hacs-pyscript

Release 1.5.0

Aug 25, 2023

Contents

This HACS custom integration for Home Assistant allows you to write Python functions and scripts that can implement a wide range of automation, logic and triggers. State variables are bound to Python variables and services are callable as Python functions, so it's easy and concise to implement logic.

CHAPTER 1

Contents

1.1 Overview

This HACS custom integration allows you to write Python functions and scripts that can implement a wide range of automation, logic and triggers. State variables are bound to Python variables and services are callable as Python functions, so it's easy and concise to implement logic.

Functions you write can be configured to be called as a service or run upon time, state-change or event triggers. Functions can also call any service, fire events and set state variables. Functions can sleep or wait for additional changes in state variables or events, without slowing or affecting other operations. You can think of these functions as small programs that run in parallel, independently of each other, and they could be active for extended periods of time.

State, event and time triggers are specified by Python function decorators (the "@" lines immediately before each function definition). A state trigger can be any Python expression using state variables - the trigger is evaluated only when a state variable it references changes, and the trigger occurs when the expression is true or non-zero. A time trigger could be a single event (eg: date and time), a repetitive event (eg: at a particular time each day or weekday, daily relative to sunrise or sunset or any regular time period within an optional range) or using cron syntax (where events occur periodically based on a concise specification of ranges of minutes, hours, days of week, days of month and months). An event trigger specifies the event type, and an optional Python trigger test based on the event data that runs the Python function if true.

Pyscript implements a Python interpreter using the ast parser output, in a fully async manner. That allows several of the "magic" features to be implemented in a seamless Pythonic manner, such as binding of variables to states and functions to services. Pyscript supports imports, although by default the valid import list is restricted for security reasons (there is a configuration option `allow_all_imports` to allow all imports). Pyscript supports almost all Python language features except generators, `yield`, and defining special class methods. (see language limitations). Pyscript provides a handful of additional built-in functions that connect to HASS features, like logging, accessing state variables as strings (if you need to compute their names dynamically), running and managing tasks, sleeping and waiting for triggers.

Pyscript also provides a kernel that interfaces with the Jupyter front-ends (eg, notebook, console, lab and VSC). That allows you to develop and test pyscript code interactively. Plus you can interact with much of HASS by looking at state variables, calling services etc, in a similar way to HASS CLI, although the CLI provides status on many other parts of HASS.

For more information about the Jupyter kernel, see the README. There is also a Jupyter notebook tutorial, which can be downloaded and run interactively in Jupyter notebook or VSC connected to your live HASS with pyscript.

Pyscript provides functionality that complements the existing automations, templates and triggers. Pyscript is most similar to AppDaemon, and some similarities and differences are discussed in this Wiki page. Pyscript with Jupyter makes it extremely easy to learn, use and debug. Pyscripts presents a simplified and more integrated binding for Python scripting than Python Scripts, which requires a lot more expertise and scaffolding using direct access to Home Assistant internals.

1.2 Installation

1.2.1 Option 1: HACS

Under HACS -> Integrations, select "+", search for `pyscript` and install it.

1.2.2 Option 2: Manual

From the latest release download the zip file `hass-custom-pyscript.zip`

```
cd YOUR_HASS_CONFIG_DIRECTORY    # same place as configuration.yaml
mkdir -p custom_components/pyscript
cd custom_components/pyscript
unzip hass-custom-pyscript.zip
```

Alternatively, you can install the current GitHub master version by cloning and copying:

```
mkdir SOME_LOCAL_WORKSPACE
cd SOME_LOCAL_WORKSPACE
git clone https://github.com/custom-components/pyscript.git
mkdir -p YOUR_HASS_CONFIG_DIRECTORY/custom_components
cp -pr pyscript/custom_components/pyscript YOUR_HASS_CONFIG_DIRECTORY/custom_
↪components
```

1.2.3 Install Jupyter Kernel

Installing the Pyscript Jupyter kernel is optional but highly recommended. The steps to install and use it are in this README.

1.3 Configuration

- Pyscript can be configured using the UI, or via yaml. To use the UI, go to the Configuration -> Integrations page and selection "+" to add `Pyscript Python scripting`. After that, you can change the settings anytime by selecting Options under Pyscript in the Configuration page.

