Templating

This is an advanced feature of Home Assistant. You'll need a basic understanding of:

- Home Assistant architecture, especially states.
- The State object.

Templating is a powerful feature that allows you to control information going into and out of the system. It is used for:

- Formatting outgoing messages in, for example, the notify platforms and Alexa integration.
- Process incoming data from sources that provide raw data, like MQTT, `rest` sensor or the `command_line` sensor.
- Automation Templating.

Building templates

Templating in Home Assistant is powered by the Jinja2 templating engine. This means that we are using their syntax and make some custom Home Assistant variables available to templates during rendering. Jinja2 supports a wide variety of operations:

- Mathematical operation
- Comparisons
- Logic

We will not go over the basics of the syntax, as Jinja2 does a great job of this in their templates documentation .

The frontend has a template editor tool to help develop and debug templates. Navigate to Developer Tools > Template, create your template in the *Template editor* and check the results on the right.

Templates can get big pretty fast. To keep a clear overview, consider using YAML multiline strings to define your templates:

```yaml
script:
  msg_who_is_home:
    sequence:
      - service: notify.notify
        data:
          message: >
            {% if is_state('device_tracker.paulus', 'home') %}
              Ha, Paulus is home!
            {% else %}
              Paulus is at {{ states('device_tracker.paulus') }}.
            {% endif %}
```

IMPORTANT TEMPLATE RULES

There are a few very important rules to remember when adding templates to YAML:

> You **must** surround single-line templates with double quotes (`"`) or single quotes (`'`).
>
> It is advised that you prepare for undefined variables by using `if ... is not none` or the `default` filter, or both.
>
> It is advised that when comparing numbers, you convert the number(s) to a `float` or an `int` by using the respective filter.
>
> While the `float` and `int` filters do allow a default fallback value if the conversion is unsuccessful, they do not provide the ability to catch undefined variables.

Remembering these simple rules will help save you from many headaches and endless hours of frustration when using automation templates.

ENABLED JINJA EXTENSIONS

Jinja supports a set of language extensions that add new functionality to the language. To improve the experience of writing Jinja templates, we have enabled the following extensions:

- Loop Controls (`break` and `continue`)

REUSING TEMPLATES

You can write reusable Jinja templates by adding them to a `custom_templates` folder under your configuration directory. All template files must have the `.jinja` extension and be less than 5MiB. Templates in this folder will be loaded at startup. To reload the

templates without restarting Home Assistant, invoke the `homeassistant.reload_custom_templates` service.

Once the templates are loaded, Jinja includes and imports will work using `config/custom_templates` as the base directory.

For example, you might define a macro in a template in `config/custom_templates/formatter.jinja`:

```
{% macro format_entity(entity_id) %}
{{ state_attr(entity_id, 'friendly_name') }} - {{ states(entity_id) }}
{% endmacro %}
```

In your automations, you could then reuse this macro by importing it:

```
{% from 'formatter.jinja' import format_entity %}
{{ format_entity('sensor.temperature') }}
```

Home Assistant template extensions

Extensions allow templates to access all of the Home Assistant specific states and adds other convenience functions and filters.

LIMITED TEMPLATES

Templates for some triggers as well as `trigger_variables` only support a subset of the Home Assistant template extensions. This subset is referred to as “Limited Templates”.

STATES

Not supported in limited templates.

- Iterating `states` will yield each state object.
- Iterating `states.domain` will yield each state object of that domain.
- `states.sensor.temperature` returns the state object for `sensor.temperature` (avoid when possible, see note below).
- `states` can also be used as a function, `states(entity_id, rounded=False, with_unit=False)`, which returns the state string (not the state object) of the given entity, `unknown` if it doesn’t exist, and `unavailable` if the object exists but is not available.

    - The optional arguments `rounded` and `with_unit` control the formatting of sensor state strings, please see the examples below.

- `states.sensor.temperature.state_with_unit` formats the state string in the same way as if calling `states('sensor.temperature', rounded=True, with_unit=True)`.
- `is_state` compares an entity's state with a specified state or list of states and returns `True` or `False`. `is_state('device_tracker.paulus', 'home')` will test if the given entity is the specified state. `is_state('device_tracker.paulus', ['home', 'work'])` will test if the given entity is any of the states in the list.
- `state_attr('device_tracker.paulus', 'battery')` will return the value of the attribute or None if it doesn't exist.
- `is_state_attr('device_tracker.paulus', 'battery', 40)` will test if the given entity attribute is the specified state (in this case, a numeric value). Note that the attribute can be `None` and you want to check if it is `None`, you need to use `state_attr('sensor.my_sensor', 'attr') is none` or `state_attr('sensor.my_sensor', 'attr') == None` (note the difference in the capitalization of none in both versions).
- `has_value('sensor.my_sensor')` will test if the given entity is not unknown or unavailable. Can be used as a filter or a test.

Warning

Avoid using `states.sensor.temperature.state`, instead use `states('sensor.temperature')`. It is strongly advised to use the `states()`, `is_state()`, `state_attr()` and `is_state_attr()` as much as possible, to avoid errors and error message when the entity isn't ready yet (e.g., during Home Assistant startup).

