Mycroft Documentation

Release v21.2.1

Matthew Scholefield

Nov 30, 2021

CONTENTS:

CHAPTER

ONE

MYCROFT SKILLS API

Reference for use during Skill creation

1.1 mycroft package

1.2 mycroft.skills

1.2.1 MycroftSkill class - Base class for all Mycroft skills

class mycroft.**MycroftSkill**(*name=None*, *bus=None*, *use_settings=True*)

Base class for mycroft skills providing common behaviour and parameters to all Skill implementations.

For information on how to get started with creating mycroft skills see https://mycroft.ai/documentation/skills/introduction-developing-skills/

Parameters

- **name** (*str*) – skill name
- **bus** (*MycroftWebsocketClient*) – Optional bus connection
- **use_settings** (*bool*) – Set to false to not use skill settings at all

acknowledge()

Acknowledge a successful request.

This method plays a sound to acknowledge a request that does not require a verbal response. This is intended to provide simple feedback to the user that their request was handled successfully.

add_event(*name*, *handler*, *handler_info=None*, *once=False*)

Create event handler for executing intent or other event.

Parameters

- **name** (*string*) – IntentParser name
- **handler** (*func*) – Method to call
- **handler_info** (*string*) – Base message when reporting skill event handler status on messagebus.
- **once** (*bool, optional*) – Event handler will be removed after it has been run once.

ask_selection(*options*, *dialog=''*, *data=None*, *min_conf=0.65*, *numeric=False*)

Read options, ask dialog question and wait for an answer.

This automatically deals with fuzzy matching and selection by number e.g.

- "first option"
- "last option"
- "second option"
- "option number four"

Parameters

- **options** (*list*) – list of options to present user
- **dialog** (*str*) – a dialog id or string to read AFTER all options
- **data** (*dict*) – Data used to render the dialog
- **min_conf** (*float*) – minimum confidence for fuzzy match, if not reached return None
- **numeric** (*bool*) – speak options as a numeric menu

Returns list element selected by user, or None

Return type string

ask_yesno(*prompt*, *data=None*)

Read prompt and wait for a yes/no answer

This automatically deals with translation and common variants, such as 'yeah', 'sure', etc.

Parameters

- **prompt** (*str*) – a dialog id or string to read
- **data** (*dict*) – response data

Returns

'yes', 'no' or whatever the user response if not one of those, including None

Return type string

bind(*bus*)

Register messagebus emitter with skill.

Parameters **bus** – Mycroft messagebus connection

cancel_all_repeating_events()

Cancel any repeating events started by the skill.

cancel_scheduled_event(*name*)

Cancel a pending event. The event will no longer be scheduled to be executed

Parameters **name** (*str*) – reference name of event (from original scheduling)

config_core

Mycroft global configuration. (dict)

converse(*message=None*)

Handle conversation.

This method gets a peek at utterances before the normal intent handling process after a skill has been invoked once.

To use, override the converse() method and return True to indicate that the utterance has been handled.

utterances and lang are depreciated

Parameters **message** – a message object containing a message type with an optional JSON data packet

Returns True if an utterance was handled, otherwise False

Return type bool

default_shutdown()
Parent function called internally to shut down everything.

Shuts down known entities and calls skill specific shutdown method.

disable_intent(*intent_name*)
Disable a registered intent if it belongs to this skill.

Parameters **intent_name** (*string*) – name of the intent to be disabled

Returns True if disabled, False if it wasn't registered

Return type bool

enable_intent(*intent_name*)
(Re)Enable a registered intent if it belongs to this skill.

Parameters **intent_name** – name of the intent to be enabled

Returns True if enabled, False if it wasn't registered

Return type bool

file_system
Filesystem access to skill specific folder. See mycroft.filesystem for details.

find_resource(*res_name*, *res_dirname=None*)
Find a resource file.

Searches for the given filename using this scheme:

1. Search the resource lang directory:

   <skill>/<res_dirname>/<lang>/<res_name>

2. Search the resource directory:

   <skill>/<res_dirname>/<res_name>

3. Search the locale lang directory or other subdirectory:

   <skill>/locale/<lang>/<res_name> or

   <skill>/locale/<lang>/…/<res_name>

Parameters

- **res_name** (*string*) – The resource name to be found
- **res_dirname** (*string, optional*) – A skill resource directory, such 'dialog', 'vocab', 'regex' or 'ui'. Defaults to None.

Returns The full path to the resource file or None if not found

Return type string

get_intro_message()
Get a message to speak on first load of the skill.

Useful for post-install setup instructions.

Returns message that will be spoken to the user

Return type str

get_response(*dialog=''*, *data=None*, *validator=None*, *on_fail=None*, *num_retries=-1*)
Get response from user.

If a dialog is supplied it is spoken, followed immediately by listening for a user response. If the dialog is omitted listening is started directly.

The response can optionally be validated before returning.

Example:

```
color = self.get_response('ask.favorite.color')
```

Parameters

- **dialog** (*str*) – Optional dialog to speak to the user
- **data** (*dict*) – Data used to render the dialog
- **validator** (*any*) – Function with following signature:

```
def validator(utterance):
    return utterance != "red"
```

- **on_fail** (*any*) – Dialog or function returning literal string to speak on invalid input. For example:

```
def on_fail(utterance):
    return "nobody likes the color red, pick another"
```

- **num_retries** (*int*) – Times to ask user for input, -1 for infinite NOTE: User can not respond and timeout or say "cancel" to stop

Returns User's reply or None if timed out or canceled

Return type str

get_scheduled_event_status(*name*)
Get scheduled event data and return the amount of time left

Parameters **name** (*str*) – reference name of event (from original scheduling)

Returns the time left in seconds

Return type int

Raises **Exception** – Raised if event is not found

handle_disable_intent(*message*)
Listener to disable a registered intent if it belongs to this skill.

handle_enable_intent(*message*)
Listener to enable a registered intent if it belongs to this skill.

handle_remove_cross_context(*message*)
Remove global context from intent service.

handle_set_cross_context(*message*)
Add global context to intent service.

handle_settings_change(*message*)
Update settings if the remote settings changes apply to this skill.

