_text()`

**Examples**

```
## Not run:
"\"What would be a suitable e-mail address for cupcake company?\" |>
  answer_as_regex_match("^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\. [a-zA-Z]{2,}$") |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
#   What would be a suitable e-mail address for cupcake company?
#
#   You must answer with a response that matches this regex format:
#     ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\. [a-zA-Z]{2,}$
#     (use no other characters)
# --- Receiving response from LLM provider: ---
#   sweet.treats.cupcakes@gmail.com
# [1] "sweet.treats.cupcakes@gmail.com"
```

```

"what would be a suitable e-mail address for cupcake company?" |>
  add_text("Give three ideas.") |>
  answer_as_regex_match(
    "[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.\\.[a-zA-Z]{2,}",
    mode = "extract_matches"
  ) |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
#   What would be a suitable e-mail address for cupcake company?
#
#   Give three ideas.
#
#   You must answer with a response that matches this regex format:
#     [a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.\\.[a-zA-Z]{2,}
# --- Receiving response from LLM provider: ---
#   Here are three potential email addresses for a cupcake company:
#
#   1. sweettreats.cupcakes@yummail.com
#   2. cupcakes.and.love@flourpower.net
#   3. thecupcakery@gmail.com
# [1] "sweettreats.cupcakes@yummail.com" "cupcakes.and.love@flourpower.net"
# "thecupcakery@gmail.com"

## End(Not run)

```

---

answer\_as\_text

*Make LLM answer as a constrained text response*

---

## Description

Make LLM answer as a constrained text response

## Usage

```

answer_as_text(
  prompt,
  max_words = NULL,
  max_characters = NULL,
  add_instruction_to_prompt = TRUE
)

```

## Arguments

prompt	A single string or a <code>tidyprompt()</code> object
max_words	(optional) Maximum number of words allowed in the response. If specified, responses exceeding this limit will fail validation
max_characters	(optional) Maximum number of characters allowed in the response. If specified, responses exceeding this limit will fail validation

```

add_instruction_to_prompt
    (optional) Add instruction for replying within the constraints to the prompt text.
    Set to FALSE for debugging if extractions/validations are working as expected
    (without instruction the answer should fail the validation function, initiating a
    retry)

```

### Value

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM response conforms to the specified constraints

### See Also

Other `pre_built_prompt_wraps`: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other `answer_as_prompt_wraps`: `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`

### Examples

```

## Not run:
"What is a large language model?" |>
  answer_as_text(max_words = 10) |>
  send_prompt()
# --- Sending request to LLM provider (llama3.1:8b): ---
# What is a large language model?
#
# You must provide a text response. The response must be at most 10 words.
# --- Receiving response from LLM provider: ---
# A type of AI that processes and generates human-like text.
# [1] "A type of AI that processes and generates human-like text."

## End(Not run)

```

---

answer\_by\_chain\_of\_thought

*Set chain of thought mode for a prompt*

---

### Description

This function enables chain of thought mode for evaluation of a prompt or a `tidyprompt()`. In chain of thought mode, the large language model (LLM) is asked to think step by step to arrive at a final answer. It is hypothesized that this may increase LLM performance at solving complex tasks. Chain of thought mode is inspired by the method described in Wei et al. (2022).

**Usage**

```
answer_by_chain_of_thought(
  prompt,
  extract_from_finish_brackets = TRUE,
  extraction_lenience = TRUE
)
```

**Arguments**

`prompt` A single string or a `tidyprompt()` object

`extract_from_finish_brackets` A logical indicating whether the final answer should be extracted from the text inside the "FINISH..." brackets.

`extraction_lenience` A logical indicating whether the extraction function should be lenient. If TRUE, the extraction function will attempt to extract the final answer even if it cannot be extracted from within the brackets, by extracting everything after the final occurrence of 'FINISH' (if present). This may be useful for smaller LLMs which may not follow the output format as strictly

**Value**

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM follows the chain of thought mode in answering the prompt

**References**

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. [doi:10.48550/arXiv.2201.11903](https://doi.org/10.48550/arXiv.2201.11903)

**See Also**

Other `pre_built_prompt_wraps`: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other `answer_by_prompt_wraps`: `answer_by_react()`

**Examples**

```
## Not run:
"\"What is 5 + 5?\" |>
  answer_by_chain_of_thought() |>
  answer_as_integer() |>
  send_prompt()
# --- Sending request to LLM provider (llama3.1:8b): ---
# You are given a user's prompt.
# To answer the user's prompt, you need to think step by step to arrive at a final answer.
```



```

#
# ----- START OF USER'S PROMPT -----
# What is 5 + 5?
#
# You must answer with only an integer (use no other characters).
# ----- END OF USER'S PROMPT -----
#
# What are the steps you would take to answer the user's prompt?
# Describe your thought process in the following format:
# >> step 1: <step 1 description>
# >> step 2: <step 2 description>
# (etc.)
#
# When you are done, you must type:
# FINISH[<put here your final answer to the user's prompt>]
#
# Make sure your final answer follows the logical conclusion of your thought process.
# --- Receiving response from LLM provider: ---
# >> step 1: Recognize that the user's prompt is a simple arithmetic question asking for
# the sum of two numbers.
#
# >> step 2: Identify the specific numbers mentioned in the prompt, which are 5 and 5.
#
# >> step 3: Recall basic addition facts from memory or knowledge.
# In this case, knowing that  $5 + 0 = 5$ ,  $5 + 1 = 6$ ,  $5 + 2 = 7$ , and so on up to  $5 + 4 = 9$ ,
# it is clear that the next step in adding 5 to itself would be to keep the same sum.
#
# >> step 4: Apply the addition principle by considering the original number (5) as being
# added to itself once more. Since  $5 + 1 = 6$  and the process of adding 1 increases the total
# sum, then  $5 + 2$  must equal 7, thus indicating a direct progression in the sum as the additive
# amount increases.
#
# >> step 5: From steps 3 and 4, conclude that since 5 added to itself (once) would be 10,
# then when it is added again (to itself), it would continue from the last result which was
# achieved by adding one more than the original number. Therefore, it must follow the pattern
# established in step 3 where each additional amount increases the total sum.
#
# >> step 6: Now knowing that the operation of adding a second 5 to the first one results in
# increasing the sum to the next integer after 10 (based on steps 3 and 4), determine what the
# next integer is after 10. Knowing that integers are whole numbers without fractions, it's
# clear that if you add another unit to 10, it becomes 11.
#
# FINISH[10]
# [1] 10

## End(Not run)

```

## Description

This function enables ReAct mode for the evaluation of a prompt or a `tidyprompt()`. In ReAct mode, the large language model (LLM) is asked to think step by step, each time detailing a thought, action, and observation, to eventually arrive at a final answer. It is hypothesized that this may increase LLM performance at solving complex tasks. ReAct mode is inspired by the method described in Yao et al. (2022).

## Usage

```
answer_by_react(  
  prompt,  
  extract_from_finish_brackets = TRUE,  
  extraction_lenience = TRUE  
)
```

## Arguments

<code>prompt</code>	A single string or a <code>tidyprompt()</code> object
<code>extract_from_finish_brackets</code>	A logical indicating whether the final answer should be extracted from the text inside the "FINISH..." brackets
<code>extraction_lenience</code>	A logical indicating whether the extraction function should be lenient. If TRUE, the extraction function will attempt to extract the final answer even if it cannot be extracted from within the brackets, by extracting everything after the final occurrence of 'FINISH' (if present). This may be useful for smaller LLMs which may not follow the output format as strictly

## Details

Please note that ReAct mode may be most useful if in combination with tools that the LLM can use. See, for example, `'add_tools()'` for enabling R function calling, or, for example, `'answer_as_code()'` with `'output_as_tool = TRUE'` for enabling R code evaluation as a tool.

## Value

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM follows the ReAct mode in answering the prompt

## References

Yao, S., Wu, Y., Cheung, W., Wang, Z., Narasimhan, K., & Kong, L. (2022). ReAct: Synergizing Reasoning and Acting in Language Models. [doi:10.48550/arXiv.2210.03629](https://arxiv.org/abs/2210.03629)

## See Also

[answer\\_using\\_tools\(\)](#) [answer\\_using\\_r\(\)](#)

Other pre\_built\_prompt\_wraps: [add\\_text\(\)](#), [answer\\_as\\_boolean\(\)](#), [answer\\_as\\_integer\(\)](#), [answer\\_as\\_json\(\)](#), [answer\\_as\\_list\(\)](#), [answer\\_as\\_named\\_list\(\)](#), [answer\\_as\\_regex\\_match\(\)](#), [answer\\_as\\_text\(\)](#),

`answer_by_chain_of_thought()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`,  
`prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other `answer_by_prompt_wraps`: `answer_by_chain_of_thought()`

## Examples

```
## Not run:
"What is 5 + 5?" |>
  answer_by_react() |>
  answer_as_integer() |>
  send_prompt()
# --- Sending request to LLM provider (llama3.1:8b): ---
# You are given a user's prompt.
# To answer the user's prompt, you need to think step by step,
# take an action if needed, and then return a final answer.
#
# ----- START OF USER'S PROMPT -----
# What is 5 + 5?
#
# You must answer with only an integer (use no other characters).
# ----- END OF USER'S PROMPT -----
#
# Use the following structure:
# Thought: <describe your thought process>
# Action: <if needed, describe the action you take (e.g., look up information)>
# Observation: <describe the result or observation from the action>
# (Repeat Thought -> Action -> Observation as necessary)
#
# When you are done, you must type:
# FINISH[<put here your final answer to the user's prompt>]
#
# Ensure your final answer aligns with your reasoning and observations.
# --- Receiving response from LLM provider: ---
# Thought: The problem is asking for the sum of two numbers, 5 and 5.
#
# Action: None needed, as this is a simple arithmetic operation that can be performed mentally.
#
# Observation: I can easily add these two numbers together in my mind to get the result.
#
# Thought: To find the sum, I will simply add the two numbers together: 5 + 5 = ?
#
# Action: Perform the addition.
#
# Observation: The result of adding 5 and 5 is 10.
#
# FINISH[10]
# [1] 10

## End(Not run)
```

---

 answer\_using\_r

*Enable LLM to draft and execute R code*


---

## Description

This function adds a prompt wrap to a `tidyprompt()` that instructs the LLM to answer the prompt with R code. There are various options to customize the behavior of this prompt wrap, concerning the evaluation of the R code, the packages that may be used, the objects that already exist in the R session, and if the console output that should be sent back to the LLM.

## Usage

```
answer_using_r(
  prompt,
  add_text = "You must code in the programming language 'R' to answer this prompt.",
  pkgs_to_use = c(),
  objects_to_use = list(),
  list_packages = TRUE,
  list_objects = TRUE,
  skim_dataframes = TRUE,
  evaluate_code = FALSE,
  r_session_options = list(),
  output_as_tool = FALSE,
  return_mode = c("full", "code", "console", "object", "formatted_output", "llm_answer")
)
```

## Arguments

<code>prompt</code>	A single string or a <code>tidyprompt()</code> object
<code>add_text</code>	Single string which will be added to the prompt text, informing the LLM that they must code in R to answer the prompt
<code>pkgs_to_use</code>	A character vector of package names that may be used in the R code that the LLM will generate. If evaluating the R code, these packages will be pre-loaded in the R session
<code>objects_to_use</code>	A named list of objects that may be used in the R code that the LLM will generate. If evaluating the R code, these objects will be pre-loaded in the R session. The names of the list will be used as the object names in the R session
<code>list_packages</code>	Logical indicating whether the LLM should be informed about the packages that may be used in their R code (if TRUE, a list of the loaded packages will be shown in the initial prompt)
<code>list_objects</code>	Logical indicating whether the LLM should be informed about the existence of 'objects_to_use' (if TRUE, a list of the objects plus their types will be shown in the initial prompt)

skim_dataframes	Logical indicating whether the LLM should be informed about the structure of dataframes present in 'objects_to_use' (if TRUE, a skim summary of each data.frame type object will be shown in the initial prompt). This uses the function <code>skim_with_labels_and_levels()</code>
evaluate_code	Logical indicating whether the R code should be evaluated. If TRUE, the R code will be evaluated in a separate R session (using 'callr' to create an isolated R session via <code>r_session</code> ). Note that setting this to 'TRUE' means that code generated by the LLM will run on your system; use this setting with caution
r_session_options	A list of options to pass to the <code>r_session</code> . This can be used to customize the R session. See <code>r_session_options</code> for the available options. If no options are provided, the default options will be used but with 'system_profile' and 'user_profile' set to FALSE
output_as_tool	Logical indicating whether the console output of the evaluated R code should be sent back to the LLM, meaning the LLM will use R code as a tool to formulate a final answer to the prompt. If TRUE, the LLM can decide if they can answer the prompt with the output, or if they need to modify their R code. Once the LLM does not provide new R code (i.e., the prompt is being answered) this prompt wrap will end (it will continue for as long as the LLM provides R code). When this option is enabled, the resulting <code>prompt_wrap()</code> will be of type 'tool'. If TRUE, the return mode will also always be set to 'llm_answer'
return_mode	Single string indicating the return mode. One of: <ul style="list-style-type: none"> <li>'full': Return a list with the final LLM answer, the extracted R code, and (if argument 'evaluate_code' is TRUE) the output of the R code</li> <li>'code': Return only the extracted R code</li> <li>'console': Return only the console output of the evaluated R code</li> <li>'object': Return only the object produced by the evaluated R code</li> <li>'formatted_output': Return a formatted string with the extracted R code and its console output, and a print of the last object (this is identical to how it would be presented to the LLM if 'output_as_tool' is TRUE)</li> <li>'llm_answer': Return only the final LLM answer</li> </ul> <p>When choosing 'console' or 'object', an additional instruction will be added to the prompt text to inform the LLM about the expected output of the R code. If 'output_as_tool' is TRUE, the return mode will always be set to 'llm_answer' (as the LLM will be using the R code as a tool to answer the prompt)</p>

## Details

For the evaluation of the R code, the 'callr' package is required. Please note: automatic evaluation of generated R code may be dangerous to your system; you must use this function with caution.

## Value

A `tidyprompt()` object with the `prompt_wrap()` added to it, which will handle R code generation and possibly evaluation

**See Also**

[answer\\_using\\_tools\(\)](#)

Other pre\_built\_prompt\_wraps: [add\\_text\(\)](#), [answer\\_as\\_boolean\(\)](#), [answer\\_as\\_integer\(\)](#), [answer\\_as\\_json\(\)](#), [answer\\_as\\_list\(\)](#), [answer\\_as\\_named\\_list\(\)](#), [answer\\_as\\_regex\\_match\(\)](#), [answer\\_as\\_text\(\)](#), [answer\\_by\\_chain\\_of\\_thought\(\)](#), [answer\\_by\\_react\(\)](#), [answer\\_using\\_sql\(\)](#), [answer\\_using\\_tools\(\)](#), [prompt\\_wrap\(\)](#), [quit\\_if\(\)](#), [set\\_system\\_prompt\(\)](#)

Other answer\_using\_prompt\_wraps: [answer\\_using\\_sql\(\)](#), [answer\\_using\\_tools\(\)](#)

**Examples**

```
## Not run:
# Prompt to value calculated with R
avg_miles_per_gallon <- paste0(
  "Using my data,",
  " calculate the average miles per gallon (mpg) for cars with 6 cylinders."
) |>
answer_as_integer() |>
answer_using_r(
  pkgs_to_use = c("dplyr"),
  objects_to_use = list(mtcars = mtcars),
  evaluate_code = TRUE,
  output_as_tool = TRUE
) |>
send_prompt()
avg_miles_per_gallon

# Prompt to linear model object in R
model <- paste0(
  "Using my data, create a statistical model",
  " investigating the relationship between two variables."
) |>
answer_using_r(
  objects_to_use = list(data = mtcars),
  evaluate_code = TRUE,
  return_mode = "object"
) |>
prompt_wrap(
  validation_fn = function(x) {
    if (!inherits(x, "lm"))
      return(llm_feedback("The output should be a linear model object."))
    return(x)
  }
) |>
send_prompt()
summary(model)

# Prompt to plot object in R
plot <- paste0(
  "Create a scatter plot of miles per gallon (mpg) versus",
  " horsepower (hp) for the cars in my data.",
  " Use different colors to represent the number of cylinders (cyl).",

```

```

    " Be very creative and make the plot look nice but also a little crazy!"
  ) |>
  answer_using_r(
    pkgs_to_use = c("ggplot2"),
    objects_to_use = list(mtcars = mtcars),
    evaluate_code = TRUE,
    return_mode = "object"
  ) |>
  send_prompt()
plot

## End(Not run)

```

---

answer\_using\_sql      *Enable LLM to draft and execute SQL queries on a database*

---

## Description

Enable LLM to draft and execute SQL queries on a database

## Usage

```

answer_using_sql(
  prompt,
  add_text = paste0("You must code in SQL to answer this prompt.",
    " You must provide all SQL code between ```sql and ```.", "\n\n",
    "Never make assumptions about the possible values in the tables.\n",
    "Instead, execute SQL queries to retrieve information you need."),
  conn,
  list_tables = TRUE,
  describe_tables = TRUE,
  evaluate_code = FALSE,
  output_as_tool = FALSE,
  return_mode = c("full", "code", "object", "formatted_output", "llm_answer")
)

