hass-apps

Release 0.20200319.0

Mar 19, 2020

Contents

This is a collection of useful apps to empower Home Assistant even more.

The apps are built on top of the AppDaemon framework. Each has its own, detailled documentation and includes a sample configuration. Read the *Getting Started* chapter and start empowering your smart home.

CHAPTER 1

Getting Started

1.1 Installation in Hass.io

In order to use hass-apps in the hass.io ecosystem, you first need to set up an AppDaemon add-on. The recommended add-on is this one.

When you have that up and running, head over to *Installation in Docker* and choose a hass-apps version to install, **BUT instead of storing the chosen string in a ‘‘requirements.txt‘‘ file, you add it to the ‘‘python_packages‘‘ setting of the AppDaemon add-on using hass.io’s web interface.** It should look like this:

```
"python_packages": [
    "hass-apps"
]
```

1.2 Installation in Docker

Note: AppDaemon version 3.0.2 or later is required for this to work.

1. When you have the official AppDaemon container up and running, create a file named `requirements.txt` in your `apps` directory (or one of its sub-directories) with one of the following contents.
   a) To always have the latest stable version of hass-apps installed when AppDaemon starts:

   ```
   hass-apps
   ```

   b) To install a specific version of hass-apps (e.g. v0.20181005.0):

   ```
   hass-apps==0.20181005.0
   ```

   c) To always have the latest development version installed (don’t do this unless you know what you’re doing):

```
https://github.com/efficiosoft/hass-apps/archive/master.zip
```

3. Continue with the *Configuration* as normal.

1.3 Manual Installation

Hass-apps is a collection of apps for AppDaemon, hence AppDaemon is a dependency of hass-apps and will automatically be installed alongside.

The project itself is developed on GNU/Linux, but since there are no platform-specific Python modules used it should run everywhere Python and AppDaemon are available. However, we'll assume an installation on GNU/Linux for the rest of this guide. Feel free to apply it to your own operating system.

The minimum required Python version is 3.5. To find out what you have installed, run `python3 --version`. If your version of Python is recent enough, you may continue with installing.

It is strongly recommended to install hass-apps (+ it's dependencies like AppDaemon) into a virtualenv, separated even from Home Assistant in order to avoid conflicts with different versions of dependency packages.

Other huge benefits of the virtualenv installation are that you neither need root privileges nor do you pollute your system.with numerous tiny packages that are complicated to remove, should you sometime wish to do so.

The following simple steps will guide you through the installation process.

1. If you use a distribution like Debian or Ubuntu which doesn't ship `venv` with Python by default, install it first. Whithout installing `python3-venv`, you'd end up with a crippled virtualenv with pip, the Python package manager, not available. Of course you do need root privileges for this particular step.

```
sudo apt install python3-venv
```

2. Then, create the virtualenv. We do this in a directory named `appdaemon` in this example inside the user's home directory.

```
 mkdir ~/appdaemon
python3 -m venv ~/appdaemon/venv
```

3. Activate the virtualenv.

```
cd ~/appdaemon
source venv/bin/activate
```

4. Now install some common packages.

```
pip install --upgrade pip setuptools wheel
```

5. And finally, install hass-apps.

   a) Install the latest stable version from PyPi (preferred).

   ```
   pip install --upgrade hass-apps
   ```

   b) Or, as an alternative, install the state from the Git repository to get even the latest changes. But please keep in mind that this shouldn't be considered stable and isn't guaranteed to work all the time. Don't use the development version in production unless you have a good reason to do so.

   ```
   pip install --upgrade https://github.com/efficiosoft/hass-apps/archive/master.
   ↪zip
   ```

1.4 Configuration

When you followed the above steps for installing hass-apps, you automatically installed AppDaemon as well. Configuring AppDaemon is out of the scope of this tutorial, but there is a Configuration Section in the AppDaemon Documentation which describes what to do. We assume that you've got a working AppDaemon 4.x for now.

1. Get yourself a nice cup of coffee or tea. You'll surely need it.
2. Store the file hass_apps_loader.py in your AppDaemon's `apps` directory. This is just a stub which imports the real app's code.
3. Pick one or more apps you want to use.
4. Copy the sample configuration provided for each app in the docs to a new YAML file in your AppDaemon's `apps` directory and start editing it. Adapt the sample configuration as necessary. Documentary comments explaining what the different settings mean are included. The sample configurations can also be found in the GitHub repository under `docs/apps/<app_name>/sample-apps.yaml`.
5. AppDaemon should have noticed the changes made to the `apps` directory and start the new app(s) automatically.

You're done, enjoy hass-apps!

CHAPTER 2

Upgrading

As with every software, hass-apps and its dependencies need to be upgraded regularly in order to get the latest fixes, security updates, feature additions and enhancements that are incorporated every now and then.

2.1 Upgrade in Hass.io or Docker

When you followed the tutorial for *Installation in Hass.io* or *Installation in Docker* and decided for automatic upgrading, you don't need to do anything. Just ensure that your configuration stays compatible with the new hass-apps versions and restart the AppDaemon container (or the add-on in case of hass.io) from time to time.

If you explicitly decided for a specific version of hass-apps, change the version number in the `requirements.txt` file you once created (or the add-on settings) to the latest one and restart AppDaemon.

2.2 Upgrade Manually

When you've installed hass-apps using the *Manual Installation* method, simply repeat the procedure from step 3 onwards in order to upgrade.

Note: AppDaemon doesn't detect changes in the imported modules automatically and needs to be restarted manually for the upgrade to take effect.

CHAPTER 3

Schedy

Schedy is a highly-configurable, comfortable to use multi-purpose scheduler for Home Assistant that controls different types of actors such as switches and thermostats based on powerful rules while still facilitating manual intervention at any time.

The goal is to provide an easy solution for conventional scheduling (e.g. by time of day and day of week) while leaving advanced users plenty of room for customization with arbitrarily complex rules.

Note: Excited? A *Tutorial* is provided for getting up and running quickly.

These key features are implemented in Schedy. More are added continuously.

- Schedules (based on time, days of week/month, month, year and more)
- Multiple schedules for different purposes, occasions or seasons
- One schedule can control a group of actors at once
- Unlimited number of actor groups (Schedy calls them rooms), each having its own schedule
- Configurable re-scheduling after manual adjustments
- Optional synchronization of manual changes among all actors in a room
- Dynamic values based on expressions written in Python, allowing for arbitrarily complex rules that can consider any information available to Home Assistant
- Event-driven system enables external control by ordinary Home Assistant events
- Re-sending until actors report a change back (for unreliable networks)
- Collection of individually configurable statistical parameters regarding Schedy's operation
- Configurable logging

The scenarios for which you might need a scheduler are numerous. Here are just some ideas:

- advanced heating setup based on day, time, presence etc.
- motion, daylight and time-triggered lights

- controlling roller shutters based on time, sun and wind conditions
- ... and much more

This documentation is written for both beginners that want to get started with Schedy and advanced users needing a reference book for implementing complex scenarios.

In order to get started, it is recommended to read the *chapter about the concept* first and then proceed to the *Tutorial*.

3.1 The Concept

Schedy is a multi-purpose scheduler for Home Assistant.

When one thinks of a schedule, he usually imagines to configure values (such as temperatures) for different times of the day and days of week. That's of course possible with Schedy in a convenient manner, but it can do a lot more as well.

Scheduling here basically means linking time frames (and/or state conditions) to the states actors should adopt.

3.1.1 Why not use Automations?

You may now ask: Why should I use a third-party solution when I have automations in Home Assistant right at hand? Well, that's a legitimate question. But have you ever tried to implement a flexible, easily maintainable schedule for heating, roller shutters or lights using plain automations? Maybe even one that cooperates with presence or motion detection? If not, believe me, that's no fun and will get really confusing sooner than later.

Besides this practical reasons why automations are not suited well for scheduling, take a look at what automations really do: reacting to triggers. Triggers can be described as events - they happen once, cause the automation to fire and are then gone. Possible triggers could be "I get home" or "Someone turns on the TV". But if you, for instance, start Home Assistant after the TV was already turned on, your automation won't fire at all.

In contrast to automations, Schedy maps time (and optionally state) to state. Instead of waiting for the events "It's 8.00pm" and ""Someone turns on the TV", Schedy checks "Is it after 8.00pm?" and ""Is the TV turned on?" and, if so, ensures the corresponding scheduled state, such as "Living room lights off" is in place.

Note: Automations react to triggers (events/state changes), Schedy reacts to time and/or state..

Don't get me wrong, automations are great and Schedy doesn't try to obsolete them, but they simply aren't suited well for scheduling.

3.1.2 How it Works

While reading this documentation and working with Schedy, you'll stumble across different terms that you have to understand first.

An **actor** is an entity that can be controlled by Home Assistant. A switch is an actor that can have the states `on` and `off`, for instance. A thermostat is one that can be set to different temperature values or be turned off completely. There are far more possibilities for what can be used as an actor in Schedy, but that's enough for now.

The purpose of a **schedule**, which usually consists of multiple **schedule rules**, is to define what state actors should be in at which times. Apart from the rich set of available constraints for specifying a schedule rule's period of validity, Schedy's schedules do also support **expressions** that can easily be written in-line in Python to let the state of arbitrary entities in Home Assistant influence the scheduled value, allowing for decisions based on, for instance, presence or motion.

Finally, Schedy operates on so-called rooms. A **room** is an unit with a schedule and one or more actors that are controlled simultaneously by that schedule.

That's basically it. Plug all these components together and you get a really powerful scheduler that can satisfy both basic and advanced needs. The next chapter is a tutorial for getting Schedy up and running quickly.

3.2 Tutorial

In this tutorial, you'll learn how to set up a basic heating schedule with some cool features using Schedy.

Note: You are highly recommended to read the *chapter about Schedy's conception* before proceeding.

This tutorial's purpose is to get up and running quickly, which is why explanations aren't very detailed here, but the individual sections tell you where to read more about particular features.

Contents

3.2.1 Objective

The goal is to have Schedy control thermostats in a flat with four rooms, a **living room**, a **bedroom** and two **kids rooms**. In the living room, there are two radiators, hence we've got two thermostats there.

In each room, there's a window with window sensor attached. We want the heatings in the particular room to be turned off when a window is opened and the previous setting be restored when it's closed again.

Furthermore, we make some enhancements to our schedules, allowing for dynamic schedule switching and more. Stay tuned!

3.2.2 Configuration Skeleton

Our first step is to create a basic configuration which defines our rooms and actors and save it as `schedy_heating.yaml` in AppDaemon's `apps` directory:

```
schedy_heating:  # This is our app instance name.
  module: hass_apps_loader
  class: SchedyApp

  actor_type: thermostat

  rooms:

    living:
      actors:
        climate.living_1:
        climate.living_2:
      schedule:

    bed:
      actors:
        climate.bed_1:
      schedule:

    kids1:
      actors:
        climate.kids1_1:
      schedule:

    kids2:
      actors:
        climate.kids2_1:
      schedule:
```

During the following steps, only configuration changes and additions are shown. A full sample configuration like it looks after all steps have been applied can be found at the *end of this tutorial*.

You may also want to consult the *full reference* of all available settings.

3.2.3 Reading the Log

Schedy uses AppDaemon's regular logging functionality to inform you about what's going on. How to access these logs depends on the way you set up AppDaemon, but by default they're just printed to stdout. Consult AppDaemon's documentation for details.

You'll need to watch the log often as you proceed with this tutorial, so make sure you know how to do it.

3.2.4 Configuring Some Heating Times

Obviously, schedules are the most powerful part of Schedy. Unfortunately, that means they can get a little complex when advanced features are used heavily. This tutorial just configures simple heating times, but you may need to have a comprehensive look at the *chapter about schedules* at some point.

We want to keep it simple for now. During nights or when no other temperature has been configured, the heating should be turned off in all rooms.

As schedules are evaluated rule by rule from top to bottom until a matching rule was found, we create a new rule as fallback at the end of each room's schedule. But wait, that would be redundant! Fortunately, there is the `schedule_append` section we can use to append something to the schedules of all rooms at once. This goes into our config:

```
schedule_append:
- v: "OFF"
```

Now, each room gets its own heating times.

1. Living room:

   ```
   schedule:
   # We set different heating times for weekdays and weekends.
   - { v: 20, start: "06:00", end: "07:30", weekdays: 1-5 }
   - { v: 20, start: "15:00", end: "22:30", weekdays: 1-5 }
   - { v: 20, start: "08:00", end: "23:30", weekdays: 6-7 }
   ```

2. Bedroom:

   ```
   schedule:
   # The bedroom should always have 14 degrees to sleep well in there.
   - v: 14
   ```

3. Kids rooms:

   ```
   # We use the exact same schedule for both kids1 and kids2.
   schedule:
   - { v: 20, start: "06:00", end: "07:30", weekdays: 1-5 }
   - { v: 20, start: "15:00", end: "19:00", weekdays: 1-5 }
   - { v: 20, start: "07:30", end: "20:00", weekdays: 6-7 }
   ```

Now save the configuration and watch your new schedules in action. You can play with the times of some rules and change them back and forth to verify Schedy applies everything correctly.