  Alternatively, for yaml configuration, add `pyscript:` to `<config>/configuration.yaml`. Pyscript has two optional configuration parameters that allow any python package to be imported and exposes the `hass` variable as a global (both options default to `false`):

  ```
  pyscript:
    allow_all_imports: true
    hass_is_global: true
  ```

- Add files with a suffix of `.py` in the folder `<config>/pyscript`.
- Restart HASS after installing pyscript.
- Whenever you change a script file or app, pyscript will automatically reload the changed files. To reload all files and apps, call the `pyscript.reload` service with the optional `global_ctx` parameter to `*`.
- Watch the HASS log for `pyscript` errors and logger output from your scripts.

1.4 Tutorial

1.4.1 Jupyter Tutorial

The best way to learn about pyscript is to interactively step through the Jupyter tutorial. After you have installed the pyscript Jupyter kernel, the tutorial can be downloaded with:

```
wget https://github.com/craigbarratt/hass-pyscript-jupyter/raw/master/pyscript_
↪tutorial.ipynb
```

and open it with:

```
jupyter notebook pyscript_tutorial.ipynb
```

You can step through each command by hitting <Shift>Enter. There are various ways to navigate and run cells in Jupyter that you can read in the Jupyter documentation.

1.4.2 Writing your first script

Create a file `example.py` in the `<config>/pyscript` folder (you can use any file name, so long as it ends in `.py`) that contains:

```
@service
def hello_world(action=None, id=None):
    """hello_world example using pyscript."""
    log.info(f"hello world: got action {action} id {id}")
    if action == "turn_on" and id is not None:
        light.turn_on(entity_id=id, brightness=255)
    elif action == "fire" and id is not None:
        event.fire(id, param1=12, pararm2=80)
```

After starting Home Assistant, use the Service tab in the Developer Tools page to call the service `pyscript.hello_world` with parameters

```
action: hello
id: world
```

The function decorator `@service` means `pyscript.hello_world` is registered as a service. The expected service parameters are keyword arguments to the function. This function prints a log message showing the `action` and `id` that the service was called with. Then, if the action is `"turn_on"` and the `id` is specified, the `light.turn_on` service is called. Otherwise, if the action is `"fire"` then an event type with that `id` is fired with the given parameters. You can experiment by calling the service with different parameters. (Of course, it doesn't make much sense to have a function that either does nothing, calls another service, or fires an event, but, hey, this is just an example.)

Note: You'll need to look at the log messages to see the output (unless you are using Jupyter, in which case all log messages will be displayed, independent of the log setting). The log message won't be visible unless the `Logger` is enabled at least for level `info`, for example:

```
logger:
  default: info
  logs:
    custom_components.pyscript: info
```

1.4.3 An example using triggers

Here's another example:

```
@state_trigger("security.rear_motion == '1' or security.side_motion == '1'")
@time_active("range(sunset - 20min, sunrise + 15min)")
def motion_light_rear():
    """Turn on rear light for 5 minutes when there is motion and it's dark"""
    log.info(f"triggered; turning on the light")
    light.turn_on(entity_id="light.outside_rear", brightness=255)
    task.sleep(300)
    light.turn_off(entity_id="light.outside_rear")
```

This introduces two new function decorators

- `@state_trigger` describes the condition(s) that trigger the function (the other two trigger types are `@time_trigger` and `@event_trigger`, which we'll describe below). This condition is evaluated each time the variables it refers to change, and if it evaluates to `True` or non-zero then the trigger occurs.
- `@time_active` describes a time range that is checked whenever a potential trigger occurs. The Python function is only executed if the `@time_active` criteria is met. In this example the time range is from 20 minutes before sunset to 15 minutes after sunrise, ie: from dusk to dawn. Whenever the trigger is `True` and the active conditions are met, the function is executed as a new task. The trigger logic doesn't wait for the function to finish; it goes right back to checking for the next condition. The function turns on the rear outside light, waits for 5 minutes, and then turns it off.

However, this example has a problem. During those 5 minutes, any additional motion event will cause another instance of the function to be executed. You might have dozens of them running, which is perfectly ok for `pyscript`, but probably not the behavior you want, since as each earlier one finishes the light will be turned off, which could be much less than 5 minutes after the most recent motion event.