STATES EXAMPLES

The next two statements result in the same value if the state exists. The second one will result in an error if the state does not exist.

```
{{ states('device_tracker.paulus') }}
{{ states.device_tracker.paulus.state }}
```

Print out a list of all the sensor states:

```
{% for state in states.sensor %}
  {{ state.entity_id }}={{ state.state }},
{% endfor %}
```

Print out a list of all the sensor states sorted by `entity_id`:

```
{% for state in states.sensor | sort(attribute='entity_id') %}
  {{ state.entity_id }}={{ state.state }},
```

```
{% endfor %}
```

Entities that are on:

```
{{ ['light.kitchen', 'light.dining_room'] | select('is_state', 'on') | list }}
```

Other state examples:

```
{% if is_state('device_tracker.paulus', 'home') %}
  Ha, Paulus is home!
{% else %}
  Paulus is at {{ states('device_tracker.paulus') }}.
{% endif %}

#check sensor.train_departure_time state
{% if states('sensor.train_departure_time') in ("unavailable", "unknown") %}
  {{ ... }}

{% if has_value('sensor.train_departure_time') %}
  {{ ... }}


{% set state = states('sensor.temperature') %}{{ state | float + 1 if is_number(state)
else "invalid temperature" }}

{% set state = states('sensor.temperature') %}{{ (state | float * 10) | round(2) if
is_number(state)}}

{% set state = states('sensor.temperature') %}
{% if is_number(state) and state | float > 20 %}
  It is warm!
{% endif %}

{{ as_timestamp(states.binary_sensor.garage_door.last_changed) }}

{{ as_local(states.binary_sensor.garage_door.last_changed) }}

{{ as_timestamp(now()) - as_timestamp(states.binary_sensor.garage_door.last_changed) }}

{{ as_local(states.sensor.time.last_changed) }}

{{ states('sensor.expires') | as_datetime }}

# Make a list of states
{{ ['light.kitchen', 'light.dining_room'] | map('states') | list }}
```

FORMATTING SENSOR STATES

The examples below show the output of a temperature sensor with state `20.001`, unit `°C` and user configured presentation rounding set to 1 decimal.

The following example results in the number `20.001`:

```
{{ states('sensor.temperature') }}
```

The following example results in the string `"20.0 °C"`:

```
{{ states('sensor.temperature', with_unit=True) }}
```

The following example result in the string `"20.001 °C"`:

```
{{ states('sensor.temperature', with_unit=True, rounded=False) }}
```

The following example results in the number `20.0`:

```
{{ states('sensor.temperature', rounded=True) }}
```

The following example results in the number `20.001`:

```
{{ states.sensor.temperature.state }}
```

The following example results in the string `"20.0 °C"`:

```
{{ states.sensor.temperature.state_with_unit }}
```

ATTRIBUTES

Not supported in limited templates.

You can print an attribute with `state_attr` if state is defined.

ATTRIBUTES EXAMPLES

```
{% if states.device_tracker.paulus %}
  {{ state_attr('device_tracker.paulus', 'battery') }}
{% else %}
  ??
{% endif %}
```

With strings:

```
{% set tracker_name = "paulus"%}

{% if states("device_tracker." + tracker_name) != "unknown" %}
  {{ state_attr("device_tracker." + tracker_name, "battery")}}
{% else %}
  ??
{% endif %}
```

List of friendly names:

```
{{ ['binary_sensor.garage_door', 'binary_sensor.front_door'] | map('state_attr',
'friendly_name') | list }}
```

List of lights that are on with a brightness of 255:

```
{{ ['light.kitchen', 'light.dinig_room'] | select('is_state', 'on') |
select('is_state_attr', 'brightness', 255) | list }}
```

STATE TRANSLATED

Not supported in limited templates.

The `state_translated` function returns a translated state of an entity using a language that is currently configured in the general settings.

STATE TRANSLATED EXAMPLES

```
{{ states("sun.sun") }}             # below_horizon
{{ state_translated("sun.sun") }}   # Below horizon
{{ "sun.sun" | state_translated }}  # Below horizon
```

```
{{ states("binary_sensor.movement_backyard") }}             # on
{{ state_translated("binary_sensor.movement_backyard") }}   # Detected
{{ "binary_sensor.movement_backyard" | state_translated }}  # Detected
```

WORKING WITH GROUPS

Not supported in limited templates.

The `expand` function and filter can be used to sort entities and expand groups. It outputs a sorted array of entities with no duplicates.

EXPAND EXAMPLES

```
{% for tracker in expand('device_tracker.paulus', 'group.child_trackers') %}
  {{ state_attr(tracker.entity_id, 'battery') }}
  {%- if not loop.last %}, {% endif -%}
{% endfor %}
```

The same thing can also be expressed as a filter:

```
{{ expand(['device_tracker.paulus', 'group.child_trackers'])
  | selectattr("attributes.battery", 'defined')
  | join(', ', attribute="attributes.battery") }}
```

```
{% for energy in expand('group.energy_sensors') if is_number(energy.state) %}
  {{ energy.state }}
  {%- if not loop.last %}, {% endif -%}
{% endfor %}
```

The same thing can also be expressed as a test:

```
{{ expand('group.energy_sensors')
  | selectattr("state", 'is_number') | join(', ') }}
```

ENTITIES

- `is_hidden_entity(entity_id)` returns whether an entity has been hidden. Can also be used as a test.

ENTITIES EXAMPLES

```
{{ area_entities('kitchen') | reject('is_hidden_entity') }} # Gets a list of visible entities in the kitchen area
```

DEVICES

- `device_entities(device_id)` returns a list of entities that are associated with a given device ID. Can also be used as a filter.
- `device_attr(device_or_entity_id, attr_name)` returns the value of `attr_name` for the given device or entity ID. Can also be used as a filter. Not supported in limited templates.
- `is_device_attr(device_or_entity_id, attr_name, attr_value)` returns whether the value of `attr_name` for the given device or entity ID matches `attr_value`. Can also be used as a test. Not supported in limited templates.
- `device_id(entity_id)` returns the device ID for a given entity ID or device name. Can also be used as a filter.