Grouping Similar Rules Into Sub-Schedules

The schedules we created so far work fine, but they are quite verbose and contain some redundancy. Let's utilize a cool feature of Schedy to get rid of that redundancy and make our rules more concise: *sub-schedules*.

The only rooms this really makes sense for are the living room and the kids rooms, as they contain multiple rules with common properties (like `v` and `weekdays`).

1. Living room:

   ```
   schedule:
   - v: 20
     rules:
     - weekdays: 1-5
       rules:
       - { start: "06:00", end: "07:30" }
       - { start: "15:00", end: "22:30" }
     - weekdays: 6-7
       rules:
       - { start: "08:00", end: "23:30" }
   ```

2. Kids rooms:

```
schedule:
- v: 20
  rules:
  - weekdays: 1-5
    rules:
    - { start: "06:00", end: "07:30" }
    - { start: "15:00", end: "19:00" }
  - weekdays: 6-7
    rules:
    - { start: "07:30", end: "20:00" }
```

You see that the schedules didn't get shorter, but we now have a clear hirarchy of rules and don't need to repeat `v` and `weekdays` over and over anymore. Structuring your schedules this way is by no means required, but it does increase readability and maintainability as your schedules get more complex. Some sophisticated features can take even more advantage of sub-schedules, as you'll see later.

Consolidating the Kids Rooms' Schedules

The schedules for both kids rooms are identical. It would be nice to have the schedule only once. We use the *schedule snippets feature* and create a schedule snippet named `"kids"`:

```
schedule_snippets:
  kids:
  - v: 20
    rules:
    - weekdays: 1-5
      rules:
      - { start: "06:00", end: "07:30" }
      - { start: "15:00", end: "19:00" }
    - weekdays: 6-7
      rules:
      - { start: "07:30", end: "20:00" }
```

Now, we include that snippet in the schedules of the kids rooms:

```
schedule:
- x: "IncludeSchedule(schedule_snippets['kids'])"
```

Done!

3.2.5 Adding Window Sensors

We're just following the *official guide for open window detection* here.

The rule which turns the heatings off when a window is open is placed in the `schedule_prepend` section:

```
schedule_prepend:
- x: "Mark(OFF, Mark.OVERLAY) if not is_empty(filter_entities('binary_sensor', state=
↪'on', window_room=room_name)) else Next()"
```

Why that rule works as it does is explained in more detail in the guide linked above.

We now map our sensors to the rooms they belong to with help of `customize.yaml`:

```
binary_sensor.living_window_1:
  window_room: living
binary_sensor.bed_window_1:
  window_room: bed
binary_sensor.kids1_window_1:
  window_room: kids1
binary_sensor.kids2_window_1:
  window_room: kids2
```

Adding more than one sensor per room would be very simple, as you can see.

Finally, we tell Schedy to re-evaluate the room's schedule when a sensor changes its state. For that, we just add them to the `watched_entities` lists of the particular rooms. Here is an example for `living`, the others are analogous:

```
watched_entities:
- binary_sensor.living_window_1
```

3.2.6 Automatic Re-Scheduling After Manual Adjustments

It would be cool to be able to change the temperature in a room unplanned and have Schedy apply the regular schedule again after some period of time. For this purpose, there is the `rescheduling_delay` setting that can be set per room.

Let's enable it in living room and bedroom and set it to two hours (120 minutes):

```
living:
  rescheduling_delay: 120
  # ...

bed:
  rescheduling_delay: 120
  # ...
```

3.2.7 Stopping the Kids From Playing With the Thermostats

Our kids are still young and hit every button they can reach. Why not fix the temperature in the kids rooms to what is dictated by the schedule? We disable `allow_manual_changes` and Schedy will revert any manual change as soon as it's performed:

```
kids1:
  allow_manual_changes: false
  # ...

kids2:
  allow_manual_changes: false
  # ...
```

3.2.8 Switching Schedules as Needed

Wouldnt it be nice to be able to switch the schedules when, for instance, we have holidays and are home over the day? Nothing simpler than that with Schedy.

We add an `input_select` in Home Assistant:

```
input_select:
  heating_mode:
    name: Heating Mode
    options:
    - Normal
    - Parents Home
    - All Home
```

Then, we adapt the schedules accordingly. The pattern we follow is *this one*, should you need help understanding what's going on here.

1. Living room:

```
schedule:
- v: 20
  rules:
  - weekdays: 1-5
    rules:
    - rules:
      - x: "Next() if state('input_select.heating_mode') == 'Normal' else Break()"
      - { start: "06:00", end: "07:30" }
      - { start: "15:00", end: "22:30" }
    - rules:
      - x: "Next() if state('input_select.heating_mode') != 'Normal' else Break()"
      - { start: "08:00", end: "23:30" }
  - weekdays: 6-7
    rules:
    - { start: "08:00", end: "23:30" }
```

2. Kids rooms:

```
schedule_snippets:
  kids:
  - v: 20
    rules:
    - weekdays: 1-5
      rules:
      - rules:
        - x: "Next() if state('input_select.heating_mode') != 'All Home' else␣
↪Break()"
        - { start: "06:00", end: "07:30" }
        - { start: "15:00", end: "19:00" }
      - rules:
        - x: "Next() if state('input_select.heating_mode') == 'All Home' else␣
↪Break()"
        - { start: "07:30", end: "20:00" }
    - weekdays: 6-7
      rules:
      - { start: "07:30", end: "20:00" }
```

Don't forget to add `input_select.heating_mode` to the list of entities watched for state changes. Instead of adding it to all three concerned rooms, we simply add it to the global list and have it count for all rooms:

```
watched_entities:
- input_select.heating_mode
```

3.2.9 Using `expression_environment` to Make Rules More Concise

We've got four schedule rules with expressions that all use `state('input_select.heating_mode')` to query the heating mode currently selected from Home Assistant. This is quite repetitive and makes the rules long and unwieldy.

There is the `expression_environment` setting, which allows us to built custom Python objects we can then use in all our rule expressions. We utilize this functionality and create a new function, `heating_mode()`:

```
expression_environment: |
  def heating_mode():
      return state("input_select.heating_mode")
```

The individual rules then change to something like:

```
- x: "Next() if heating_mode() == 'All Home' else Break()"
```

The remaining ones are left to do for you.

3.2.10 Final Configuration

Here is the final outcome of our work as a full Schedy configuration.

```
schedy_heating:  # This is our app instance name.
  module: hass_apps_loader
  class: SchedyApp

  actor_type: thermostat

  expression_environment: |
    def heating_mode():
        return state("input_select.heating_mode")

  schedule_snippets:
    kids:
    - v: 20
      rules:
      - weekdays: 1-5
        rules:
        - rules:
          - x: "Next() if heating_mode() != 'All Home' else Break()"
          - { start: "06:00", end: "07:30" }
          - { start: "15:00", end: "19:00" }
        - rules:
          - x: "Next() if heating_mode() == 'All Home' else Break()"
          - { start: "07:30", end: "20:00" }
      - weekdays: 6-7
        rules:
        - { start: "07:30", end: "20:00" }

  watched_entities:
  - input_select.heating_mode

  schedule_prepend:
  - x: "Mark(OFF, Mark.OVERLAY) if not is_empty(filter_entities('binary_sensor',␣
↪state='on', window_room=room_name)) else Next()"
```

(continues on next page)

(continued from previous page)

```
schedule_append:
- v: "OFF"

rooms:

  living:
    rescheduling_delay: 120
    actors:
      climate.living_1:
      climate.living_2:
    watched_entities:
    - binary_sensor.living_window_1
    schedule:
    - v: 20
      rules:
      - weekdays: 1-5
        rules:
        - rules:
          - x: "Next() if heating_mode() == 'Normal' else Break()"
          - { start: "06:00", end: "07:30" }
          - { start: "15:00", end: "22:30" }
        - rules:
          - x: "Next() if heating_mode() != 'Normal' else Break()"
          - { start: "08:00", end: "23:30" }
      - weekdays: 6-7
        rules:
        - { start: "08:00", end: "23:30" }

  bed:
    rescheduling_delay: 120
    actors:
      climate.bed_1:
    watched_entities:
    - binary_sensor.bed_window_1
    schedule:

  kids1:
    allow_manual_changes: false
    actors:
      climate.kids1_1:
    watched_entities:
    - binary_sensor.kids1_window_1
    schedule:
    - x: "IncludeSchedule(schedule_snippets['kids'])"

  kids2:
    allow_manual_changes: false
    actors:
      climate.kids2_1:
    watched_entities:
    - binary_sensor.kids2_window_1
    schedule:
    - x: "IncludeSchedule(schedule_snippets['kids'])"
```

And the Home Assistant part:

```
customize:
  binary_sensor.living_window_1:
    window_room: living
  binary_sensor.bed_window_1:
    window_room: bed
  binary_sensor.kids1_window_1:
    window_room: kids1
  binary_sensor.kids2_window_1:
    window_room: kids2

input_select:
  heating_mode:
    name: Heating Mode
    options:
    - Normal
    - Parents Home
    - All Home
```

3.2.11 Ok, And Now?

Enjoy your new, powerful schedules! Consult the following chapters for more detailed information on *creating advanced rules*, *supported actor types*, *events* and *statistics collection*. The *Tips & Tricks* chapter may give you some more inspiration after all.

3.3 Configuration

There is no graphical user interface for configuring Schedy. You'll have to express your configuration and schedules in a YAML file. However, if you've ever written an automation or script in Home Assistant, this is nothing you should be worried about.

This is a full configuration example with comments on each available setting. Copy it to a `.yaml` file in your `apps` directory and adapt it to your needs. The default values are provided as well. If you don't need a particular setting, omit it or leave it commented out.

```
schedy:  # An arbitrary name for this instance of Schedy,
         # needed e.g. for addressing events to it.
  # Obligatory settings that tell appdaemon where to find the app.
  # You shouldn't need to change these two.
  module: hass_apps_loader
  class: SchedyApp

  # Enable debugging output
  #debug: false


  # By default, Schedy tries to restore its previous state at startup
  # from the records stored in Home Assistant. This includes scheduled
  # values, manual adjustments and re-scheduling times - just everything.
  # When this behaviour is not desired, change this setting to true
  # and Schedy will just apply the schedules at startup, no matter what
  # the previous actor states were.
  #reset_at_startup: false
```

(continues on next page)

(continued from previous page)

```
  # If you enable this option, potentially harmful expressions received
  # in schedy_set_value events are evaluated.
  #expressions_from_events: false

  # This Python script is executed when a schedule rule expression needs to be
  # evaluated. All modules you import and variables you set here will be available␣
↪when
  # your expression is evaluated. You can even declare classes and functions. The
  # result types and expression helpers can be used in this script as well.
  # Note: Don't expect this script to be executed as often as the number of rules
  # you have in your schedules. The expression evaluation environment is built the
  # first time a rule needs to be evaluated and then reused for all subsequent
  # rules during that schedule evaluation turn.
  #expression_environment: |
  #  import math
  #  import time as _time
  #  something = "value"


  # Chose the type of actors that should be controlled by this instance
  # of Schedy.
  # Possible types can be found in the documentation.
  #actor_type: <required>

  # In the following config block, you may define templates with
  # settings that affect multiple actors in your setup. These can then be
  # used as a starting point for the configuration of individual actors.
  # Nested inclusion of templates is possible as well.
  actor_templates:

    # By default, an actor inherits its settings from the "default" template.
    default:

      # Optionally have this template based on another one.
      #template: other_template

      # Actor type-specific settings can be found in the documentation.
      #foo: bar
      #...

    # More templates ...
    #other_template:
    # ...


  # Add schedule rules you want to have prepended to each room's schedule
  # automatically here.
  schedule_prepend:

  # Add schedule rules you want to have appended to each room's schedule
  # automatically here.
  schedule_append:

  # Optionally, configure schedule snippets (lists of rules) that can
  # be included by expressions dynamically. See the documentation for
  # an example on how to use them.
```

(continues on next page)