There is a special function provided to ensure just one function uniquely handles a task, if that's the behavior you prefer. Here's the improved example:

```
@state_trigger("security.rear_motion == '1' or security.side_motion == '1'")
@time_active("range(sunset - 20min, sunrise + 20min)")
def motion_light_rear():
    """Turn on rear light for 5 minutes when there is motion and it's dark"""
    task.unique("motion_light_rear")
    log.info(f"triggered; turning on the light")
    light.turn_on(entity_id="light.outside_rear", brightness=255)
    task.sleep(300)
    light.turn_off(entity_id="light.outside_rear")
```

The `task.unique` function will terminate any task that previously called `task.unique("motion_light_rear")`, and our instance will survive. (The function takes a 2nd argument

that causes the opposite to happen: the older task survives and we are terminated - so long!)

As before, this example will turn on the light for 5 minutes, but when there is a new motion event, the old function (which is part way through waiting for 5 minutes) is terminated, and we start another 5 minute timer. The effect is the light will stay on for 5 minutes after the last motion event, and stays on until there are no motion events for at least 5 minutes. If instead the second argument to `task.unique` is set, that means the new task is terminated instead. The result is that the light will go on for 5 minutes following a motion event, and any new motion events during that time will be ignored, since each new triggered function will be terminated. Depending on your application, either behavior might be preferred.

There are some other improvements we could make. We could check if the light is already on so we don't have to turn it on again, by checking the relevant state variable:

```
@state_trigger("security.rear_motion == '1' or security.side_motion == '1'")
@time_active("range(sunset - 20min, sunrise + 20min)")
def motion_light_rear():
    """Turn on rear light for 5 minutes when there is motion and it's dark"""
    task.unique("motion_light_rear")
    log.info(f"triggered; turning on the light")
    if light.outside_rear != "on":
        light.turn_on(entity_id="light.outside_rear", brightness=255)
    task.sleep(300)
    light.turn_off(entity_id="light.outside_rear")
```

You could also create another function that calls `task.unique("motion_light_rear")` if the light is manually turned on (by doing a `@state_trigger` on the relevant state variable), so that the motion logic is stopped when there is a manual event that you want to override the motion logic.

We've introduced some of the main features. Now for some more formal descriptions of the decorators and the handful of extra built-in functions available.

1.5 Reference

1.5.1 Configuration

Pyscript can be configured using the UI, or via yaml. To use the UI, go to the Configuration -> Integrations page and select "+" to add `Pyscript Python scripting`. After that, you can change the settings anytime by selecting "Options" under Pyscript in the Configuration -> Integrations page. You will need to select "reload" under Pyscript, or call the `pyscript.reload` service for the new settings to take effect.

Alternatively, for yaml configuration, add `pyscript:` to `<config>/configuration.yaml`. You can't mix these two methods - your initial choice determines how you should update these settings later. If you want to switch configuration methods you will need to uninstall and reinstall pyscript.

Pyscript has two optional configuration parameters that allow any python package to be imported and exposes the `hass` variable as a global (both options default to `false`). Assuming you didn't use the UI to configure pyscript, these can be set in `<config>/configuration.yaml`:

```
pyscript:
  allow_all_imports: true
  hass_is_global: true
```

It is recommended you put your pyscript configuration its own `yaml` file in the `pyscript` folder. That way changes to that file will be automatically detected and will trigger a reload, which includes rereading the configuration parameters:

```
pyscript: !include pyscript/config.yaml
```

The settings and behavior of your code can be controlled by additional user-defined yaml configuration settings. If you configured pyscript using the UI flow, you can still add additional configuration settings via yaml. Since they are free-form (no fixed schema) there is no UI configuration available for these additional settings.

Each application should have its configuration stored using the application's name under an `apps` entry. For example, applications `my_app1` and `my_app2` would be configured as:

```
pyscript:
  allow_all_imports: true
  apps:
    my_app1:
      # any settings for my_app1 go here
    my_app2:
      # any settings for my_app2 go here
```

If the include above is used, the `pyscript/config.yaml` would look like:

```
allow_all_imports: true
apps:
  my_app1:
    # any settings for my_app1 go here
  my_app2:
    # any settings for my_app2 go here
```

The `apps` section could also include other configuration files if you prefer to put each application's configuration in its own file.

An application will not be loaded unless it has a configuration entry (even if it is empty). That provides an easy was to enable or disable an application.

An application's configuration (eg, all the settings below `my_app1` in the example above) are available in the variable `pyscript.app_config` in the global scope of the application's main file (eg, `apps/my_app1.py` or `apps/my_app1/__init__.py`). Additionally, all the pyscript configuration settings are available via the variable `pyscript.config`, which also includes all the application configuration below the `apps` key. However, in a future release the `apps` entry will be removed so that apps do not have access to another app's configuration (which might include token, passwords or keys). See *this section* for more information. Note that `pyscript.app_config` is not defined in regular scripts - only in each application's main file.