```

## Arguments

prompt	A single string or a <code>tidyprompt()</code> object
add_text	Single string which will be added to the prompt text, informing the LLM that they must use SQL to answer the prompt
conn	A <code>DBIConnection</code> object to the SQL database
list_tables	Logical indicating whether to list tables available in the database in the prompt text
describe_tables	Logical indicating whether to describe the tables available in the database in the prompt text. If <code>TRUE</code> , the columns of each table will be listed

evaluate_code	Logical indicating whether to evaluate the SQL code. If TRUE, the SQL code will be executed on the database and the results will be returned. Use with caution, as this allows the LLM to execute arbitrary SQL code
output_as_tool	Logical indicating whether to return the output as a tool result. If TRUE, the output of the SQL query will be sent back to the LLM as a tool result. The LLM can then provide a final answer or try another query. This can continue until the LLM provides a final answer without any SQL code
return_mode	Character string indicating the return mode. Options are: <ul style="list-style-type: none"> <li>• "full": Return a list containing the SQL code, output, and formatted output</li> <li>• "code": Return only the SQL code</li> <li>• "object": Return only the query result object</li> <li>• "formatted_output": Return the formatted output: a string detailing the SQL code and query result object. This is identical to how the LLM would see the output when output_as_tool is TRUE</li> <li>• "llm_answer": Return the LLM answer. If output as tool is TRUE, the return mode will always be "llm_answer" (since the LLM uses SQL to provide a final answer)</li> </ul>

### Value

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the LLM will use SQL to answer the prompt

### See Also

Other pre\_built\_prompt\_wraps: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_tools()`, `prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other answer\_using\_prompt\_wraps: `answer_using_r()`, `answer_using_tools()`

### Examples

```
## Not run:
# Create an in-memory SQLite database
conn <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")

# Create a sample table of customers
DBI::dbExecute(conn, "
CREATE TABLE
  customers (
    id INTEGER PRIMARY KEY,
    name TEXT,
    email TEXT,
    country TEXT
  );
")

# Insert some sample customer data
```



```

DBI::dbExecute(conn, "
INSERT INTO
  customers (name, email, country)
VALUES
  ('Alice', 'alice@example.com', 'USA'),
  ('Bob', 'bob@example.com', 'Canada'),
  ('Charlie', 'charlie@example.com', 'UK'),
  ('Diana', 'diana@example.com', 'USA');
")

# Create another sample table for orders
DBI::dbExecute(conn, "
CREATE TABLE orders (
  order_id INTEGER PRIMARY KEY,
  customer_id INTEGER,
  product TEXT,
  amount REAL,
  order_date TEXT,
  FOREIGN KEY(customer_id) REFERENCES customers(id)
);
")

# Insert some sample orders
DBI::dbExecute(conn, "
INSERT INTO
  orders (customer_id, product, amount, order_date)
VALUES
  (1, 'Widget', 19.99, '2024-01-15'),
  (1, 'Gadget', 29.99, '2024-01-17'),
  (2, 'Widget', 19.99, '2024-02-10'),
  (3, 'SuperWidget', 49.99, '2024-03-05'),
  (4, 'Gadget', 29.99, '2024-04-01'),
  (1, 'Thingamajig', 9.99, '2024-04-02');
")

# Ask LLM a question which it will answer using the SQL database:
"Where are my customers from?" |>
  answer_using_sql(
    conn = conn,
    evaluate_code = TRUE,
    output_as_tool = TRUE
  ) |>
  send_prompt(llm_provider_openai())
# --- Sending request to LLM provider (gpt-4o-mini): ---
# Where are my customers from?
#
# You must code in SQL to answer this prompt. You must provide all SQL code
# between ```sql and ```.
#
# Never make assumptions about the possible values in the tables.
# Instead, execute SQL queries to retrieve information you need.
#
# These tables are available in the database:

```

```

# customers, orders
#
# Table descriptions:
# - customers
# Columns: id, name, email, country
#
# - orders
# Columns: order_id, customer_id, product, amount, order_date
#
# Your SQL query will be executed on the database. The results will be sent back
# to you. After seeing the results, you can either provide a final answer or try
# another SQL query. When you provide your final answer, do not include any SQL code.
# --- Receiving response from LLM provider: ---
# ```sql
# SELECT DISTINCT country FROM customers;
# ```
# --- Sending request to LLM provider (gpt-4o-mini): ---
# --- SQL code: ---
# SELECT DISTINCT country FROM customers;
#
# --- Query results: ---
# country
# 1 USA
# 2 Canada
# 3 UK
# --- Receiving response from LLM provider: ---
# Based on the query results, your customers are from the following countries:
# USA, Canada, and UK.
# [1] "Based on the query results, your customers are from the following countries:
# USA, Canada, and UK."

## End(Not run)

```

---

answer\_using\_tools      *Enable LLM to call R functions*

---

## Description

This function adds the ability for the a LLM to call R functions. Users can specify a list of functions that the LLM can call, and the prompt will be modified to include information, as well as an accompanying extraction function to call the functions (handled by [send\\_prompt\(\)](#)). Documentation for the functions is extracted from the help file (if available), or from documentation added by [tools\\_add\\_docs\(\)](#)

## Usage

```

answer_using_tools(
  prompt,
  tools = list(),
  type = c("text-based", "auto", "openai", "ollama")
)

```

**Arguments**

prompt	A single string or a <code>tidyprompt()</code> object
tools	An R function or a list of R functions that the LLM can call. If the function has been documented in a help file (e.g., because it is part of a package), the documentation will be parsed from the help file. If it is a custom function, documentation should be added with <code>tools_add_docs()</code>
type	(optional) The way that tool calling should be enabled. 'auto' will automatically determine the type based on 'llm_provider\$api_type' (note that this does not consider model compatibility, and could lead to errors; set 'type' manually if errors occur). 'openai' and 'ollama' will set the relevant API parameters. 'text-based' will provide function definitions in the prompt, extract function calls from the LLM response, and call the functions, providing the results back via <code>llm_feedback()</code> . 'text-based' always works, but may be inefficient for APIs that support tool calling natively. However, 'text-based' may be more reliable and flexible, especially when combining with other prompt wraps. 'openai' and 'ollama' may not allow for retries if the function call did not provide the expected result. Note that when using 'openai' or 'ollama', tool calls are not counted as interactions and may continue indefinitely (use with caution)

**Details**

Note that this method of function calling is purely text-based. This makes it suitable for any LLM and any LLM provider. However, 'native' function calling (where the LLM model provider restricts the model to special tokens that can be used to call functions) may perform better in terms of accuracy and efficiency. 'tidyprompt' may support 'native' function calling in the future

**Value**

A `tidyprompt()` with an added `prompt_wrap()` which will allow the LLM to call R functions

**See Also**

`answer_using_r()` `tools_get_docs()`

Other pre\_built\_prompt\_wraps: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `prompt_wrap()`, `quit_if()`, `set_system_prompt()`

Other answer\_using\_prompt\_wraps: `answer_using_r()`, `answer_using_sql()`

Other tools: `tools_add_docs()`, `tools_get_docs()`

**Examples**

```
## Not run:
# When using functions from base R or R packages,
# documentation is automatically extracted from help files:
"What are the files in my current directory?" |>
  answer_using_tools(dir) |> # 'dir' function is from base R
  send_prompt()
```

```

## End(Not run)

# Custom functions may also be provided;
#   in this case, some documentation is extracted from the function's formals;
#   descriptions may be added manually. See below

# Example fake weather function to add to the prompt:
temperature_in_location <- function(
  location = c("Amsterdam", "Utrecht", "Enschede"),
  unit = c("Celcius", "Fahrenheit")
) {
  location <- match.arg(location)
  unit <- match.arg(unit)

  temperature_celcius <- switch(
    location,
    "Amsterdam" = 32.5,
    "Utrecht" = 19.8,
    "Enschede" = 22.7
  )

  if (unit == "Celcius") {
    return(temperature_celcius)
  } else {
    return(temperature_celcius * 9/5 + 32)
  }
}

# Generate documentation for a function
#   (based on formals, & help file if available)
docs <- tools_get_docs(temperature_in_location)

# The types get inferred from the function's formals
# However, descriptions are still missing as the function is not from a package
# We can modify the documentation object to add descriptions:
docs$description <- "Get the temperature in a location"
docs$arguments$unit$description <- "Unit in which to return the temperature"
docs$arguments$location$description <- "Location for which to return the temperature"
docs$return$description <- "The temperature in the specified location and unit"
# (See `?tools_add_docs` for more details on the structure of the documentation)

# When we are satisfied with the documentation, we can add it to the function:
temperature_in_location <- tools_add_docs(temperature_in_location, docs)

## Not run:
# Now the LLM can use the function:
"Hi, what is the weather in Enschede? Give me Celcius degrees" |>
  answer_using_tools(temperature_in_location) |>
  send_prompt()

## End(Not run)

```

---

`chat_history`*Create or validate chat\_history object*

---

## Description

This function creates and validates a `chat_history` object, ensuring that it matches the expected format with `'role'` and `'content'` columns. It has separate methods for `data.frame` and character inputs and includes a helper function to add a system prompt to the chat history.

## Usage

```
chat_history(chat_history)
```

## Arguments

`chat_history` A single string, a `data.frame` with `'role'` and `'content'` columns, or `NULL`. If a `data.frame` is provided, it should contain `'role'` and `'content'` columns, where `'role'` is either `'user'`, `'assistant'`, or `'system'`, and `'content'` is a character string representing a chat message

## Value

A valid chat history `data.frame` (of class `chat_history`)

## Examples

```
chat <- "Hi there!" |>
  chat_history()
chat

chat_from_df <- data.frame(
  role = c("user", "assistant"),
  content = c("Hi there!", "Hello! How can I help you today?")
) |>
  chat_history()
chat_from_df

# `add_msg_to_chat_history()` may be used to add messages to a chat history
chat_from_df <- chat_from_df |>
  add_msg_to_chat_history("Calculate 2+2 for me, please!")
chat_from_df

# You can also continue conversations which originate from `send_prompt()`:
## Not run:
result <- "Hi there!" |>
  send_prompt(return_mode = "full")
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi there!
# --- Receiving response from LLM provider: ---
```

```

# It's nice to meet you. Is there something I can help you with, or would you
# like to chat?

# Access the chat history from the result:
chat_from_send_prompt <- result$chat_history

# Add a message to the chat history:
chat_history_with_new_message <- chat_from_send_prompt |>
  add_msg_to_chat_history("Let's chat!")

# The new chat history can be input for a new tidyprompt:
prompt <- tidyprompt(chat_history_with_new_message)

# You can also take an existing tidyprompt and add the new chat history to it;
# this way, you can continue a conversation using the same prompt wraps
prompt$set_chat_history(chat_history_with_new_message)

# send_prompt() also accepts a chat history as input:
new_result <- chat_history_with_new_message |>
  send_prompt(return_mode = "full")

# You can also create a persistent chat history object from
# a chat history data frame; see ?`persistent_chat-class`
chat <- `persistent_chat-class`$new(llm_provider_ollama(), chat_from_send_prompt)
chat$chat("Let's chat!")

## End(Not run)

```

---

construct\_prompt\_text *Construct prompt text from a [tidyprompt](#) object*

---

## Description

Construct prompt text from a [tidyprompt](#) object

## Usage

```
construct_prompt_text(x, llm_provider = NULL)
```

## Arguments

x	A <a href="#">tidyprompt</a> object
llm_provider	An optional <a href="#">llm_provider</a> object. This may sometimes affect the prompt text construction

## Value

The constructed prompt text

**See Also**

Other tidyprompt: [get\\_chat\\_history\(\)](#), [get\\_prompt\\_wraps\(\)](#), [is\\_tidyprompt\(\)](#), [set\\_chat\\_history\(\)](#), [tidyprompt\(\)](#), [tidyprompt-class](#)

**Examples**

```
prompt <- tidyprompt("Hi!")
print(prompt)

# Add to a tidyprompt using a prompt wrap:
prompt <- tidyprompt("Hi!") |>
  add_text("How are you?")
print(prompt)

# Strings can be input for prompt wraps; therefore,
# a call to tidyprompt() is not necessary:
prompt <- "Hi" |>
  add_text("How are you?")

# Example of adding extraction & validation with a prompt_wrap():
prompt <- "Hi" |>
  add_text("What is 5 + 5?") |>
  answer_as_integer()

## Not run:
# tidyprompt objects are evaluated by send_prompt(), which will
# handle construct the prompt text, send it to the LLM provider,
# and apply the extraction and validation functions from the tidyprompt object
prompt |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi
#
# What is 5 + 5?
#
# You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
# 10
# [1] 10

# See prompt_wrap() and send_prompt() for more details

## End(Not run)

# `tidyprompt` objects may be validated with these helpers:
is_tidyprompt(prompt) # Returns TRUE if input is a valid tidyprompt object

# Get base prompt text
base_prompt <- prompt$base_prompt

# Get all prompt wraps
prompt_wraps <- prompt$get_prompt_wraps()
```

```

# Alternative:
prompt_wraps <- get_prompt_wraps(prompt)

# Construct prompt text
prompt_text <- prompt$construct_prompt_text()
# Alternative:
prompt_text <- construct_prompt_text(prompt)

# Set chat history (affecting also the base prompt)
chat_history <- data.frame(
  role = c("user", "assistant", "user"),
  content = c("What is 5 + 5?", "10", "And what is 5 + 6?")
)
prompt$set_chat_history(chat_history)

# Get chat history
chat_history <- prompt$get_chat_history()

```

---

df\_to\_string

*Convert a dataframe to a string representation*


---

### Description

Converts a data frame to a string format, intended for sending it to a LLM (or for display or logging).

### Usage

```
df_to_string(df, how = c("wide", "long"))
```

### Arguments

df	A data.frame object to be converted to a string
how	In what way the df should be converted to a string; either "wide" or "long". "wide" presents column names on the first row, followed by the row values on each new row. "long" presents the values of each row together with the column names, repeating for every row after two lines of whitespace

### Value

A single string representing the df

### See Also

Other text\_helpers: [skim\\_with\\_labels\\_and\\_levels\(\)](#), [vector\\_list\\_to\\_string\(\)](#)



## Examples

```
cars |>
  head(5) |>
  df_to_string(how = "wide")
```

```
cars |>
  head(5) |>
  df_to_string(how = "long")
```

---

extract\_from\_return\_list

*Function to extract a specific element from a list*

---

## Description

This function is intended as a helper function for piping with output from `send_prompt()` when using `return_mode = "full"`. It allows to extract a specific element from the list returned by `send_prompt()`, which can be useful for further piping.

## Usage

```
extract_from_return_list(list, name_of_element = "response")
```

## Arguments

`list` A list, typically the output from `send_prompt()` with `return_mode = "full"`  
`name_of_element` A character string with the name of the element to extract from the list

## Value

The extracted element from the list

## Examples

```
## Not run:
response <- "Hi!" |>
  send_prompt(llm_provider_ollama(), return_mode = "full") |>
  extract_from_return_list("response")
response
# [1] "It's nice to meet you. Is there something I can help you with,
# or would you like to chat?"

## End(Not run)
```

---

get_chat_history	<i>Get the chat history of a <a href="#">tidyprompt</a> object</i>
------------------	--

---

### Description

This function gets the chat history of the [tidyprompt](#) object. The chat history is constructed from the base prompt, system prompt, and chat history field. The returned object will be the chat history with the system prompt as the first message with role 'system' and the the base prompt as the last message with role 'user'.