(continued from previous page)

```
schedule_snippets:

  #summer:
  #- ...


# When you use expressions in your schedules that query the state of
# entities, you should tell Schedy which entities the schedules depend
# on. It can then watch for state changes of these and re-evaluate
# schedules automatically.
# Note: Entities listed here trigger a re-evaluation in all rooms. For
# entities only used in the schedules of particular rooms, use the
# per-room watched_entities list.
watched_entities:

  # Each entry has to be a string consisting of up to three
  # colon-separated parts:
  #
  # 1. The id of the entity to watch. This is mandatory.
  # 2. Which attributes to watch for changes of, either a single
  #    attribute or a comma-separated list.
  #    The special value "all" listens for changes of any attribute. Try
  #    to avoid this whenever possible as it can increase the load
  #    significantly.
  #    Default: "state"
  # 3. The re-evaluation mode as known from the schedy_reevaluate event.
  #    The special mode "ignore" causes no re-evaluation, it just
  #    suppresses the warnings generated when the entity is queried from
  #    an expression.
  #    Default: "reevaluate"
  #
  # Examples:
  # - "binary_sensor.motion"
  # - "binary_sensor.motion:all"
  # - "binary_sensor.motion::reset"
  # - "binary_sensor.motion:state,other_attribute:reset"
  # - "binary_sensor.motion:all:ignore"


# Configure your rooms here.
rooms:

  # Create such a block for every room you want to control.
  #living:

    # An alternative friendly name to display in logs.
    #friendly_name: ...

    # When you disable this setting, Schedy won't allow actors to
    # change their value to something different from the scheduled
    # one or the one set by a schedy_set_value event. Actors
    # that change their value are then set back to the wanted one
    # immediately.
    #allow_manual_changes: true

    # This setting controls whether changes reported by one actor
    # should automatically be replicated to the other ones in this
```

(continues on next page)

(continued from previous page)

```
    # particular room.
    #replicate_changes: true

    # Set this value to a number of minutes and Schedy will
    # automatically again apply the schedule after a manual change has
    # been made. If you, for instance, use the thermostat actor type,
    # change the target temperature at one of your thermostats and
    # this value is set to 120, Schedy will again apply the schedule
    # two hours after you made the change.
    # 0 means not re-schedule before the next scheduled value change
    # occurs.
    #rescheduling_delay: 0

    # All actors of this room go here.
    #actors:

      # This could be a thermostat.
      # NOTE: Don't forget the colon after the entity id.
      #climate.living1:

        # Choose the template this actor should inherit its settings
        # from.
        # By default, an actor inherits its settings from the "default"
        # template, given that you defined it.
        #template: default

        # We could, for instance, overwrite the delta defined in the
        # template for this particular thermostat only.
        #delta: 0.0

    # The room's schedule, consisting of multiple rules.
    schedule:
    #- ...

    # The same as the global watched_entities above, but these only
    #  trigger a re-evaluation for this particular room.
    watched_entities:
    #- ...


# Configure statistical parameters to be collected.
statistics:

  # Pick an arbitrary name for the parameter instance.
  #some_name:
    # The type of parameter as found in the actor'S documentation.
    #type: <required>
    # More parameter-specific settings ...

  # More parameter instances ...
```

3.4 Actors

Schedy supports controlling different types of actors such as thermostats or switches.

You first need to specify the desired actor type at the top level of Schedy's configuration:

```
actor_type: <name of actor type>
```

Then go on and add actors to your rooms. The available configuration parameters and supported values for scheduling are explained on the actor-specific pages.

Note: You have to decide for one actor type per instance of Schedy you run. If you need to control different types of actors, create an instance of Schedy for each, like so.

```
schedy_lights:
  module: hass_apps_loader
  class: SchedyApp
  actor_type: switch
  # ...

schedy_heating:
  module: hass_apps_loader
  class: SchedyApp
  actor_type: thermostat
  # ...
```

Of course, the same room names may then be used in each of these app instances, since they run completely independent of each other.

Currently, the following actor types are available:

3.4.1 Custom Actor

Warning: This feature is experimental. When you're using it, feedback is very welcome.

Note: This is a topic targeted at advanced users. It might be hard to understand for newcomers.

The `custom` actor can be used if maximum control and flexibility is required, as it allows you to write custom hooks (pieces of Python code) that link schedule results to entity states. In fact, you could even implement advanced types like *Thermostat* with this one.

While this actor is probably not for daily use, it gives you the power you need when implementing something really fancy.

Note: When you're extensively using the custom actor type for something that could be interesting to other people as well, please consider filing your idea as an issue on GitHub to maybe get it included in Schedy natively. Thank you!

Understanding the Custom Actor

The purpose of every actor type is to provide a mapping between values generated by a schedule and states of entities. These two terms, **value** and **state**, are crucial for understanding how the custom actor works.

A value returned by a schedule may in fact be any Python object, be it a string, number, boolean or even a custom type. The work to be done by an implementation of the custom actor type is then to execute the Home Assistant services needed to reach the state you want a particular value to stand for. This work has to be acomplished in the so-called `send` hook, which is just normal Python code in which you can, for example, call services.

Note: You need to realize that values and states are two different things your custom actor implementation needs to link to each other..

However, the same work has to be done in the other direction as well. It needs to be possible to map a given entity state to the value that, when returned from a schedule, would cause that state to be achieved. This is handled by the `state` hook. It gets all state attributes of the watched entity and must return the value this state is caused by.

These two hooks, the `send` and the `state` hook, are executed similarly to the *expressions used in schedules*. Both simple expressions (single-line) and whole statements (multi-line) are possible. When using whole statements, the result has to be stored in the global `result` variable as usual.

Inside the hooks, the following variables are available for you to work with:

- `state` or `value`: The input depending on the type of hook.
- `entity_id`: The actor's entity id.
- `config`: The custom configuration dictionary as defined in the actor configuration.
- `app`: The `appdaemon.plugins.hass.hassapi.Hass` object to be used for calling services etc.
- `actor`: The `CustomActor` object. The only purpose I could imagine for using this object directly is for generating custom log messages, e.g. for debugging purposes. You could do:

```
actor.log("I'm going to send the value {}."
          .format(repr(value)),
          level="DEBUG")
```

Configuration

This is a full actor configuration with comments on each available setting. The default values are provided as well. If you don't need a particular setting, omit it or leave it commented out.

```
# This hook should perform all actions required for moving the actor
# to the state corresponding to the value given as "value". The result
# of the hook isn't respected. You'll probably use this for calling
# some services.
#send_hook: <required>

# This hook is executed when a state update is received from the
# watched entity. It has the dictionary with all received state attributes
# available in the variable "state". The result has to be the scheduling
# value corresponding to this state. If the result is None, the state
# change is ignored.
# When you don't define a state hook, the actor doesn't react to state
# changes at all. Inferring values from states is then impossible, which
# also disables change replication between actors in a room. Should you
# really decide to not map states back to values, make sure you set the
# "send_retries" setting to a low value, because the actor won't be able
# to notice when it has reached the desired state and always send until
# the configured number of retries is exceeded.
#state_hook: ...
```

(continues on next page)

(continued from previous page)

```
# This hook is optional and may be used to preprocess a value generated
# by scheduling before it is stored and passed on to the send hook. The
# value is available under the name "value". The result of this hook
# has to be the updated value or None, in which case the value change
# is discarded and no send hook is executed.
#filter_value_hook: ...

# The config parameter is a dictionary which is available in all hooks
# under the name "config". It can be filled with anything you like and
# hence is useful for reusing a custom actor template, each time with
# different settings. You may want to store things like attribute names
# to be used by your custom hooks in the config dictionary.
# It's empty by default.
config:
```

3.4.2 Generic Actor

Warning: This actor type has been superseeded by the *Generic Actor Version 2*. Use that instead.

The `generic` actor can be used for controlling different types of entities like numbers or media players, even those having multiple adjustable attributes such as roller shutters with tilt and position.

It works by defining a set of values and, for each of these values, what service has to be called in order to reach the state represented by that value. Together with a wildcard for undefined values, this is a quite powerful mechanism.

Instead of a single value such as `"on"` or `"off"`, you may also generate a tuple with multiple values like `(50, 75)` or `("on", 10)` in your schedule rules, where each slot in that tuple corresponds to a different attribute of the entity.

Configuration

This is a full actor configuration with comments on each available setting. The default values are provided as well. If you don't need a particular setting, omit it or leave it commented out.

```
# List the entity attributes to be controlled by this actor.
attributes:

- # The attribute of the entity to be watched for state changes.
  # Use "state" for the entity state or any other attribute from the
  # attributes dictionary.
  # When you set this to null, you define a write-only attribute which's
  # current value isn't reflected in the entity's state and thus can't be
  # determined by Schedy. The configured service is called when setting a
  # value, but the value can never be considered committed and re-sending
  # is done according to the send_retries and send_retry_interval settings.
  #attribute: null

  # Here, the possible values of that attribute are configured.
  # Values can be floats, integers, strings or null.
  values:

    #some_value:
```

(continues on next page)

(continued from previous page)

```
      # The service that needs to be called in order to make the attribute
      # show this value.
      #service: <required>

      # The data to be passed to the service.
      #service_data: {}

      # Whether to include the entity id as "entity_id" in the service data.
      #include_entity_id: true

      # The parameter as which the actual value should be passed in the
      # service data.
      # null means not include the value in the service data.
      #value_parameter: null

    #other_value:
      # ...

    # The keyword "_other_" stands for a so-called wildcard value that
    # matches any value not explicitly defined here. You may want to use
    # the wildcard value to catch arbitrary numbers, for example.
    # If you define all possible values explicitly and don't need a wildcard,
    # simply leave it out.
    #_other_:
      # ...

- # Second attribute...


# By default, services for changing the different attributes are called
# in the order you defined the attributes. Set this flag to true to have
# the order in which services are called reversed.
#call_reversed: false


# For some types of actors that need multiple attributes to be
# controlled, there are sometimes attributes that aren't important in
# a particular state. For instance, a light that's turned off doesn't
# care about the brightness or color set.
# For these kind of actors, you can configure short values: values which
# only need to have the first N attributes set.
# You of course need to configure the attributes in an order that makes
# sense, with mandatory ones first and state-specific ones later.
short_values:
# When the first attribute is set to "off", don't consider further
# attributes.
- ["off"]
# When the first attribute is set to "on" and the second's value is
# something catched by the wildcard value, ignore further attributes.
- ["on", "_other_"]
```

Supported Values

The generic actor can be used in two ways. When just a single attribute should be controlled, every value for which a service has been configured in the `values` section of the actor configuration may be returned by a schedule. If you

have the wildcard value `_other_` configured, any value is accepted.

Examples:

```
- v: "on"
- x: "-40 if is_on(...) else Next()"
```

As soon as you add multiple attributes to control, a list or tuple with a value for each attribute is expected. The order is the same in which the attributes were specified in the configuration.

Examples:

```
- v: ['on', 20]
- x: "(-40, 'something') if is_on(...) else Next()"
```

Note: When specifying the values `on` and `off`, enclose them in quotes as shown above to inform the YAML parser you don't mean the booleans `True` and `False` instead.

3.4.3 Generic Actor Version 2

The `generic2` actor can be used for controlling different types of entities like numbers or media players, even those having multiple adjustable attributes such as roller shutters with tilt and position.

It works by defining a set of values and, for each of these values, what services have to be called in order to reach the state represented by that value.

Instead of a single value such as `"on"` or `"off"`, you may also generate a tuple of multiple values like `(50, 75)` or `("on", 10)` in your schedule rules, where each slot in that tuple corresponds to a different attribute of the entity.

If you want to see how this actor type can be used, have a look at the *Switch*.

Configuration

This is a full actor configuration with comments on each available setting. The default values are provided as well. If you don't need a particular setting, omit it or leave it commented out.

```
# Here you configure the attributes of the entity to be controlled by the schedule.
attributes:
  # The attribute to be controlled, this could be e.g. "state" or "brightness".
  # A value of null creates a write-only attribute. This has to be used when you want
  # to control a property whose current value is not reflected in any of the entity's
  # attributes. Don't do this if not really necessary, since doing so means that
  # Schedy won't be able to verify that the value has been transmitted correctly. If
  # you must use a write-only attribute, you might also want to set send_retries to a
  # low value in order to avoid excessive network load.
- attribute: first
- attribute: second
- ...

# Here you configure the values you want to be able to return from your schedule.
values:
  # Each value is a list of the values for the individual attributes configured above.
  # Schedy compares the entity's current attributes against the values defined here
  # in order to find the value currently active.
  # The special attribute value "*" is a wildcard and will, when used, match any
```

(continues on next page)

(continued from previous page)

```
  # value of that particular attribute.
  # Additionally, you don't have to include all attributes in every single value,
  # only the first N attributes which values are provided for are compared against
  # the entity's state for the value to match.
- value: ["on", "*"]
  # The services that have to be called in order to make the actor report this value.
  calls:
    # Which service to call
  - service: ...
    # Optionally, provide a mapping with data to be passed with the service call.
    # You can use "{attr1}" as a placeholder for the value set for the first␣
↪attribute,
    # "{attr2}" for the value of the second attribute and so on to pass the correct
    # attribute values to the service call as needed in order to bring the entity
    # to the state represented by the value you returned from your schedule.
    # The placeholder "{entity_id}" can be used to insert the actor's entity id.
    # For instance, if the value
    #   ["on", 75]
    # was returned by a schedule, the following sample would render to:
    #   {"param1": "something", "param2": 75}
    data:
      param1: "something"
      param2: "{attr2}"
    # Set to false if you don't want the entity_id field to be included in service
    # data automatically.
    #include_entity_id: true

# More values#
- ...

# Set this to true if you want Schedy to treat string attributes of an entity the
# same, no matter if they're reported in lower or upper case. This is handy for some
# MQTT devices, for instance, which sometimes report a state of "ON", while others say
# "on".
#ignore_case: false
```

Supported Values

Every value that has been configured in the `values` section of the actor configuration may be returned from a schedule.