Note that if you used the UI flow to configure pyscript, the `allow_all_imports` and `hass_is_global` configuration settings will be ignored in the yaml file. In that case you should omit them from the yaml, and just use yaml for pyscript app configuration.

At startup, pyscript loads the following files. It also automatically unloads and reloads files when they are changed, renamed, created or deleted:

<config>/pyscript/*.py all files with a `.py` suffix in the top-level directory are autoloaded.

<config>/pyscript/scripts//*.py** all files with a `.py` suffix below the `scripts` directory, recursively to any depth, are autoloaded. This is useful for organizing your scripts into subdirectories that are related in some way.

<config>/pyscript/apps/<app_name>/__init__.py every `__init__.py` file in a subdirectory in the `apps` subdirectory is autoloaded, provided `app_name` exists in the pyscript `yaml` configuration under `apps`. This package form is most convenient for sharing pyscript code, since all the files for one application are stored in their own directory.

<config>/pyscript/apps/<app_name>.py all files in the `apps` subdirectory with a `.py` suffix are autoloaded, unless the package form above was loaded instead, and provided `app_name` exists in the pyscript

yaml configuration under apps (that allows each app to be disabled by simply removing its configuration and reloading).

The idea is that the top-level directory pyscript and pyscript/scripts are for your own use (and are always loaded), and pyscript/apps/ is for things you've gotten from others or things you've written with the intent to reuse or share (and only are loaded if there is a configuration entry, which can be empty).

Any file name that starts with # is not loaded, and similarly scripts anywhere below a directory name that starts with #, are not loaded. That's a convenient way to disable a specific script or entire directory below scripts - you can simply rename it with or without the leading # to disable or enable it, which automatically triggers a reload. Think of it as "commenting" the file name, rather than having to delete it or move it outside the pyscript directory.

Like regular Python, functions within one source file can call each other, and can share global variables (if necessary), but just within that one file. Each file has its own separate global context. Each Jupyter session also has its own separate global context, so functions, triggers, variables and services defined in each interactive session are isolated from the script files and other Jupyter sessions. Pyscript provides some utility functions to switch global contexts, which allows an interactive Jupyter session to interact directly with functions and global variables created by a script file, or even another Jupyter session.

The optional <config>/pyscript/modules subdirectory can contain modules (files with a .py extension) or packages (directories that contain at least a __init__.py file) that can be imported by any other pyscript files, applications or modules. The module form is ignored if the package form is present. They are not autoloaded. Any modules or packages in <config>/pyscript/modules that are modified will be unloaded when they are modified, and any scripts or apps that depend on those modules will be reloaded. Importing modules and packages from <config>/pyscript/modules are not restricted if allow_all_imports is False. Typically common functions or features would be implemented in a module or package, and then imported and used by scripts in <config>/pyscript or applications in <config>/pyscript/apps.

Even if you can't directly call one function from another script file, HASS state variables are global and services can be called from any script file.

1.5.2 Reloading Scripts

Manually reloading scripts via the pyscript.reload service is no longer necessary starting in 1.2.0, since changes are automatically detected and a reload is triggered.

By default, calling the reload service only reloads scripts, apps or modules that have changed since they were last loaded. A change means the script, app or module file contents or modification time has changed. Additionally, an app is considered changed if its yaml configuration has changed. When a change is detected, additional files are also reloaded if they depend on a changed file. Additional files include all other files in a particular module or app (meaning the entire module or app is reloaded if any of its files or imports are changed), and if a module has changed, any other module, app or script that imports that module directly or indirectly is reloaded too.

A file is also considered changed if it is newly created or deleted (or "commented" by renaming it or a parent directory to start with #). When reload detects a deleted file, the prior global context and all its triggers are deleted, just as when a file has changed.

If you want to reload a specific script or file, you can touch it to change its modification time. Alternatively, the pyscript.reload service takes an optional parameter global_ctx which specifies the name of a specific global context to reload. That is the one file considered to be changed by reload, and other changes are ignored. For example, specifying file.example will just reload example.py, and apps.my_app1 will just reload apps/my_app1.py. See the *Global Context* section for the mapping of file or app names to global context names. Additional files might still be reloaded too (all other files in the module or app, and any modules, apps or scripts that import a module if global_ctx was set to a module).