### Usage

```
get_chat_history(x)
```

### Arguments

x                    A [tidyprompt](#) object

### Value

A dataframe containing the chat history

### See Also

[chat\\_history\(\)](#)

Other tidyprompt: [construct\\_prompt\\_text\(\)](#), [get\\_prompt\\_wraps\(\)](#), [is\\_tidyprompt\(\)](#), [set\\_chat\\_history\(\)](#), [tidyprompt\(\)](#), [tidyprompt-class](#)

### Examples

```
prompt <- tidyprompt("Hi!")
print(prompt)

# Add to a tidyprompt using a prompt wrap:
prompt <- tidyprompt("Hi!") |>
  add_text("How are you?")
print(prompt)

# Strings can be input for prompt wraps; therefore,
# a call to tidyprompt() is not necessary:
prompt <- "Hi" |>
  add_text("How are you?")

# Example of adding extraction & validation with a prompt_wrap():
prompt <- "Hi" |>
  add_text("What is 5 + 5?") |>
  answer_as_integer()

## Not run:
```

```

# tidyprompt objects are evaluated by send_prompt(), which will
#   handle construct the prompt text, send it to the LLM provider,
#   and apply the extraction and validation functions from the tidyprompt object
prompt |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
#   Hi
#
#   What is 5 + 5?
#
#   You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
#   10
# [1] 10

# See prompt_wrap() and send_prompt() for more details

## End(Not run)

# `tidyprompt` objects may be validated with these helpers:
is_tidyprompt(prompt) # Returns TRUE if input is a valid tidyprompt object

# Get base prompt text
base_prompt <- prompt$base_prompt

# Get all prompt wraps
prompt_wraps <- prompt$get_prompt_wraps()
# Alternative:
prompt_wraps <- get_prompt_wraps(prompt)

# Construct prompt text
prompt_text <- prompt$construct_prompt_text()
# Alternative:
prompt_text <- construct_prompt_text(prompt)

# Set chat history (affecting also the base prompt)
chat_history <- data.frame(
  role = c("user", "assistant", "user"),
  content = c("What is 5 + 5?", "10", "And what is 5 + 6?")
)
prompt$set_chat_history(chat_history)

# Get chat history
chat_history <- prompt$get_chat_history()

```

---

get\_prompt\_wraps

*Get prompt wraps from a [tidyprompt](#) object*


---

## Description

Get prompt wraps from a [tidyprompt](#) object

**Usage**

```
get_prompt_wraps(x, order = c("default", "modification", "evaluation"))
```

**Arguments**

x	A <a href="#">tidyprompt</a> object
order	The order to return the wraps. Options are: <ul style="list-style-type: none"> <li>• "default": as originally added to the object</li> <li>• "modification": as ordered for modification of the base prompt; ordered by type: check, unspecified, mode, tool, break. This is the order in which prompt wraps are applied during <a href="#">construct_prompt_text()</a></li> <li>• "evaluation": ordered for evaluation of the LLM response; ordered by type: tool, mode, break, unspecified, check. This is the order in which wraps are applied to the LLM output during <a href="#">send_prompt()</a></li> </ul>

**Value**

A list of prompt wrap objects (see [prompt\\_wrap\(\)](#))

**See Also**

Other tidyprompt: [construct\\_prompt\\_text\(\)](#), [get\\_chat\\_history\(\)](#), [is\\_tidyprompt\(\)](#), [set\\_chat\\_history\(\)](#), [tidyprompt\(\)](#), [tidyprompt-class](#)

**Examples**

```
prompt <- tidyprompt("Hi!")
print(prompt)

# Add to a tidyprompt using a prompt wrap:
prompt <- tidyprompt("Hi!") |>
  add_text("How are you?")
print(prompt)

# Strings can be input for prompt wraps; therefore,
# a call to tidyprompt() is not necessary:
prompt <- "Hi" |>
  add_text("How are you?")

# Example of adding extraction & validation with a prompt_wrap():
prompt <- "Hi" |>
  add_text("What is 5 + 5?") |>
  answer_as_integer()

## Not run:
# tidyprompt objects are evaluated by send_prompt(), which will
# handle construct the prompt text, send it to the LLM provider,
# and apply the extraction and validation functions from the tidyprompt object
prompt |>
  send_prompt(llm_provider_ollama())
```

```

# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi
#
# What is 5 + 5?
#
# You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
# 10
# [1] 10

# See prompt_wrap() and send_prompt() for more details

## End(Not run)

# `tidyprompt` objects may be validated with these helpers:
is_tidyprompt(prompt) # Returns TRUE if input is a valid tidyprompt object

# Get base prompt text
base_prompt <- prompt$base_prompt

# Get all prompt wraps
prompt_wraps <- prompt$get_prompt_wraps()
# Alternative:
prompt_wraps <- get_prompt_wraps(prompt)

# Construct prompt text
prompt_text <- prompt$construct_prompt_text()
# Alternative:
prompt_text <- construct_prompt_text(prompt)

# Set chat history (affecting also the base prompt)
chat_history <- data.frame(
  role = c("user", "assistant", "user"),
  content = c("What is 5 + 5?", "10", "And what is 5 + 6?")
)
prompt$set_chat_history(chat_history)

# Get chat history
chat_history <- prompt$get_chat_history()

```

---

is\_tidyprompt

*Check if object is a [tidyprompt](#) object*


---

## Description

Check if object is a [tidyprompt](#) object

## Usage

```
is_tidyprompt(x)
```

**Arguments**

x                    An object to check

**Value**

TRUE if the object is a valid [tidyprompt](#) object, otherwise FALSE

**See Also**

Other tidyprompt: [construct\\_prompt\\_text\(\)](#), [get\\_chat\\_history\(\)](#), [get\\_prompt\\_wraps\(\)](#), [set\\_chat\\_history\(\)](#), [tidyprompt\(\)](#), [tidyprompt-class](#)

**Examples**

```
prompt <- tidyprompt("Hi!")
print(prompt)

# Add to a tidyprompt using a prompt wrap:
prompt <- tidyprompt("Hi!") |>
  add_text("How are you?")
print(prompt)

# Strings can be input for prompt wraps; therefore,
# a call to tidyprompt() is not necessary:
prompt <- "Hi" |>
  add_text("How are you?")

# Example of adding extraction & validation with a prompt_wrap():
prompt <- "Hi" |>
  add_text("What is 5 + 5?") |>
  answer_as_integer()

## Not run:
# tidyprompt objects are evaluated by send_prompt(), which will
# handle construct the prompt text, send it to the LLM provider,
# and apply the extraction and validation functions from the tidyprompt object
prompt |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi
#
# What is 5 + 5?
#
# You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
# 10
# [1] 10

# See prompt_wrap() and send_prompt() for more details

## End(Not run)
```

```

# `tidyprompt` objects may be validated with these helpers:
is_tidyprompt(prompt) # Returns TRUE if input is a valid tidyprompt object

# Get base prompt text
base_prompt <- prompt$base_prompt

# Get all prompt wraps
prompt_wraps <- prompt$get_prompt_wraps()
# Alternative:
prompt_wraps <- get_prompt_wraps(prompt)

# Construct prompt text
prompt_text <- prompt$construct_prompt_text()
# Alternative:
prompt_text <- construct_prompt_text(prompt)

# Set chat history (affecting also the base prompt)
chat_history <- data.frame(
  role = c("user", "assistant", "user"),
  content = c("What is 5 + 5?", "10", "And what is 5 + 6?")
)
prompt$set_chat_history(chat_history)

# Get chat history
chat_history <- prompt$get_chat_history()

```

---

llm\_break

*Create an llm\_break object*


---

## Description

This object is used to break an extraction and validation loop defined in a `prompt_wrap()` as evaluated by `send_prompt()`. When an extraction or validation function returns this object, the loop will be broken and no further extraction or validation functions are applied; instead, `send_prompt()` will be able to return the result at that point. This may be useful in scenarios where it is determined the LLM is unable to provide a response to a prompt.

## Usage

```
llm_break(object_to_return = NULL, success = FALSE)
```

## Arguments

<code>object_to_return</code>	The object to return as the response result from <code>send_prompt()</code> when this object is returned from an extraction or validation function.
<code>success</code>	A logical indicating whether the <code>send_prompt()</code> loop break should nonetheless be considered as a successful completion of the extraction and validation process. If <code>FALSE</code> , the <code>object_to_return</code> must be <code>NULL</code> (as the response result of

`send_prompt()` will always be 'NULL' when the evaluation was unsuccessful); if FALSE, `send_prompt()` will also print a warning about the unsuccessful evaluation. If TRUE, the `object_to_return` will be returned as the response result of `send_prompt()` (and `send_prompt()` will print no warning about unsuccessful evaluation).

### Value

An list of class "llm\_break" containing the object to return and a logical indicating whether the evaluation was successful

### See Also

`llm_feedback()`

Other prompt\_wrap: `llm_feedback()`, `prompt_wrap()`

Other prompt\_evaluation: `llm_feedback()`, `send_prompt()`

### Examples

```
# Example usage within an extraction function similar to the one in 'quit_if()':
extraction_fn <- function(x) {
  quit_detect_regex <- "NO ANSWER"

  if (grepl(quit_detect_regex, x)) {
    return(llm_break(
      object_to_return = NULL,
      success = TRUE
    ))
  }

  return(x)
}

## Not run:
result <- "How many months old is the cat of my uncle?" |>
  answer_as_integer() |>
  prompt_wrap(
    modify_fn = function(prompt) {
      paste0(
        prompt, "\n\n",
        "Type only 'NO ANSWER' if you do not know."
      )
    },
    extraction_fn = extraction_fn,
    type = "break"
  ) |>
  send_prompt()
result
# NULL

## End(Not run)
```



---

llm_feedback	<i>Create an llm_feedback object</i>
--------------	--------------------------------------

---

### Description

This object is used to send feedback to a LLM when a LLM reply does not successfully pass an extraction or validation function (as handled by `send_prompt()` and defined using `prompt_wrap()`). The feedback text is sent back to the LLM. The extraction or validation function should then return this object with the feedback text that should be sent to the LLM.

### Usage

```
llm_feedback(text, tool_result = FALSE)
```

### Arguments

<code>text</code>	A character string containing the feedback text. This will be sent back to the LLM after not passing an extractor or validator function
<code>tool_result</code>	A logical indicating whether the feedback is a tool result. If TRUE, <code>send_prompt()</code> will not remove it from the chat history when cleaning the context window during repeated interactions

### Value

An object of class "llm\_feedback" (or "llm\_feedback\_tool\_result") containing the feedback text to send back to the LLM

### See Also

`llm_break()`

Other prompt\_wrap: `llm_break()`, `prompt_wrap()`

Other prompt\_evaluation: `llm_break()`, `send_prompt()`

### Examples

```
# Example usage within a validation function similar to the one in 'answer_as_integer()':
validation_fn <- function(x, min = 0, max = 100) {
  if (x != floor(x)) { # Not a whole number
    return(llm_feedback(
      "You must answer with only an integer (use no other characters).")
    ))
  }
  if (!is.null(min) && x < min) {
    return(llm_feedback(glue::glue(
      "The number should be greater than or equal to {min}."
    )))
  }
  if (!is.null(max) && x > max) {
```

```

    return(llm_feedback(glue::glue(
      "The number should be less than or equal to {max}."
    )))
  }
  return(TRUE)
}

# This validation_fn would be part of a prompt_wrap();
# see the `answer_as_integer()` function for an example of how to use it

```

---

llm\_provider-class      *LlmProvider R6 Class*


---

## Description

This class provides a structure for creating [llm\\_provider](#) objects with different implementations of `$complete_chat()`. Using this class, you can create an [llm\\_provider](#) object that interacts with different LLM providers, such as Ollama, OpenAI, or other custom providers.

## Public fields

`parameters` A named list of parameters to configure the [llm\\_provider](#). Parameters may be appended to the request body when interacting with the LLM provider API

`verbose` A logical indicating whether interaction with the LLM provider should be printed to the console

`url` The URL to the LLM provider API endpoint for chat completion

`api_key` The API key to use for authentication with the LLM provider API

`api_type` The type of API to use (e.g., "openai", "ollama"). This is used to determine certain specific behaviors for different APIs, for instance, as is done in the [answer\\_as\\_json\(\)](#) function

`handler_fns` A list of functions that will be called after the completion of a chat. See [\\$add\\_handler\\_fn\(\)](#)

## Methods

### Public methods:

- [llm\\_provider-class\\$new\(\)](#)
- [llm\\_provider-class\\$set\\_parameters\(\)](#)
- [llm\\_provider-class\\$complete\\_chat\(\)](#)
- [llm\\_provider-class\\$add\\_handler\\_fn\(\)](#)
- [llm\\_provider-class\\$set\\_handler\\_fns\(\)](#)
- [llm\\_provider-class\\$clone\(\)](#)

**Method** `new()`: Create a new [llm\\_provider](#) object

*Usage:*

```
llm_provider-class$new(
  complete_chat_function,
  parameters = list(),
  verbose = TRUE,
  url = NULL,
  api_key = NULL,
  api_type = "unspecified"
)
```

*Arguments:*

`complete_chat_function` Function that will be called by the `llm_provider` to complete a chat. This function should take a list containing at least `'$chat_history'` (a data frame with `'role'` and `'content'` columns) and return a response object, which contains:

- `'completed'`: A dataframe with `'role'` and `'content'` columns, containing the completed chat history
- `'http'`: A list containing a list `'requests'` and a list `'responses'`, containing the HTTP requests and responses made during the chat completion

`parameters` A named list of parameters to configure the `llm_provider`. These parameters may be appended to the request body when interacting with the LLM provider. For example, the `model` parameter may often be required. The `'stream'` parameter may be used to indicate that the API should stream. Parameters should not include the `chat_history`, or `'api_key'` or `'url'`, which are handled separately by the `llm_provider` and `'$complete_chat()'`. Parameters should also not be set when they are handled by prompt wraps

`verbose` A logical indicating whether interaction with the LLM provider should be printed to the console

`url` The URL to the LLM provider API endpoint for chat completion (typically required, but may be left NULL in some cases, for instance when creating a fake LLM provider)

`api_key` The API key to use for authentication with the LLM provider API (optional, not required for, for instance, Ollama)

`api_type` The type of API to use (e.g., "openai", "ollama"). This is used to determine certain specific behaviors for different APIs (see for example the `answer_as_json()` function)

*Returns:* A new `llm_provider` R6 object

**Method** `set_parameters()`: Helper function to set the parameters of the `llm_provider` object. This function appends new parameters to the existing parameters list.

*Usage:*

```
llm_provider-class$set_parameters(new_parameters)
```

*Arguments:*

`new_parameters` A named list of new parameters to append to the existing parameters list

*Returns:* The modified `llm_provider` object

**Method** `complete_chat()`: Sends a chat history (see `chat_history()` for details) to the LLM provider using the configured `$complete_chat()`. This function is typically called by `send_prompt()` to interact with the LLM provider, but it can also be called directly.

*Usage:*

```
llm_provider-class$complete_chat(input)
```

*Arguments:*

input A string, a data frame which is a valid chat history (see [chat\\_history\(\)](#)), or a list containing a valid chat history under key 'chat\_history'

*Returns:* The response from the LLM provider

**Method** `add_handler_fn()`: Helper function to add a handler function to the `llm_provider` object. Handler functions are called after the completion of a chat and can be used to modify the response before it is returned by the `llm_provider`. Each handler function should take the response object as input (first argument) as well as 'self' (the `llm_provider` object) and return a modified response object. The functions will be called in the order they are added to the list.

*Usage:*

```
llm_provider-class$add_handler_fn(handler_fn)
```

*Arguments:*

handler\_fn A function that takes the response object plus 'self' (the `llm_provider` object) as input and returns a modified response object

*Details:* If a handler function returns a list with a 'break' field set to TRUE, the chat completion will be interrupted and the response will be returned at that point. If a handler function returns a list with a 'done' field set to FALSE, the handler functions will continue to be called in a loop until the 'done' field is not set to FALSE.

**Method** `set_handler_fns()`: Helper function to set the handler functions of the `llm_provider` object. This function replaces the existing handler functions list with a new list of handler functions. See `$add_handler_fn()` for more information

*Usage:*

```
llm_provider-class$set_handler_fns(handler_fns)
```

*Arguments:*

handler\_fns A list of handler functions to set

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
llm_provider-class$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

**See Also**

Other `llm_provider`: [llm\\_provider\\_google\\_gemini\(\)](#), [llm\\_provider\\_groq\(\)](#), [llm\\_provider\\_mistral\(\)](#), [llm\\_provider\\_ollama\(\)](#), [llm\\_provider\\_openai\(\)](#), [llm\\_provider\\_openrouter\(\)](#), [llm\\_provider\\_xai\(\)](#)

**Examples**

```
# Example creation of a llm_provider-class object:
llm_provider_openai <- function(
  parameters = list(
    model = "gpt-4o-mini",
    stream = getOption("tidyprompt.stream", TRUE)
```

```

    ),
    verbose = getOption("tidyprompt.verbose", TRUE),
    url = "https://api.openai.com/v1/chat/completions",
    api_key = Sys.getenv("OPENAI_API_KEY")
  ) {
    complete_chat <- function(chat_history) {
      headers <- c(
        "Content-Type" = "application/json",
        "Authorization" = paste("Bearer", self$api_key)
      )

      body <- list(
        messages = lapply(seq_len(nrow(chat_history)), function(i) {
          list(role = chat_history$role[i], content = chat_history$content[i])
        })
      )

      for (name in names(self$parameters))
        body[[name]] <- self$parameters[[name]]

      request <- httr2::request(self$url) |>
        httr2::req_body_json(body) |>
        httr2::req_headers(!!!headers)

      request_llm_provider(
        chat_history,
        request,
        stream = self$parameters$stream,
        verbose = self$verbose,
        api_type = self$api_type
      )
    }

    return(`llm_provider-class`$new(
      complete_chat_function = complete_chat,
      parameters = parameters,
      verbose = verbose,
      url = url,
      api_key = api_key,
      api_type = "openai"
    ))
  }

llm_provider <- llm_provider_openai()

## Not run:
llm_provider$complete_chat("Hi!")
# --- Sending request to LLM provider (gpt-4o-mini): ---
# Hi!
# --- Receiving response from LLM provider: ---
# Hello! How can I assist you today?

## End(Not run)

```

---

llm\_provider\_google\_gemini

*Create a new Google Gemini LLM provider*


---

### Description

This function creates a new [llm\\_provider](#) object that interacts with the Google Gemini API. Streaming is not yet supported in this implementation.