Examples:

```
- v: "on"
- x: "-40 if is_on(...) else Next()"
```

As soon as you configure multiple slots (attributes to be controlled), a list or tuple with a value for each attribute is expected. The order is the same in which the slots were specified in the configuration.

Examples:

```
- v: ['on', 20]
- x: "(-40, 'something') if is_on(...) else Next()"
```

Note: When specifying the values `on` and `off`, enclose them in quotes as shown above to inform the YAML parser

you don't mean the booleans `True` and `False` instead.

3.4.4 Switch

The `switch` actor is used to control binary on/off switches. Internally, it's a *Generic Actor Version 2*, but with a much simpler configuration, namely none at all.

Note: It calls the generic `homeassistant.turn_on` and `homeassistant.turn_off` services and hence can as well be used for other entity types supporting to be turned on and off this way. However, they need to provide `"on"` and `"off"` as their `state`.

Especially, this is true for `input_boolean` and `light` entities.

For completeness, this is the configuration you had to use if you wanted to build this switch actor out of the *Generic Actor Version 2* yourself:

```
actor_type: generic2
actor_templates:
  default:
    attributes:
    - attribute: state
    values:
    - value: ["on"]
      calls:
      - service: homeassistant.turn_on
    - value: ["off"]
      calls:
      - service: homeassistant.turn_off
    ignore_case: true
```

Supported Values

You need to return the strings `"on"` or `"off"` from your schedules for the switch actor to work. It's that simple.

3.4.5 Thermostat

The `thermostat` actor is used to control the temperature of climate entities.

Often, people ask me whether Schedy can be used with their particular heating setup. I always tend to repeat myself in these situations, hence I want to explain here what the exact preconditions for using Schedy for heating control actually are.

1. You need at least one thermostat in each room you want to control. Such a thermostat must be recognized as a climate entity in Home Assistant, and setting the target temperature from the Home Assistant web interface should work reliably. Wall thermostats can be controlled the same way as radiator thermostats, as long as they fulfill these conditions as well. If you only have a switchable heater and an external temperature sensor, have a look at Home Assistant's Generic Thermostat platform to build a virtual thermostat first.
2. If your thermostat is used for both heating and cooling, there has to be an automatic HVAC mode which does heating/cooling based on the difference between current and set target temperature. Schedy will only switch the HVAC mode between on and off (exact names can be configured) and set the target temperature according to the room's schedule.

If you are happy with these points and your setup fulfills them, there should be nothing stopping you from integrating Schedy's great scheduling capabilities with your home's heating. You can then go on and create a Schedy configuration with thermostat actors.

Configuration

This is a full actor configuration with comments on each available setting. The default values are provided as well. If you don't need a particular setting, omit it or leave it commented out.

```
# Delta that is added to the temperature value sent to this
# thermostat in order to correct potential inaccuracies of
# the temperature sensor.
#delta: 0

# The minimum/maximum temperature the thermostat supports.
# If configured, temperatures outside the supported range are changed
# to the minimum/maximum value before they're sent to the thermostat.
# null means there is no limitation.
#min_temp: null
#max_temp: null

# When this is set to something different than "OFF", Schedy will
# rewrite the value OFF into this temperature before sending it to
# the thermostat. You can set it to 4.0 degrees (if your thermostat
# supports this low value) in order to prevent frost-induced damage
# to your heating setup.
# This setting is required when you want to send OFF to thermostats with
# disabled HVAC mode support.
#off_temp: "OFF"

# Set this to false if your thermostat doesn't support HVAC modes.
# Please note that you won't be able to turn it off completely without
# HVAC mode support. Remember to also configure off_temp when you
# disable this feature.
#supports_hvac_modes: true

# These two settings can be used to tweak the names of the HVAC modes.
#hvac_mode_on: heat
#hvac_mode_off: "off"
```

Supported Values

Your schedules must generate valid temperature values. Those can be integers (`20`) or floats (`21.5`). Strings are tried to be converted to numbers automatically for you.

A special value is `OFF`, which is an object available in the evaluation environment when using the thermostat actor type. If this object is returned from an expression, it will turn the thermostats off. The equivalent for the `OFF` object to use when using plain values instead of expressions is the string `"OFF"` (case-insensitive).

Note: When working with the `Add()` *postprocessor* and the result is `OFF`, it will stay `OFF`, no matter what's being added to it.

Statistical Parameters

The following statistical parameters are available when using this actor type. In order to learn how to configure a statistical parameter, see the chapter about *Statistics*.

`temp_delta`

This parameter measures the difference between target and current temperature for all thermostats in the associated rooms. It can be used to control a source of heating energy, such as a fuel oven, with Home Assistant automations.

Options provided because this is a `TempDeltaParameter`:

- `off_value`: Specify how to handle thermostats which are turned off. Specify either the number to assume as the delta or `null`, which causes the thermostat to be excluded from statistics collection. The default value is `0`.

Options provided because this is a `RoomBasedParameter`:

- `rooms`: A list of rooms this parameter should consider. When you leave the list empty, all rooms are considered.

Note: When you see something like `some_id` or `other_id` in the following examples, these are meant to be replaced by entity ids of individual actors.

The parameter calculates the minimum, average and maximum of all collected values and adds them as `min`, `avg` and `max` attributes to the parameter entity.

Reacting to changes of the attributes can easily be done with the `numeric_state` trigger, together with a `value_template` like `{{ state.attributes.max }}` in Home Assistant.

Options provided because this is a `MinAvgMaxParameter`:

- `factors`: Specify a factor which an individual value should be multiplied with before adding it to the list of values. Note that this doesn't change the weighting of a value for calculating the average, it instead changes the value itself. The default factor is `1`.

```
factors:
  some_id: 1.5
  other_id: 2
```

- `weights`: Specify how individual values should be weighted when calculating the average value. The default weight is `1` and a weight of `0` causes the value to be excluded completely. You may want to use this feature to indicate that some values are more or less important than others and have this fact reflected in the statistics.

```
weights:
  some_id: 0.5
  other_id: 0
```

3.4.6 Common Settings

There are some settings common among all available actor types.

```
# An alternative friendly name to display in logs.
#friendly_name: ...

# This setting tells Schedy how often it should try sending a
# value to the actor. If the actor reports the set value back, no
```

(continues on next page)

(continued from previous page)

```
# further retry is made. You may find this useful if the connection
# between Home Assistant and your actor is unreliable. Set to 0 in
# order to disable retrying entirely.
#send_retries: 10
# How many seconds to wait before retrying.
#send_retry_interval: 30
```

3.5 Schedules

A schedule controls the state of actors in a room. In its simplest form, this means specifying which state should be set at which times statically, like in a timetable.

However, this is not flexible enough for more sophisticated needs, which is why schedules can be extended with dynamic rules, turning them into Python scripts that can, for instance, access the state of Home Assistant entities easily.

To get started, begin with static schedules. Once you feel comfortable writing them, you may proceed to dynamic expressions.

3.5.1 The Basics: Static Schedules

A schedule controls the state of actors in a room over time. It consists of a set of rules. What these rules define is dependent upon the type of actor. Our examples here use the `thermostat` actor type and hence define temperatures.

Each rule must at least define a value:

```
schedule:
- value: 16
```

This schedule would just always set the temperature to `16` degrees, nothing else. Of course, schedules wouldn't make a lot sense if they couldn't do more than this.

For `value`, there is a shortcut `v` to make rules more compact. We'll use that from now on.

Scheduling Based on Time of the Day

Here is another one:

```
schedule:
- v: 21.5
  start: "7:00"
  end: "22:00"
  name: Fancy Rule
- v: 16
```

This schedule shares the 16 degrees rule with the previous one, but additionally, it got a new rule at the top. The new first rule overwrites the second and will set a temperature of `21.5` degrees, but only from 7.00 am to 10.00 pm. This is because it's placed before the 16 degrees-rule and Schedy evaluates rules from top to bottom. From 10.00 pm to next day 7.00 am, the `16` degrees do still apply.

Note: This is how schedules work. The first matching rule wins and determines the value to set. Consequently, you should design your schedules with the most specific rules at the top and gradually generalize to wider time frames

towards the bottom. Finally, there should be a fallback rule without time restrictions at all to ensure you have no time slot left without a value defined for.

The `name` parameter we specified here is completely optional and doesn't influence how the rule is interpreted. A rule's name is shown in logs and may be useful for troubleshooting.

For more fine-grained control, you may also specify seconds in addition to hour and minute. `22:00:30` means 10.00 pm + 30 seconds, for instance. Spanning rules beyond midnight (`start` >= `end`) is possible as well.

You can now write rules that specify the value over the day, but you still can't create different schedules for, for instance, the days of the week. Let's do this next.

Constraints

```
schedule:
- v: 22
  weekdays: 1-5
  start: "7:00"
  end: "22:00"

- v: 22
  weekdays: 6,7
  start: "7:45"

- v: 15
```

With your knowledge so far, this should be self-explanatory. The only new parameter is `weekdays`, which is a so called constraint.

Constraints can be used to limit the days on which the rule should start to be active. There are a number of these constraints, namely:

- `years`: limit the years (e.g. `years: 2016-2018`); only years from 1970 to 2099 are supported
- `months`: limit based on months of the year (e.g. `months: 1-3, 10-12` for Jan, Feb, Mar, Oct, Nov and Dec)
- `days`: limit based on days of the month (e.g. `days: 1-15, 22` for the first half of the month + the 22nd)
- `weeks`: limit based on the weeks of the year
- `weekdays`: limit based on the days of the week, from 1 (Monday) to 7 (Sunday)
- `start_date`: A date of the form `{ year: 2018, month: 2, day: 3 }` before which the rule should not be considered. Any of the three fields may be omitted, in which case the particular field is populated with the current date at validation time. If an invalid date such as `{ year: 2018, month: 2, day: 29 }` is provided, the next valid date (namely 2018-03-01 in this case) is assumed.
- `end_date`: A date of the form `{ year: 2018, month: 2, day: 3 }` after which the rule should not be considered anymore. As with `start_date`, any of the three fields may be omitted. If an invalid date such as `{ year: 2018, month: 2, day: 29 }` is provided, the nearest prior valid date (namely 2018-02-28 in this case) is assumed.

A date needs to fulfill all constraints you defined for a rule to be considered active at that specific date.

The format used to specify values for the first five types of constraints is similar to that of crontab files. We call it range specification, and only integers are supported, no decimal values.

- `x`: the single number `x`

- `x-y` where `x < y`: range of numbers from `x` to `y`, including `x` and `y`
- `x-y/z` where `x < y`: range of numbers from `x` to `y`, including `x` and `y`, going in steps of `z`
- `*`: range of all numbers
- `*/z`: range of all numbers, going in steps of `z`
- `a,b`, where `a` and `b` are any of the previous: the numbers represented by `a` and `b` joined together
- ... and so on
- Any spaces are ignored.

If an exclamation mark (`!`) is prepended to the range specification, its values are inverted. For instance, the constraint `weekdays: "!4-5,7"` expands to `weekdays: 1,2,3,6` and `months: "!3"` is equivalent to `months: 1-2,4-12`.

Note: The `!` sign has a special meaning in YAML, hence inverted specifications have to be enclosed in quotes.

Rules Spanning Multiple Days

Now let's come back to the 16-degrees rule we wrote above and figure out why that actually counts as a fallback for the whole day. Here's the rule we have so far.

```
- v: 16
```

If you omit the `start` parameter, Schedy assumes that you mean midnight (`0:00`) and fills that in for you. When `end` is not specified (as has been done here), Schedy sets `0:00` for it as well. However, a rule that ends the same moment it starts at wouldn't make sense. We expect it to count for the whole day instead.