DEVICES EXAMPLES

```
{{ device_attr('deadbeefdeadbeefdeadbeefdeadbeef', 'manufacturer') }}  # Sony
```

```
{{ is_device_attr('deadbeefdeadbeefdeadbeefdeadbeef', 'manufacturer', 'Sony') }}  # true
```

```
{{ device_id('sensor.sony') }}  # deadbeefdeadbeefdeadbeefdeadbeef
```

CONFIG ENTRIES

- `config_entry_id(entity_id)` returns the config entry ID for a given entity ID. Can also be used as a filter.
- `config_entry_attr(config_entry_id, attr)` returns the value of `attr` for the config entry of the given entity ID. Can also be used as a filter. The following attributes are allowed: `domain`, `title`, `state`, `source`, `disabled_by`. Not supported in limited templates.

CONFIG ENTRIES EXAMPLES

```
{{ config_entry_id('sensor.sony') }}  # deadbeefdeadbeefdeadbeefdeadbeef
```

```
{{ config_entry_attr(config_entry_id('sensor.sony'), 'title') }}  # Sony Bravia TV
```

FLOORS

- `floors()` returns the full list of floor IDs.
- `floor_id(lookup_value)` returns the floor ID for a given device ID, entity ID, area ID, or area name. Can also be used as a filter.
- `floor_name(lookup_value)` returns the floor name for a given device ID, entity ID, area ID, or floor ID. Can also be used as a filter.
- `floor_areas(floor_name_or_id)` returns the list of area IDs tied to a given floor ID or name. Can also be used as a filter.

FLOORS EXAMPLES

```
{{ floors() }}  # ['floor_id']
```

```
{{ floor_id('First floor') }}  # 'first_floor'
```

```
{{ floor_id('my_device_id') }}  # 'second_floor'
```

```
{{ floor_id('sensor.sony') }}  # 'first_floor'
```

```
{{ floor_name('first_floor') }}  # 'First floor'
```

```
{{ floor_name('my_device_id') }}  # 'Second floor'
```

```
{{ floor_name('sensor.sony') }}  # 'First floor'
```

```
{{ floor_areas('first_floor') }}  # ['living_room', 'kitchen']
```

AREAS

- `areas()` returns the full list of area IDs

- `area_id(lookup_value)` returns the area ID for a given device ID, entity ID, or area name. Can also be used as a filter.
- `area_name(lookup_value)` returns the area name for a given device ID, entity ID, or area ID. Can also be used as a filter.
- `area_entities(area_name_or_id)` returns the list of entity IDs tied to a given area ID or name. Can also be used as a filter.
- `area_devices(area_name_or_id)` returns the list of device IDs tied to a given area ID or name. Can also be used as a filter.

AREAS EXAMPLES

```
{{ areas() }}  # ['area_id']
```

```
{{ area_id('Living Room') }}  # 'deadbeefdeadbeefdeadbeefdeadbeef'
```

```
{{ area_id('my_device_id') }}  # 'deadbeefdeadbeefdeadbeefdeadbeef'
```

```
{{ area_id('sensor.sony') }}  # 'deadbeefdeadbeefdeadbeefdeadbeef'
```

```
{{ area_name('deadbeefdeadbeefdeadbeefdeadbeef') }}  # 'Living Room'
```

```
{{ area_name('my_device_id') }}  # 'Living Room'
```

```
{{ area_name('sensor.sony') }}  # 'Living Room'
```

```
{{ area_entities('deadbeefdeadbeefdeadbeefdeadbeef') }}  # ['sensor.sony']
```

```
{{ area_devices('Living Room') }}  # ['my_device_id']
```

ENTITIES FOR AN INTEGRATION

- `integration_entities(integration)` returns a list of entities that are associated with a given integration, such as `hue` or `zwave_js`.
- `integration_entities(config_entry_title)` if you have multiple entries set-up for an integration, you can also use the title you've set for the integration in case you only want to target a specific entry.

If there is more than one entry with the same title, the entities for all the matching entries will be returned, even if the entries are for different integrations. It's not possible to search for entities of an untitled integration.

INTEGRATIONS EXAMPLES

```
{{ integration_entities('hue') }}  # ['light.hue_light_upstairs',
'light.hue_light_downstairs']
```

```
{{ integration_entities('Hue bridge downstairs') }}  # ['light.hue_light_downstairs']
```

LABELS

- `labels()` returns the full list of label IDs, or those for a given area ID, device ID, or entity ID.
- `label_id(lookup_value)` returns the label ID for a given label name.
- `label_name(lookup_value)` returns the label name for a given label ID.
- `label_areas(label_name_or_id)` returns the list of area IDs tied to a given label ID or name.
- `label_devices(label_name_or_id)` returns the list of device IDs tied to a given label ID or name.
- `label_entities(label_name_or_id)` returns the list of entity IDs tied to a given label ID or name.

Each of the label template functions can also be used as a filter.

LABELS EXAMPLES

```
{{ labels() }}  # ['christmas_decorations', 'energy_saver', 'security']
```

```
{{ labels("living_room") }}  # ['christmas_decorations', 'energy_saver']
```

```
{{ labels("my_device_id") }}  # ['security']
```

```
{{ labels("light.christmas_tree") }}  # ['christmas_decorations']
```

```
{{ label_id('Energy saver') }}  # 'energy_saver'
```

```
{{ label_name('energy_saver') }}  # 'Energy saver'
```

```
{{ label_areas('security') }}  # ['driveway', 'garden', 'porch']
```

```
{{ label_devices('energy_saver') }}  # ['deadbeefdeadbeefdeadbeefdeadbeef']
```

```
{{ label_entities('security') }}  # ['camera.driveway', 'binary_sensor.motion_garden',
'camera.porch']
```

ISSUES

- `issues()` returns all open issues as a mapping of (domain, issue_id) tuples to the issue object.
- `issue(domain, issue_id)` returns a specific issue for the provided domain and issue_id.