### Usage

```
llm_provider_google_gemini(
  parameters = list(model = "gemini-1.5-flash"),
  verbose = getOption("tidyprompt.verbose", TRUE),
  url = "https://generativelanguage.googleapis.com/v1beta/models/",
  api_key = Sys.getenv("GOOGLE_AI_STUDIO_API_KEY")
)
```

### Arguments

parameters	<p>A named list of parameters. Currently the following parameters are required:</p> <ul style="list-style-type: none"> <li>model: The name of the model to use (see: <a href="https://ai.google.dev/gemini-api/docs/models/gemini">https://ai.google.dev/gemini-api/docs/models/gemini</a>)</li> </ul> <p>Additional parameters are appended to the request body; see the Google AI Studio API documentation for more information: <a href="https://ai.google.dev/gemini-api/docs/text-generation">https://ai.google.dev/gemini-api/docs/text-generation</a> and <a href="https://github.com/google/generative-ai-docs/blob/main/site/en/gemini-api/docs/get-started/rest.ipynb">https://github.com/google/generative-ai-docs/blob/main/site/en/gemini-api/docs/get-started/rest.ipynb</a></p>
verbose	A logical indicating whether the interaction with the LLM provider should be printed to the console
url	The URL to the Google Gemini API endpoint for chat completion
api_key	The API key to use for authentication with the Google Gemini API (see: <a href="https://aistudio.google.com/app/a">https://aistudio.google.com/app/a</a> )

### Value

A new [llm\\_provider](#) object for use of the Google Gemini API

### See Also

Other llm\_provider: [llm\\_provider-class](#), [llm\\_provider\\_groq\(\)](#), [llm\\_provider\\_mistral\(\)](#), [llm\\_provider\\_ollama\(\)](#), [llm\\_provider\\_openai\(\)](#), [llm\\_provider\\_openrouter\(\)](#), [llm\\_provider\\_xai\(\)](#)

**Examples**

```
# Various providers:
ollama <- llm_provider_ollama()
openai <- llm_provider_openai()
openrouter <- llm_provider_openrouter()
mistral <- llm_provider_mistral()
groq <- llm_provider_groq()
xai <- llm_provider_xai()
gemini <- llm_provider_google_gemini()

# Initialize with settings:
ollama <- llm_provider_ollama(
  parameters = list(
    model = "llama3.2:3b",
    stream = TRUE
  ),
  verbose = TRUE,
  url = "http://localhost:11434/api/chat"
)

# Change settings:
ollama$verbose <- FALSE
ollama$parameters$stream <- FALSE
ollama$parameters$model <- "llama3.1:8b"

## Not run:
# Try a simple chat message with '$complete_chat()':
response <- ollama$complete_chat("Hi!")
response
# $role
# [1] "assistant"
#
# $content
# [1] "How's it going? Is there something I can help you with or would you like
# to chat?"
#
# $http
# Response [http://localhost:11434/api/chat]
# Date: 2024-11-18 14:21
# Status: 200
# Content-Type: application/json; charset=utf-8
# Size: 375 B

# Use with send_prompt():
"Hi" |>
  send_prompt(ollama)
# [1] "How's your day going so far? Is there something I can help you with or
# would you like to chat?"

## End(Not run)
```

---

llm\_provider\_groq      *Create a new Groq LLM provider*

---

## Description

This function creates a new `llm_provider` object that interacts with the Groq API.

## Usage

```
llm_provider_groq(
  parameters = list(model = "llama-3.1-8b-instant", stream = TRUE),
  verbose = getOption("tidyprompt.verbose", TRUE),
  url = "https://api.groq.com/openai/v1/chat/completions",
  api_key = Sys.getenv("GROQ_API_KEY")
)
```

## Arguments

parameters	<p>A named list of parameters. Currently the following parameters are required:</p> <ul style="list-style-type: none"> <li>• model: The name of the model to use</li> <li>• stream: A logical indicating whether the API should stream responses</li> </ul> <p>Additional parameters are appended to the request body; see the Groq API documentation for more information: <a href="https://console.groq.com/docs/api-reference#chat-create">https://console.groq.com/docs/api-reference#chat-create</a></p>
verbose	A logical indicating whether the interaction with the LLM provider should be printed to the console
url	The URL to the Groq API endpoint for chat completion
api_key	The API key to use for authentication with the Groq API

## Value

A new `llm_provider` object for use of the Groq API

## See Also

Other `llm_provider`: `llm_provider-class`, `llm_provider_google_gemini()`, `llm_provider_mistral()`, `llm_provider_ollama()`, `llm_provider_openai()`, `llm_provider_openrouter()`, `llm_provider_xai()`

## Examples

```
# Various providers:
ollama <- llm_provider_ollama()
openai <- llm_provider_openai()
openrouter <- llm_provider_openrouter()
mistral <- llm_provider_mistral()
groq <- llm_provider_groq()
```



```

xai <- llm_provider_xai()
gemini <- llm_provider_google_gemini()

# Initialize with settings:
ollama <- llm_provider_ollama(
  parameters = list(
    model = "llama3.2:3b",
    stream = TRUE
  ),
  verbose = TRUE,
  url = "http://localhost:11434/api/chat"
)

# Change settings:
ollama$verbose <- FALSE
ollama$parameters$stream <- FALSE
ollama$parameters$model <- "llama3.1:8b"

## Not run:
# Try a simple chat message with '$complete_chat()':
response <- ollama$complete_chat("Hi!")
response
# $role
# [1] "assistant"
#
# $content
# [1] "How's it going? Is there something I can help you with or would you like
# to chat?"
#
# $http
# Response [http://localhost:11434/api/chat]
# Date: 2024-11-18 14:21
# Status: 200
# Content-Type: application/json; charset=utf-8
# Size: 375 B

# Use with send_prompt():
"Hi" |>
  send_prompt(ollama)
# [1] "How's your day going so far? Is there something I can help you with or
# would you like to chat?"

## End(Not run)

```

---

llm\_provider\_mistral *Create a new Mistral LLM provider*

---

## Description

This function creates a new [llm\\_provider](#) object that interacts with the Mistral API.

**Usage**

```
llm_provider_mistral(
  parameters = list(model = "ministral-3b-latest", stream =
    getOption("tidyprompt.stream", TRUE)),
  verbose = getOption("tidyprompt.verbose", TRUE),
  url = "https://api.mistral.ai/v1/chat/completions",
  api_key = Sys.getenv("MISTRAL_API_KEY")
)
```

**Arguments**

parameters	A named list of parameters. Currently the following parameters are required: <ul style="list-style-type: none"> <li>• model: The name of the model to use</li> <li>• stream: A logical indicating whether the API should stream responses</li> </ul> Additional parameters are appended to the request body; see the Mistral API documentation for more information: <a href="https://docs.mistral.ai/api/#tag/chat">https://docs.mistral.ai/api/#tag/chat</a>
verbose	A logical indicating whether the interaction with the LLM provider should be printed to the consol
url	The URL to the Mistral API endpoint for chat completion
api_key	The API key to use for authentication with the Mistral API

**Value**

A new [llm\\_provider](#) object for use of the Mistral API

**See Also**

Other llm\_provider: [llm\\_provider-class](#), [llm\\_provider\\_google\\_gemini\(\)](#), [llm\\_provider\\_groq\(\)](#), [llm\\_provider\\_ollama\(\)](#), [llm\\_provider\\_openai\(\)](#), [llm\\_provider\\_openrouter\(\)](#), [llm\\_provider\\_xai\(\)](#)

**Examples**

```
# Various providers:
ollama <- llm_provider_ollama()
openai <- llm_provider_openai()
openrouter <- llm_provider_openrouter()
mistral <- llm_provider_mistral()
groq <- llm_provider_groq()
xai <- llm_provider_xai()
gemini <- llm_provider_google_gemini()

# Initialize with settings:
ollama <- llm_provider_ollama(
  parameters = list(
    model = "llama3.2:3b",
    stream = TRUE
  ),
  verbose = TRUE,
  url = "http://localhost:11434/api/chat"
```

```

)

# Change settings:
ollama$verbose <- FALSE
ollama$parameters$stream <- FALSE
ollama$parameters$model <- "llama3.1:8b"

## Not run:
# Try a simple chat message with '$complete_chat()':
response <- ollama$complete_chat("Hi!")
response
# $role
# [1] "assistant"
#
# $content
# [1] "How's it going? Is there something I can help you with or would you like
# to chat?"
#
# $http
# Response [http://localhost:11434/api/chat]
# Date: 2024-11-18 14:21
# Status: 200
# Content-Type: application/json; charset=utf-8
# Size: 375 B

# Use with send_prompt():
"Hi" |>
  send_prompt(ollama)
# [1] "How's your day going so far? Is there something I can help you with or
# would you like to chat?"

## End(Not run)

```

---

llm\_provider\_ollama    *Create a new Ollama LLM provider*

---

## Description

This function creates a new [llm\\_provider](#) object that interacts with the Ollama API.

## Usage

```

llm_provider_ollama(
  parameters = list(model = "llama3.1:8b", stream = getOption("tidyprompt.stream", TRUE)),
  verbose = getOption("tidyprompt.verbose", TRUE),
  url = "http://localhost:11434/api/chat"
)

```

**Arguments**

parameters	<p>A named list of parameters. Currently the following parameters are required:</p> <ul style="list-style-type: none"> <li>• model: The name of the model to use</li> <li>• stream: A logical indicating whether the API should stream responses</li> </ul> <p>Additional parameters may be passed by adding them to the parameters list; these parameters will be passed to the Ollama API via the body of the POST request. Note that various Ollama options need to be set in a list named 'options' within the parameters list (e.g., context window size is represented in <code>\$parameters\$options\$num_ctx</code>)</p>
verbose	A logical indicating whether the interaction with the LLM provider should be printed to the console
url	The URL to the Ollama API endpoint for chat completion (typically: "http://localhost:11434/api/chat")

**Value**

A new `llm_provider` object for use of the Ollama API

**See Also**

Other `llm_provider`: `llm_provider_class`, `llm_provider_google_gemini()`, `llm_provider_groq()`, `llm_provider_mistral()`, `llm_provider_openai()`, `llm_provider_openrouter()`, `llm_provider_xai()`

**Examples**

```
# Various providers:
ollama <- llm_provider_ollama()
openai <- llm_provider_openai()
openrouter <- llm_provider_openrouter()
mistral <- llm_provider_mistral()
groq <- llm_provider_groq()
xai <- llm_provider_xai()
gemini <- llm_provider_google_gemini()

# Initialize with settings:
ollama <- llm_provider_ollama(
  parameters = list(
    model = "llama3.2:3b",
    stream = TRUE
  ),
  verbose = TRUE,
  url = "http://localhost:11434/api/chat"
)

# Change settings:
ollama$verbose <- FALSE
ollama$parameters$stream <- FALSE
ollama$parameters$model <- "llama3.1:8b"

## Not run:
```

```

# Try a simple chat message with '$complete_chat()':
response <- ollama$complete_chat("Hi!")
response
# $role
# [1] "assistant"
#
# $content
# [1] "How's it going? Is there something I can help you with or would you like
# to chat?"
#
# $http
# Response [http://localhost:11434/api/chat]
# Date: 2024-11-18 14:21
# Status: 200
# Content-Type: application/json; charset=utf-8
# Size: 375 B

# Use with send_prompt():
"Hi" |>
  send_prompt(ollama)
# [1] "How's your day going so far? Is there something I can help you with or
# would you like to chat?"

## End(Not run)

```

---

```
llm_provider_openai Create a new OpenAI LLM provider
```

---

## Description

This function creates a new [llm\\_provider](#) object that interacts with the Open AI API.

## Usage

```

llm_provider_openai(
  parameters = list(model = "gpt-4o-mini", stream = getOption("tidyprompt.stream", TRUE)),
  verbose = getOption("tidyprompt.verbose", TRUE),
  url = "https://api.openai.com/v1/chat/completions",
  api_key = Sys.getenv("OPENAI_API_KEY")
)

```

## Arguments

parameters	<p>A named list of parameters. Currently the following parameters are required:</p> <ul style="list-style-type: none"> <li>• model: The name of the model to use</li> <li>• api_key: The API key to use for authentication with the OpenAI API. This should be a project API key (not a user API key)</li> <li>• url: The URL to the OpenAI API (may also be an alternative endpoint that provides a similar API.)</li> </ul>
------------	---

- stream: A logical indicating whether the API should stream responses

Additional parameters are appended to the request body; see the OpenAI API documentation for more information: <https://platform.openai.com/docs/api-reference/chat>

verbose	A logical indicating whether the interaction with the LLM provider should be printed to the console. Default is TRUE.
url	The URL to the OpenAI API endpoint for chat completion (typically: "https://api.openai.com/v1/chat/completions")
api_key	The API key to use for authentication with the OpenAI API

### Value

A new `llm_provider` object for use of the OpenAI API

### See Also

Other `llm_provider`: `llm_provider_class`, `llm_provider_google_gemini()`, `llm_provider_groq()`, `llm_provider_mistral()`, `llm_provider_ollama()`, `llm_provider_openrouter()`, `llm_provider_xai()`

### Examples

```
# Various providers:
ollama <- llm_provider_ollama()
openai <- llm_provider_openai()
openrouter <- llm_provider_openrouter()
mistral <- llm_provider_mistral()
groq <- llm_provider_groq()
xai <- llm_provider_xai()
gemini <- llm_provider_google_gemini()

# Initialize with settings:
ollama <- llm_provider_ollama(
  parameters = list(
    model = "llama3.2:3b",
    stream = TRUE
  ),
  verbose = TRUE,
  url = "http://localhost:11434/api/chat"
)

# Change settings:
ollama$verbose <- FALSE
ollama$parameters$stream <- FALSE
ollama$parameters$model <- "llama3.1:8b"

## Not run:
# Try a simple chat message with '$complete_chat()':
response <- ollama$complete_chat("Hi!")
response
# $role
# [1] "assistant"
#
# $content
```

```

# [1] "How's it going? Is there something I can help you with or would you like
# to chat?"
#
# $http
# Response [http://localhost:11434/api/chat]
# Date: 2024-11-18 14:21
# Status: 200
# Content-Type: application/json; charset=utf-8
# Size: 375 B

# Use with send_prompt():
"Hi" |>
  send_prompt(ollama)
# [1] "How's your day going so far? Is there something I can help you with or
# would you like to chat?"

## End(Not run)

```

---

llm\_provider\_openrouter

*Create a new OpenRouter LLM provider*

---

## Description

This function creates a new `llm_provider` object that interacts with the OpenRouter API.

## Usage

```

llm_provider_openrouter(
  parameters = list(model = "qwen/qwen-2.5-7b-instruct", stream =
    getOption("tidyprompt.stream", TRUE)),
  verbose = getOption("tidyprompt.verbose", TRUE),
  url = "https://openrouter.ai/api/v1/chat/completions",
  api_key = Sys.getenv("OPENROUTER_API_KEY")
)

```

## Arguments

<code>parameters</code>	<p>A named list of parameters. Currently the following parameters are required:</p> <ul style="list-style-type: none"> <li><code>model</code>: The name of the model to use</li> <li><code>stream</code>: A logical indicating whether the API should stream responses</li> </ul> <p>Additional parameters are appended to the request body; see the OpenRouter API documentation for more information: <a href="https://openrouter.ai/docs/parameters">https://openrouter.ai/docs/parameters</a></p>
<code>verbose</code>	A logical indicating whether the interaction with the LLM provider should be printed to the console.
<code>url</code>	The URL to the OpenRouter API endpoint for chat completion
<code>api_key</code>	The API key to use for authentication with the OpenRouter API

**Value**

A new `llm_provider` object for use of the OpenRouter API

**See Also**

Other `llm_provider`: `llm_provider-class`, `llm_provider_google_gemini()`, `llm_provider_groq()`, `llm_provider_mistral()`, `llm_provider_ollama()`, `llm_provider_openai()`, `llm_provider_xai()`

**Examples**

```
# Various providers:
ollama <- llm_provider_ollama()
openai <- llm_provider_openai()
openrouter <- llm_provider_openrouter()
mistral <- llm_provider_mistral()
groq <- llm_provider_groq()
xai <- llm_provider_xai()
gemini <- llm_provider_google_gemini()

# Initialize with settings:
ollama <- llm_provider_ollama(
  parameters = list(
    model = "llama3.2:3b",
    stream = TRUE
  ),
  verbose = TRUE,
  url = "http://localhost:11434/api/chat"
)

# Change settings:
ollama$verbose <- FALSE
ollama$parameters$stream <- FALSE
ollama$parameters$model <- "llama3.1:8b"

## Not run:
# Try a simple chat message with '$complete_chat()':
response <- ollama$complete_chat("Hi!")
response
# $role
# [1] "assistant"
#
# $content
# [1] "How's it going? Is there something I can help you with or would you like
# to chat?"
#
# $http
# Response [http://localhost:11434/api/chat]
# Date: 2024-11-18 14:21
# Status: 200
# Content-Type: application/json; charset=utf-8
# Size: 375 B
```



```

# Use with send_prompt():
"Hi" |>
  send_prompt(ollama)
# [1] "How's your day going so far? Is there something I can help you with or
# would you like to chat?"

## End(Not run)

```

---

llm\_provider\_xai      *Create a new XAI (Grok) LLM provider*

---

## Description

This function creates a new `llm_provider` object that interacts with the XAI API.