The skill settings downloader uses a single API call to retrieve the settings for all skills. This is done to limit the number API calls. A "mycroft.skills.settings.changed" event is emitted for each skill that had their settings changed. Only update this skill's settings if its remote settings were among those changed

initialize()
Perform any final setup needed for the skill.

Invoked after the skill is fully constructed and registered with the system.

property lang
Get the configured language.

load_data_files(*root_directory=None*)
Called by the skill loader to load intents, dialogs, etc.

Parameters **root_directory** (*str*) – root folder to use when loading files.

load_regex_files(*root_directory*)
Load regex files found under the skill directory.

Parameters **root_directory** (*str*) – root folder to use when loading files

load_vocab_files(*root_directory*)
Load vocab files found under root_directory.

Parameters **root_directory** (*str*) – root folder to use when loading files

property location
Get the JSON data struction holding location information.

property location_pretty
Get a more 'human' version of the location as a string.

property location_timezone
Get the timezone code, such as 'America/Los_Angeles'

log
Skill logger instance

make_active()
Bump skill to active_skill list in intent_service.

This enables converse method to be called even without skill being used in last 5 minutes.

register_entity_file(*entity_file*)
Register an Entity file with the intent service.

An Entity file lists the exact values that an entity can hold. For example:

=== ask.day.intent === Is it {weekend}?

=== weekend.entity === Saturday Sunday

Parameters **entity_file** (*string*) – name of file that contains examples of an entity. Must end with '.entity'

register_intent(*intent_parser*, *handler*)
Register an Intent with the intent service.

Parameters

- **intent_parser** – Intent, IntentBuilder object or padatious intent file to parse utterance for the handler.

- **handler** (*func*) – function to register with intent

register_intent_file(*intent_file*, *handler*)
Register an Intent file with the intent service.

For example:

=== food.order.intent === Order some {food}. Order some {food} from {place}. I'm hungry. Grab some {food} from {place}.

Optionally, you can also use <register_entity_file> to specify some examples of {food} and {place}

In addition, instead of writing out multiple variations of the same sentence you can write:

=== food.order.intent === (Order | Grab) some {food} (from {place} |). I'm hungry.

Parameters

- **intent_file** – name of file that contains example queries that should activate the intent. Must end with '.intent'
- **handler** – function to register with intent

register_regex(*regex_str*)
Register a new regex. :param regex_str: Regex string

register_resting_screen()
Registers resting screen from the resting_screen_handler decorator.

This only allows one screen and if two is registered only one will be used.

register_vocabulary(*entity*, *entity_type*)
Register a word to a keyword

Parameters

- **entity** – word to register
- **entity_type** – Intent handler entity to tie the word to

reload_skill
allow reloading (default True)

remove_context(*context*)
Remove a keyword from the context manager.

remove_cross_skill_context(*context*)
Tell all skills to remove a keyword from the context manager.

remove_event(*name*)
Removes an event from bus emitter and events list.

Parameters **name** (*string*) – Name of Intent or Scheduler Event

Returns True if found and removed, False if not found

Return type bool

report_metric(*name*, *data*)
Report a skill metric to the Mycroft servers.

Parameters

- **name** (*str*) – Name of metric. Must use only letters and hyphens
- **data** (*dict*) – JSON dictionary to report. Must be valid JSON

root_dir
Member variable containing the absolute path of the skill's root directory. E.g. /opt/mycroft/skills/my-skill.me/

schedule_event(*handler*, *when*, *data=None*, *name=None*, *context=None*)
Schedule a single-shot event.

Parameters

- **handler** – method to be called
- **when** (*datetime/int/float*) – datetime (in system timezone) or number of seconds in the future when the handler should be called
- **data** (*dict, optional*) – data to send when the handler is called
- **name** (*str, optional*) – reference name NOTE: This will not warn or replace a previously scheduled event of the same name.
- **context** (*dict, optional*) – context (dict, optional): message context to send when the handler is called

schedule_repeating_event(*handler*, *when*, *frequency*, *data=None*, *name=None*, *context=None*)
Schedule a repeating event.

Parameters

- **handler** – method to be called
- **when** (*datetime*) – time (in system timezone) for first calling the handler, or None to initially trigger <frequency> seconds from now
- **frequency** (*float/int*) – time in seconds between calls
- **data** (*dict, optional*) – data to send when the handler is called
- **name** (*str, optional*) – reference name, must be unique
- **context** (*dict, optional*) – context (dict, optional): message context to send when the handler is called

send_email(*title*, *body*)
Send an email to the registered user's email.

Parameters

- **title** (*str*) – Title of email
- **body** (*str*) – HTML body of email. This supports simple HTML like bold and italics

set_context(*context*, *word=''*, *origin=''*)
Add context to intent service

Parameters

- **context** – Keyword
- **word** – word connected to keyword
- **origin** – origin of context

set_cross_skill_context(*context*, *word=''*)
Tell all skills to add a context to intent service

Parameters

- **context** – Keyword

- **word** – word connected to keyword

settings_change_callback
Set to register a callback method that will be called every time the skills settings are updated. The referenced method should include any logic needed to handle the updated settings.

shutdown()
Optional shutdown proceedure implemented by subclass.

This method is intended to be called during the skill process termination. The skill implementation must shutdown all processes and operations in execution.

speak(*utterance*, *expect_response=False*, *wait=False*, *meta=None*)
Speak a sentence.

Parameters

- **utterance** (*str*) – sentence mycroft should speak
- **expect_response** (*bool*) – set to True if Mycroft should listen for a response immediately after speaking the utterance.
- **wait** (*bool*) – set to True to block while the text is being spoken.
- **meta** – Information of what built the sentence.

speak_dialog(*key*, *data=None*, *expect_response=False*, *wait=False*)
Speak a random sentence from a dialog file.

Parameters

- **key** (*str*) – dialog file key (e.g. "hello" to speak from the file "locale/en-us/hello.dialog")
- **data** (*dict*) – information used to populate sentence
- **expect_response** (*bool*) – set to True if Mycroft should listen for a response immediately after speaking the utterance.
- **wait** (*bool*) – set to True to block while the text is being spoken.

stop()
Optional method implemented by subclass.

translate(*text*, *data=None*)
Load a translatable single string resource

The string is loaded from a file in the skill's dialog subdirectory 'dialog/<lang>/<text>.dialog'

The string is randomly chosen from the file and rendered, replacing mustache placeholders with values found in the data dictionary.