ISSUES EXAMPLES

```
{{ issues() }}  # { ("homeassistant", "deprecated_yaml_ping"): {...}, ("cloud",
"legacy_subscription"): {...} }
```

```
{{ issue('homeassistant', 'python_version') }}  # {"breaks_in_ha_version": "2024.4",
"domain": "homeassistant", "issue_id": "python_version", "is_persistent": False, ...}
```

IMMEDIATE IF (IIF)

A common case is to conditionally return a value based on another value. For example, return a “Yes” or “No” when the light is on or off.

This can be written as:

```
{% if is_state('light.kitchen', 'on') %}
  Yes
{% else %}
  No
{% endif %}
```

Or using a shorter syntax:

```
{{ 'Yes' if is_state('light.kitchen', 'on') else 'No' }}
```

Additionally, to the above, you can use the `iif` function/filter, which is an immediate if.

Syntax: `iif(condition, if_true, if_false, if_none)`

`iif` returns the value of `if_true` if the condition is truthy, the value of `if_false` if it’s `falsy` and the value of `if_none` if it’s `None`. An empty string, an empty mapping or an an empty list, are all falsy, refer to the Python documentation for an in depth explanation.

`if_true` is optional, if it’s omitted `True` is returned if the condition is truthy. `if_false` is optional, if it’s omitted `False` is returned if the condition is falsy. `if_none` is optional, if it’s omitted the value of `if_false` is returned if the condition is `None`.

Examples using `iif`:

```
{{ iif(is_state('light.kitchen', 'on'), 'Yes', 'No') }}

{{ is_state('light.kitchen', 'on') | iif('Yes', 'No') }}

{{ (states('light.kitchen') == 'on') | iif('Yes', 'No') }}
```

Warning

The immediate if filter does not short-circuit like you might expect with a typical conditional statement. The `if_true`, `if_false` and `if_none` expressions will all be evaluated and the filter will simply return one of the resulting values. This means you cannot use this filter to prevent executing an expression which would result in an error.
For example, if you wanted to select a field from `trigger` in an automation based on the platform you might go to make this template: `trigger.platform == 'event' | iif(trigger.event.data.message, trigger.to_state.state)`. This won't work because both expressions will be evaluated and one will fail since the field doesn't exist. Instead you have to do this `trigger.event.data.message if trigger.platform == 'event' else trigger.to_state.state`. This form of the expression short-circuits so if the platform is `event` the expression `trigger.to_state.state` will never be evaluated and won't cause an error.

TIME

`now()`, `relative_time()`, `today_at()`, and `utcnow()` are not supported in limited templates.

- `now()` returns a datetime object that represents the current time in your time zone.
  - You can also use: `now().second`, `now().minute`, `now().hour`, `now().day`, `now().month`, `now().year`, `now().weekday()` and `now().isoweekday()` and other `datetime` attributes and functions.
  - Using `now()` will cause templates to be refreshed at the start of every new minute.
- `utcnow()` returns a datetime object of the current time in the UTC timezone.
  - For specific values: `utcnow().second`, `utcnow().minute`, `utcnow().hour`, `utcnow().day`, `utcnow().month`, `utcnow().year`, `utcnow().weekday()` and `utcnow().isoweekday()`.
  - Using `utcnow()` will cause templates to be refreshed at the start of every new minute.
- `today_at(value)` converts a string containing a military time format to a datetime object with today's date in your time zone.
  - Using `today_at()` will cause templates to be refreshed at the start of every new minute.

```
# Is the current time past 10:15?
{{ now() > today_at("10:15") }}
```

- `as_datetime(value, default)` converts a string containing a timestamp, or valid UNIX timestamp, to a datetime object. If that fails, it returns the `default` value or, if omitted, raises an error. When the input is already a datetime object it will be returned as is. in case the input is a datetime.date object, midnight will be added as time. This function can also be used as a filter.
- `as_timestamp(value, default)` converts datetime object or string to UNIX timestamp. If that fails, returns the `default` value, or if omitted raises an error. This function can also be used as a filter.
- `as_local()` converts datetime object to local time. This function can also be used as a filter.
- `strptime(string, format, default)` parses a string based on a format and returns a datetime object. If that fails, it returns the `default` value or, if omitted, raises an error.
- `time_since(datetime, precision)` converts a datetime object into its human-readable time string. The time string can be in seconds, minutes, hours, days, months, and years. `precision` takes an integer (full number) and indicates the number of units returned. The last unit is rounded. For example: `precision = 1` could return “2 years” while `precision = 2` could return “1 year 11 months”. This function can also be used as a filter. If the datetime is in the future, returns 0 seconds. A precision of 0 returns all available units, default is 1.
- `time_until(datetime, precision)` converts a datetime object into a human-readable time string. The time string can be in seconds, minutes, hours, days, months, and years. `precision` takes an integer (full number) and indicates the number of units returned. The last unit is rounded. For example: `precision = 1` could return “2 years” while `precision = 2` could return “1 year 11 months”. This function can also be used as a filter. If the datetime is in the past, returns 0 seconds. A precision of 0 returns all available units, default is 1.
- `timedelta` returns a timedelta object and accepts the same arguments as the Python `datetime.timedelta` function – days, seconds, microseconds, milliseconds, minutes, hours, weeks.

```
# 77 minutes before current time.
{{ now() - timedelta( hours = 1, minutes = 17 ) }}
```

- `as_timedelta(string)` converts a string to a timedelta object. Expects data in the format `DD HH:MM:SS.uuuuuu`, `DD HH:MM:SS,uuuuuu`, or as specified by ISO 8601 (e.g. `P4DT1H15M20S` which is equivalent to `4 1:15:20`) or PostgreSQL’s day-time interval format (e.g. `3 days 04:05:06`) This function can also be used as a filter.

```
# Renders to "00:10:00"
{{ as_timedelta("PT10M") }}
```

- Filter `timestamp_local(default)` converts a UNIX timestamp to the ISO format string representation as date/time in your local timezone. If that fails, returns

the `default` value, or if omitted raises an error. If a custom string format is needed in the string, use `timestamp_custom` instead.