In order to express what we actually want, we'd have to set `end` to `"00:00+1d"`, which tells Schedy that there is one midnight between the start and end times. For convenience, Schedy automatically assumes one midnight between start and end when you don't specify a number of days explicitly and the start time is prior or equal to the end time, as in our case.

Note: You don't need to care about setting `+?d` yourself unless one of your rules should span more than 24 hours, requiring `+1d` or greater.

Having written out what Schedy assumes automatically would result in the following rule, which behaves exactly identical to what we begun with.

```
- { v: 16, start: "0:00", end: "0:00+1d" }
```

Note: The rule has been rewritten to take just a single line. This is no special feature of Schedy, it's rather normal YAML. But writing rules this way is often more readable, especially if you need to create multiple similar ones which, for instance, only differ in weekdays, time or value.

Let's get back to *Constraints* briefly. We know that constraints limit the days on which a rule starts to be active. This explanation is not correct in all cases, as you'll see now.

There are some days, such as the last day of a month, which can't be expressed using constraints explicitly. To allow targeting such days anyway, the `start` parameter of a rule accepts a day shifting suffix as well. Your constraints are checked for some date, but the rule starts being active some days earlier or later, relative to the matching date.

Even though you can't specify the last day of a month, you can well specify the 1st. This rule is active on the last day of February from 6.00 pm to 10.00 pm, no matter if in a leap year or not:

```
- { v: 22, start: "18:00-1d", end: "22:00", days: 1, months: 3 }
```

This one even runs until March 1st, 10.00 pm:

```
- { v: 22, start: "18:00-1d", end: "22:00+1d", days: 1, months: 3 }
```

As you noted, the day shift of `start` can be negative as well, but not that of `end`, meaning your rules can't span backwards in time. This design decision was made in order to keep rules readable and the evaluation algorithm simple. It neither has a technical reason nor does it reduce the expressiveness of rules.

Rules with Sub-Schedules

Imagine you need to turn on heating three times a day for one hour, but only on working days from January to April. The obvious way of doing this is to define four rules:

```
schedule:
- { v: 23, start: "06:00", end: "07:00", months: "1-4", weekdays: "1-5" }
- { v: 20, start: "11:30", end: "12:30", months: "1-4", weekdays: "1-5" }
- { v: 20, start: "18:00", end: "19:00", months: "1-4", weekdays: "1-5" }
- { v: "OFF" }
```

But what if you want to extend the schedule to heat on Saturdays as well? You'd end up changing this at three different places.

The more elegant way involves so-called sub-schedule rules. Look at this:

```
schedule:
- months: 1-4
  weekdays: 1-6
  rules:
  - { v: 23, start: "06:00", end: "07:00" }
  - { v: 20, start: "11:30", end: "12:30" }
  - { v: 20, start: "18:00", end: "19:00" }
- v: "OFF"
```

The first, outer rule containing the `rules` parameter isn't considered for evaluation itself. Instead, it's child rules (those defined under `rules:`) are considered, but with all constraints of the outer rule (`months` and `weekdays` in this case) applied to them.

Note: The delegation of constraints works not only for one level of sub-schedules. Sub-schedules can be nested as deep as desired and constraints are cumulated correctly.

We can go even further and move the `v: 20` one level up, so that it counts for all child rules which don't have their own `v` defined:

```
schedule:
- v: 20
  months: 1-4
  weekdays: 1-6
  rules:
  - { start: "06:00", end: "07:00", v: 23 }
  - { start: "11:30", end: "12:30" }
```

(continues on next page)

(continued from previous page)

```
  - { start: "18:00", end: "19:00" }
- v: "OFF"
```

Note how the `v` for a rule is chosen. To find the value to use for a particular rule, the rule is first considered itself. In case it has no own `v` defined, all sub-schedule rules that led to this rule are then traversed and scanned for a `v` until one is found. When looking at the indentation of the YAML, this lookup is done from right to left, so that the innermost value is used. The exact same approach is taken for `start` and `end`.

Note: Values for `v`, `start` and `end` declared at inner rules (more indented in YAML) take precedence over those set at outer rules and render the outer values ineffective.

I've to admit that this was a small and well arranged example, but the benefit becomes clearer when you start to write longer schedules, maybe with separate sections for the different seasons.

Note: A rule with sub-schedule attached (one that has a `rules` parameter) is **never** evaluated itself. Only the innermost rules (those with no sub-schedule) count as rules for determining the room's value. This snippet, for instance, has no effect at all:

```
- v: 20
  weekdays: 1-3
  rules:
  - months: 1-4,9-12
    rules:
    # Note there are NO rules in the innermost sub-schedule, hence no rule
    # is evaluated by Schedy, making all this completely ineffective.
```

With this knowledge, writing basic Schedy schedules should be straightforward.

The next chapter deals with expressions, which finally give you the power to do whatever you can do with Python, right inside your schedules.

3.5.2 Dynamic Expressions

As an alternative to fixed values, Schedy accepts so called expressions in schedule rules.

Expressions are a powerful way of expressing a value to be sent to actors dynamically in relation to anything you can think of. This power comes from the fact that expressions are just normal Python code which is evaluated at runtime. All expressions are pre-compiled at startup to make their later evaluation really performant.

Writing Expressions

Note: In contrast to plain values, which are denoted as `value` or `v`, expressions have to be set as the `expression` (or `x`) parameter of a schedule rule. And since expressions have to be strings, we enclose them in quotation marks to prevent the YAML parser from guessing, which may otherwise lead to obscure errors with certain expressions.

Expressions must return a kind of value the used actor type understands. Take the thermostat actor as an example. It needs a temperature value which can either be an integer (`19`) or floating point value (`20.5`). What type of value an individual actor needs is explained in the *chapter specific to the actor type*.

Expressions vs. Statements

The string provided as the `x` parameter of a schedule rule is treated as a simple Python expression. Each of the following is a valid expression.

- `5`
- `True`
- `'off'`
- `17 if is_on('binary_sensor.absent') else Next()`

Writing expressions that way is short and great for things like binary decisions. However, there might be situations in which you need to make more complex weightings that would get confusing when written as a single line expression. That's why you may as well use whole statements.

As soon as the string given as an expression contains line-breaks, it's treated as a series of whole statements rather than an expression. In YAML, a schedule rule with such a multi-line expression can be denoted as follows.

```
- x: |
    a = 2
    b = 5
    result = a * b
```

The string is introduced by a `|`, and all following lines need to be indented by a custom (but consistent) number of spaces.

You may in fact write arbitrary Python code in such a script, including import statements and class or function definitions. The only requirement is that at the end of the execution, the final result is stored in the global `result` variable.

Note: The string really has to consist of more than one line to be treated as a statement. The following example doesn't contain line-breaks and hence would be considered as an uncompilable expression.

```
- x: |
    result = 42
```

While this is a valid single-line expression and would compile just fine:

```
- x: |
    42
```

Controlling the Evaluation Flow

There are special types available for creating objects you can return from an expression in order to influence the way your schedule is processed.

- `Abort()`, which causes schedule lookup to be aborted immediately. The value will not be changed in this case.
- `Break(levels=1)`, which causes lookup of one (or multiple nested) sub-schedule(s) to be aborted immediately. The evaluation will continue after the sub-schedule(s).
- `IncludeSchedule(schedule)`, which dynamically inserts a sub-schedule rule with the given `Schedule` object after the current rule. Especially, `Break()` and `Inherit()` in included schedules do behave as if the included schedule was a regular sub-schedule.

- `Inherit()`, which causes the value or expression of the nearest ancestor rule to be used as result for the current rule. See the next section for a more detailed explanation.
- `Next()`, which causes the rule to be treated as if it didn't exist at all. If one exists, the next rule is evaluated in this case.

For all of these types, *usage examples* are provided.

Expressions and Sub-Schedules

In general, there is no difference between using plain values and advanced expressions in both rules with a sub-schedule attached to them (so-called sub-schedule rules) and the rules contained in these sub-schedules. But with expressions, you gain a lot more flexibility.

As you know from *Rules with Sub-Schedules*, rules of sub-schedules inherit their `v` parameter from the nearest ancestor rule having it defined, should they miss an own one. Basically, this is true for the `x` parameter as well.

With an expression as the `x` value of the rule inside a sub-schedule, you get the flexibility to conditionally overwrite the ancestor rule's value or expression. Should an expression return `Inherit()`, the next ancestor rule's value or expression is used. When compared to static values, returning `Inherit()` is the equivalent of omitting the `v` parameter completely, but with the benefit of deciding dynamically about whether to omit it or not.

The whole process can be described as follows. To find the result for a particular rule inside a sub-schedule, the `v`/`x` parameters of the rule and it's ancestor rules are evaluated from inside to outside (from right to left when looking at the indentation of the YAML syntax) until one results in something different to `Inherit()`.

`Inherit()` even works accross the boundaries of a schedule snippet, because Schedy internally converts a rule which returned `IncludeSchedule()` into a sub-schedule rule, with the included schedule attached to it.

Examples

Considering the State of Entities

Let's say we use the thermostat actor type and have a switch that should prepare our bathroom for taking a bath. It's name is `switch.take_a_bath`. We write the following schedule for the room `bathroom`.

```
schedule:
- x: "22 if is_on('switch.take_a_bath') else Next()"
- v: 19
```

Last step is to tell Schedy to watch for changes of the state of `switch.take_a_bath`, so that it can re-evaluate the schedule of the bathroom when the switch is toggled. We add the following to the room's configuration:

```
watched_entities:
- "switch.take_a_bath"
```

We're done! Now, whenever we toggle the `take_a_bath` switch, the schedule is re-evaluated and our first schedule rule executes. The rule is evaluating our custom expression, checking the state of the `take_a_bath` switch and, if it's enabled, causes the temperature to be set to 22 degrees. However, if the switch is off, the rule is ignored completely due to the `Next()` we return in that case and the second rule is processed, which always evaluates to 19 degrees.

What's so nice about these `... if ... else ...` expressions in Python is that they're almost always self-explanatory. We'll use them extensively in the following examples.

Use of `Add()` and `Next()`

This is something I once used in my own heating configuration at home:

```
schedule_prepend:
- x: "Add(-3) if is_on('input_boolean.absent') else Next()"
watched_entities:
- "input_boolean.absent"
```

What does this? Well, the first thing we see is that the rule is placed inside the `schedule_prepend` section. That means, it is valid for every room and always the first rule being evaluated.

I've defined an `input_boolean` called `absent` in Home Assistant. Whenever I leave the house, this gets enabled. If I return, it's turned off again. In order for Schedy to notice the toggling, I added it to the global `watched_entities` configuration.

Now let's get back to the schedule rule. When it evaluates, it checks the state of `input_boolean.absent`. If the switch is turned on, it evaluates to `Add(-3)`, otherwise to `Next()`.

`Add(-3)` is a so-called *postprocessor*. Think of it as a temporary value that is remembered and used later, after a real result was found.

Now, my regular schedule starts being evaluated, which, of course, is different for every room. Rules are evaluated just as normal. If one returns a result, that is used as the temperature and evaluation stops. But wait, there was the `Add(-3)`, wasn't it? Hence `-3` is now added to the final result.

With this minimal configuration effort, I added an useful away-mode which throttles all thermostats in the house as soon as I leave.

Think of a device tracker that is able to report the distance between you and your home. Having such one set up, you could even implement dynamic throttling that slowly decreases as you near with almost zero configuration effort.

Conditional Sub-Schedules Using `Break()`

When in a sub-schedule, returning `Break()` from an expression will skip the remaining rules of that sub-schedule and continue evaluation after it. You can use it together with `Next()` to create a conditional sub-schedule, for instance. Again, we assume to write a schedule for the thermostat actor type.

```
schedule:
- v: 20
  rules:
  - x: "Next() if is_on('input_boolean.include_sub_schedule') else Break()"
  - { start: "07:00", end: "09:00" }
  - { start: "12:00", end: "22:00" }
  - v: 17
 - v: "OFF"

watched_entities:
- "input_boolean.include_sub_schedule"
```

The rules 2-4 of the sub-schedule will only be respected when `input_boolean.include_sub_schedule` is on. Otherwise, evaluation continues with the last rule, setting the value to `OFF`.

Note: Since `schedule_prepend`, a room's individual schedule and `schedule_append` are just sub-schedules chained internally, returning `Break()` from a top-level rule of one of these three sections causes evaluation to be continued with the next section.