Parameters

- **text** (*str*) – The base filename (no extension needed)
- **data** (*dict, optional*) – a JSON dictionary

Returns A randomly chosen string from the file

Return type str

translate_list(*list_name*, *data=None*)
Load a list of translatable string resources

The strings are loaded from a list file in the skill's dialog subdirectory. 'dialog/<lang>/<list_name>.list'

The strings are loaded and rendered, replacing mustache placeholders with values found in the data dictionary.

Parameters

- **list_name** (*str*) – The base filename (no extension needed)
- **data** (*dict, optional*) – a JSON dictionary

Returns

The loaded list of strings with items in consistent positions regardless of the language.

Return type list of str

translate_namedvalues(*name*, *delim=','*)

Load translation dict containing names and values.

This loads a simple CSV from the 'dialog' folders. The name is the first list item, the value is the second. Lines prefixed with # or // get ignored

Parameters

- **name** (*str*) – name of the .value file, no extension needed
- **delim** (*char*) – delimiter character used, default is ','

Returns name and value dictionary, or empty dict if load fails

Return type dict

translate_template(*template_name*, *data=None*)

Load a translatable template.

The strings are loaded from a template file in the skill's dialog subdirectory. 'dialog/<lang>/<template_name>.template'

The strings are loaded and rendered, replacing mustache placeholders with values found in the data dictionary.

Parameters

- **template_name** (*str*) – The base filename (no extension needed)
- **data** (*dict, optional*) – a JSON dictionary

Returns The loaded template file

Return type list of str

update_scheduled_event(*name*, *data=None*)

Change data of event.

Parameters

- **name** (*str*) – reference name of event (from original scheduling)
- **data** (*dict*) – event data

voc_match(*utt*, *voc_filename*, *lang=None*, *exact=False*)

Determine if the given utterance contains the vocabulary provided.

By default the method checks if the utterance contains the given vocab thereby allowing the user to say things like "yes, please" and still match against "Yes.voc" containing only "yes". An exact match can be requested.

The method first checks in the current Skill's .voc files and secondly in the "res/text" folder of mycroft-core. The result is cached to avoid hitting the disk each time the method is called.

Parameters

- **utt** (*str*) – Utterance to be tested
- **voc_filename** (*str*) – Name of vocabulary file (e.g. 'yes' for 'res/text/en-us/yes.voc')
- **lang** (*str*) – Language code, defaults to self.long
- **exact** (*bool*) – Whether the vocab must exactly match the utterance

Returns True if the utterance has the given vocabulary it

Return type bool

1.2.2 CommonIoTSkill class

class mycroft.skills.common_iot_skill.**CommonIoTSkill**(*name=None*, *bus=None*, *use_settings=True*)

Bases: *mycroft.skills.mycroft_skill.mycroft_skill.MycroftSkill*, abc.ABC

Skills that want to work with the CommonIoT system should extend this class. Subclasses will be expected to implement two methods, *can_handle* and *run_request*. See the documentation for those functions for more details on how they are expected to behave.

Subclasses may also register their own entities and scenes. See the register_entities and register_scenes methods for details.

This class works in conjunction with a controller skill. The controller registers vocabulary and intents to capture IoT related requests. It then emits messages on the messagebus that will be picked up by all skills that extend this class. Each skill will have the opportunity to declare whether or not it can handle the given request. Skills that acknowledge that they are capable of handling the request will be considered candidates, and after a short timeout, a winner, or winners, will be chosen. With this setup, a user can have several IoT systems, and control them all without worry that skills will step on each other.

bind(*bus*)

Overrides MycroftSkill.bind.

This is called automatically during setup, and need not otherwise be used.

Subclasses that override this method must call this via super in their implementation.

Parameters **bus** –

abstract can_handle(*request: mycroft.skills.common_iot_skill.IoTRequest*)

Determine if an IoTRequest can be handled by this skill.

This method must be implemented by all subclasses.

An IoTRequest contains several properties (see the documentation for that class). This method should return True if and only if this skill can take the appropriate 'action' when considering all other properties of the request. In other words, a partial match, one in which any piece of the IoTRequest is not known to this skill, and is not None, this should return (False, None).

Parameters **request** – IoTRequest

Returns: (boolean, dict) True if and only if this skill knows about all the properties set on the IoTRequest, and a dict containing callback_data. If this skill is chosen to handle the request, this dict will be supplied to *run_request*.

Note that the dictionary will be sent over the bus, and thus must be JSON serializable.

get_entities()
Get a list of custom entities.

This is intended to be overridden by subclasses, though it it not required (the default implementation will return an empty list).

The strings returned by this function will be registered as ENTITY values with the intent parser. Skills should provide group names, user aliases for specific devices, or anything else that might represent a THING or a set of THINGs, e.g. 'bedroom', 'lamp', 'front door.' This allows commands that don't explicitly include a THING to still be handled, e.g. "bedroom off" as opposed to "bedroom lights off."

get_scenes()
Get a list of custom scenes.

This method is intended to be overridden by subclasses, though it is not required. The strings returned by this function will be registered as SCENE values with the intent parser. Skills should provide user defined scene names that they are aware of and capable of handling, e.g. "relax," "movie time," etc.

register_entities_and_scenes()
This method will register this skill's scenes and entities.

This should be called in the skill's *initialize* method, at some point after *get_entities* and *get_scenes* can be expected to return correct results.

abstract run_request(*request: mycroft.skills.common_iot_skill.IoTRequest*, *callback_data: dict*)
Handle an IoT Request.

All subclasses must implement this method.

When this skill is chosen as a winner, this function will be called. It will be passed an IoTRequest equivalent to the one that was supplied to *can_handle*, as well as the *callback_data* returned by *can_handle*.

Parameters

- **request** – IoTRequest
- **callback_data** – dict

speak(*utterance*, **args*, ***kwargs*)
Speak a sentence.