If you want to manually reload all scripts, apps and modules, set global_ctx to * when you call pyscript.reload. Alternatively, the Reload option under pyscript in the in the Configuration -> Integrations page will always do a full reload.

Any function definitions, services and triggers in the reloaded files are re-created. Jupyter sessions are not affected. Any currently running functions (ie, functions that have been triggered and are actively executing Python code or waiting inside `task.sleep()` or `task.wait_until()`) are not stopped on reload - they continue to run until they finish (return). You can terminate these running functions too on reload if you wish by calling `task.unique()` in the script file preamble (ie, outside any function definition), so it is executed on load and reload, which will terminate any running functions that have previously called `task.unique()` with the same argument.

1.5.3 State Variables

These are typically called entities or `entity_id` in HASS.

State variables can be accessed in any Python code simply by name. State variables (also called `entity_id`) are of the form `DOMAIN.name`, where `DOMAIN` is typically the name of the component that sets that variable. You can set a state variable by assigning to it.

State variables only have string values, so you will need to convert them to `int` or `float` if you need their numeric value.

State variables have attributes that can be accessed by adding the name of the attribute, as in `DOMAIN.name.attr`. The attribute names and their meaning depend on the component that sets them, so you will need to look at the State tab in the Developer Tools to see the available attributes. You can set an attribute directly by assigning `DOMAIN.name.attr = value`.

In cases where you need to compute the name of the state variable dynamically, or you need to set or get all the state attributes, you can use the built-in functions `state.get()`, `state.getattr()`, `state.set()` and `state.setattr()`; see *State Functions*.

The function `state.names(domain=None)` returns a list of all state variable names (ie, `entity_id`s) of a domain. If `domain` is not specified, it returns all HASS state variable (entity) names.

Also, service names (which are called as functions) take priority over state variable names, so if a component has a state variable name that collides with one of its services, you'll need to use `state.get(name)` to access that state variable.

Accessing state variables that don't exist will throw a `NameError` exception, and accessing an attribute that doesn't exist will throw a `AttributeError` exception. One exception (!) to this is that in a `@state_trigger` expression, undefined state variables and attributes will evaluate to `None` instead of throwing an exception.

You can assign a state variable (or the return value of `state.get()`) to a normal Python variable, and that variable will capture the value and attributes of the state variable at the time of the assignment. So, for example, after this assignment:

```
test1_state = binary_sensor.test1
```

the variable `test1_state` captures both the value and attributes of `binary_sensor.test1`. Later, if `binary_sensor.test1` changes, `test1_state` continues to represent the previous value and attributes at the time of the assignment.

State variables also support virtual methods that are service calls with that `entity_id`. For any state variable `DOMAIN.ENTITY`, any services registered by `DOMAIN`, eg: `DOMAIN.SERVICE`, that have an `entity_id` parameter can be called as a method `DOMAIN.ENTITY.SERVICE()`. Each of these statements are equivalent:

```
service.call("DOMAIN", "SERVICE", entity_id="DOMAIN.ENTITY", other_param=123)
DOMAIN.SERVICE(entity_id="DOMAIN.ENTITY", other_param=123)
DOMAIN.ENTITY.SERVICE(other_param=123)
```

In the case the service has only one other parameter in addition to `entity_id`, a further shorthand is that the method can be called with just a positional, rather than keyword, argument. So if the service only takes two parameters, `entity_id` and `other_param`, this additional form is equivalent to each of the above statements:

```
DOMAIN.ENTITY.SERVICE(123)
```

Here's an example using `input_number`, assuming it has been configured to create an entity `input_number.test`. These statements all do the same thing (and the last one works because `set_value` only takes one other parameter):

```
service.call("input_number", "set_value", entity_id="input_number.test", value=13)
input_number.set_value(entity_id="input_number.test", value=13)
input_number.test.set_value(value=13)
input_number.test.set_value(13)
```

Two additional virtual attribute values are available when you use a variable directly as `DOMAIN.entity.attr` or call `state.get("DOMAIN.entity.attr")`:

- `last_changed` is the last UTC time the state value was changed (not the attributes)
- `last_updated` is the last UTC time the state entity was updated

If you need to compute how many seconds ago the `binary_sensor.test1` state changed, you could do this:

```
from datetime import datetime as dt
from datetime import timezone as timezone

num_seconds_ago = (dt.now(tz=timezone.utc) - binary_sensor.test1.last_changed).total_
↪seconds()
```

Note that these virtual attributes and methods take precedence over any entity attributes that have the same name. If an entity has attributes with those names and you need to access them, use `state.getattr(name)`.