## Usage

```

llm_provider_xai(
  parameters = list(model = "grok-beta", stream = getOption("tidyprompt.stream", TRUE)),
  verbose = getOption("tidyprompt.verbose", TRUE),
  url = "https://api.x.ai/v1/chat/completions",
  api_key = Sys.getenv("XAI_API_KEY")
)

```

## Arguments

<code>parameters</code>	<p>A named list of parameters. Currently the following parameters are required:</p> <ul style="list-style-type: none"> <li>• <code>model</code>: The name of the model to use</li> <li>• <code>stream</code>: A logical indicating whether the API should stream responses</li> </ul> <p>Additional parameters are appended to the request body; see the XAI API documentation for more information: <a href="https://docs.x.ai/api/endpoints#chat-completions">https://docs.x.ai/api/endpoints#chat-completions</a></p>
<code>verbose</code>	A logical indicating whether the interaction with the LLM provider should be printed to the console. Default is TRUE.
<code>url</code>	The URL to the XAI API endpoint for chat completion
<code>api_key</code>	The API key to use for authentication with the XAI API

## Value

A new `llm_provider` object for use of the XAI API

## See Also

Other `llm_provider`: [llm\\_provider-class](#), [llm\\_provider\\_google\\_gemini\(\)](#), [llm\\_provider\\_groq\(\)](#), [llm\\_provider\\_mistral\(\)](#), [llm\\_provider\\_ollama\(\)](#), [llm\\_provider\\_openai\(\)](#), [llm\\_provider\\_openrouter\(\)](#)

**Examples**

```

# Various providers:
ollama <- llm_provider_ollama()
openai <- llm_provider_openai()
openrouter <- llm_provider_openrouter()
mistral <- llm_provider_mistral()
groq <- llm_provider_groq()
xai <- llm_provider_xai()
gemini <- llm_provider_google_gemini()

# Initialize with settings:
ollama <- llm_provider_ollama(
  parameters = list(
    model = "llama3.2:3b",
    stream = TRUE
  ),
  verbose = TRUE,
  url = "http://localhost:11434/api/chat"
)

# Change settings:
ollama$verbose <- FALSE
ollama$parameters$stream <- FALSE
ollama$parameters$model <- "llama3.1:8b"

## Not run:
# Try a simple chat message with '$complete_chat()':
response <- ollama$complete_chat("Hi!")
response
# $role
# [1] "assistant"
#
# $content
# [1] "How's it going? Is there something I can help you with or would you like
# to chat?"
#
# $http
# Response [http://localhost:11434/api/chat]
# Date: 2024-11-18 14:21
# Status: 200
# Content-Type: application/json; charset=utf-8
# Size: 375 B

# Use with send_prompt():
"Hi" |>
  send_prompt(ollama)
# [1] "How's your day going so far? Is there something I can help you with or
# would you like to chat?"

## End(Not run)

```

---

`llm_verify`*Have LLM check the result of a prompt (LLM-in-the-loop)*

---

### Description

This function will wrap a prompt with a check for a LLM to accept or decline the result of the prompt, providing feedback if the result is declined. The evaluating LLM will be presented with the original prompt and the result of the prompt, and will be asked to verify if the answer is satisfactory (using chain of thought reasoning to arrive at a boolean decision). If the result is declined, the chain of thought responsible for the decision will be summarized and sent back to the original LLM that was asked to evaluate the prompt, so that it may retry the prompt.

Note that this function is experimental and, because it relies on chain of thought reasoning by a LLM about the answer of another LLM, it may not always provide accurate results and can increase the token cost of evaluating a prompt.

### Usage

```
llm_verify(  
    prompt,  
    question = "Is the answer satisfactory?",  
    llm_provider = NULL,  
    max_words_feedback = 50  
)
```

### Arguments

<code>prompt</code>	A single string or a <a href="#">tidyprompt</a> object
<code>question</code>	The question to ask the LLM to verify the result of the prompt. The LLM will be presented the original prompt, its result, and this question. The LLM will be asked to provide a boolean answer to this question. If TRUE, the result of the prompt will be accepted; if FALSE, the result will be declined
<code>llm_provider</code>	A <a href="#">llm_provider</a> object which will be used to verify the evaluation led to a satisfactory result. If not provided, the same LLM provider as the prompt was originally evaluated with will be used
<code>max_words_feedback</code>	The maximum number of words allowed in the summary of why the result was declined. This summary is sent back to the LLM originally asked to evaluate the prompt

### Details

The original prompt text shown to the LLM is built from the base prompt as well as all prompt wraps that have a modify function but do not have an extraction or validation function. This is to ensure that no redundant validation is performed by the evaluating LLM on instructions which have already been validated by functions in those prompt wraps.

## Value

A `tidyprompt` with an added `prompt_wrap()` which will add a check for a LLM to accept or decline the result of the prompt, providing feedback if the result is declined

## Examples

```
## Not run:
"What is 'Enschede'?" |>
  answer_as_text(max_words = 50) |>
  llm_verify() |>
  send_prompt()
# --- Sending request to LLM provider (gpt-4o-mini): ---
# What is 'Enschede'?!
#
# You must provide a text response. The response must be at most 50 words.
# --- Receiving response from LLM provider: ---
# Enschede is a city in the Netherlands, located in the eastern part near the German border.
# It is known for its vibrant culture, history, and universities, particularly the
# University of Twente, as well as its textiles industry and beautiful parks.
# --- Sending request to LLM provider (gpt-4o-mini): ---
# You are given a user's prompt.
# To answer the user's prompt, you need to think step by step to arrive at a final answer.
#
# ----- START OF USER'S PROMPT -----
# >>> An assistant was asked:
#
# What is 'Enschede'?!
#
# >>> The assistant answered:
#
# [1] "Enschede is a city in the Netherlands, located in the eastern part near
# the German border. It is known for its vibrant culture, history, and universities,
# particularly the University of Twente, as well as its textiles industry and
# beautiful parks."
#
# >>> Is the answer satisfactory?
#
# You must answer with only TRUE or FALSE (use no other characters).
# ----- END OF USER'S PROMPT -----
#
# What are the steps you would take to answer the user's prompt?
# Describe your thought process in the following format:
#   >> step 1: <step 1 description>
#   >> step 2: <step 2 description>
#   (etc.)
#
# When you are done, you must type:
#   FINISH[<put here your final answer to the user's prompt>]
#
# Make sure your final answer follows the logical conclusion of your thought process.
# --- Receiving response from LLM provider: ---
# >> step 1: Read the user's prompt carefully to understand what is being asked
```

```

# regarding the assistant's answer about Enschede.
# >> step 2: Analyze the assistant's provided answer for accuracy and completeness,
# ensuring it conveys relevant information about Enschede.
# >> step 3: Assess whether the answer includes key aspects, such as the city's
# location, cultural significance, educational institutions, and any notable
# historical references or industries tied to the city.
# >> step 4: Determine if the assistant's answer sufficiently addresses the
# question "What is Enschede?" and provides a clear and informative overview of the city.
# >> step 5: Conclude whether the assistant's answer meets the standards for being
# satisfactory in terms of detail and correctness.
#
# FINISH[TRUE]
# [1] "Enschede is a city in the Netherlands, located in the eastern part near the
# German border. It is known for its vibrant culture, history, and universities,
# particularly the University of Twente, as well as its textiles industry and beautiful parks."

## End(Not run)

```

---

persistent\_chat-class *PersistentChat R6 class*

---

## Description

A class for managing a persistent chat with a large language model (LLM).

While 'tidyprompt' is primarily focused on automatic interactions with LLMs through `send_prompt()` using a `tidyprompt` object with `prompt_wrap()`, this class may be useful for having a manual conversation with an LLM. (It may specifically be used to continue a chat history which was returned by `send_prompt()` with `return_mode = "full"`.)

## Public fields

`chat_history` A `chat_history()` object  
`llm_provider` A `llm_provider` object

## Methods

### Public methods:

- `persistent_chat-class$new()`
- `persistent_chat-class$chat()`
- `persistent_chat-class$reset_chat_history()`
- `persistent_chat-class$clone()`

**Method** `new()`: Initialize the PersistentChat object

*Usage:*

```
persistent_chat-class$new(llm_provider, chat_history = NULL)
```

*Arguments:*

llm\_provider A [llm\\_provider](#) object  
 chat\_history (optional) A [chat\\_history\(\)](#) object  
*Returns:* The initialized PersistentChat object

**Method** chat(): Add a message to the chat history and get a response from the LLM

*Usage:*

```
persistent_chat-class$chat(msg, role = "user", verbose = TRUE)
```

*Arguments:*

msg Message to add to the chat history  
 role Role of the message  
 verbose Whether to print the interaction to the console

*Returns:* The response from the LLM

**Method** reset\_chat\_history(): Reset the chat history

*Usage:*

```
persistent_chat-class$reset_chat_history()
```

*Returns:* NULL

**Method** clone(): The objects of this class are cloneable with this method.

*Usage:*

```
persistent_chat-class$clone(deep = FALSE)
```

*Arguments:*

deep Whether to make a deep clone.

## See Also

[llm\\_provider](#) [chat\\_history\(\)](#)

## Examples

```
# Create a persistent chat with any LLM provider
chat <- `persistent_chat-class`$new(llm_provider_ollama())

## Not run:
chat$chat("Hi! Tell me about Twente, in a short sentence?")
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi! Tell me about Twente, in a short sentence?
# --- Receiving response from LLM provider: ---
# Twente is a charming region in the Netherlands known for its picturesque
# countryside and vibrant culture!

chat$chat("How many people live there?")
# --- Sending request to LLM provider (llama3.1:8b): ---
# How many people live there?
# --- Receiving response from LLM provider: ---
# The population of Twente is approximately 650,000 inhabitants, making it one of
# the largest regions in the Netherlands.
```

```

# Access the chat history:
chat$chat_history

# Reset the chat history:
chat$reset_chat_history()

# Continue a chat from the result of `send_prompt()`:
result <- "Hi there!" |>
  answer_as_integer() |>
  send_prompt(return_mode = "full")
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi there!
#
# You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
# 42
chat <- `persistent_chat-class`$new(llm_provider_ollama(), result$chat_history)
chat$chat("Why did you choose that number?")
# --- Sending request to LLM provider (llama3.1:8b): ---
# Why did you choose that number?
# --- Receiving response from LLM provider: ---
# I chose the number 42 because it's a reference to Douglas Adams' science fiction
# series "The Hitchhiker's Guide to the Galaxy," in which a supercomputer named
# Deep Thought is said to have calculated the "Answer to the Ultimate Question of
# Life, the Universe, and Everything" as 42.

## End(Not run)

```

---

prompt\_wrap

*Wrap a prompt with functions for modification and handling the LLM response*

---

## Description

This function takes a single string or a [tidyprompt](#) object and adds a new prompt wrap to it.

A prompt wrap is a set of functions that modify the prompt text, extract a value from the LLM response, and validate the extracted value.

The functions are used to ensure that the prompt and LLM response are in the correct format and meet the specified criteria; they may also be used to provide the LLM with feedback or additional information, like the result of a tool call or some evaluated code.

Advanced prompt wraps may also include functions that directly handle the response from a LLM API or configure API parameters.

## Usage

```
prompt_wrap(
```

```

prompt,
modify_fn = NULL,
extraction_fn = NULL,
validation_fn = NULL,
handler_fn = NULL,
parameter_fn = NULL,
type = c("unspecified", "mode", "tool", "break", "check"),
name = NULL
)

```

## Arguments

prompt	A string or a <a href="#">tidyprompt</a> object
modify_fn	A function that takes the previous prompt text (as first argument) and returns the new prompt text
extraction_fn	A function that takes the LLM response (as first argument) and attempts to extract a value from it. Upon successful extraction, the function should return the extracted value. If the extraction fails, the function should return a <a href="#">llm_feedback()</a> message to initiate a retry. A <a href="#">llm_break()</a> can be returned to break the extraction and validation loop, ending <a href="#">send_prompt()</a>
validation_fn	A function that takes the (extracted) LLM response (as first argument) and attempts to validate it. Upon successful validation, the function should return TRUE. If the validation fails, the function should return a <a href="#">llm_feedback()</a> message to initiate a retry. A <a href="#">llm_break()</a> can be returned to break the extraction and validation loop, ending <a href="#">send_prompt()</a>
handler_fn	A function that takes a 'completion' object (a result of a request to a LLM, as returned by <a href="#">\$complete_chat()</a> of a <a href="#">llm_provider</a> object) as first argument and the <a href="#">llm_provider</a> object as second argument. The function should return a (modified or identical) completion object. This can be used for advanced side effects, like logging, or native tool calling, or keeping track of token usage. See <a href="#">llm_provider</a> for more information; handler_fn is attached to the <a href="#">llm_provider</a> object that is being used. For example usage, see source code of <a href="#">answer_using_tools()</a>
parameter_fn	A function that takes the <a href="#">llm_provider</a> object which is being used with <a href="#">send_prompt()</a> and returns a named list of parameters to be set in the <a href="#">llm_provider</a> object via its <a href="#">\$set_parameters()</a> method. This can be used to configure specific parameters of the <a href="#">llm_provider</a> object when evaluating the prompt. For example, <a href="#">answer_as_json()</a> may set different parameters for different APIs related to JSON output. This function is typically only used with advanced prompt wraps that require specific settings in the <a href="#">llm_provider</a> object
type	The type of prompt wrap. Must be one of: <ul style="list-style-type: none"> <li>"unspecified": The default type, typically used for prompt wraps which request a specific format of the LLM response, like <a href="#">answer_as_integer()</a></li> <li>"mode": For prompt wraps that change how the LLM should answer the prompt, like <a href="#">answer_by_chain_of_thought()</a> or <a href="#">answer_by_react()</a></li> <li>"tool": For prompt wraps that enable the LLM to use tools, like <a href="#">answer_using_tools()</a> or <a href="#">answer_using_r()</a> when 'output_as_tool' = TRUE</li> </ul>



- "break": For prompt wraps that may break the extraction and validation loop, like `quit_if()`. These are applied before type "unspecified" as they may instruct the LLM to not answer the prompt in the manner specified by those prompt wraps
- "check": For prompt wraps that apply a last check to the final answer, after all other prompt wraps have been evaluated. These prompt wraps may only contain a validation function, and are applied after all other prompt wraps have been evaluated. These prompt wraps are even applied after an earlier prompt wrap has broken the extraction and validation loop with `llm_break()`

Types are used to determine the order in which prompt wraps are applied. When constructing the prompt text, prompt wraps are applied to the base prompt in the following order: 'check', 'unspecified', 'break', 'mode', 'tool'. When evaluating the LLM response and applying extraction and validation functions, prompt wraps are applied in the reverse order: 'tool', 'mode', 'break', 'unspecified', 'check'. Order among the same type is preserved in the order they were added to the prompt.

name                    An optional name for the prompt wrap. This can be used to identify the prompt wrap in the `tidyprompt` object

## Details

For advanced use, `modify_fn`, `extraction_fn`, and `validation_fn` may take the `llm_provider` object (as used with `send_prompt()`) as second argument, and the 'http\_list' (a list of all HTTP requests and responses made during `send_prompt()`) as third argument. Use of these arguments is not required, but can be useful for more complex prompt wraps which require additional information about the LLM provider or requests made so far. The functions (including `parameter_fn`) also have access to the object `self` (not a function argument; it is attached to the environment of the function) which contains the `tidyprompt` object that the prompt wrap is a part of. This can be used to access other prompt wraps, or to access the prompt text or other information about the prompt. For instance, other prompt wraps can be accessed through `self$get_prompt_wraps()`.

## Value

A `tidyprompt` object with the `prompt_wrap()` appended to it

## See Also

`tidyprompt send_prompt()`

Other `prompt_wrap`: `llm_break()`, `llm_feedback()`

Other `pre_built_prompt_wraps`: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `quit_if()`, `set_system_prompt()`

## Examples

```
# A custom prompt_wrap may be created during piping
```

```

prompt <- "Hi there!" |>
  prompt_wrap(
    modify_fn = function(base_prompt) {
      paste(base_prompt, "How are you?", sep = "\n\n")
    }
  )
prompt

# (Shorter notation of the above:)
prompt <- "Hi there!" |>
  prompt_wrap(\(x) paste(x, "How are you?", sep = "\n\n"))

# It may often be preferred to make a function which takes a prompt and
# returns a wrapped prompt:
my_prompt_wrap <- function(prompt) {
  modify_fn <- function(base_prompt) {
    paste(base_prompt, "How are you?", sep = "\n\n")
  }

  prompt_wrap(prompt, modify_fn)
}
prompt <- "Hi there!" |>
  my_prompt_wrap()

# For more advanced examples, take a look at the source code of the
# pre-built prompt wraps in the tidyprompt package, like
# answer_as_boolean, answer_as_integer, add_tools, answer_as_code, etc.
# Below is the source code for the 'answer_as_integer' prompt wrap function:

#' Make LLM answer as an integer (between min and max)
#'
#' @param prompt A single string or a [tidyprompt()] object
#' @param min (optional) Minimum value for the integer
#' @param max (optional) Maximum value for the integer
#' @param add_instruction_to_prompt (optional) Add instruction for replying
#' as an integer to the prompt text. Set to FALSE for debugging if extractions/validations
#' are working as expected (without instruction the answer should fail the
#' validation function, initiating a retry)
#'
#' @return A [tidyprompt()] with an added [prompt_wrap()] which
#' will ensure that the LLM response is an integer.
#'
#' @export
#'
#' @example inst/examples/answer_as_integer.R
#'
#' @family pre_built_prompt_wraps
#' @family answer_as_prompt_wraps
answer_as_integer <- function(
  prompt,
  min = NULL,
  max = NULL,
  add_instruction_to_prompt = TRUE

```

```

) {
  instruction <- "You must answer with only an integer (use no other characters)."
```

```

  if (!is.null(min) && !is.null(max)) {
    instruction <- paste(instruction, glue::glue(
      "Enter an integer between {min} and {max}."
    ))
  } else if (!is.null(min)) {
    instruction <- paste(instruction, glue::glue(
      "Enter an integer greater than or equal to {min}."
    ))
  } else if (!is.null(max)) {
    instruction <- paste(instruction, glue::glue(
      "Enter an integer less than or equal to {max}."
    ))
  }

  modify_fn <- function(original_prompt_text) {
    if (!add_instruction_to_prompt) {
      return(original_prompt_text)
    }

    glue::glue("{original_prompt_text}\n\n{instruction}")
  }

  extraction_fn <- function(x) {
    extracted <- suppressWarnings(as.numeric(x))
    if (is.na(extracted)) {
      return(llm_feedback(instruction))
    }
    return(extracted)
  }

  validation_fn <- function(x) {
    if (x != floor(x)) { # Not a whole number
      return(llm_feedback(instruction))
    }

    if (!is.null(min) && x < min) {
      return(llm_feedback(glue::glue(
        "The number should be greater than or equal to {min}."
      )))
    }
    if (!is.null(max) && x > max) {
      return(llm_feedback(glue::glue(
        "The number should be less than or equal to {max}."
      )))
    }
    return(TRUE)
  }

  prompt_wrap(
    prompt,

```

```

    modify_fn, extraction_fn, validation_fn,
    name = "answer_as_integer"
  )
}

```

---

quit\_if

*Make evaluation of a prompt stop if LLM gives a specific response*


---

### Description

This function is used to wrap a [tidyprompt\(\)](#) object and ensure that the evaluation will stop if the LLM says it cannot answer the prompt. This is useful in scenarios where it is determined the LLM is unable to provide a response to a prompt.