- Filter `timestamp_utc(default)` converts a UNIX timestamp to the ISO format string representation representation as date/time in UTC timezone. If that fails, returns the `default` value, or if omitted raises an error. If a custom string format is needed in the string, use `timestamp_custom` instead.
- Filter `timestamp_custom(format_string, local=True, default)` converts an UNIX timestamp to its string representation based on a custom format, the use of a local timezone is the default. If that fails, returns the `default` value, or if omitted raises an error. Supports the standard Python time formatting options .

Tip

UNIX timestamp is the number of seconds that have elapsed since 00:00:00 UTC on 1 January 1970. Therefore, if used as a function's argument, it can be substituted with a numeric value (`int` or `float`).

Important

If your template is returning a timestamp that should be displayed in the frontend (e.g., as a sensor entity with `device_class: timestamp`), you have to ensure that it is the ISO 8601 format (meaning it has the "T" separator between the date and time portion). Otherwise, frontend rendering on macOS and iOS devices will show an error. The following value template would result in such an error:
`{{ states.sun.sun.last_changed }}` => `2023-07-30 20:03:49.253717+00:00` (missing "T" separator)
To fix it, enforce the ISO conversion via `isoformat()`:
`{{ states.sun.sun.last_changed.isoformat() }}` => `2023-07-30T20:03:49.253717+00:00` (contains "T" separator)

```
{{ 120 | timestamp_local }}
```

TO/FROM JSON

The `to_json` filter serializes an object to a JSON string. In some cases, it may be necessary to format a JSON string for use with a webhook, as a parameter for command-line utilities or any number of other applications. This can be complicated in a template, especially when dealing with escaping special characters. Using the `to_json` filter, this is handled automatically.

`to_json` also accepts boolean arguments for `pretty_print`, which will pretty print the JSON with a 2-space indent to make it more human-readable, and `sort_keys`, which

will sort the keys of the JSON object, ensuring that the resulting string is consistent for the same input.

If you need to generate JSON that will be used by a parser that lacks support for Unicode characters, you can add `ensure_ascii=True` to have `to_json` generate Unicode escape sequences in strings.

The `from_json` filter operates similarly, but in the other direction, de-serializing a JSON string back into an object.

TO/FROM JSON EXAMPLES

TEMPLATE

```
{% set temp = {'temperature': 25, 'unit': '°C'} %}
stringified object: {{ temp }}
object|to_json: {{ temp|to_json(sort_keys=True) }}
```

OUTPUT

```
stringified object: {'temperature': 25, 'unit': '°C'}
object|to_json: {"temperature": 25, "unit": "°C"}
```

Conversely, `from_json` can be used to de-serialize a JSON string back into an object to make it possible to easily extract usable data.

TEMPLATE

```
{% set temp = '{"temperature": 25, "unit": "°C"}'|from_json %}
The temperature is {{ temp.temperature }}{{ temp.unit }}
```

OUTPUT

```
The temperature is 25°C
```

IS DEFINED

Sometimes a template should only return if a value or object is defined, if not, the supplied default value should be returned. This can be useful to validate a JSON payload. The `is_defined` filter allows to throw an error if a value or object is not defined.

Example using `is_defined` to parse a JSON payload:

```
{{ value_json.val | is_defined }}
```

This will throw an error `UndefinedError: 'value_json' is undefined` if the JSON payload has no `val` attribute.

VERSION

- `version()` Returns a AwesomeVersion object for the value given inside the brackets.
  - This is also available as a filter (`| version`).

Examples:

- `{{ version("2099.9.9") > "2000.0.0" }}` Will return `True`
- `{{ version("2099.9.9") < "2099.10" }}` Will return `True`
- `{{ "2099.9.9" | version < "2099.10" }}` Will return `True`
- `{{ (version("2099.9.9") - "2100.9.10").major }}` Will return `True`
- `{{ (version("2099.9.9") - "2099.10.9").minor }}` Will return `True`
- `{{ (version("2099.9.9") - "2099.9.10").patch }}` Will return `True`

DISTANCE

Not supported in limited templates.

- `distance()` measures the distance between home, an entity, or coordinates. The unit of measurement (kilometers or miles) depends on the system's configuration settings.
- `closest()` will find the closest entity.

DISTANCE EXAMPLES

If only one location is passed in, Home Assistant will measure the distance from home.

```
Using Lat Lng coordinates: {{ distance(123.45, 123.45) }}

Using State: {{ distance(states.device_tracker.paulus) }}

These can also be combined in any combination:
{{ distance(123.45, 123.45, 'device_tracker.paulus') }}
{{ distance('device_tracker.anne_therese', 'device_tracker.paulus') }}
```

CLOSEST EXAMPLES

The closest function and filter will find the closest entity to the Home Assistant location:

```
Query all entities: {{ closest(states) }}
Query all entities of a specific domain: {{ closest(states.device_tracker) }}
Query all entities in group.children: {{ closest('group.children') }}
Query all entities in group.children: {{ closest(states.group.children) }}
```

Find entities closest to a coordinate or another entity. All previous arguments still apply for second argument.

```
Closest to a coordinate: {{ closest(23.456, 23.456, 'group.children') }}
Closest to an entity: {{ closest('zone.school', 'group.children') }}
Closest to an entity: {{ closest(states.zone.school, 'group.children') }}
```