1.5.4 Calling services

Any service can be called by using the service name as a function, with keyword parameters to specify the service parameters. You'll need to look up the service in the Service tab of Developer Tools to find the exact name and parameters. For example, inside any function you can call:

```
myservice.flash_light(light_name="front", light_color="red")
```

which calls the `myservice.flash_light` service with the indicated parameters. Obviously those parameter values could be any Python expression, and this call could be inside a loop, an if statement or any other Python code.

Starting in HASS 2023.7, services can return a response, which is a dictionary of values. You need to set the keyword parameter `return_response=True` to get the response. For example:

```
service_response = myservice.flash_light(light_name="front", light_color="red",␣
↪return_response=True)
```

The function `service.call(domain, name, **kwargs)` can also be used to call a service when you need to compute the domain or service name dynamically. For example, the above service could also be called by:

```
service.call("myservice", "flash_light", light_name="front", light_color="red")
```

When making a service call, either using the `service.call` function or the service name as the function, you can optionally pass the keyword argument `blocking=True` if you would like to wait for the service to finish execution

before continuing execution in your function. If the service returns a response (new feature starting in HASS 2023.7), it will be returned by the call if you set the optional keyword parameter `return_response=True`.

1.5.5 Firing events

Any event can be triggered by calling `event.fire(event_type, **kwargs)`. It takes the `event_type` as a first argument, and any keyword parameters as the event parameters. The `event_type` could be a user-defined string, or it could be one of the built-in events. You can access the names of those built-in events by importing from `homeassistant.const`, eg:

```
from homeassistant.const import EVENT_CALL_SERVICE
```

1.5.6 Function Trigger Decorators

There are four decorators for defining state, time, event and MQTT triggers, and two decorators for defining whether any trigger actually causes the function to run (i.e., is active), based on state-based expressions or one or more time-windows. The decorators should appear immediately before the function they refer to. A single function can have any or all of the decorator types specified. Multiple trigger decorators of the same type can be added to a single function, but only one `@state_active`, `@time_active` or `@task_unique` can be used per function.

A Python function with decorators is still a normal Python function that can be called by any other Python function. The decorators have no effect in the case where you call it directly from another function.

@state_trigger

```
@state_trigger(str_expr, ..., state_hold=None, state_hold_false=None, state_check_
↪now=False, kwargs=None, watch=None)
```

`@state_trigger` takes one or more string arguments that contain any expression based on one or more state variables, and evaluates to `True` or `False` (or non-zero or zero). Whenever any of the state variables or attribute values mentioned in the expression change (or specified via the `watch` argument), the expression is evaluated and the trigger occurs if it evaluates to `True` (or non-zero). For each state variable, eg: `domain.name`, the prior value is also available to the expression as `domain.name.old` in case you want to condition the trigger on the prior value too. Attribute values can be used in the expression too, using the forms `domain.name.attr` and `domain.name.old.attr` for the new and old attribute values respectively.

Multiple `str_expr` arguments are logically "or"ed together into a single expression. Any argument can alternatively be a list or set of strings, and they are treated the same as multiple arguments by "or"ing them together. Alternatively, independent state triggers can be specified by using multiple `@state_trigger` decorators.

Optional arguments are:

`state_check_now=False` If set, the `@state_trigger` expression is evaluated immediately when the trigger function is defined (typically at startup), and the trigger occurs if the expression evaluates to `True` or non-zero. Normally the expression is only evaluated when a state variable changes, and not when the trigger function is first defined. This option is the same as in the `task.wait_until` function, except the default value is `True` in that case. Note that if you specify `state_check_now=True`, entries in `state_trigger` that are plain state variable names (which mean trigger on any change) are ignored during the initial check - only expressions are checked.

`state_hold=None` A numeric duration in seconds that delays executing the trigger function for this amount of time. If the state trigger expression changes back to `False` during that time, the trigger is canceled and a wait for a new trigger begins. If the state trigger expression changes, but is still `True` then the `state_hold` time is not restarted - the trigger will still occur that number of seconds after the first state trigger.

state_hold_false=None If set, the state trigger edge is triggered (triggers on a `False` to `True` transition), versus the default of level triggered (triggers when `True`). The `@state_trigger` expression must evaluate to `False` for this duration in seconds before the next trigger can occur. A value of `0` requires the expression be `False` before a trigger, but with no minimum time in that state. If the expression evaluates to `True` during the `state_hold_false` period, that trigger is ignored, and when the expression next is `False` the `state_hold_false` period starts over.