### Usage

```

quit_if(
  prompt,
  quit_detect_regex = "NO ANSWER",
  instruction =
    paste0("If you think that you cannot provide a valid answer, you must type:\n",
      "'NO ANSWER' (use no other characters)"),
  success = TRUE,
  response_result = c("null", "llm_response", "regex_match")
)

```

### Arguments

prompt	A single string or a <a href="#">tidyprompt()</a> object
quit_detect_regex	A regular expression to detect in the LLM's response which will cause the evaluation to stop. The default will detect the string "NO ANSWER" in the response
instruction	A string to be added to the prompt to instruct the LLM how to respond if it cannot answer the prompt. The default is "If you think that you cannot provide a valid answer, you must type: 'NO ANSWER' (use no other characters)". This parameter can be set to NULL if no instruction is needed in the prompt
success	A logical indicating whether the <a href="#">send_prompt()</a> loop break should nonetheless be considered as a successful completion of the extraction and validation process. If FALSE, the <code>object_to_return</code> must will always be set to NULL and thus parameter 'response_result' must also be set to 'null'; if FALSE, <a href="#">send_prompt()</a> will also print a warning about the unsuccessful evaluation. If TRUE, the <code>object_to_return</code> will be returned as the response result of <a href="#">send_prompt()</a> (and <a href="#">send_prompt()</a> will print no warning about unsuccessful evaluation); parameter 'response_result' will then determine what is returned as the response result of <a href="#">send_prompt()</a> .

response\_result

A character string indicating what should be returned when the `quit_detect_regex` is detected in the LLM's response. The default is 'null', which will return NULL as the response result of `send_prompt()`. Under 'llm\_response', the full LLM response will be returned as the response result of `send_prompt()`. Under 'regex\_match', the part of the LLM response that matches the `quit_detect_regex` will be returned as the response result of `send_prompt()`

### Value

A `tidyprompt()` with an added `prompt_wrap()` which will ensure that the evaluation will stop upon detection of the `quit_detect_regex` in the LLM's response

### See Also

Other pre\_built\_prompt\_wraps: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `prompt_wrap()`, `set_system_prompt()`

Other miscellaneous\_prompt\_wraps: `add_text()`, `set_system_prompt()`

### Examples

```
## Not run:
  "What the favourite food of my cat on Thursday mornings?" |>
  quit_if() |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
#   What the favourite food of my cat on Thursday mornings?
#
#   If you think that you cannot provide a valid answer, you must type:
#   'NO ANSWER' (use no other characters)
# --- Receiving response from LLM provider: ---
#   NO ANSWER
#   NULL

## End(Not run)
```

---

r\_json\_schema\_to\_example

*Generate an example object from a JSON schema*

---

### Description

This function generates an example JSON object from a JSON schema. This is used when enforcing a JSON schema through text-based handling (requiring an example to be added to the prompt text).

**Usage**

```
r_json_schema_to_example(schema)
```

**Arguments**

schema            A list (R object) representing a JSON schema

**Value**

A list (R object) which matches the JSON schema definition

**See Also**

Other json: [answer\\_as\\_json\(\)](#)

**Examples**

```
base_prompt <- "How can I solve 8x + 7 = -23?"

# This example will show how to enforce JSON format in the response,
# with and without a schema, using the 'answer_as_json()' prompt wrap.
# If you use type = 'auto', the function will automatically detect the
# best way to enforce JSON based on the LLM provider you are using.
# Note that the default type is 'text-based', which will work for any provider/model

#### Enforcing JSON without a schema: ####

## Not run:
## Text-based (works for any provider/model):
# Adds request to prompt for a JSON object
# Extracts JSON from textual response (feedback for retry if no JSON received)
# Parses JSON to R object
json_1 <- base_prompt |>
  answer_as_json() |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# How can I solve 8x + 7 = -23?
#
# Your must format your response as a JSON object.
# --- Receiving response from LLM provider: ---
# Here is the solution to the equation formatted as a JSON object:
#
# ```
# {
#   "equation": "8x + 7 = -23",
#   "steps": [
#     {
#       "step": "Subtract 7 from both sides of the equation",
#       "expression": "-23 - 7"
#     },
#     {
#       "step": "Simplify the expression on the left side",
```

```

#   "result": "-30"
# },
# {
#   "step": "Divide both sides by -8 to solve for x",
#   "expression": "-30 / -8"
# },
# {
#   "step": "Simplify the expression on the right side",
#   "result": "3.75"
# }
# ],
# "solution": {
#   "x": 3.75
# }
# }
# ```

## Ollama:
# - Sets 'format' parameter to 'json', enforcing JSON
# - Adds request to prompt for a JSON object, as is recommended by the docs
# - Parses JSON to R object
json_2 <- base_prompt |>
  answer_as_json(type = "auto") |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# How can I solve  $8x + 7 = -23$ ?
#
# Your must format your response as a JSON object.
# --- Receiving response from LLM provider: ---
# {"steps": [
#   "Subtract 7 from both sides to get  $8x = -30$ ",
#   "Simplify the right side of the equation to get  $8x = -30$ ",
#   "Divide both sides by 8 to solve for x, resulting in  $x = -30/8$ ",
#   "Simplify the fraction to find the value of x"
# ],
# "value_of_x": "-3.75"}

## OpenAI-type API without schema:
# - Sets 'response_format' parameter to 'json_object', enforcing JSON
# - Adds request to prompt for a JSON object, as is required by the API
# - Parses JSON to R object
json_3 <- base_prompt |>
  answer_as_json(type = "auto") |>
  send_prompt(llm_provider_openai())
# --- Sending request to LLM provider (gpt-4o-mini): ---
# How can I solve  $8x + 7 = -23$ ?
#
# Your must format your response as a JSON object.
# --- Receiving response from LLM provider: ---
# {
#   "solution_steps": [

```

```

#   {
#     "step": 1,
#     "operation": "Subtract 7 from both sides",
#     "equation": "8x + 7 - 7 = -23 - 7",
#     "result": "8x = -30"
#   },
#   {
#     "step": 2,
#     "operation": "Divide both sides by 8",
#     "equation": "8x / 8 = -30 / 8",
#     "result": "x = -3.75"
#   }
# ],
# "solution": {
#   "x": -3.75
# }
# }

## End(Not run)

#### Enforcing JSON with a schema: ####

# Make a list representing a JSON schema,
# which the LLM response must adhere to:
json_schema <- list(
  name = "steps_to_solve", # Required for OpenAI API
  description = NULL, # Optional for OpenAI API
  schema = list(
    type = "object",
    properties = list(
      steps = list(
        type = "array",
        items = list(
          type = "object",
          properties = list(
            explanation = list(type = "string"),
            output = list(type = "string")
          ),
          required = c("explanation", "output"),
          additionalProperties = FALSE
        )
      ),
      final_answer = list(type = "string")
    ),
    required = c("steps", "final_answer"),
    additionalProperties = FALSE
  )
# 'strict' parameter is set as argument 'answer_as_json()'
)
# Note: when you are not using an OpenAI API, you can also pass just the
# internal 'schema' list object to 'answer_as_json()' instead of the full

```



```

# 'json_schema' list object

# Generate example R object based on schema:
r_json_schema_to_example(json_schema)

## Not run:
## Text-based with schema (works for any provider/model):
# - Adds request to prompt for a JSON object
# - Adds schema to prompt
# - Extracts JSON from textual response (feedback for retry if no JSON received)
# - Validates JSON against schema with 'jsonvalidate' package (feedback for retry if invalid)
# - Parses JSON to R object
json_4 <- base_prompt |>
  answer_as_json(schema = json_schema) |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# How can I solve  $8x + 7 = -23$ ?
#
# Your must format your response as a JSON object.
#
# Your JSON object should match this example JSON object:
# {
#   "steps": [
#     {
#       "explanation": "...",
#       "output": "..."
#     }
#   ],
#   "final_answer": "..."
# }
# --- Receiving response from LLM provider: ---
# Here is the solution to the equation:
#
# ---
# {
#   "steps": [
#     {
#       "explanation": "First, we want to isolate the term with 'x' by
#       subtracting 7 from both sides of the equation.",
#       "output": " $8x + 7 - 7 = -23 - 7$ "
#     },
#     {
#       "explanation": "This simplifies to:  $8x = -30$ ",
#       "output": " $8x = -30$ "
#     },
#     {
#       "explanation": "Next, we want to get rid of the coefficient '8' by
#       dividing both sides of the equation by 8.",
#       "output": " $(8x) / 8 = (-30) / 8$ "
#     },
#     {
#       "explanation": "This simplifies to:  $x = -3.75$ ",
#       "output": " $x = -3.75$ "
#     }
#   ]
# }

```

```

#   }
# ],
# "final_answer": "-3.75"
# }
# ```

## Ollama with schema:
# - Sets 'format' parameter to 'json', enforcing JSON
# - Adds request to prompt for a JSON object, as is recommended by the docs
# - Adds schema to prompt
# - Validates JSON against schema with 'jsonvalidate' package (feedback for retry if invalid)
json_5 <- base_prompt |>
  answer_as_json(json_schema, type = "auto") |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# How can I solve  $8x + 7 = -23$ ?
#
# Your must format your response as a JSON object.
#
# Your JSON object should match this example JSON object:
# {
#   "steps": [
#     {
#       "explanation": "...",
#       "output": "..."
#     }
#   ],
#   "final_answer": "..."
# }
# --- Receiving response from LLM provider: ---
# {
#   "steps": [
#     {
#       "explanation": "First, subtract 7 from both sides of the equation to
#       isolate the term with x.",
#       "output": " $8x = -23 - 7$ "
#     },
#     {
#       "explanation": "Simplify the right-hand side of the equation.",
#       "output": " $8x = -30$ "
#     },
#     {
#       "explanation": "Next, divide both sides of the equation by 8 to solve for x.",
#       "output": " $x = -30 / 8$ "
#     },
#     {
#       "explanation": "Simplify the right-hand side of the equation.",
#       "output": " $x = -3.75$ "
#     }
#   ],
#   "final_answer": "-3.75"
# }

```

```

## OpenAI with schema:
# - Sets 'response_format' parameter to 'json_object', enforcing JSON
# - Adds json_schema to the API request, API enforces JSON adhering schema
# - Parses JSON to R object
json_6 <- base_prompt |>
  answer_as_json(json_schema, type = "auto") |>
  send_prompt(llm_provider_openai())
# --- Sending request to LLM provider (gpt-4o-mini): ---
# How can I solve  $8x + 7 = -23$ ?
# --- Receiving response from LLM provider: ---
# {"steps":[
# {"explanation":"Start with the original equation.",
# "output":" $8x + 7 = -23$ "},
# {"explanation":"Subtract 7 from both sides to isolate the term with x.",
# "output":" $8x + 7 - 7 = -23 - 7$ "},
# {"explanation":"Simplify the left side and the right side of the equation.",
# "output":" $8x = -30$ "},
# {"explanation":"Now, divide both sides by 8 to solve for x.",
# "output":" $x = -30 / 8$ "},
# {"explanation":"Simplify the fraction by dividing both the numerator and the
# denominator by 2.",
# "output":" $x = -15 / 4$ "}
# ], "final_answer":"x = -15/4"}

## End(Not run)

```

---

send\_prompt

*Send a prompt to a LLM provider*


---

## Description

This function is responsible for sending prompts to a LLM provider for evaluation. The function will interact with the LLM provider until a successful response is received or the maximum number of interactions is reached. The function will apply extraction and validation functions to the LLM response, as specified in the prompt wraps (see [prompt\\_wrap\(\)](#)). If the maximum number of interactions

## Usage

```

send_prompt(
  prompt,
  llm_provider = llm_provider_ollama(),
  max_interactions = 10,
  clean_chat_history = TRUE,
  verbose = NULL,
  stream = NULL,
  return_mode = c("only_response", "full")
)

```

**Arguments**

prompt	A string or a <code>tidyprompt</code> object
llm_provider	<code>llm_provider</code> object (default is <code>llm_provider_ollama()</code> ). This object and its settings will be used to evaluate the prompt. Note that the 'verbose' and 'stream' settings in the LLM provider will be overruled by the 'verbose' and 'stream' arguments in this function when those are not NULL. Furthermore, advanced <code>tidyprompt</code> objects may carry '\$parameter_fn' functions which can set parameters in the llm_provider object (see <code>prompt_wrap()</code> and <code>llm_provider</code> for more).
max_interactions	Maximum number of interactions allowed with the LLM provider. Default is 10. If the maximum number of interactions is reached without a successful response, 'NULL' is returned as the response (see return value). The first interaction is the initial chat completion
clean_chat_history	If the chat history should be cleaned after each interaction. Cleaning the chat history means that only the first and last message from the user, the last message from the assistant, all messages from the system, and all tool results are kept in a 'clean' chat history. This clean chat history is used when requesting a new chat completion. (i.e., if a LLM repeatedly fails to provide a correct response, only its last failed response will be included in the context window). This may increase the LLM performance on the next interaction
verbose	If the interaction with the LLM provider should be printed to the console. This will overrule the 'verbose' setting in the LLM provider
stream	If the interaction with the LLM provider should be streamed. This setting will only be used if the LLM provider already has a 'stream' parameter (which indicates there is support for streaming). Note that when 'verbose' is set to FALSE, the 'stream' setting will be ignored
return_mode	One of 'full' or 'only_response'. See return value

**Value**

- If return mode 'only\_response', the function will return only the LLM response after extraction and validation functions have been applied (NULL is returned when unsuccessful after the maximum number of interactions).
- If return mode 'full', the function will return a list with the following elements:
  - 'response' (the LLM response after extraction and validation functions have been applied; NULL is returned when unsuccessful after the maximum number of interactions),
  - 'interactions' (the number of interactions with the LLM provider),
  - 'chat\_history' (a dataframe with the full chat history which led to the final response),
  - 'chat\_history\_clean' (a dataframe with the cleaned chat history which led to the final response; here, only the first and last message from the user, the last message from the assistant, and all messages from the system are kept),
  - 'start\_time' (the time when the function was called),
  - 'end\_time' (the time when the function ended),

- 'duration\_seconds' (the duration of the function in seconds), and
- 'http\_list' (a list with all HTTP responses made during the interactions; as returned by `llm_provider$complete_chat()`).

### See Also

[tidyprompt](#), [prompt\\_wrap\(\)](#), [llm\\_provider](#), [llm\\_provider\\_ollama\(\)](#), [llm\\_provider\\_openai\(\)](#)

Other `prompt_evaluation`: [llm\\_break\(\)](#), [llm\\_feedback\(\)](#)

### Examples

```
## Not run:
"Hi!" |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi!
# --- Receiving response from LLM provider: ---
# It's nice to meet you. Is there something I can help you with, or would you like to chat?
# [1] "It's nice to meet you. Is there something I can help you with, or would you like to chat?"

"Hi!" |>
  send_prompt(llm_provider_ollama(), return_mode = "full")
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi!
# --- Receiving response from LLM provider: ---
# It's nice to meet you. Is there something I can help you with, or would you like to chat?
# $response
# [1] "It's nice to meet you. Is there something I can help you with, or would you like to chat?"
#
# $chat_history
# ...
#
# $chat_history_clean
# ...
#
# $start_time
# [1] "2024-11-18 15:43:12 CET"
#
# $end_time
# [1] "2024-11-18 15:43:13 CET"
#
# $duration_seconds
# [1] 1.13276
#
# $http_list
# $http_list[[1]]
# Response [http://localhost:11434/api/chat]
# Date: 2024-11-18 14:43
# Status: 200
# Content-Type: application/x-ndjson
# <EMPTY BODY>
```

```

"Hi!" |>
  add_text("What is 5 + 5?") |>
  answer_as_integer() |>
  send_prompt(llm_provider_ollama(), verbose = FALSE)
# [1] 10

## End(Not run)

```

---

set\_chat\_history      *Set the chat history of a [tidyprompt](#) object*

---

### Description

This function sets the chat history for a [tidyprompt](#) object. The chat history will also set the base prompt and system prompt (the last message of the chat history should be of role 'user' and will be used as the base prompt; the first message of the chat history may be of the role 'system' and will then be used as the system prompt).