The actual definition of this result type is `Break(levels=1)`, which means that you may optionally pass a parameter called `levels` to `Break()`. This parameter controls how many levels of nested sub-schedules to break out of. The implicit default value `1` will only abort the innermost sub-schedule (the one currently in). However, you may want to directly abort its parent schedule as well by returning `Break(2)`. In the above example, this would actually break the room's schedule and hence continue evaluating the `schedule_append` section.

Here's another example with multiple nested sub-schedules utilizing `Break()`. It is used by an user of Schedy to turn his bathroom floor heating on at specific times, but only when the outside temperature is 5 degrees or lower. It additionally differenciates between away, holiday and normal modes.

```
schedule:
- v: "on"
  rules:
  # don't turn on when it's > 5 degrees outside
  - x: "Break() if float(state('sensor.outside_temperature') or 0) > 5 else Next()"

  # don't turn on when in away mode
  - x: "Break() if is_on('input_boolean.away') else Next()"

  # on weekends and during holidays, turn on from 09:00 to 10:30
  - rules:
    - x: "Next() if is_on('input_boolean.holidays') else Break()"
      weekdays: "!6-7"
    - { start: "09:00", end: "10:30" }

  # on normal working days, turn on from 06:30 to 07:00
  - weekdays: 1-5
    rules:
    - { start: "06:30", end: "07:00"}

# at all other times, turn off
- v: 'off'

watched_entities:
- "sensor.outside_temperature"
- "input_boolean.away"
- "input_boolean.holidays"
```

Including Schedules Dynamically with `IncludeSchedule()`

The `IncludeSchedule()` result type for expressions can be used to insert a set of schedule rules right at the position of the current rule. This comes handy when a set of rules needs to be chosen depending on the state of entities or is reused in multiple rooms.

Note: If you just want to prevent yourself from repeating the same static constraints over and over for multiple consecutive rules that are used only once in your configuration, use the *sub-schedule feature* of the normal rule syntax instead.

You can reference any schedule defined under `schedule_snippets` in the configuration, hence we create one to play with for our heating setup:

```
schedule_snippets:
  vacation:
  - { v: 21, start: "08:30", end: "23:00" }
  - { v: 16 }
```

Now, we include the snippet into a room's schedule:

```
schedule:
- x: "IncludeSchedule(schedule_snippets['vacation']) if is_on('input_boolean.vacation
↪') else Next()"
# when not in vacation mode, have the normal per-room schedule
- { v: 21, start: "07:00", end: "21:30", weekdays: 1-5 }
- { v: 21, start: "08:00", end: "23:00", weekdays: 6-7 }
- { v: 16 }

watched_entities:
- "input_boolean.vacation"
```

It turns out that you could have done the exact same without including a snippet by adding the vacation rules directly to the room's schedule, but doing it this way makes the configuration more readable, easier to maintain and avoids redundancy in case you want to include the `vacation` snippet into other rooms as well.

Other use cases for `IncludeSchedule` are selecting different schedules based on presence (maybe even long holidays vs. short absence) or weather sensors.

Note: Splitting up schedules doesn't bring any extra power to Schedy's scheduling capabilities, but it can make configurations much more readable as they grow.

What to Use `Abort()` for

The `Abort` return type is most useful for disabling Schedy's scheduling mechanism depending on the state of entities. You might implement on/off switches for disabling the schedules with it, like so:

```
schedule_prepend:
- name: global schedule on/off switch
  x: "Abort() if is_off('input_boolean.schedy') else Next()"
- name: per-room schedule on/off switch
  x: "Abort() if is_off('input_boolean.schedy_room_' + room_name) else Next()"

# These should trigger a re-evaluation in every room.
watched_entities:
- "input_boolean.schedy"

# And for these it is sufficient to re-evaluate the corresponding room only.
rooms:
  living:
    watched_entities:
    - "input_boolean.schedy_room_living"
  kitchen:
    watched_entities:
    - "input_boolean.schedy_room_kitchen"
```

As soon as `Abort()` is returned, schedule evaluation is aborted and the value stays unchanged.

Using the Generic `Postprocess()` Postprocessor

The `Postprocess()` *postprocessor* lets you alter the result of scheduling in arbitrary ways. It takes a callable which is then called with the result as its argument and should return the eventually altered result.

In this example, we use `Postprocess()` with lambda closures (in-line functions that generate their return value with only a single expression) to limit the scheduled value to the range from `16` to `22`. This could be useful for a temperature, for instance.

```
- x: "Postprocess(lambda result: max(16, result))"
- x: "Postprocess(lambda result: min(result, 22))"
```

You could of course have done this with a single postprocessor as well.

```
- x: "Postprocess(lambda result: max(16, min(result, 22)))"
```

Instead of lambda closures, normal functions may also be used. Here is an identically behaving, quite verbose implementation.

```
- x: |
    def limit(r):
        if r < 16:
            return 16
        if r > 22:
            return 22
        return r

    result = Postprocess(limit)
```

Here's another one which actually behaves like `Add(-3)`.

```
- x: "Postprocess(lambda result: result - 3)"
```

Note: As you know, evaluation stops at the first rule generating a result. Hence you need to ensure the rules returning postprocessors are placed before the rules that generate the results to be postprocessed, not after them.

Expression Helpers

For generating meaningful values with expressions, you usually need access to the state of entities in Home Assistant and other data, such as the current date and time. Several functions and variables are available for usage in your expressions, these will be described in this chapter.

Note: Depending on the actor type you're using, there may be additional helpers available. See the documentation of the particular actor type.

Basic Helpers

app

`app: SchedyApp`

There is an object available under the name `app` which represents the `appdaemon.plugins.hass.hassapi.Hass` object of Schedy. You could, for instance, retrieve values of input sliders via the normal AppDaemon API.

room_name

```
room_name: str
```

A string representing the name of the room the expression is evaluated for as set in Schedy's configuration (not the friendly name).

schedule_snippets

```
schedule_snippets: Dict[str, Schedule]
```

A dictionary containing all configured schedule snippets, indexed by their name for use with `IncludeSchedule()`.

is_empty

```
is_empty(iterable: Iterable) -> bool
```

Returns whether the given iterable is empty.

next() is used for testing the iterable. For iterators, this has the side effect of the first item being consumed, but it avoids generating all values just for decision about emptiness.

round_to_step

```
round_to_step(value: Union[float, int], step: Union[float, int],
decimal_places: int = None) -> Union[float, int]
```

Round the value to the nearest step and, optionally, the given number of decimal places.

Examples:

```
round_to_step(34, 25) == 25
round_to_step(0.665, 0.2, 1) == 0.6
```

Date and Time

datetime

```
datetime: ModuleType
```

Python's `datetime` module.

now

```
now: datetime.datetime
```

A `datetime.datetime` object containing the current date and time.

date

```
date: datetime.date
```

A shortcut for `now.date()`.

time

```
time: datetime.time
```

A shortcut for `now.time()`.

State Helpers

These helpers can be used to retrieve the state of entities from Home Assistant.

is_on

```
is_on(entity_id: str) -> bool
```

Returns `True` if the state of the given entity is `"on"` (case-insensitive), `False` otherwise.

is_off

```
is_off(entity_id: str) -> bool
```

Returns `True` if the state of the given entity is `"off"` (case-insensitive), `False` otherwise.

Note: There is a difference between using `is_off(...)` and `not is_on(...)`. These helper functions only compare the state of the specified entity to the values `"off"` and `"on"`, respectively. If you want to treat a non-existing entity (which's state is returned as `None`) as if it was `"off"`, you have to use `not is_on(...)` since `is_off(...)` would return `False` in this case.

---

state

```
state(entity_id: str = None, attribute: str = None) -> Any
```

A shortcut for `app.get_state()`.

It generates a warning when an entity is queried for which no watch has been configured via `watched_entities`. That's why you should always use this helper instead of calling `app.get_state()` directly.

filter_entities

```
filter_entities(entities: Union[str, List[str]] = None, **criteria: Any) ->
Iterable[str]
```

From a given set of entities, this function yields only those with a state and/or attributes matching all given criteria.

Entities may either be specified as a single string (full entity id or domain), a list of such strings, or as `None`, which means all entities found in Home Assistant.

Examples:

```
# entities with a state of "on"
for entity in filter_entities(state="on"):
    ...

# binary_sensor and input_boolean entities having a room attribute with the value
↪"living"
for entity in filter_entities(["binary_sensor", "input_boolean"], room="living"):
    ...
```

Schedule Helpers

Note: This is a topic targeted at advanced users. It might be hard to understand for newcomers.

These helpers can be used to evaluate schedule snippets from within an expression. That could be useful to make decisions based on the result a particular schedule snippet would provide when evaluated at a given point in time, even in the future.

Warning: Prospective evaluation of schedule snippets can only provide reliable results for such ones not including expressions that reference to the state of entities, because there is no way for Schedy to foresee state changes. Schedule snippets only having rules with plain values instead of expressions are however always safe in this regard.

`ScheduleEvaluationResult` is a type defined as `Tuple[Any, Set[str], Rule]`. The first item is the value generated by the schedule, the second a set with markers applied to the result and the third is the `Rule` object which generated the value. You'll normally only want the first item, the actual value.

schedule.evaluate

`schedule.evaluate(schedule: Schedule, when: datetime.datetime = None) -> Optional[ScheduleEvaluationResult]`

Evaluates the given schedule at the given point in time. If `when` is not specified, the current date and time is assumed. When no result could be generated (e.g. because a rule evaluated to `Abort()` or all evaluated to `Next()`), `None` is returned instead of a `ScheduleEvaluationResult`.

Example:

```
result = schedule.evaluate(
    schedule_snippets["snip"],
    when=now+datetime.timedelta(hours=1),
)
if result:
    value = result[0]
    # do something with the value
```

schedule.next_results

```
schedule.next_results(schedule: Schedule, start: datetime.datetime =
None, end: datetime.datetime = None) -> Generator[Tuple[datetime.datetime,
ScheduleEvaluationResult], None, None]
```

This function let's you iterate over future results of a given schedule snippet. Every `Tuple[datetime.datetime, ScheduleEvaluationResult]` represents a point in time at which the result will change. With the `start` and `end` parameters, you can limit the time range to consider. The default is to start at the current time and continue infinitely. The first result generated is always that for the `start` time, the last one that for the `end` time.

Example:

```
results = schedule.next_results(
    schedule_snippets["snip"],
    end=now+datetime.timedelta(hours=10),
)
for when, (value, markers, rule) in results:
    # do something with the value
```

Pattern Helpers

These helpers can be used to calculate values based on some pre-defined patterns.

pattern.linear

```
pattern.linear(start_value: Union[float, int], end_value: Union[float, int],
percentage: Union[float, int]) -> float
```

Calculate the value at a given `percentage` between `start_value` and `end_value`. The borders can be crossed when `percentage` is outside the range `0..100`.

Postprocessing Results

Note: This is a topic targeted at advanced users. It might be hard to understand for newcomers.

There are situations in which it would come handy to post-process the later result of scheduling in a specific way without knowing what that result will actually be. One such situation for the thermostat actor type could be lowering the temperature by a certain number of degrees when nobody is home. For such needs, there is a concept called postprocessors.

In the evaluation environment, there are a number of types which, when returned, tell Schedy you want to generate a postprocessor that is going to alter the later result. Namely, there are:

- `Add(x)` to add a value `x` to the result.
- `Multiply(x)` to multiply the result with `x`.
- `Invert()` to invert the result. This negates numbers, inverts boolean values and swaps the strings `"on"` and `"off"` for each other.
- `Postprocess(func)`, where `func` is a callable that takes the result as its only argument and returns the post-processed result. This can conveniently be used with lambda-closures .

When an expression results in such a postprocessor object, it is stored until a subsequent rule returns some real result. Then, the stored postprocessors are applied to that result one by one in the order they were generated.

See the *Examples* page for usage examples.

Result Markers

Note: This is a topic targeted at advanced users. It might be hard to understand for newcomers.

A result generated by an expression can optionally be marked with some pre-defined markers that influence how the result is handled.

Instead of

```
- x: "21"
```

you write

```
- x: "Mark(21, Mark.OVERLAY)"
```

to mark the result `21` with the `OVERLAY` marker.

The actual syntax is `Mark(result, marker1, ..., markerN)`, so you can add multiple markers ad once. Markers can be applied to *postprocessors* as well, but they will be used for the final result. Custom postprocessors (e.g. those defined via `Postprocess()`) may also add result markers themselves.