Since closest returns a state, we can combine it with distance too.

```
{{ closest(states).name }} is {{ distance(closest(states)) }} kilometers away.
```

The last argument of the closest function has an implicit `expand`, and can take any iterable sequence of states or entity IDs, and will expand groups:

```
Closest out of given entities:
    {{ closest(['group.children', states.device_tracker]) }}
Closest to a coordinate:
    {{ closest(23.456, 23.456, ['group.children', states.device_tracker]) }}
Closest to some entity:
    {{ closest(states.zone.school, ['group.children', states.device_tracker]) }}
```

It will also work as a filter over an iterable group of entities or groups:

```
Closest out of given entities:
    {{ ['group.children', states.device_tracker] | closest }}
Closest to a coordinate:
    {{ ['group.children', states.device_tracker] | closest(23.456, 23.456) }}
Closest to some entity:
    {{ ['group.children', states.device_tracker] | closest(states.zone.school) }}
```

CONTAINS

Jinja provides by default a `in` operator how return `True` when one element is `in` a provided list. The `contains` test and filter allow you to do the exact opposite and test for a list containing an element. This is particularly useful in `select` or `selectattr` filter, as well as to check if a device has a specific attribute, a `supported_color_modes`, a specific light effect.

Some examples:

- `{{ state_attr('light.dining_room', 'effect_list') | contains('rainbow') }}` will return `true` if the light has a `rainbow` effect.

- `{{ expand('light.office') | selectattr("attributes.supported_color_modes", 'contains', 'color_temp') | list }}` will return all light that support color_temp in the office group.
- ```
  {% set current_month = now().month %}
  {% set extra_ambiance = [
    {'name':'Halloween', 'month': [10,11]},
    {'name':'Noel', 'month': [1,11,12]}
  ]%}
  {% set to_add = extra_ambiance | selectattr('month', 'contains', current_month )
  | map(attribute='name') | list  %}
  {% set to_remove = extra_ambiance | map(attribute='name') | reject('in', to_add)
  | list %}
  {{ (state_attr('input_select.light_theme', 'options') + to_add ) | unique |
  reject('in', to_remove) | list }}
  ```

  This more complex example uses the `contains` filter to match the current month with a list. In this case, it's used to generate a list of light theme to give to the `Input select: Set options` service.

NUMERIC FUNCTIONS AND FILTERS

Some of these functions can also be used in a filter. This means they can act as a normal function like this `sqrt(2)`, or as part of a filter like this `2|sqrt`.

Note

The numeric functions and filters raise an error if the input is not a valid number, optionally a default value can be specified which will be returned instead. The `is_number` function and filter can be used to check if a value is a valid number. Errors can be caught by the `default` filter.

- `{{ float("not_a_number") }}` - the template will fail to render
- `{{ "not_a_number" | sin }}` - the template will fail to render
- `{{ float("not_a_number", default="Invalid number!") }}` - renders as `"Invalid number!"`
- `{{ "not_a_number" | sin(default="Invalid number!") }}` - renders as `"Invalid number!"`