Parameters

- **utterance** (*str*) – sentence mycroft should speak
- **expect_response** (*bool*) – set to True if Mycroft should listen for a response immediately after speaking the utterance.
- **wait** (*bool*) – set to True to block while the text is being spoken.
- **meta** – Information of what built the sentence.

property supported_request_version:
mycroft.skills.common_iot_skill.IoTRequestVersion
Get the supported IoTRequestVersion

By default, this returns IoTRequestVersion.V1. Subclasses should override this to indicate higher levels of support.

The documentation for IoTRequestVersion provides a reference indicating which fields are included in each version. Note that you should always take the latest, and account for all request fields.

1.2.3 CommonPlaySkill class

class mycroft.skills.common_play_skill.**CommonPlaySkill**(*name=None*, *bus=None*)
Bases: *mycroft.skills.mycroft_skill.mycroft_skill.MycroftSkill*, abc.ABC

To integrate with the common play infrastructure of Mycroft skills should use this base class and override the two methods *CPS_match_query_phrase* (for checking if the skill can play the utterance) and *CPS_start* for launching the media.

The class makes the skill available to queries from the mycroft-playback-control skill and no special vocab for starting playback is needed.

CPS_extend_timeout(*timeout=5*)
Request Common Play Framework to wait another {timeout} seconds for an answer from this skill.

Parameters **timeout** (*int*) – Number of seconds

abstract CPS_match_query_phrase(*phrase*)
Analyze phrase to see if it is a play-able phrase with this skill.

Parameters **phrase** (*str*) – User phrase uttered after "Play", e.g. "some music"

Returns

Tuple containing a string with the appropriate matching phrase, the PlayMatch type, and optionally data to return in the callback if the match is selected.

Return type (match, CPSMatchLevel[, callback_data]) or None

CPS_play(**args*, ***kwargs*)
Begin playback of a media file or stream

Normally this method will be invoked with somthing like: self.CPS_play(url)

Advanced use can also include keyword arguments, such as: self.CPS_play(url, repeat=True)

Parameters **method** (*same as the Audioservice.play*) –

CPS_send_status(*artist=''*, *track=''*, *album=''*, *image=''*, *uri=''*, *track_length=None*, *elapsed_time=None*, *playlist_position=None*, *status=CPSTrackStatus.DISAMBIGUATION*, ***kwargs*)
Inform system of playback status.

If a skill is handling playback and wants the playback control to be aware of it's current status it can emit this message indicating that it's performing playback and can provide some standard info.

All parameters are optional so any can be left out. Also if extra non-standard parameters are added, they too will be sent in the message data.

Parameters

- **artist** (*str*) – Current track artist
- **track** (*str*) – Track name
- **album** (*str*) – Album title
- **image** (*str*) – url for image to show
- **uri** (*str*) – uri for track
- **track_length** (*float*) – track length in seconds
- **elapsed_time** (*float*) – current offset into track in seconds
- **playlist_position** (*int*) – Position in playlist of current track

CPS_send_tracklist(*tracklist*)
Inform system of playlist track info.

Provides track data for playlist

Parameters **tracklist** (*list/dict*) – Tracklist data

abstract CPS_start(*phrase*, *data*)
Begin playing whatever is specified in 'phrase'

Parameters

- **phrase** (*str*) – User phrase uttered after "Play", e.g. "some music"
- **data** (*dict*) – Callback data specified in match_query_phrase()

bind(*bus*)
Overrides the normal bind method.

Adds handlers for play:query and play:start messages allowing interaction with the playback control skill.

This is called automatically during setup, and need not otherwise be used.

stop()
Stop anything playing on the audioservice.

1.2.4 CommonQuerySkill class

class mycroft.skills.common_query_skill.**CommonQuerySkill**(*name=None*, *bus=None*)
Bases: *mycroft.skills.mycroft_skill.mycroft_skill.MycroftSkill*, abc.ABC

Question answering skills should be based on this class.

The skill author needs to implement *CQS_match_query_phrase* returning an answer and can optionally implement *CQS_action* to perform additional actions if the skill's answer is selected.

This class works in conjunction with skill-query which collects answers from several skills presenting the best one available.

CQS_action(*phrase*, *data*)
Take additional action IF the skill is selected.

The speech is handled by the common query but if the chosen skill wants to display media, set a context or prepare for sending information info over e-mail this can be implemented here.

Parameters

- **phrase** (*str*) – User phrase uttered after "Play", e.g. "some music"
- **data** (*dict*) – Callback data specified in match_query_phrase()

abstract CQS_match_query_phrase(*phrase*)
Analyze phrase to see if it is a play-able phrase with this skill.

Needs to be implemented by the skill.

Parameters **phrase** (*str*) – User phrase, "What is an aardwark"

Returns

Tuple containing a string with the appropriate matching phrase, the PlayMatch type, and optionally data to return in the callback if the match is selected.

Return type (match, CQSMatchLevel[, callback_data]) or None

bind(*bus*)
Overrides the default bind method of MycroftSkill.

This registers messagebus handlers for the skill during startup but is nothing the skill author needs to consider.

remove_noise(*phrase*)
remove noise to produce essence of question

1.2.5 FallbackSkill class

class mycroft.**FallbackSkill**(*name=None*, *bus=None*, *use_settings=True*)
Bases: *mycroft.skills.mycroft_skill.mycroft_skill.MycroftSkill*

Fallbacks come into play when no skill matches an Adapt or closely with a Padatious intent. All Fallback skills work together to give them a view of the user's utterance. Fallback handlers are called in an order determined the priority provided when the the handler is registered.

Priority	Who?	Purpose
1-4	RESERVED	Unused for now, slot for pre-Padatious if needed
5	MYCROFT	Padatious near match (conf > 0.8)
6-88	USER	General
89	MYCROFT	Padatious loose match (conf > 0.5)
90-99	USER	Uncaught intents
100+	MYCROFT	Fallback Unknown or other future use

Handlers with the numerically lowest priority are invoked first. Multiple fallbacks can exist at the same priority, but no order is guaranteed.