The following markers are available:

- `OVERLAY`: Overwrite manual value adjustments even when a configured `rescheduling_delay` would normally have prevented it. As soon as the schedule no longer evaluates to an `OVERLAY`-marked result the previous value is restored, no matter if that was the scheduled or a manually set one. Even a previous re-scheduling time is restored. An occasion for using this marker is *Open Door or Window Detection*.
- `OVERLAY_REVERT_ON_NO_RESULT`: When applied in conjunction with the `OVERLAY` marker, the overlay is cancelled as soon as a schedule evaluation produces no result (e.g. because `Abort()` was used or all rules evaluated to `Next()`). When an overlay is created without this additional marker, the value marked with `OVERLAY` stays active until the schedule really results in another value. **DEPRECATED:** This is the default behaviour of *OVERLAY* now.

Security Considerations

It has to be noted that expressions are evaluated using Python's `exec()` function. In general, this is not suited for code originating from a source you don't trust completely, because such code can potentially execute arbitrary commands on your system with the same permissions and capabilities the AppDaemon process itself has. That shouldn't be a problem for expressions you write yourself inside schedules.

This feature could however become problematic if an attacker somehow is able to emit events on your Home Assistant's event bus. To prevent expressions from being accepted in the `schedy_set_value` event, processing of such expressions is disabled by default and has to be enabled explicitly by setting `expressions_from_events: true` in your Schedy configuration.

3.6 Events

Schedy introduces two new events it listens for and which you can emit from your custom Home Assistant automations or scripts in order to control Schedy's behaviour.

- `schedy_reevaluate`: Trigger a re-evaluation of schedules. Only use this event if you can't express the criteria that should trigger a re-evaluation via the `watched_entities` configuration, e.g. when you need re-evaluation.based on time intervals instead of state changes. Parameters are:
  - `room`: the name (or list of names) of the room(s) to re-evaluate schedules for as defined in Schedy's configuration (not the `friendly_name`) (default: `null`, which means all rooms)
  - `mode`: There are two different re-evaluation modes you can choose from. (default: `"reevaluate"`)
    - `"reevaluate"`: Re-evaluate the schedule and, if the result has changed compared to the previous evaluation, apply the new value to all actors in the room. If a re-scheduling timer is running, nothing is done until that timer goes off. This is the mode you normally want when notifying Schedy about state changes of entities used in your schedule. You can trigger a `schedy_reevaluate` event in this mode as often as you like without worrying about loosing manual value changes early.
    - `"reset"`: Re-evaluate the schedule and set the resulting value to all actors in the room, no matter if it has changed or not. This mode also discardss a re-scheduling eventually planned for the future and instead performs one immediately. Use this mode in order to discard any manual adjustment at one of the actors, e.g. when presence state has changed or a master switch was toggled and you want to ensure all actors are updated. This is exactly what the built-in delayed re-scheduling does after manual adjustments when enabled.
- `schedy_set_value`: Sets a given value for a room. Parameters are:
  - `room`: the name (or list of names) of the room(s) as defined in Schedy's configuration (not the `friendly_name`)
  - `value` or `v`: a plain value as it could also have been generated by a schedule, as a simple alternative to `expression`
  - `expression` or `x`: an expression as it could also have been generated by a schedule, as an alternative to `value`
  - `force_resend`: whether to re-send the value to the actors even if it hasn't changed due to Schedy's records (default: `false`)
  - `rescheduling_delay`: a number of minutes after which Schedy should automatically switch back to the schedule; `0` disables automatic re-scheduling (default: the `rescheduling_delay` set in Schedy's configuration for the particular room)

Note: In order to pass an `expression` to the `schedy_set_value` event, you need to set `expressions_from_events: true` in Schedy's configuration. Beware the implications on security this has, as everybody with access to Home Assistant's event bus can then execute arbitrary Python code on your machine with the privileges of the user AppDaemon runs as. Weigh for yourself on whether you really need this feature.

All events have an optional `app_name` parameter that can be submitted when you have multiple instances of Schedy running for different purposes and you want to address exactly one of these instances. Its value has to be the name of the app instance as configured in AppDaemon. If you omit this parameter, all Schedy instances will react to the event. The app name is the name you start the app's configuration with:

```
schedy_heating:  # "schedy_heating" would be the value to use for app_name.
  module: hass_apps_loader
```

(continues on next page)

(continued from previous page)

```
  class: SchedyApp
  # ...
```

This is an example Home Assistant script that turns the heating in the room named `living` to `25.0` degrees and switches back to the regular schedule after one hour, given that you use the `thermostat` actor type and yourr app instance is named `schedy_heating`, of course.

```
- alias: Hot for one hour
  sequence:
  - event: schedy_set_value
    event_data:
      app_name: schedy_heating
      room: living
      v: 25.0
      rescheduling_delay: 60
```

3.7 Statistics

Schedy provides a concept for collecting statistical data about its operation at runtime.

The statistical parameters you can collect depend on the actor type used. A `switch` actor, for instance, obviously doesn't support measuring temperature differences. What parameters are available can be found in the individual actor documentations.

What's common among all parameters is that they create a new entity in Home Assistant, named `schedy_stats.<app name>_<parameter instance name>`. The state of these entities is always the empty string `""` and thus irrelevant, but their attributes are of interest. The names and meanings of attributes generated by a specific parameter type can be found in its documentation. You may then use normal Home Assistant automations to react to changes of the individual entity attributes.

A simple statistics configuration with a single instance of the `temp_delta` parameter, which is provided by the `thermostat` actor type, might look as follows:

```
statistics:
  # Pick an arbitrary name for the parameter instance.
  upstairs_temp_delta:
    # The type of parameter as found in the actor'S documentation.
    type: temp_delta
    # More parameter-specific settings:
    rooms:
      bathroom:
      kidsroom:
    #...
```

Given that the name of your Schedy app instance is `heating`, this would create an entity named `schedy_stats.heating_upstairs_temp_delta` with the attributes `min`, `avg` and `max`.

3.8 Tips & Tricks

The purpose of this chapter is to collect useful configuration snippets and tips for using Schedy in various (maybe not so common) usage scenarios.

3.8.1 Schedule Rules with Dynamic Start and End Times

The start and end time of a schedule rule are always static. They can't be computed by something like expressions at runtime. However, there is a trick you can utilize in order to get start and end times which are based on the state of entities in Home Assistant.

Let's assume you've got two entities, `input_number.start_hour` and `input_number.end_hour`. Then you could write a schedule rule without the `start` and `end` fields set, resulting in it always being valid. As the value for `x`, you configure an expression like the following.

```
'on' if time.hour >= float(state('input_number.start_hour')) and time.hour <=␣
↪float(state('input_number.end_hour')) else Next()
```

What this does is quite simple. It sets the value to "on" if the current hour is between the values configured by the two entities we introduced. If it's not, the rule is ignored and processing continues at the next rule, as always.

There is still one thing missing in order to make this work properly. Schedy needs to be notified about state changes of the used entities by adding them to the `watched_entities` configuration. How that's done is described in *this example*.

You could now make the temperature configurable via an `input_number.day_temperature` entity as well.

Now let's put this all together into a valid schedule rule:

```
- x: "state('input_number.day_temperature') if time.hour >= float(state('input_number.
↪start_hour')) and time.hour <= float(state('input_number.end_hour')) else Next()"
```

3.8.2 Reacting to Changes of Schedy's State

Each room records its state to an entity in Home Assistant. This entity is named `schedy_room.<app name>_<room name>`.

The state of such an entity is the value currently set for the room. Since values can be changed manually, this is not necessarily the one generated by the schedule. The actual scheduled value is stored as the `scheduled_value` attribute of the entity.

You can use normal Home Assistant automations to react to changes of these entities.

3.8.3 Open Door or Window Detection

When using Schedy for heating control and you've got window sensors, you might want to have the thermostats in a room turned off when a window is opened. We can achieve this with a single additional schedule rule for an unlimited number of windows.

We assume that our window sensors for the `living` room are named `binary_sensor.living_window_1` and `binary_sensor.living_window_2` and report `"on"` as their state when the particular window is opened.

To make this solution scale to multiple windows in multiple rooms without creating additional rules, we add a new custom attribute to our window sensors via the `customize.yaml` file that holds the name of the Schedy room the sensor belongs to.

```
binary_sensor.living_window_1:
  window_room: living

binary_sensor.living_window_2:
  window_room: living
```

Now, a new rule which overlais the temperature with `OFF` when a window in the current room is open is added. We place it at the top of the `schedule_prepend` configuration section to have it applied to all rooms as their first rule.

This code checks all `binary_sensor` entities found in Home Assistant for a `window_room` attribute with the current room's name as its value and a state of `"on"`. This way it finds all window sensors of the current room that report to be open. The `is_empty()` function is used with the `filter_entities()` generator to have searching aborted as soon as one open window is found rather than always checking all entities. Feel free to break this single-line expression into multiple statements if you prefer clarity over conciseness.

```
- x: "Mark(OFF, Mark.OVERLAY) if not is_empty(filter_entities('binary_sensor', window_
↪room=room_name, state='on')) else Next()"
```

Now, we add the window sensors to the `watched_entities` of the `living` room.

```
watched_entities:
- "binary_sensor.living_window_1"
- "binary_sensor.living_window_2"
```

That's it. Don't forget to restart Home Assistant after editing `customize.yaml`.

3.8.4 Motion-Triggered Lights

Scheduling lights is really easy with the `switch` actor type. Even associating motion sensors isn't too complicated with just a single additional schedule rule. The procedure is identical to that used for *Open Door or Window Detection*, except that the `binary_sensor` entities now report motion instead of open windows and the value needs to be set to `"on"` while motion is detected.

Let's assume the following:

1. You've got a room named `entrance` configured in Schedy with one or more lights as actors.
2. There'S a motion sensor `binary_sensor.entrance_motion` that switches to `on` when motion is detected.

Ok, let's get started.

1. Add a custom `motion_room: entrance` attribute to the `binary_sensor.entrance_motion` entity via `customize.yaml` to tie the motion sensor to the Schedy room it belongs to.
2. Now, a new rule which overlais the value with `"on"` while a motion sensor of the current room reports motion is added. We place it at the top of the `schedule_prepend` configuration section to have it applied to all rooms as their first rule.

   ```
   - x: "Mark('on', Mark.OVERLAY) if not is_empty(filter_entities('binary_sensor',␣
   ↪motion_room=room_name, state='on')) else Next()"
   ```

3. Add the motion sensor to the `watched_entities` of the `entrance` room.

```
watched_entities:
- "binary_sensor.entrance_motion"
```

Try it out. As long as at least one of the motion sensors in a room reports motion, the lights in that room should stay on.

If you also had brightness sensors in each room, you could now insert another rule before the one we just added to fix the value to `"off"` when it's not dark enough in the particular room.

3.9 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

3.9.1 Unreleased

Fixed

Security

Added

Changed

Deprecated

Removed

3.9.2 0.8.3 - 2020-03-19

Fixed

- Fixed placeholder insertion in `generic2` actor service data. (#50)

3.9.3 0.8.2 - 2020-02-21

Fixed

- Hotfix for an issue causing infinite recursion at delayed actor initialization on AppDaemon 4 (see home-assistant/appdaemon#773).

3.9.4 0.8.1 - 2020-02-05

Fixed

- Fixed possible `TypeError` when using the `generic2` actor type. (#47)
- The `generic2` actor type now correctly handles service names in the usual `<domain>.<name>` format.

3.9.5 0.8.0 - 2020-02-03

Fixed

- Fixed indentation in sample configuration in docs.

Added

- Added new `generic2` actor type which is more flexible than the old `generic`.

Changed

- The `switch` actor type is now driven by the new `generic2` actor type. Functionality and syntax stays all the same.

3.9.6 0.7.0 - 2019-12-04

Fixed

- An overlay will now be applied even when the scheduled value won't change.

Changed

- Infinite retrying of value sending to an actor is no longer supported, meaning `send_retries: -1` is now a configuration error. Use a reasonably high value instead if you really need excessive retrying.
- Simplified internal handling of `IncludeSchedule()`. If this causes problems with existing configurations, please file an issue.

Removed

- The previous name `Skip` for the `Next` expression result type has been removed.
- The `OVERLAY_REVERT_ON_NO_RESULT` marker has been removed, it's the default now.

3.9.7 0.6.0 - 2019-09-27

Fixed

- Fixed a regression due to which setting an actor's `send_retries: 0` led to infinite re-sending if the actor didn't respond as expected.
- Fixed a race condition between `Mark.OVERLAY` and re-scheduling timers. (#35)

Changed

- The `Skip` expression result type has been renamed to `Next`, which better describes its purpose.
- The behaviour of the `OVERLAY_REVERT_ON_NO_RESULT` result marker now is the default with `OVERLAY`. The marker will be removed.

Deprecated

- 0.7: The previous name `Skip` for the `Next` expression result type will be removed.
- 0.7: The `OVERLAY_REVERT_ON_NO_RESULT` marker will be removed, it's the default now.