This may be useful when one wants to change the base prompt, system prompt, and chat history of a [tidyprompt](#) object while retaining other fields like the list of prompt wraps.

### Usage

```
set_chat_history(x, chat_history)
```

### Arguments

x                    A [tidyprompt](#) object  
chat\_history        A valid chat history (see [chat\\_history\(\)](#))

### Value

The updated [tidyprompt](#) object

### See Also

[chat\\_history\(\)](#)

Other [tidyprompt](#): [construct\\_prompt\\_text\(\)](#), [get\\_chat\\_history\(\)](#), [get\\_prompt\\_wraps\(\)](#), [is\\_tidyprompt\(\)](#), [tidyprompt\(\)](#), [tidyprompt-class](#)

### Examples

```

prompt <- tidyprompt("Hi!")
print(prompt)

# Add to a tidyprompt using a prompt wrap:
prompt <- tidyprompt("Hi!") |>
  add_text("How are you?")
print(prompt)

```

```

# Strings can be input for prompt wraps; therefore,
# a call to tidyprompt() is not necessary:
prompt <- "Hi" |>
  add_text("How are you?")

# Example of adding extraction & validation with a prompt_wrap():
prompt <- "Hi" |>
  add_text("What is 5 + 5?") |>
  answer_as_integer()

## Not run:
# tidyprompt objects are evaluated by send_prompt(), which will
# handle construct the prompt text, send it to the LLM provider,
# and apply the extraction and validation functions from the tidyprompt object
prompt |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi
#
# What is 5 + 5?
#
# You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
# 10
# [1] 10

# See prompt_wrap() and send_prompt() for more details

## End(Not run)

# `tidyprompt` objects may be validated with these helpers:
is_tidyprompt(prompt) # Returns TRUE if input is a valid tidyprompt object

# Get base prompt text
base_prompt <- prompt$base_prompt

# Get all prompt wraps
prompt_wraps <- prompt$get_prompt_wraps()
# Alternative:
prompt_wraps <- get_prompt_wraps(prompt)

# Construct prompt text
prompt_text <- prompt$construct_prompt_text()
# Alternative:
prompt_text <- construct_prompt_text(prompt)

# Set chat history (affecting also the base prompt)
chat_history <- data.frame(
  role = c("user", "assistant", "user"),
  content = c("What is 5 + 5?", "10", "And what is 5 + 6?")
)
prompt$set_chat_history(chat_history)

```

```
# Get chat history
chat_history <- prompt$get_chat_history()
```

---

```
set_system_prompt      Set system prompt of a tidyprompt object
```

---

## Description

Set the system prompt for a prompt. The system prompt will be added as a message with role 'system' at the start of the chat history when this prompt is evaluated by `send_prompt()`.

## Usage

```
set_system_prompt(prompt, system_prompt)
```

## Arguments

`prompt` A single string or a `tidyprompt()` object  
`system_prompt` A single character string representing the system prompt

## Details

The system prompt will be stored in the `tidyprompt()` object as '`$system_prompt`'.

## Value

A `tidyprompt()` with the system prompt set

## See Also

Other `pre_built_prompt_wraps`: `add_text()`, `answer_as_boolean()`, `answer_as_integer()`, `answer_as_json()`, `answer_as_list()`, `answer_as_named_list()`, `answer_as_regex_match()`, `answer_as_text()`, `answer_by_chain_of_thought()`, `answer_by_react()`, `answer_using_r()`, `answer_using_sql()`, `answer_using_tools()`, `prompt_wrap()`, `quit_if()`

Other `miscellaneous_prompt_wraps`: `add_text()`, `quit_if()`

## Examples

```
prompt <- "Hi there!" |>
  set_system_prompt("You are an assistant who always answers in very short poems.")
prompt$system_prompt
```

```
## Not run:
prompt |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi there!
```



```
# --- Receiving response from LLM provider: ---
# Hello to you, I say,
# Welcome here, come what may!
# How can I assist today?
# [1] "Hello to you, I say,\nWelcome here, come what may!\nHow can I assist today?"

## End(Not run)
```

---

`skim_with_labels_and_levels`

*Skim a dataframe and include labels and levels*

---

## Description

This function takes a `data.frame` and returns a skim summary with variable names, labels, and levels for categorical variables. It is a wrapper around the `skimr::skim()` function.

## Usage

```
skim_with_labels_and_levels(data)
```

## Arguments

`data`            A `data.frame` to be skimmed

## Value

A `data.frame` with variable names, labels, levels, and a skim summary

## See Also

Other `text_helpers`: [df\\_to\\_string\(\)](#), [vector\\_list\\_to\\_string\(\)](#)

## Examples

```
# First add some labels to 'mtcars':
mtcars$car <- rownames(mtcars)
mtcars$car <- factor(mtcars$car, levels = rownames(mtcars))
attr(mtcars$car, "label") <- "Name of the car"

# Then skim the data:
mtcars |>
  skim_with_labels_and_levels()
```

---

`tidyprompt`*Create a `tidyprompt` object*

---

## Description

This is a wrapper around the `tidyprompt` constructor.

## Usage

```
tidyprompt(input)
```

## Arguments

<code>input</code>	A string, a chat history, a list containing a chat history under key <code>'\$chat_history'</code> , or a <code>tidyprompt</code> object
--------------------	--

## Details

Different types of input are accepted for initialization of a `tidyprompt` object:

- A single character string. This will be used as the base prompt
- A dataframe which is a valid chat history (see `chat_history()`)
- A list containing a valid chat history under `'$chat_history'` (e.g., a result from `send_prompt()` when using `'return_mode' = "full"`)
- A `tidyprompt` object. This will be checked for validity and, if valid, the fields are copied to the object which is returned from this method

When passing a dataframe or list with a chat history, the last row of the chat history must have role `'user'`; this row will be used as the base prompt. If the first row of the chat history has role `'system'`, it will be used as the system prompt.

## Value

A `tidyprompt` object

## See Also

Other `tidyprompt`: `construct_prompt_text()`, `get_chat_history()`, `get_prompt_wraps()`, `is_tidyprompt()`, `set_chat_history()`, `tidyprompt-class`

## Examples

```
prompt <- tidyprompt("Hi!")
print(prompt)

# Add to a tidyprompt using a prompt wrap:
prompt <- tidyprompt("Hi!") |>
  add_text("How are you?")
```

```

print(prompt)

# Strings can be input for prompt wraps; therefore,
# a call to tidyprompt() is not necessary:
prompt <- "Hi" |>
  add_text("How are you?")

# Example of adding extraction & validation with a prompt_wrap():
prompt <- "Hi" |>
  add_text("What is 5 + 5?") |>
  answer_as_integer()

## Not run:
# tidyprompt objects are evaluated by send_prompt(), which will
# handle construct the prompt text, send it to the LLM provider,
# and apply the extraction and validation functions from the tidyprompt object
prompt |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
# Hi
#
# What is 5 + 5?
#
# You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
# 10
# [1] 10

# See prompt_wrap() and send_prompt() for more details

## End(Not run)

# `tidyprompt` objects may be validated with these helpers:
is_tidyprompt(prompt) # Returns TRUE if input is a valid tidyprompt object

# Get base prompt text
base_prompt <- prompt$base_prompt

# Get all prompt wraps
prompt_wraps <- prompt$get_prompt_wraps()
# Alternative:
prompt_wraps <- get_prompt_wraps(prompt)

# Construct prompt text
prompt_text <- prompt$construct_prompt_text()
# Alternative:
prompt_text <- construct_prompt_text(prompt)

# Set chat history (affecting also the base prompt)
chat_history <- data.frame(
  role = c("user", "assistant", "user"),
  content = c("What is 5 + 5?", "10", "And what is 5 + 6?")
)

```

```

prompt$set_chat_history(chat_history)

# Get chat history
chat_history <- prompt$get_chat_history()

```

---

tidyprompt-class      *Tidyprompt R6 Class*

---

## Description

A [tidyprompt](#) object contains a base prompt and a list of [prompt\\_wrap\(\)](#) objects. It provides structured methods to modify the prompt while simultaneously adding logic to extract from and validate the LLM response. Besides a base prompt, a [tidyprompt](#) object may contain a system prompt and a chat history which precede the base prompt.

## Public fields

`base_prompt` The base prompt string. The base prompt be modified by prompt wraps during [construct\\_prompt\\_text\(\)](#); the modified prompt text will be used as the final message of role 'user' during [send\\_prompt\(\)](#)

`system_prompt` A system prompt string. This will be added at the start of the chat history as role 'system' during [send\\_prompt\(\)](#)

## Methods

### Public methods:

- [tidyprompt-class\\$new\(\)](#)
- [tidyprompt-class\\$is\\_valid\(\)](#)
- [tidyprompt-class\\$add\\_prompt\\_wrap\(\)](#)
- [tidyprompt-class\\$get\\_prompt\\_wraps\(\)](#)
- [tidyprompt-class\\$construct\\_prompt\\_text\(\)](#)
- [tidyprompt-class\\$set\\_chat\\_history\(\)](#)
- [tidyprompt-class\\$get\\_chat\\_history\(\)](#)
- [tidyprompt-class\\$clone\(\)](#)

**Method** `new()`: Initialize a [tidyprompt](#) object

*Usage:*

```
tidyprompt-class$new(input)
```

*Arguments:*

`input` A string, a chat history, a list containing a chat history under key '\$chat\_history', or a [tidyprompt](#) object

*Details:* Different types of input are accepted for initialization of a [tidyprompt](#) object:

- A single character string. This will be used as the base prompt
- A dataframe which is a valid chat history (see [chat\\_history\(\)](#))

- A list containing a valid chat history under '\$chat\_history' (e.g., a result from `send_prompt()` when using 'return\_mode' = "full")
- A `tidyprompt` object. This will be checked for validity and, if valid, the fields are copied to the object which is returned from this method

When passing a dataframe or list with a chat history, the last row of the chat history must have role 'user'; this row will be used as the base prompt. If the first row of the chat history has role 'system', it will be used as the system prompt.

*Returns:* A `tidyprompt` object

**Method** `is_valid()`: Check if the `tidyprompt` object is valid.

*Usage:*

```
tidyprompt-class$is_valid()
```

*Returns:* TRUE if valid, otherwise FALSE

**Method** `add_prompt_wrap()`: Add a `prompt_wrap()` to the `tidyprompt` object.

*Usage:*

```
tidyprompt-class$add_prompt_wrap(prompt_wrap)
```

*Arguments:*

`prompt_wrap` A `prompt_wrap()` object

*Returns:* The updated `tidyprompt` object

**Method** `get_prompt_wraps()`: Get list of `prompt_wrap()` objects from the `tidyprompt` object.

*Usage:*

```
tidyprompt-class$get_prompt_wraps(
  order = c("default", "modification", "evaluation")
)
```

*Arguments:*

`order` The order to return the wraps. Options are:

- "default": as originally added to the object
- "modification": as ordered for modification of the base prompt; ordered by type: check, unspecified, mode, tool, break. This is the order in which prompt wraps are applied during `construct_prompt_text()`
- "evaluation": ordered for evaluation of the LLM response; ordered by type: tool, mode, break, unspecified, check. This is the order in which wraps are applied to the LLM output during `send_prompt()`

*Returns:* A list of `prompt_wrap()` objects.

**Method** `construct_prompt_text()`: Construct the complete prompt text.

*Usage:*

```
tidyprompt-class$construct_prompt_text(llm_provider = NULL)
```

*Arguments:*

`llm_provider` Optional `llm_provider` object. This may sometimes affect the prompt text construction

*Returns:* A string representing the constructed prompt text

**Method** `set_chat_history()`: This function sets the chat history for the `tidyprompt` object. The chat history will also set the base prompt and system prompt (the last message of the chat history should be of role 'user' and will be used as the base prompt; the first message of the chat history may be of the role 'system' and will then be used as the system prompt). This may be useful when one wants to change the base prompt, system prompt, and chat history of a `tidyprompt` object while retaining other fields like the prompt wraps.

*Usage:*

```
tidyprompt-class$set_chat_history(chat_history)
```

*Arguments:*

`chat_history` A valid chat history (see `chat_history()`)

*Returns:* The updated `tidyprompt` object

**Method** `get_chat_history()`: This function gets the chat history of the `tidyprompt` object. The chat history is constructed from the base prompt, system prompt, and chat history field. The returned object will be the chat history with the system prompt as the first message with role 'system' and the the base prompt as the last message with role 'user'.

*Usage:*

```
tidyprompt-class$get_chat_history()
```

*Returns:* A dataframe containing the chat history

**Method** `clone()`: The objects of this class are cloneable with this method.

*Usage:*

```
tidyprompt-class$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## See Also

Other `tidyprompt`: `construct_prompt_text()`, `get_chat_history()`, `get_prompt_wraps()`, `is_tidyprompt()`, `set_chat_history()`, `tidyprompt()`

## Examples

```
prompt <- tidyprompt("Hi!")
print(prompt)

# Add to a tidyprompt using a prompt wrap:
prompt <- tidyprompt("Hi!") |>
  add_text("How are you?")
print(prompt)

# Strings can be input for prompt wraps; therefore,
# a call to tidyprompt() is not necessary:
prompt <- "Hi" |>
  add_text("How are you?")
```

```

# Example of adding extraction & validation with a prompt_wrap():
prompt <- "Hi" |>
  add_text("What is 5 + 5?") |>
  answer_as_integer()

## Not run:
# tidyprompt objects are evaluated by send_prompt(), which will
#   handle construct the prompt text, send it to the LLM provider,
#   and apply the extraction and validation functions from the tidyprompt object
prompt |>
  send_prompt(llm_provider_ollama())
# --- Sending request to LLM provider (llama3.1:8b): ---
#   Hi
#
#   What is 5 + 5?
#
#   You must answer with only an integer (use no other characters).
# --- Receiving response from LLM provider: ---
#   10
# [1] 10

# See prompt_wrap() and send_prompt() for more details

## End(Not run)

# `tidyprompt` objects may be validated with these helpers:
is_tidyprompt(prompt) # Returns TRUE if input is a valid tidyprompt object

# Get base prompt text
base_prompt <- prompt$base_prompt

# Get all prompt wraps
prompt_wraps <- prompt$get_prompt_wraps()
# Alternative:
prompt_wraps <- get_prompt_wraps(prompt)

# Construct prompt text
prompt_text <- prompt$construct_prompt_text()
# Alternative:
prompt_text <- construct_prompt_text(prompt)

# Set chat history (affecting also the base prompt)
chat_history <- data.frame(
  role = c("user", "assistant", "user"),
  content = c("What is 5 + 5?", "10", "And what is 5 + 6?")
)
prompt$set_chat_history(chat_history)

# Get chat history
chat_history <- prompt$get_chat_history()

```

tools\_add\_docs

*Add tidyprompt function documentation to a function***Description**

This function adds documentation to a custom function. This documentation is used to extract information about the function's name, description, arguments, and return value. This information is used to provide an LLM with information about the functions, so that the LLM can call R functions. The intended use of this function is to add documentation to custom functions that do not have help files; `tools_get_docs()` may generate documentation from a help file when the function is part of base R or a package.

If a function already has documentation, the documentation added by this function may overwrite it. If you wish to modify existing documentation, you may make a call to `tools_get_docs()` to extract the existing documentation, modify it, and then call `tools_add_docs()` to add the modified documentation.

**Usage**

```
tools_add_docs(func, docs)
```

**Arguments**

`func` A function object

`docs`

A list with the following elements:

- `'name'`: (optional) The name of the function. If not provided, the function name will be extracted from the function object. Use this parameter to override the function name if necessary
- `'description'`: A description of the function and its purpose
- `'arguments'`: A named list of arguments with descriptions. Each argument is a list which may contain:
  - `'description'`: A description of the argument and its purpose. Not required or used for native function calling (e.g., with OpenAI), but recommended for text-based function calling
  - `'type'`: The type of the argument. This should be one of: `'integer'`, `'numeric'`, `'logical'`, `'string'`, `'match.arg'`, `'vector integer'`, `'vector numeric'`, `'vector logical'`, `'vector string'`. For arguments which are named lists, `'type'` should be a named list which contains the types of the elements. For type `'match.arg'`, the possible values should be passed as a vector under `'default_value'`. `'type'` is required for native function calling (with, e.g., OpenAI) but may also be useful to provide for text-based function calling, in which it will be added to the prompt introducing the function
  - `'default_value'`: The default value of the argument. This is only required when `'type'` is set to `'match.arg'`. It should then be a vector of possible values for the argument. In other cases, it is not required; for



native function calling, it is not used in other cases; for text-based function calling, it may be useful to provide the default value, which will be added to the prompt introducing the function

- 'return': A list with the following elements:
  - 'description': A description of the return value or the side effects of the function

## Value

The function object with the documentation added as an attribute ('tidyprompt\_tool\_docs')

## See Also

Other tools: [answer\\_using\\_tools\(\)](#), [tools\\_get\\_docs\(\)](#)

## Examples

```
## Not run:
# When using functions from base R or R packages,
# documentation is automatically extracted from help files:
"what are the files in my current directory?" |>
  answer_using_tools(dir) |> # 'dir' function is from base R
  send_prompt()