A Fallback can either observe or consume an utterance. A consumed utterance will not be see by any other Fallback handlers.

default_shutdown()
Remove all registered handlers and perform skill shutdown.

classmethod make_intent_failure_handler(*bus*)
Goes through all fallback handlers until one returns True

register_fallback(*handler*, *priority*)
Register a fallback with the list of fallback handlers and with the list of handlers registered by this instance

classmethod remove_fallback(*handler_to_del*)
Remove a fallback handler.

Parameters **handler_to_del** – reference to handler

Returns (bool) True if at least one handler was removed, otherwise False

remove_instance_handlers()
Remove all fallback handlers registered by the fallback skill.

1.2.6 AudioService class

class mycroft.skills.audioservice.**AudioService**(*bus*)
Bases: object

AudioService class for interacting with the audio subsystem

Parameters **bus** – Mycroft messagebus connection

available_backends()
Return available audio backends.

Returns dict with backend names as keys

property is_playing
True if the audioservice is playing, else False.

next()
Change to next track.

pause()
Pause playback.

play(*tracks=None*, *utterance=None*, *repeat=None*)
Start playback.

Parameters

- **tracks** – track uri or list of track uri's Each track can be added as a tuple with (uri, mime) to give a hint of the mime type to the system
- **utterance** – forward utterance for further processing by the audio service.
- **repeat** – if the playback should be looped

prev()
Change to previous track.

queue(*tracks=None*)
Queue up a track to playing playlist.

Parameters **tracks** – track uri or list of track uri's Each track can be added as a tuple with (uri, mime) to give a hint of the mime type to the system

resume()
Resume paused playback.

seek(*seconds=1*)
Seek X seconds.

Parameters **seconds** (*int*) – number of seconds to seek, if negative rewind

seek_backward(*seconds=1*)
Rewind X seconds

Parameters **seconds** (*int*) – number of seconds to rewind

seek_forward(*seconds=1*)
Skip ahead X seconds.

Parameters **seconds** (*int*) – number of seconds to skip

stop()
Stop the track.

track_info()
Request information of current playing track.

Returns Dict with track info.

1.2.7 intent_handler decorator

mycroft.**intent_handler**(*intent_parser*)
Decorator for adding a method as an intent handler.

1.2.8 intent_file_handler decorator

mycroft.**intent_file_handler**(*intent_file*)
Decorator for adding a method as an intent file handler.

This decorator is deprecated, use intent_handler for the same effect.

1.2.9 adds_context decorator

mycroft.**adds_context**(*context*, *words=''*)
Decorator adding context to the Adapt context manager.

Parameters

- **context** (*str*) – context Keyword to insert
- **words** (*str*) – optional string content of Keyword

1.2.10 removes_context decorator

mycroft.**removes_context**(*context*)
Decorator removing context from the Adapt context manager.

Parameters **context** (*str*) – Context keyword to remove

1.3 mycroft.audio

1.3.1 mycroft.audio

mycroft.audio.**is_speaking**()
Determine if Text to Speech is occurring

Returns True while still speaking

Return type bool

mycroft.audio.**stop_speaking**()
Stop mycroft speech.

TODO: Skills should only be able to stop speech they've initiated

mycroft.audio.**wait_while_speaking**()
Pause as long as Text to Speech is still happening

Pause while Text to Speech is still happening. This always pauses briefly to ensure that any preceeding request to speak has time to begin.

1.4 mycroft.filesystem

1.4.1 mycroft.filesystem

class mycroft.filesystem.**FileSystemAccess**(*path*)
A class for providing access to the mycroft FS sandbox.

Intended to be attached to skills at initialization time to provide a skill-specific namespace.

exists(*filename*)
Check if file exists in the namespace.

Parameters **filename** (*str*) – a path relative to the namespace. subdirs not currently supported.

Returns True if file exists, else False.

Return type bool

open(*filename*, *mode*)
Open a file in the provided namespace.

Get a handle to a file (with the provided mode) within the skill-specific namespace.

Parameters

- **filename** (*str*) – a path relative to the namespace. subdirs not currently supported.
- **mode** (*str*) – a file handle mode

Returns an open file handle.

path
Member value containing the root path of the namespace

1.5 mycroft.util

1.5.1 mycroft.util package

The mycroft.util package includes functions for common operations such as playing audio files, parsting and creating natural text as well as many components used internally in Mycroft such as cache directory lookup, path resolution. etc.

Below _some_ of the functions that are of interest to skill developers are listed.

1.5.1.1 LOG

mycroft.util.**LOG**(*name*)

Custom logger class that acts like logging.Logger The logger name is automatically generated by the module of the caller

Usage:

```
>>> LOG.debug('My message: %s', debug_str)
13:12:43.673 - :<module>:1 - DEBUG - My message: hi
>>> LOG('custom_name').debug('Another message')
13:13:10.462 - custom_name - DEBUG - Another message
```

1.5.1.2 play_wav

mycroft.util.**play_wav**(*uri*, *environment=None*)

Play a wav-file.

This will use the application specified in the mycroft config and play the uri passed as argument. The function will return directly and play the file in the background.

Parameters

- **uri** – uri to play
- **environment** (*dict*) – optional environment for the subprocess call

Returns: subprocess.Popen object or None if operation failed

1.5.1.3 play_mp3

mycroft.util.**play_mp3**(*uri*, *environment=None*)

Play a mp3-file.

This will use the application specified in the mycroft config and play the uri passed as argument. The function will return directly and play the file in the background.

Parameters

- **uri** – uri to play
- **environment** (*dict*) – optional environment for the subprocess call

Returns: subprocess.Popen object or None if operation failed

1.5.1.4 play_ogg

mycroft.util.**play_ogg**(*uri*, *environment=None*)

Play an ogg-file.

This will use the application specified in the mycroft config and play the uri passed as argument. The function will return directly and play the file in the background.

Parameters

- **uri** – uri to play
- **environment** (*dict*) – optional environment for the subprocess call

Returns: subprocess.Popen object, or None if operation failed

1.5.1.5 resolve_resource_file

mycroft.util.**resolve_resource_file**(*res_name*)

Convert a resource into an absolute filename.