- `float(value, default)` function will attempt to convert the input to a `float`. If that fails, returns the `default` value, or if omitted raises an error.
- `float(default)` filter will attempt to convert the input to a `float`. If that fails, returns the `default` value, or if omitted raises an error.
- `is_number` will return `True` if the input can be parsed by Python's `float` function and the parsed input is not `inf` or `nan`, in all other cases returns `False`. Note
  ```

that a Python `bool` will return `True` but the strings `"True"` and `"False"` will both return `False`. Can be used as a filter.

- `int(value, default)` function is similar to `float`, but converts to an `int` instead. Like `float`, it has a filter form, and an error is raised if the `default` value is omitted. Fractional part is discarded: `int("1.5")` is `1`.
- `bool(value, default)` function converts the value to either `true` or `false`. The following values are considered to be `true`: boolean `true`, non-zero `int`s and `float`s, and the strings `"true"`, `"yes"`, `"on"`, `"enable"`, and `"1"` (case-insensitive). `false` is returned for the opposite values: boolean `false`, integer or floating-point `0`, and the strings `"false"`, `"no"`, `"off"`, `"disable"`, and `"0"` (also case-insensitive). If the value is not listed here, the function returns the `default` value, or if omitted raises an error. This function is intended to be used on states of binary sensors, switches, or similar entities, so its behavior is different from Python's built-in `bool` conversion, which would consider e.g. `"on"`, `"off"`, and `"unknown"` all to be `true`, but `""` to be `false`; if that is desired, use `not not value` or a similar construct instead.
  Like `float` and `int`, `bool` has a filter form. Using `none` as the default value is particularly useful in combination with the immediate if filter: it can handle all three possible cases in a single line.
- `log(value, base, default)` will take the logarithm of the input. When the base is omitted, it defaults to `e` - the natural logarithm. If `value` or `base` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can also be used as a filter.
- `sin(value, default)` will return the sine of the input. If `value` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `cos(value, default)` will return the cosine of the input. If `value` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `tan(value, default)` will return the tangent of the input. If `value` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `asin(value, default)` will return the arcus sine of the input. If `value` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `acos(value, default)` will return the arcus cosine of the input. If `value` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `atan(value, default)` will return the arcus tangent of the input. If `value` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `atan2(y, x, default)` will return the four quadrant arcus tangent of y / x. If `y` or `x` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can be used as a filter.

- `sqrt(value, default)` will return the square root of the input. If `value` can't be converted to a `float`, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `max([x, y, ...])` will obtain the largest item in a sequence. Uses the same parameters as the built-in max filter.
- `min([x, y, ...])` will obtain the smallest item in a sequence. Uses the same parameters as the built-in min filter.
- `average([x, y, ...], default)` will return the average value of the sequence. If list is empty or contains non-numeric value, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `median([x, y, ...], default)` will return the median value of the sequence. If list is empty or contains non-numeric value, returns the `default` value, or if omitted raises an error. Can be used as a filter.
- `statistical_mode([x, y, ...], default)` will return the statistical mode value (most frequent occurrence) of the sequence. If the list is empty, it returns the `default` value, or if omitted raises an error. It can be used as a filter.
- `e` mathematical constant, approximately 2.71828.
- `pi` mathematical constant, approximately 3.14159.
- `tau` mathematical constant, approximately 6.28318.
- Filter `round(precision, method, default)` will convert the input to a number and round it to `precision` decimals. Round has four modes and the default mode (with no mode specified) will round-to-even . If the input value can't be converted to a `float`, returns the `default` value, or if omitted raises an error.
  - `round(precision, "floor", default)` will always round down to `precision` decimals
  - `round(precision, "ceil", default)` will always round up to `precision` decimals
  - `round(1, "half", default)` will always round to the nearest .5 value. `precision` should be 1 for this mode
- Filter `value_one|bitwise_and(value_two)` perform a bitwise and(&) operation with two values.
- Filter `value_one|bitwise_or(value_two)` perform a bitwise or(|) operation with two values.
- Filter `value_one|bitwise_xor(value_two)` perform a bitwise xor(^) operation with two values.
- Filter `ord` will return for a string of length one an integer representing the Unicode code point of the character when the argument is a Unicode object, or the value of the byte when the argument is an 8-bit string.
- Filter `multiply(arg)` will convert the input to a number and multiply it by `arg`. Useful in list operations in conjunction with `map`.
- Filter `add(arg)` will convert the input to a number and add it to `arg`. Useful in list operations in conjunction with `map`.

COMPLEX TYPE CHECKING

In addition to strings and numbers, Python (and Jinja) supports lists, sets, and dictionaries. To help you with testing these types, you can use the following tests:

- `x is list` will return whether `x` is a `list` or not (e.g. `[1, 2] is list` will return `True`).
- `x is set` will return whether `x` is a `set` or not (e.g. `{1, 2} is set` will return `True`).
- `x is tuple` will return whether `x` is a `tuple` or not (e.g. `(1, 2) is tuple` will return `True`).
- `x is datetime` will return whether `x` is a `datetime` or not (e.g. `datetime(2020, 1, 1, 0, 0, 0) is datetime` will return `True`).
- `x is string_like` will return whether `x` is a string, bytes, or bytearray object.

Note that, in Home Assistant, Jinja has built-in tests for `boolean` (`True`/`False`), `callable` (any function), `float` (a number with a decimal), `integer` (a number without a decimal), `iterable` (a value that can be iterated over such as a `list`, `set`, `string`, or generator), `mapping` (mainly `dict` but also supports other dictionary like types), `number` (`float` or `int`), `sequence` (a value that can be iterated over and indexed such as `list` and `string`), and `string`.

TYPE CONVERSIONS

While Jinja natively supports the conversion of an iterable to a `list`, it does not support conversion to a `tuple` or `set`. To help you with using these types, you can use the following functions:

- `set(x)` will convert any iterable `x` to a `set` (e.g. `set([1, 2]) == {1, 2}`)
- `tuple(x)` will convert any iterable `x` to a `tuple` (e.g. `tuple("abc") == ("a", "b", "c")`)

Note that, in Home Assistant, to convert a value to a `list`, a `string`, an `int`, or a `float`, Jinja has built-in functions with names that correspond to each type.

FUNCTIONS AND FILTERS TO PROCESS RAW DATA

These functions are used to process raw value's in a `bytes` format to values in a native Python type or vice-versa. The `pack` and `unpack` functions can also be used as a filter. They make use of the Python 3 `struct` library. See: Python struct library documentation

- Filter `value | pack(format_string)` will convert a native type to a `bytes` type object. This will call function `struct.pack(format_string, value)`. Returns `None` if an error occurs or when `format_string` is invalid.
- Function `pack(value, format_string)` will convert a native type to a `bytes` type object. This will call function `struct.pack(format_string, value)`. Returns `None` if an error occurs or when `format_string` is invalid.
- Filter `value | unpack(format_string, offset=0)` will try to convert a `bytes` object into a native Python object. The `offset` parameter defines the offset position in bytes from the start of the input `bytes` based buffer. This will call function `struct.unpack_from(format_string, value, offset=offset)`. Returns `None` if an error occurs or when `format_string` is invalid. Note that the filter `unpack` will only return the first `bytes` object, despite the function `struct.unpack_from` supporting to return multiple objects (e.g. with `format_string` being `">hh"`.
- Function `unpack(value, format_string, offset=0)` will try to convert a `bytes` object into a native Python object. The `offset` parameter defines the offset position in bytes from the start of the input `bytes` based buffer. This will call function `struct.unpack_from(format_string, value, offset=offset)`. Returns `None` if an error occurs or when `format_string` is invalid. Note that the function `unpack` will only return the first `bytes` object, despite the function `struct.unpack_from` supporting to return multiple objects (e.g. with `format_string` being `">hh"`.

Note

Some examples:

- `{{ 0xDEADBEEF | pack(">I") }}` - renders as `b"\xde\xad\xbe\xef"`
- `{{ pack(0xDEADBEEF, ">I") }}` - renders as `b"\xde\xad\xbe\xef"`
- `{{ "0x%X" % 0xDEADBEEF | pack(">I") | unpack(">I") }}` - renders as `0xDEADBEEF`
- `{{ "0x%X" % 0xDEADBEEF | pack(">I") | unpack(">H", offset=2) }}` - renders as `0xBEEF`

STRING FILTERS

- Filter `urlencode` will convert an object to a percent-encoded ASCII text string (e.g., for HTTP requests using `application/x-www-form-urlencoded`).
- Filter `slugify(separator="_")` will convert a given string into a “slug”.
- Filter `ordinal` will convert an integer into a number defining a position in a series (e.g., `1st`, `2nd`, `3rd`, `4th`, etc).

- Filter `value | base64_decode` Decodes a base 64 string to a string, by default utf-8 encoding is used.
- Filter `value | base64_decode("ascii")` Decodes a base 64 string to a string, using ascii encoding.
- Filter `value | base64_decode(None)` Decodes a base 64 string to raw bytes.

Some examples:

- `{{ "aG9tZWFzc2lzdGFudA==" | base64_decode }}` - renders as `homeassistant`
- `{{ "aG9tZWFzc2lzdGFudA==" | base64_decode(None) }}` - renders as `b'homeassistant'`

REGULAR EXPRESSIONS

For more information on regular expressions See: Python regular expression operations

- Test `string is match(find, ignorecase=False)` will match the find expression at the beginning of the string using regex.
- Test `string is search(find, ignorecase=False)` will match the find expression anywhere in the string using regex.
- Filter `string|regex_replace(find='', replace='', ignorecase=False)` will replace the find expression with the replace string using regex.
- Filter `value | regex_findall(find='', ignorecase=False)` will find all regex matches of the find expression in `value` and return the array of matches.
- Filter `value | regex_findall_index(find='', index=0, ignorecase=False)` will do the same as `regex_findall` and return the match at index.

Processing incoming data

The other part of templating is processing incoming data. It allows you to modify incoming data and extract only the data you care about. This will only work for platforms and integrations that mention support for this in their documentation.

It depends per integration or platform, but it is common to be able to define a template using the `value_template` configuration key. When a new value arrives, your template will be rendered while having access to the following values on top of the usual Home Assistant extensions:

Variable	Description
`value`	The incoming value.
`value_json`	The incoming value parsed as JSON.

This means that if the incoming values looks like the sample below:

```
{
  "on": "true",
  "temp": 21
}
```

The template for `on` would be:

```
"{{value_json.on}}"
```

Nested JSON in a response is supported as well:

```
{
  "sensor": {
    "type": "air",
    "id": "12345"
  },
  "values": {
    "temp": 26.09,
    "hum": 56.73
  }
}
```

Just use the “Square bracket notation” to get the value.

```
"{{ value_json['values']['temp'] }}"
```

The following overview contains a couple of options to get the needed values:

```
# Incoming value:
{"primes": [2, 3, 5, 7, 11, 13]}

# Extract first prime number
{{ value_json.primes[0] }}

# Format output
{{ "%+.1f" | value_json }}

# Math
{{ value_json | float * 1024 if is_number(value_json) }}
{{ float(value_json) * (2**10) if is_number(value_json) }}
{{ value_json | log if is_number(value_json) }}
{{ log(1000, 10) }}
{{ sin(pi / 2) }}
```

```
{{ cos(tau) }}
{{ tan(pi) }}
{{ sqrt(e) }}

# Timestamps
{{ value_json.tst | timestamp_local }}
{{ value_json.tst | timestamp_utc }}
{{ value_json.tst | timestamp_custom('%Y', True) }}
```

To evaluate a response, go to **Developer Tools > Template**, create your output in "Template editor", and check the result.

```
{% set value_json=
    {"name":"Outside",
     "device":"weather-ha",
     "data":
        {"temp":"24C",
         "hum":"35%"
         } }%}

{{value_json.data.hum[:-1]}}
```

USING TEMPLATES WITH THE MQTT INTEGRATION

The MQTT integration relies heavily on templates. Templates are used to transform incoming payloads (value templates) to status updates or incoming service calls (command templates) to payloads that configure the MQTT device.

USING VALUE TEMPLATES WITH MQTT

For incoming data a value template translates incoming JSON or raw data to a valid payload. Incoming payloads are rendered with possible JSON values, so when rendering the `value_json` can be used access the attributes in a JSON based payload.

Note

Example value template:
With given payload:

```
{ "state": "ON", "temperature": 21.902 }
```

Template `{{ value_json.temperature | round(1) }}` renders to `21.9`.
Additional the MQTT entity attributes `entity_id`, `name` and `this` can be used as variables in the template. The `this` attribute refers to the entity state of the MQTT item.

USING COMMAND TEMPLATES WITH MQTT

For service calls command templates are defined to format the outgoing MQTT payload to the device. When a service call is executed `value` can be used to generate the correct payload to the device.

Note

Example command template:
With given value `21.9` template `{"temperature": {{ value }} }` renders to:

```
{
  "temperature": 21.9
}
```

Additional the MQTT entity attributes `entity_id`, `name` and `this` can be used as variables in the template. The `this` attribute refers to the entity state of the MQTT item.

Some more things to keep in mind

ENTITY_ID THAT BEGINS WITH A NUMBER

If your template uses an `entity_id` that begins with a number (example: `states.device_tracker.2008_gmc`) you must use a bracket syntax to avoid errors caused by rendering the `entity_id` improperly. In the example given, the correct syntax for the device tracker would be: `states.device_tracker['2008_gmc']`

PRIORITY OF OPERATORS

The default priority of operators is that the filter (`|`) has priority over everything except brackets. This means that:

```
{{ states('sensor.temperature') | float / 10 | round(2) }}
```

Would round `10` to 2 decimal places, then divide `states('sensor.temperature')` by `10` (rounded to 2 decimal places so 10.00). This behavior is maybe not the one expected, but priority rules imply that.