## End(Not run)

# Custom functions may also be provided;
# in this case, some documentation is extracted from the function's formals;
# descriptions may be added manually. See below

# Example fake weather function to add to the prompt:
temperature_in_location <- function(
  location = c("Amsterdam", "Utrecht", "Enschede"),
  unit = c("Celcius", "Fahrenheit")
) {
  location <- match.arg(location)
  unit <- match.arg(unit)

  temperature_celcius <- switch(
    location,
    "Amsterdam" = 32.5,
    "Utrecht" = 19.8,
    "Enschede" = 22.7
  )

  if (unit == "Celcius") {
    return(temperature_celcius)
  } else {
    return(temperature_celcius * 9/5 + 32)
  }
}
```

```

# Generate documentation for a function
# (based on formals, & help file if available)
docs <- tools_get_docs(temperature_in_location)

# The types get inferred from the function's formals
# However, descriptions are still missing as the function is not from a package
# We can modify the documentation object to add descriptions:
docs$description <- "Get the temperature in a location"
docs$arguments$unit$description <- "Unit in which to return the temperature"
docs$arguments$location$description <- "Location for which to return the temperature"
docs$return$description <- "The temperature in the specified location and unit"
# (See `?tools_add_docs` for more details on the structure of the documentation)

# When we are satisfied with the documentation, we can add it to the function:
temperature_in_location <- tools_add_docs(temperature_in_location, docs)

## Not run:
# Now the LLM can use the function:
"Hi, what is the weather in Enschede? Give me Celcius degrees" |>
  answer_using_tools(temperature_in_location) |>
  send_prompt()

## End(Not run)

```

---

tools\_get\_docs

*Extract documentation from a function*


---

## Description

This function extracts documentation from a help file (if available, i.e., when the function is part of a package) or from documentation added by `tools_add_docs()`. The extracted documentation includes the function's name, description, arguments, and return value. This information is used to provide an LLM with information about the functions, so that the LLM can call R functions.

## Usage

```
tools_get_docs(func, name = NULL)
```

## Arguments

func	A function object. The function should belong to a package and have documentation available in a help file, or it should have documentation added by <code>tools_add_docs()</code>
name	The name of the function if already known (optional). If not provided it will be extracted from the documentation or the function object's name

## Details

This function will prioritize documentation added by `tools_add_docs()` over documentation from a help file. Thus, it is possible to override the help file documentation by adding custom documentation

## Value

A list with documentation for the function. See `tools_add_docs()` for more information on the contents

## See Also

Other tools: `answer_using_tools()`, `tools_add_docs()`

## Examples

```
## Not run:
# When using functions from base R or R packages,
# documentation is automatically extracted from help files:
"What are the files in my current directory?" |>
  answer_using_tools(dir) |> # 'dir' function is from base R
  send_prompt()

## End(Not run)

# Custom functions may also be provided;
# in this case, some documentation is extracted from the function's formals;
# descriptions may be added manually. See below

# Example fake weather function to add to the prompt:
temperature_in_location <- function(
  location = c("Amsterdam", "Utrecht", "Enschede"),
  unit = c("Celcius", "Fahrenheit")
) {
  location <- match.arg(location)
  unit <- match.arg(unit)

  temperature_celcius <- switch(
    location,
    "Amsterdam" = 32.5,
    "Utrecht" = 19.8,
    "Enschede" = 22.7
  )

  if (unit == "Celcius") {
    return(temperature_celcius)
  } else {
    return(temperature_celcius * 9/5 + 32)
  }
}

# Generate documentation for a function
```

```

# (based on formals, & help file if available)
docs <- tools_get_docs(temperature_in_location)

# The types get inferred from the function's formals
# However, descriptions are still missing as the function is not from a package
# We can modify the documentation object to add descriptions:
docs$description <- "Get the temperature in a location"
docs$arguments$unit$description <- "Unit in which to return the temperature"
docs$arguments$location$description <- "Location for which to return the temperature"
docs$return$description <- "The temperature in the specified location and unit"
# (See `?tools_add_docs` for more details on the structure of the documentation)

# When we are satisfied with the documentation, we can add it to the function:
temperature_in_location <- tools_add_docs(temperature_in_location, docs)

## Not run:
# Now the LLM can use the function:
"Hi, what is the weather in Enschede? Give me Celcius degrees" |>
  answer_using_tools(temperature_in_location) |>
  send_prompt()

## End(Not run)

```

---

user\_verify

*Have user check the result of a prompt (human-in-the-loop)*


---

## Description

This function is used to have a user check the result of a prompt. After evaluation of the prompt and applying prompt wraps, the user is presented with the result and asked to accept or decline. If the user declines, they are asked to provide feedback to the large language model (LLM) so that the LLM can retry the prompt.

## Usage

```
user_verify(prompt)
```

## Arguments

prompt            A single string or a [tidyprompt](#) object

## Value

A [tidyprompt](#) with an added [prompt\\_wrap\(\)](#) which will add a check for the user to accept or decline the result of the prompt, providing feedback if the result is declined

**Examples**

```

## Not run:
  "Tell me a fun fact about yourself!" |>
    user_verify() |>
    send_prompt()
# --- Sending request to LLM provider (gpt-4o-mini): ---
# Tell me a fun fact about yourself!
# --- Receiving response from LLM provider: ---
# I don't have personal experiences or feelings, but a fun fact about me is that
# I can generate text in multiple languages! From English to Spanish, French, and
# more, I'm here to help with diverse linguistic needs.
#
# --- Evaluation of tidyprompt resulted in:
# [1] "I don't have personal experiences or feelings, but a fun fact about me is
# that I can generate text in multiple languages! From English to Spanish, French,
# and more, I'm here to help with diverse linguistic needs."
#
# --- Accept or decline
# * If satisfied, type nothing
# * If not satisfied, type feedback to the LLM
# Type: Needs to be funnier!
# --- Sending request to LLM provider (gpt-4o-mini): ---
# Needs to be funnier!
# --- Receiving response from LLM provider: ---
# Alright, how about this: I once tried to tell a joke, but my punchline got lost
# in translation! Now, I just stick to delivering 'byte-sized' humor!
#
# --- Evaluation of tidyprompt resulted in:
# [1] "Alright, how about this: I once tried to tell a joke, but my punchline got
# lost in translation! Now, I just stick to delivering 'byte-sized' humor!"
#
# --- Accept or decline
# * If satisfied, type nothing
# * If not satisfied, type feedback to the LLM
# Type:
# * Result accepted
# [1] "Alright, how about this: I once tried to tell a joke, but my punchline got
# lost in translation! Now, I just stick to delivering 'byte-sized' humor!"

## End(Not run)

```

---

vector\_list\_to\_string *Convert a named or unnamed list/vector to a string representation*

---

**Description**

Converts a named or unnamed list/vector to a string format, intended for sending it to an LLM (or for display or logging).

**Usage**

```
vector_list_to_string(obj, how = c("inline", "expanded"))
```

**Arguments**

obj	A list or vector (named or unnamed) to be converted to a string.
how	In what way the object should be converted to a string; either "inline" or "expanded". "inline" presents all key-value pairs or values as a single line. "expanded" presents each key-value pair or value on a separate line.

**Value**

A single string representing the list/vector.

**See Also**

Other text\_helpers: [df\\_to\\_string\(\)](#), [skim\\_with\\_labels\\_and\\_levels\(\)](#)

**Examples**

```
named_vector <- c(x = 10, y = 20, z = 30)

vector_list_to_string(named_vector, how = "inline")

vector_list_to_string(named_vector, how = "expanded")
```

# Index

- \* **answer\_as\_prompt\_wraps**
  - answer\_as\_boolean, 6
  - answer\_as\_integer, 7
  - answer\_as\_json, 8
  - answer\_as\_list, 17
  - answer\_as\_named\_list, 19
  - answer\_as\_regex\_match, 20
  - answer\_as\_text, 22
- \* **answer\_by\_prompt\_wraps**
  - answer\_by\_chain\_of\_thought, 23
  - answer\_by\_react, 25
- \* **answer\_using\_prompt\_wraps**
  - answer\_using\_r, 28
  - answer\_using\_sql, 31
  - answer\_using\_tools, 34
- \* **chat\_history**
  - chat\_history, 37
- \* **json**
  - answer\_as\_json, 8
  - r\_json\_schema\_to\_example, 77
- \* **llm\_provider**
  - llm\_provider\_class, 50
  - llm\_provider\_google\_gemini, 54
  - llm\_provider\_groq, 56
  - llm\_provider\_mistral, 57
  - llm\_provider\_ollama, 59
  - llm\_provider\_openai, 61
  - llm\_provider\_openrouter, 63
  - llm\_provider\_xai, 65
- \* **miscellaneous\_helpers**
  - extract\_from\_return\_list, 41
- \* **miscellaneous\_prompt\_wraps**
  - add\_text, 5
  - quit\_if, 76
  - set\_system\_prompt, 88
- \* **pre\_built\_prompt\_wraps**
  - add\_text, 5
  - answer\_as\_boolean, 6
  - answer\_as\_integer, 7
  - answer\_as\_json, 8
  - answer\_as\_list, 17
  - answer\_as\_named\_list, 19
  - answer\_as\_regex\_match, 20
  - answer\_as\_text, 22
  - answer\_by\_chain\_of\_thought, 23
  - answer\_by\_react, 25
  - answer\_using\_r, 28
  - answer\_using\_sql, 31
  - answer\_using\_tools, 34
  - prompt\_wrap, 71
  - quit\_if, 76
  - set\_system\_prompt, 88
- \* **prompt\_evaluation**
  - llm\_break, 47
  - llm\_feedback, 49
  - send\_prompt, 83
- \* **prompt\_wrap**
  - llm\_break, 47
  - llm\_feedback, 49
  - prompt\_wrap, 71
- \* **text\_helpers**
  - df\_to\_string, 40
  - skim\_with\_labels\_and\_levels, 89
  - vector\_list\_to\_string, 101
- \* **tidyprompt**
  - construct\_prompt\_text, 38
  - get\_chat\_history, 42
  - get\_prompt\_wraps, 43
  - is\_tidyprompt, 45
  - set\_chat\_history, 86
  - tidyprompt, 90
  - tidyprompt-class, 92
- \* **tools**
  - answer\_using\_tools, 34
  - tools\_add\_docs, 96
  - tools\_get\_docs, 98
  - ..., 24, 26
- add\_msg\_to\_chat\_history, 3

- add\_text, [5](#), [7](#), [8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [26](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_as\_boolean, [6](#), [6](#), [8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [26](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_as\_integer, [6](#), [7](#), [7](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [26](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_as\_integer(), [72](#)
- answer\_as\_json, [6–8](#), [8](#), [18](#), [20](#), [21](#), [23](#), [24](#), [26](#), [30](#), [32](#), [35](#), [73](#), [77](#), [78](#), [88](#)
- answer\_as\_json(), [50](#), [51](#), [72](#)
- answer\_as\_key\_value, [15](#)
- answer\_as\_list, [6–8](#), [10](#), [17](#), [20](#), [21](#), [23](#), [24](#), [26](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_as\_list(), [20](#)
- answer\_as\_named\_list, [6–8](#), [10](#), [18](#), [19](#), [21](#), [23](#), [24](#), [26](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_as\_named\_list(), [18](#)
- answer\_as\_regex\_match, [6–8](#), [10](#), [18](#), [20](#), [20](#), [23](#), [24](#), [26](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_as\_text, [6–8](#), [10](#), [18](#), [20](#), [21](#), [22](#), [24](#), [26](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_by\_chain\_of\_thought, [6–8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [23](#), [27](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_by\_chain\_of\_thought(), [72](#)
- answer\_by\_react, [6–8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [25](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_by\_react(), [72](#)
- answer\_using\_r, [6–8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [27](#), [28](#), [32](#), [35](#), [73](#), [77](#), [88](#)
- answer\_using\_r(), [26](#), [35](#), [72](#)
- answer\_using\_sql, [6–8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [27](#), [30](#), [31](#), [35](#), [73](#), [77](#), [88](#)
- answer\_using\_tools, [6–8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [27](#), [30](#), [32](#), [34](#), [73](#), [77](#), [88](#), [97](#), [99](#)
- answer\_using\_tools(), [26](#), [30](#), [72](#)
- chat\_history, [37](#)
- chat\_history(), [3](#), [4](#), [42](#), [51](#), [52](#), [69](#), [70](#), [86](#), [90](#), [92](#), [94](#)
- construct\_prompt\_text, [38](#), [42](#), [44](#), [46](#), [86](#), [90](#), [94](#)
- construct\_prompt\_text(), [44](#), [92](#), [93](#)
- df\_to\_string, [40](#), [89](#), [102](#)
- extract\_from\_return\_list, [41](#)
- get\_chat\_history, [39](#), [42](#), [44](#), [46](#), [86](#), [90](#), [94](#)
- get\_prompt\_wraps, [39](#), [42](#), [43](#), [46](#), [86](#), [90](#), [94](#)
- is\_tidyprompt, [39](#), [42](#), [44](#), [45](#), [86](#), [90](#), [94](#)
- jsonlite::fromJSON(), [9](#)
- jsonlite::toJSON(), [9](#)
- jsonvalidate::json\_validate(), [9](#), [10](#)
- llm\_break, [47](#), [49](#), [73](#), [85](#)
- llm\_break(), [49](#), [72](#), [73](#)
- llm\_feedback, [48](#), [49](#), [73](#), [85](#)
- llm\_feedback(), [19](#), [20](#), [35](#), [48](#), [72](#)
- llm\_provider, [38](#), [50–52](#), [54](#), [56–65](#), [67](#), [69](#), [70](#), [72](#), [73](#), [84](#), [85](#), [93](#)
- llm\_provider-class, [50](#)
- llm\_provider\_google\_gemini, [52](#), [54](#), [56](#), [58](#), [60](#), [62](#), [64](#), [65](#)
- llm\_provider\_groq, [52](#), [54](#), [56](#), [58](#), [60](#), [62](#), [64](#), [65](#)
- llm\_provider\_mistral, [52](#), [54](#), [56](#), [57](#), [60](#), [62](#), [64](#), [65](#)
- llm\_provider\_ollama, [52](#), [54](#), [56](#), [58](#), [59](#), [62](#), [64](#), [65](#)
- llm\_provider\_ollama(), [84](#), [85](#)
- llm\_provider\_openai, [52](#), [54](#), [56](#), [58](#), [60](#), [61](#), [64](#), [65](#)
- llm\_provider\_openai(), [85](#)
- llm\_provider\_openrouter, [52](#), [54](#), [56](#), [58](#), [60](#), [62](#), [63](#), [65](#)
- llm\_provider\_xai, [52](#), [54](#), [56](#), [58](#), [60](#), [62](#), [64](#), [65](#)
- llm\_verify, [67](#)
- persistent\_chat-class, [69](#)
- prompt\_wrap, [6–8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [27](#), [30](#), [32](#), [35](#), [48](#), [49](#), [71](#), [77](#), [88](#)
- prompt\_wrap(), [5](#), [7](#), [8](#), [10](#), [16](#), [18](#), [19](#), [21](#), [23](#), [24](#), [26](#), [29](#), [32](#), [35](#), [44](#), [47](#), [49](#), [68](#), [69](#), [73](#), [77](#), [83–85](#), [92](#), [93](#), [100](#)
- quit\_if, [6–8](#), [10](#), [18](#), [20](#), [21](#), [23](#), [24](#), [27](#), [30](#), [32](#), [35](#), [73](#), [76](#), [88](#)
- quit\_if(), [73](#)
- r\_json\_schema\_to\_example, [10](#), [77](#)
- r\_json\_schema\_to\_example(), [9](#)
- r\_session, [29](#)
- r\_session\_options, [29](#)
- send\_prompt, [48](#), [49](#), [83](#)



`send_prompt()`, [4](#), [34](#), [41](#), [44](#), [47–49](#), [51](#), [69](#),  
[72](#), [73](#), [76](#), [77](#), [88](#), [90](#), [92](#), [93](#)  
`set_chat_history`, [39](#), [42](#), [44](#), [46](#), [86](#), [90](#), [94](#)  
`set_system_prompt`, [6–8](#), [10](#), [18](#), [20](#), [21](#), [23](#),  
[24](#), [27](#), [30](#), [32](#), [35](#), [73](#), [77](#), [88](#)  
`skim_with_labels_and_levels`, [40](#), [89](#), [102](#)  
`skim_with_labels_and_levels()`, [29](#)  
`skimr::skim()`, [89](#)

`tidyprompt`, [4](#), [38](#), [39](#), [42–46](#), [67–69](#), [71–73](#),  
[84–86](#), [88](#), [90](#), [90](#), [92–94](#), [100](#)  
`tidyprompt()`, [5–10](#), [16–19](#), [21–24](#), [26](#), [28](#),  
[29](#), [31](#), [32](#), [35](#), [76](#), [77](#), [88](#)  
`tidyprompt-class`, [3](#), [92](#)  
`tools_add_docs`, [35](#), [96](#), [99](#)  
`tools_add_docs()`, [34](#), [35](#), [96](#), [98](#), [99](#)  
`tools_get_docs`, [35](#), [97](#), [98](#)  
`tools_get_docs()`, [35](#), [96](#)

`user_verify`, [100](#)

`vector_list_to_string`, [40](#), [89](#), [101](#)