Resource names are in the form: 'filename.ext' or 'path/filename.ext'

The system wil look for $XDG_DATA_DIRS/mycroft/res_name first (defaults to ~/.local/share/mycroft/res_name), and if not found will look at /opt/mycroft/res_name, then finally it will look for res_name in the 'mycroft/res' folder of the source code package.

Example

With mycroft running as the user 'bob', if you called `resolve_resource_file('snd/beep.wav')` it would return either: '$XDG_DATA_DIRS/mycroft/beep.wav', '/home/bob/.mycroft/snd/beep.wav' or '/opt/mycroft/snd/beep.wav' or '.../mycroft/res/snd/beep.wav' where the '...' is replaced by the path where the package has been installed.

Parameters **res_name** (*str*) – a resource path/name

Returns (str) path to resource or None if no resource found

1.5.1.6 get_cache_directory

mycroft.util.**get_cache_directory**(*domain=None*)

Get a directory for caching data.

This directory can be used to hold temporary caches of data to speed up performance. This directory will likely be part of a small RAM disk and may be cleared at any time. So code that uses these cached files must be able to fallback and regenerate the file.

Parameters **domain** (*str*) – The cache domain. Basically just a subdirectory.

Returns (str) a path to the directory where you can cache data

1.5.2 mycroft.util.log

Mycroft Logging module.

This module provides the LOG pseudo function quickly creating a logger instance for use.

The default log level of the logger created here can ONLY be set in /etc/mycroft/mycroft.conf or ~/.config/mycroft/mycroft.conf

The default log level can also be programatically be changed by setting the LOG.level parameter.

class mycroft.util.log.**LOG**(*name*)

Custom logger class that acts like logging.Logger The logger name is automatically generated by the module of the caller

Usage:

```
>>> LOG.debug('My message: %s', debug_str)
13:12:43.673 - :<module>:1 - DEBUG - My message: hi
>>> LOG('custom_name').debug('Another message')
13:13:10.462 - custom_name - DEBUG - Another message
```

classmethod debug(**args*, ***kwargs*)
Log 'msg % args' with severity 'DEBUG'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.debug("Houston, we have a %s", "thorny problem", exc_info=1)

classmethod error(**args*, ***kwargs*)
Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.error("Houston, we have a %s", "major problem", exc_info=1)

classmethod exception(**args*, ***kwargs*)
Convenience method for logging an ERROR with exception information.

classmethod info(**args*, ***kwargs*)
Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.info("Houston, we have a %s", "interesting problem", exc_info=1)

classmethod init()
Initializes the class, sets the default log level and creates the required handlers.

classmethod warning(**args*, ***kwargs*)
Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.warning("Houston, we have a %s", "bit of a problem", exc_info=1)

mycroft.util.log.**getLogger**(*name='MYCROFT'*)
Depreciated. Use LOG instead

1.5.3 mycroft.util.parse

The mycroft.util.parse module provides various parsing functions for things like numbers, times, durations etc. It's intention is to convert naturally expressed concepts into standard computer readable formats. Doing this also enables localization.

It also provides some useful associated functions like basic fuzzy matching.

The module uses lingua-franca (https://github.com/mycroftai/lingua-franca) to do most of the actual parsing. However methods may be wrapped specifically for use in Mycroft Skills.

mycroft.util.parse.**extract_datetime**(*text*, *anchorDate='DEFAULT'*, *lang=None*, *default_time=None*)
Extracts date and time information from a sentence.

Parses many of the common ways that humans express dates and times, including relative dates like "5 days from today", "tomorrow', and "Tuesday".

Vague terminology are given arbitrary values, like:

- morning = 8 AM
- afternoon = 3 PM
- evening = 7 PM

If a time isn't supplied or implied, the function defaults to 12 AM

Parameters

- **text** (*str*) – the text to be interpreted
- **anchorDate** (datetime, optional) – the date to be used for relative dating (for example, what does "tomorrow" mean?). Defaults to the current local date/time.
- **lang** (*str*) – the BCP-47 code for the language to use, None uses default
- **default_time** (*datetime.time*) – time to use if none was found in the input string.

Returns

'datetime' is the extracted date as a datetime object in the user's local timezone. 'left-over_string' is the original phrase with all date and time related keywords stripped out. See examples for further clarification Returns 'None' if no date or time related text is found.

Return type [datetime, str]

Examples

```
>>> extract_datetime(
... "What is the weather like the day after tomorrow?",
... datetime(2017, 06, 30, 00, 00)
... )
[datetime.datetime(2017, 7, 2, 0, 0), 'what is weather like']
>>> extract_datetime(
... "Set up an appointment 2 weeks from Sunday at 5 pm",
... datetime(2016, 02, 19, 00, 00)
... )
[datetime.datetime(2016, 3, 6, 17, 0), 'set up appointment']
>>> extract_datetime(
... "Set up an appointment",
... datetime(2016, 02, 19, 00, 00)
... )
None
```

Parsing functions for extracting data from natural speech.

1.5.4 mycroft.util.format

The mycroft.util.format module provides various formatting functions for things like numbers, times, etc.

The focus of these formatting functions is to create human digestible content either as speech or in display form. It is also enables localization.

The module uses lingua-franca (https://github.com/mycroftai/lingua-franca) to do most of the actual parsing. However methods may be wrapped specifically for use in Mycroft Skills.

mycroft.util.format.**nice_duration**(*duration*, *lang=None*, *speech=True*, *use_years=True*, *clock=False*, *resolution=TimeResolution.SECONDS*)

Convert duration in seconds to a nice spoken timespan

Accepts: time, in seconds, or datetime.timedelta

Examples

duration = 60 -> "1:00" or "one minute" duration = 163 -> "2:43" or "two minutes forty three seconds" duration = timedelta(seconds=120) -> "2:00" or "two minutes"

Parameters

- **duration** (*int/float/datetime.timedelta*) –
- **lang** (*str, optional*) – a BCP-47 language code, None for default
- **speech** (*bool, opt*) – format output for speech (True) or display (False)
- **use_years** (*bool, opt*) – rtn years and days if True, total days if False
- **clock** (*bool, opt*) – always format output like digital clock (see below)
- **resolution** (*mycroft.util.format.TimeResolution, optional*) – lower bound

  mycroft.util.format.TimeResolution values: TimeResolution.YEARS TimeResolution.DAYS TimeResolution.HOURS TimeResolution.MINUTES TimeResolution.SECONDS TimeResolution.MILLISECONDS

  NOTE: nice_duration will not produce milliseconds unless that resolution is passed.