For example, by default the expression `"int(sensor.temp_outside) >= 50"` will trigger every time `sensor.temp_outside` changes to a value that is 50 or more. If instead `state_hold_false=0`, the trigger will only occur when `sensor.temp_outside` changes the first time to 50 or more. It has to go back below 50 for `state_hold_false` seconds before a new trigger can occur.

When `state_hold_false` is set, the state trigger expression is evaluated at startup. If `False` the `state_hold_false` period begins. If `True`, a wait for the next `False` value begins. If `state_check_now` is also set, the trigger will also occur at startup if the expression is `True` at startup, while the `state_hold_false` logic will continue to wait until the expression is `False` for that period before the next future trigger.

kwargs=None Additional keyword arguments can be passed to the trigger function by setting `kwargs` to a dict of additional keywords and values. These will override the standard keyword arguments such as `value` or `var_name` if you include those keywords in `kwargs`. A typical use for `kwargs` is if you have multiple `@state_trigger` decorators on a single trigger function and you want to pass additional parameters (based on the trigger, such as a setting or action) to the function. That could save several lines of code in the function determining which trigger occurred.

watch=None Specifies a list or set of string state variable names (entity names) or state attributes that are monitored for this trigger. When (and only when) a variable in this set changes, the trigger expression is evaluated. Normally this set of names is automatically extracted from the `@state_trigger` expression, but there could be cases where it doesn't capture all the names (eg, if you have to use `state.get()` inside the trigger expression). You could also use `watch` to specify a subset of the names in the trigger expression, which has the effect of rendering those other variables as only conditions in the trigger expression that won't cause a trigger themselves, since the expression won't be evaluated when they change.

Here's a summary of the trigger behavior with these parameter settings:

state_check_now	state_hold_false	trigger at start?	trigger on state change?
default (`False`)	default (`None`)	no	if expr `True`
default (`False`)	0	no	if expr `True`, only after `False`
default (`False`)	10	no	if expr `True`, only after `False` for 10 sec
`True`	default (`None`)	if expr `True`	if expr `True`
`True`	0	if expr `True`	if expr `True`, only after `False`
`True`	10	if expr `True`	if expr `True`, only after `False` for 10 sec

If `state_hold` is also specified, all the entries in the table that say "if expr `True`" must remain `True` for that number of seconds for the trigger to occur.

All state variables in HASS have string values. So you'll have to do comparisons against string values or cast the variable to an integer or float. These two examples are essentially equivalent (note the use of single quotes inside the outer double quotes):

```
@state_trigger("domain.light_level == '255' or domain.light2_level == '0'")
```

```
@state_trigger("int(domain.light_level) == 255 or int(domain.light2_level) == 0")
```

although the second will throw an exception if the variable string doesn't represent a valid integer. If you want numerical inequalities you should use the second form, since string lexicographic ordering is not the same as numeric ordering.

You can also use state variable attributes in the trigger expression, with an idenfitier of the form `DOMAIN.name.attr`. Attributes maintain their original type, so there is no need to cast then to another type.

You can specify a state trigger on any change with a string that can take three forms:

- `"domain.entity"`: triggers on any change to the state variable value
- `"domain.entity.attr"`: triggers on any change to the state variable attribute `attr` value
- `"domain.entity.*"`: triggers on any change to any state variable attribute (but not its value)

For example:

```
@state_trigger("domain.light_level")
```

will trigger any time the value of `domain.light_level` changes (not its attributes), which includes the cases when that variable is first created (ie, the `old_value` is `None`) and when it is deleted (ie, the `value` is `None`).

If you use the "any change" form, there's no point in also specifying `state_hold` since the expression is always `True` whenever the state variable changes - there is no way for it to evaluate to `False` and to re-start the trigger process. If you do specify `state_hold` in this case it will simply delay the trigger by the specified time.

The trigger can include arguments with any mixture of string expressions (that are evaluated when any of the underlying state variables change) and string state variable or attribute names (that trigger whenever that variable or attribute changes).

Note that if a state variable and attributes are set to the same value, HASS doesn't generate a state change event, so the `@state_trigger` condition will not be checked. It is only evaluated each time a state variable or any of its attributes change to a new value.