Removed

- The `end_plus_days` rule parameter has been removed in favor of the new day shifts specified with `start` and `end`.
- The `expression_modules` setting has been removed in favor of the new `expression_environment`.

3.9.8 0.5.0 - 2019-07-20

Fixed

- Fixed a bug in schedule.next_results() expression helper that caused some result changes to be skipped.
- Simplified the algorithm that decides whether a rule is active or not at a given point in time. It should now handle all rules spanning multiple days correctly.

Added

- Added `expression_environment` setting which allows providing arbitrary variables for the expression evaluation environment.

Changed

- The `start` and `end` rule parameters now accept day shifts, deprecating the former `end_plus_days`.
- Constraints of rules with a sub-schedule attached are now only validated for the day at which a particular rule starts. Hence rules of such sub-schedules spanning midnight will now run until they're intended to end.
- Home Assistant 0.96 introduced breaking changes in the climate API. Operation modes have been renamed into HVAC modes, which is why the thermostat actor settings for operation modes now have new names. See the actor docs for details.

Deprecated

- 0.6: The `end_plus_days` rule parameter will be removed in favor of the new day shifts specified with `start` and `end`.
- 0.6: The `expression_modules` setting will be removed in favor of the new `expression_environment`.

Removed

- Some settings of the thermostat actor have been removed in one run with the adaptations needed to support the new climate API of Home Assistant 0.96.

3.9.9 0.4.0 - 2019-02-24

Fixed

- Fixed name of `value_parameter` setting for generic actor in docs.
- Schedules were re-evaluated when the value of a not watched attribute of a watched entity changes.

Added

- Added new result marker `OVERLAY_REVERT_ON_NO_RESULT` to cancel an overlay when the schedule produces no result.
- Result markers can now be added by postprocessors as well.
- The generic actor has received new features (short values and sending of attributes in reversed order). See the actor sample config for details.

Changed

- The wanted value of a room is not sent to actors at startup when `replicate_changes` has been disabled in the room's configuration.

Removed

- The old name `schedy_reschedule` for the `schedy_reevaluate` event has been removed.

3.9.10 0.3.0 - 2019-01-05

Fixed

- It's no longer possible to create cycles when including schedules. The backwards resolution of rule values still works, it just treats `IncludeSchedule()` results for schedules already on the stack as if they were `Inherit()` and hence ignores them.
- The `filter_entities()` state helper returned no entities in certain cases.

Added

- Schedy can now re-evaluate schedules automatically when the state of entities changes. See the new `watched_entities` settings.
- Range specifications for constraints can now be inverted by prepending them with `!`.
- Added the `Inherit()` result type to inherit the parent rule's value. `None` will continue to work as well, but `Inherit()` is more explanatory and thus preferred.
- When an expression fails to evaluate, the traceback is now logged.

Changed

- Various small improvements of the examples for using expressions.
- The `schedy_reschedule` event has been renamed to `schedy_reevaluate`. The old name will cease to work in version 0.4.
- The documentation for writing schedules has been restructured.

Deprecated

- 0.4: The old name `schedy_reschedule` for the `schedy_reevaluate` event will be removed.

Removed

- The old name `Negate` for the `Invert` postprocessor has been removed.
- The `And` and `Or` postprocessors habe been removed. Use the generic `Postprocess` instead.

3.9.11 0.2.0 - 2018-12-23

Merry Christmas to all users of hass-apps! Thank you for putting your trust in Schedy.

Fixed

- All expressions of schedule rules specified in the YAML configuration should be enclosed in quotes to force the parser to treat them as strings. A note has been added to the documentation and all examples were updated accordingly.

Added

- Added the `Postprocess` postprocessor that can be used to post-process the scheduling result in a completely custom way.

Changed

- The rules configured as `schedule_prepend`, the individual room's schedule and those configured as `schedule_append` are now combined into the final room's schedule as three separate sub-schedules. This implies that `Break()`, when returned from the top level, will now only break the individual section of the schedule it stands in. `Break()` in a `schedule_prepend` section will e.g. only cause the remaining rules of the `schedule_prepend` section to be skipped and continue with the individual room's schedule. Use `Abort()` (recommended) or `Break(2)` to achieve the old behaviour.
- The generic actor has been reworked to support controlling multiple attributes at once. Its configuration schema has changed as well, so please consult the documentation for migrating.
- Preliminary results are now called postprocessors. Syntax and names stay unchanged.
- The `Negate` postprocessor has been renamed to `Invert`. The old name will cease to work in version 0.3.

Deprecated

- 0.3: The old name `Negate` for the `Invert` postprocessor will be removed.
- 0.3: The `And` and `Or` postprocessors will be removed. Use the generic `Postprocess` instead.

3.9.12 0.1.1 - 2018-12-11

Changed

- Lowered delay after which a schedy_reschedule event is processed from 3 to 1 second.

3.9.13 0.1.0 - 2018-12-09

Added

- Initial release.

CHAPTER 4

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog.

4.1 Unreleased

4.1.1 Fixed

4.1.2 Security

4.1.3 Added

4.1.4 Changed

4.1.5 Deprecated

4.1.6 Removed

4.2 0.20200319.0

4.2.1 Changed

- Specified dependency versions based on semantic versioning.
- Schedy v0.8.3

4.3 0.20200221.0

4.3.1 Changed

- Schedy v0.8.2

4.4 0.20200205.0

4.4.1 Changed

- Schedy v0.8.1

4.5 0.20200203.0

4.5.1 Changed

- Schedy v0.8.0

4.6 0.20191210.0

4.6.1 Changed

- Simplified installation and upgrade instructions in docs

4.7 0.20191204.0

4.7.1 Changed

- Schedy v0.7.0

4.8 0.20190927.0

4.8.1 Changed

- Schedy v0.6.0

4.9 0.20190720.0

4.9.1 Changed

- Schedy v0.5.0

4.9.2 Removed

- Heaty

4.10 0.20190224.0

4.10.1 Changed

- Configuration errors are now logged in a more human-readable format.
- Schedy v0.4.0

4.11 0.20190105.0

4.11.1 Changed

- Schedy v0.3.0

4.12 0.20181223.1

4.12.1 Changed

- Some documentation changes

4.13 0.20181223.0

4.13.1 Changed

- Schedy v0.2.0

4.13.2 Removed

- Removed the motion_light app.

4.14 0.20181211.0

4.14.1 Changed

- Schedy v0.1.1

4.15 0.20181209.0

4.15.1 Added

- Schedy v0.1.0

4.15.2 Changed

- Installation in hass.io is now a lot simpler, see here.
- Installation in Docker is now a lot simpler, see here.

4.15.3 Deprecated

- The heaty app is now obsolete because of Schedy and won't receive new updates. Please migrate to Schedy. Heaty will however stay there for the foreseeable future.
- The motion_light app will be removed at the end of 2018. Schedy can control lights much more flexibly.

4.15.4 Removed

- Removed the Auto-Install Assistant because of missing resonance

4.16 0.20181005.0

4.16.1 Changed

- Re-generated sphinx configuration with version 1.7.8.
- Changed minimum version of observable dependency to 1.0.0.
- heaty v0.17.0

4.17 0.20180824.1

4.17.1 Fixed

- Fixed appdaemon dependency to version >= 3.0.0.

4.18 0.20180824.0

4.18.1 Changed

- heaty v0.16.0

4.18.2 Removed

- Removed AppDaemon 2.x support.

4.19 0.20180801.0 - 2018-08-01

4.19.1 Added

- Added a script that automates the installation process and can be run with just one single command. See here for more information.

4.19.2 Changed

- heaty v0.15.0

4.19.3 Deprecated

- AppDaemon 2.x support will be dropped in a late August 2018 release. Please switch to AppDaemon 3.x.

4.20 0.20180707.0 - 2018-07-07

4.20.1 Changed

- heaty v0.14.0
- No longer using broken `set_app_state()` feature of AppDaemon, hence AppDaemon 3.0.0+ should now work and blacklisting has been removed.

4.20.2 Deprecated

- AppDaemon 2.x support will be dropped in a late August 2018 release. Please switch to AppDaemon 3.x.

4.21 0.20180405.0 - 2018-04-05

4.21.1 Changed

- heaty v0.13.0

4.22 0.20180325.0 - 2018-03-25

4.22.1 Fixed

- Fixed wrong path to sample configuration files in docs/apps/index.rst.

4.22.2 Added

- Blacklisted AppDaemon version 3.0.0 in requirements. (#12)

4.22.3 Changed

- heaty v0.12.4

4.23 0.20180310.1 - 2018-03-10

4.23.1 Changed

- Fixed old project name in setup.py left over by mistake.

4.24 0.20180310.0 - 2018-03-10

4.24.1 Changed

- heaty v0.12.3
- Switched project name from hass_apps to hass-apps

4.25 0.20180307.0 - 2018-03-07

4.25.1 Changed

- heaty v0.12.2

4.26 0.20180305.0 - 2018-03-05

4.26.1 Changed

- heaty v0.12.1

4.27 0.20180302.0 - 2018-03-02

4.27.1 Changed

- heaty v0.12.0
- Require voluptuous >= 0.11.1.
- It is now strongly recommended to install in a separate virtualenv to avoid conflicts in versions of dependency packages that are needed by both hass_apps and Home Assistant. The Getting started section has been updated accordingly.

4.28 0.20180221.0 - 2018-02-21

4.28.1 Changed

- motion_light v0.1.1
- Ported docs, sample configurations and changelogs to sphinx + readthedocs.org.

4.29 0.20180218.0 - 2018-02-18

4.29.1 Changed

- heaty v0.11.0

4.30 0.20180209.0 - 2018-02-09

4.30.1 Changed

- heaty v0.10.2

4.31 0.20180205.2 - 2018-02-05

4.31.1 Fixed

- Fixed wrong AppDaemon version in requirements.

4.32 0.20180205.1 - 2018-02-05

4.32.1 Changed

- heaty v0.10.1

4.33 0.20180205.0 - 2018-02-05

4.33.1 Fixed

- Added missing release dates to CHANGELOG.md

4.33.2 Added

- Added CHANGELOG.md and LICENSE to Python source package.
- Added appdaemon 3 support alongside the old appdaemon 2

4.33.3 Changed

- heaty v0.10.0

4.34 0.20180203.0 - 2018-02-03

4.34.1 Changed

- heaty v0.9.4

4.35 0.20180202.1 - 2018-02-02

4.35.1 Changed

- heaty v0.9.3

4.36 0.20180202.0 - 2018-02-02

4.36.1 Changed

- heaty v0.9.2

4.37 0.20180201.0 - 2018-02-01

4.37.1 Added

- Begin using CHANGELOG.md

CHAPTER 5

Contributor Covenant Code of Conduct

5.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

5.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

5.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

5.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

5.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at r.schindler@efficiosoft.com. The project team will review and investigate all complaints, and will respond in a way that it deems appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

5.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4

Active and stable apps:

- *Schedy* - The most powerful scheduler for everything from lighting to heating

Deprecated apps that will be removed:

- None at the moment.

CHAPTER 6

Donations

Note: I work on this project in my spare time, as many free software developers do. And of course, I enjoy this work a lot. There is no and will never be a need to pay anything for using this software.

However, if you want to honor the hundreds of hours continuously spent with writing code and documentation, testing and providing support by donating me a cup of coffee, a beer in the evening, my monthly hosting fees or anything else embellishing my day a little more, that would be awesome. If you decide doing so, I want to thank you very much! Please be assured that I'm not presuming anybody to donate, it's entirely your choice.

Ensure ongoing development and support with a monthly donation, no matter how small.

Or make an one-time donation.

ETH: 0xCE6B204B6AB5B93156f4FCD373482e148753beAb
ZEC: t1RKFyt4qqtqdYfprf8HZoDHRNLNzhe35ED

CHAPTER 7

Getting Help

Note: If you run into any issue, first consult the documentation thoroughly.

A notable amount of work has gone into it and most aspects should be covered already. When this didn't help you're welcome to e.g. ask in the Home Assistant Community.

When encountering something that seems to be a bug, please open an issue on GitHub and attach complete logs with `debug: true` set in the app's configuration illustrating the issue. You won't receive help otherwise.

CHAPTER 8

Contributing

You are welcome to contribute your own apps for AppDaemon to this project. But please don't submit a pull request without talking to me first. This is because there is currently no developer documentation on how to integrate properly with the environment provided by hass-apps and I want to save you the hassle of re-designing your app after it's already written.

If you've got an interesting idea for a new app you'd like to contribute, just open an issue on GitHub and we can discuss it there.

All contributions are subject to the *Contributor Covenant Code of Conduct*. Don't contribute if you don't agree with that.