  NOTE: clock will produce digital clock-like output appropriate to resolution. Has no effect on resolutions DAYS or YEARS. Only applies to displayed output.

Returns timespan as a string

Return type str

mycroft.util.format.**nice_duration_dt**(*date1*, *date2*, *lang=None*, *speech=True*, *use_years=True*, *clock=False*, *resolution=TimeResolution.SECONDS*)

Convert duration between datetimes to a nice spoken timespan

Accepts: 2 x datetime.datetime

Examples

date1 = datetime(2019, 3, 12), date2 = datetime(2019, 1, 1) -> "seventy days"

date1 = datetime(2019, 12, 25, 20, 30), date2 = datetime(2019, 10, 31, 8, 00), speech = False -> "55d 12:30"

Parameters

- **date1** (*datetime.datetime*) –
- **date2** (*datetime.datetime*) –
- **lang** (*str, optional*) – a BCP-47 language code, None for default
- **speech** (*bool, opt*) – format output for speech (True) or display (False)
- **use_years** (*bool, opt*) – rtn years and days if True, total days if False
- **clock** (*bool, opt*) – always format output like digital clock (see below)
- **resolution** (*mycroft.util.format.TimeResolution, optional*) – lower bound

  mycroft.util.format.TimeResolution values: TimeResolution.YEARS TimeResolution.DAYS TimeResolution.HOURS TimeResolution.MINUTES TimeResolution.SECONDS

NOTE: nice_duration_dt() cannot do TimeResolution.MILLISECONDS This will silently fall back on TimeResolution.SECONDS

NOTE: clock will produce digital clock-like output appropriate to resolution. Has no effect on resolutions DAYS or YEARS. Only applies to displayed output.

Returns timespan as a string

Return type str

Formatting functions for producing natural speech from common datatypes such as numbers, dates and times.

1.5.5 mycroft.util.time

Time utils for getting and converting datetime objects for the Mycroft system. This time is based on the setting in the Mycroft config and may or may not match the system locale.

mycroft.util.time.**default_timezone**()
Get the default timezone

Based on user location settings location.timezone.code or the default system value if no setting exists.

Returns Definition of the default timezone

Return type (datetime.tzinfo)

mycroft.util.time.**now_local**(*tz=None*)
Retrieve the current time

Parameters **tz** (*datetime.tzinfo, optional*) – Timezone, default to user's settings

Returns The current time

Return type (datetime)

mycroft.util.time.**now_utc**()
Retrieve the current time in UTC

Returns The current time in Universal Time, aka GMT

Return type (datetime)

mycroft.util.time.**to_local**(*dt*)
Convert a datetime to the user's local timezone

Parameters **dt** (*datetime*) – A datetime (if no timezone, defaults to UTC)

Returns time converted to the local timezone

Return type (datetime)

mycroft.util.time.**to_system**(*dt*)
Convert a datetime to the system's local timezone

Parameters **dt** (*datetime*) – A datetime (if no timezone, assumed to be UTC)

Returns time converted to the operation system's timezone

Return type (datetime)

mycroft.util.time.**to_utc**(*dt*)
Convert a datetime with timezone info to a UTC datetime

Parameters **dt** (*datetime*) – A datetime (presumably in some local zone)

Returns time converted to UTC

Return type (datetime)

A collection of functions for handling local, system and global times.

CHAPTER

TWO

MYCROFT PLUGIN API

Reference for use during Plugin creation

2.1 Mycroft plugins

Mycroft is extendable by plugins. These plugins can add support for new Speech To Text engines, Text To Speech engines, wake word engines and add new audio playback options.

2.1.1 TTS - Base for TTS plugins

class mycroft.tts.**TTS**(*lang*, *config*, *validator*, *audio_ext='wav'*, *phonetic_spelling=True*, *ssml_tags=None*)

TTS abstract class to be implemented by all TTS engines.

It aggregates the minimum required parameters and exposes `execute(sentence)` and `validate_ssml(sentence)` functions.

Parameters

- **lang** (*str*) –
- **config** (*dict*) – Configuration for this specific tts engine
- **validator** (*TTSValidator*) – Used to verify proper installation
- **phonetic_spelling** (*bool*) – Whether to spell certain words phonetically
- **ssml_tags** (*list*) – Supported ssml properties. Ex. [‘speak’, ‘prosody’]

begin_audio()

Helper function for child classes to call in execute().

clear_cache()

Remove all cached files.

end_audio(*listen=False*)

Helper function for child classes to call in execute().

Sends the recognizer_loop:audio_output_end message (indicating that speaking is done for the moment) as well as trigger listening if it has been requested. It also checks if cache directory needs cleaning to free up disk space.

Parameters **listen** (*bool*) – indication if listening trigger should be sent.

execute(*sentence*, *ident=None*, *listen=False*)

Convert sentence to speech, preprocessing out unsupported ssml

The method caches results if possible using the hash of the sentence.

Parameters

- **sentence** – (str) Sentence to be spoken
- **ident** – (str) Id reference to current interaction
- **listen** – (bool) True if listen should be triggered at the end of the utterance.

get_tts(*sentence*, *wav_file*)
Abstract method that a tts implementation needs to implement.

Should get data from tts.

Parameters

- **sentence** (*str*) – Sentence to synthesize
- **wav_file** (*str*) – output file

Returns (wav_file, phoneme)

Return type tuple

init(*bus*)
Performs intial setup of TTS object.

Parameters **bus** – Mycroft messagebus connection

load_phonemes(*key*)
Load phonemes from cache file.

Parameters **key** (*str*) – Key identifying phoneme cache

load_spellings()
Load phonetic spellings of words as dictionary.

modify_tag(*tag*)
Override to modify each supported ssml tag.

Parameters **tag** (*str*) – SSML tag to check and possibly transform.

static remove_ssml(*text*)
Removes SSML tags from a string.

Parameters **text** (*str*) – input string

Returns input string stripped from tags.

Return type str

save_phonemes(*key*, *phonemes*)
Cache phonemes

Parameters

- **key** (*str*) – Hash key for the sentence
- **phonemes** (*str*) – phoneme string to save

validate_ssml(*utterance*)
Check if engine supports ssml, if not remove all tags.

Remove unsupported / invalid tags

Parameters **utterance** (*str*) – Sentence to validate

Returns validated_sentence

Return type str

viseme(*phonemes*)
Create visemes from phonemes.

May be implemented to convert TTS phonemes into Mycroft mouth visuals.

Parameters **phonemes** (*str*) – String with phoneme data

Returns visemes

Return type list

2.1.2 STT - base for STT plugins

class mycroft.stt.**STT**
STT Base class, all STT backends derive from this one.

abstract execute(*audio*, *language=None*)
Implementation of STT functionallity.

This method needs to be implemented by the derived class to implement the specific STT engine connection.

The method gets passed audio and optionally a language code and is expected to return a text string.

Parameters

- **audio** (*AudioData*) – audio recorded by mycroft.
- **language** (*str*) – optional language code

Returns parsed text

Return type str

static init_language(*config_core*)
Helper method to get language code from Mycroft config.

class mycroft.stt.**StreamingSTT**
ABC class for threaded streaming STT implemenations.

abstract create_streaming_thread()
Create thread for parsing audio chunks.

This method should be implemented by the derived class to return an instance derived from StreamThread to handle the audio stream and send it to the STT engine.

Returns Thread to handle audio data.

Return type *StreamThread*

execute(*audio*, *language=None*)
End the parsing thread and collect data.

stream_data(*data*)
Receiver of audio data.

Parameters **data** (*bytes*) – raw audio data.

stream_start(*language=None*)
Indicate start of new audio stream.

This creates a new thread for handling the incomming audio stream as it's collected by Mycroft.

Parameters **language** (*str*) – optional language code for the new stream.

stream_stop()
Indicate that the audio stream has ended.

This will tear down the processing thread and collect the result

Returns parsed text

Return type str

class mycroft.stt.**StreamThread**(*queue*, *language*)
ABC class to be used with StreamingSTT class implementations.

This class reads audio chunks from a queue and sends it to a parsing STT engine.

Parameters

- **queue** (*Queue*) – Input Queue
- **language** (*str*) – language code for the current language.

abstract handle_audio_stream(*audio*, *language*)
Handling of audio stream.

Needs to be implemented by derived class to process audio data and optionally update *self.text* with the current hypothesis.

Argumens: audio (bytes): raw audio data. language (str): language code for the current session.

run()
Thread entry point.

2.1.3 HotWordEngine - Base for Hotword engine plugins

class mycroft.client.speech.hotword_factory.**HotWordEngine**(*key_phrase='hey mycroft'*, *config=None*, *lang='en-us'*)

Hotword/Wakeword base class to be implemented by all wake word plugins.

Parameters

- **key_phrase** (*str*) – string representation of the wake word
- **config** (*dict*) – Configuration block for the specific wake word
- **lang** (*str*) – language code (BCP-47)

found_wake_word(*frame_data*)

Check if wake word has been found.

Checks if the wake word has been found. Should reset any internal tracking of the wake word state.

Parameters **frame_data** (*binary data*) – Deprecated. Audio data for large chunk of audio to be processed. This should not be used to detect audio data instead use update() to incrementaly update audio

Returns True if a wake word was detected, else False

Return type bool

stop()

Perform any actions needed to shut down the wake word engine.

This may include things such as unloading data or shutdown external processess.

update(*chunk*)

Updates the hotword engine with new audio data.

The engine should process the data and update internal trigger state.

Parameters **chunk** (*bytes*) – Chunk of audio data to process

2.1.4 AudioBackend - Base for audioservice backend plugins

class mycroft.audio.services.**AudioBackend**(*config*, *bus*)

Base class for all audio backend implementations.

Parameters

- **config** (*dict*) – configuration dict for the instance
- **bus** (*MessageBusClient*) – Mycroft messagebus emitter

abstract add_list(*tracks*)

Add tracks to backend's playlist.

Parameters **tracks** (*list*) – list of tracks.

abstract clear_list()

Clear playlist.

lower_volume()
Lower volume.

This method is used to implement audio ducking. It will be called when Mycroft is listening or speaking to make sure the media playing isn't interfering.

next()
Skip to next track in playlist.

pause()
Pause playback.

Stops playback but may be resumed at the exact position the pause occured.

abstract play(*repeat=False*)
Start playback.

Starts playing the first track in the playlist and will contiune until all tracks have been played.

Parameters **repeat** (*bool*) – Repeat playlist, defaults to False

previous()
Skip to previous track in playlist.

restore_volume()
Restore normal volume.

Called when to restore the playback volume to previous level after Mycroft has lowered it using lower_volume().

resume()
Resume paused playback.

Resumes playback after being paused.

seek_backward(*seconds=1*)
Rewind X seconds.

Parameters **seconds** (*int*) – number of seconds to seek, if negative jump forward.

seek_forward(*seconds=1*)
Skip X seconds.

Parameters **seconds** (*int*) – number of seconds to seek, if negative rewind

set_track_start_callback(*callback_func*)
Register callback on track start.

This method should be called as each track in a playlist is started.

shutdown()
Perform clean shutdown.

Implements any audio backend specific shutdown procedures.

abstract stop()
Stop playback.

Stops the current playback.

Returns True if playback was stopped, otherwise False

Return type bool

abstract supported_uris()
List of supported uri types.

Returns Supported uri's

Return type list

`track_info()`
Get info about current playing track.

Returns Track info containing atleast the keys artist and album.

Return type dict

`class mycroft.audio.services.RemoteAudioBackend(config, bus)`
Base class for remote audio backends.

RemoteAudioBackends will always be checked after the normal AudioBackends to make playback start locally by default.

An example of a RemoteAudioBackend would be things like Chromecasts, mopidy servers, etc.

PYTHON MODULE INDEX

m

INDEX

P

Q

R

S

T

U

V

W