When the trigger occurs and the function is executed (meaning any active checks passed too), keyword arguments are passed to the function so it can tell which state variable caused it to succeed and run, in cases where the trigger condition involves multiple variables. These are:

```
kwargs = {
    "trigger_type": "state",
    "var_name": var_name,
    "value": new_value,
    "old_value": old_value
}
```

The `value` and `old_value` represent the current and old values of the state variable `var_name` whose change caused the trigger. Those variables include the state attributes too. If the trigger occurs when the state variable is newly created, `old_value` will be `None`, and if the trigger occurs when a state variable is deleted, `value` will be `None`.

Additional keyword parameters can be passed to the trigger function by setting the optional `kwargs` parameter to a `dict` with the keyword/value pairs.

If your function needs to know any of these values, you can list the keyword arguments you need, with defaults:

```
@state_trigger("domain.light_level == '255' or domain.light2_level == '0'")
def light_turned_on(trigger_type=None, var_name=None, value=None):
    pass
```

You don't have to list all the default keyword parameters - just the ones your function needs. In contrast, if you specify additional keyword parameters via `kwargs`, you will get an excepton if the function doesn't have matching keyword arguments (unless you use the `**kwargs` catch-all in the function definition).

Using `trigger_type` is helpful if you have multiple trigger decorators. The function can now tell which type of trigger, and which of the two variables changed to cause the trigger. You can also use the keyword catch-all declaration instead:

```
@state_trigger("domain.light_level == '255' or domain.light2_level == '0'")
def light_turned_on(**kwargs)
    log.info(f"got arguments {kwargs}")
```

and all those values (including optional ones you specify with the `kwargs` argument to `@state_trigger`) will simply get passed via `kwargs` as a `dict`. That's the most useful form to use if you have multiple decorators, since each one passes different variables into the function (although all of them set `trigger_type`).

If `state_check_now` is set to `True` and the trigger occurs during its immediate check, since there is no underlying state variable change, the trigger function is called with only this argument:

```
kwargs = {
    "trigger_type": "state",
}
```

If the trigger function uses `var_name==None` as a keyword argument, it can check if it is `None` to determine whether it was called immediately or not. Similarly, if it uses the `kwargs` form, if can check if `var_name` is in `kwargs`.

If `state_hold` is specified, the arguments to the trigger function reflect the variable change that cause the first trigger, not any subsequent ones during the `state_hold` period. Also, if the `@time_active` or `@state_active` decorators are used, they will be evaluated after the `state_hold` period, but with the initial trigger variable value (ie, the value that caused the initial trigger).

Inside `str_expr`, undefined state variables, undefined state attributes, and undefined `.old` variables evaluate to `None`, rather than throwing an exception. The `.old` variable will be `None` the first time the state variable is set (since it has no prior value), and when the `str_expr` is being evaluated because a different state variable changed (only the state variable change that caused `str_expr` to be evaluated gets its prior value in `.old`; any other `.old` variables will be `None` for that evaluation).

@time_trigger

```
@time_trigger(time_spec, ..., kwargs=None)
```

`@time_trigger` takes one or more string specifications that specify time-based triggers. When multiple time triggers are specified, each are evaluated, and the earliest one is the next trigger. Then the process repeats. Alternatively, multiple time trigger specifications can be specified by using multiple `@time_trigger` decorators, although that is less efficient than passing multiple arguments to a single one.

Additional keyword parameters can be passed to the trigger function by setting the optional `kwargs` parameter to a `dict` with the keywords and values.

Several of the time specifications use a `datetime` format, which is ISO: `yyyy/mm/dd hh:mm:ss`, with the following features:

- There is no time-zone (local is assumed).
- The date is optional, and the year can be omitted with just `mm/dd`.
- The date can also be replaced by a day of the week (either full like `sunday` or 3-letters like `sun`, in your local language based on the locale; however, on Windows and other platforms that lack `locale.nl_langinfo`, the days of week default to English).
- The meaning of partial or missing dates depends on the trigger, as explained below.
- The date and time can be replaced with `now`, which means the current date and time when the trigger was first evaluated (eg, at startup or when created as an inner function or closure), and remains fixed for the lifetime of the trigger.

- The time can instead be `sunrise`, `sunset`, `noon` or `midnight`.
- If the time is missing, midnight is assumed (so `thursday` is the same as `thursday 00:00:00`)
- Seconds are optional, and can include a decimal (fractional) portion if you need finer resolution.
- The `datetime` can be followed by an optional offset of the form `[+-]number{seconds|minutes|hours|days|weeks}` with abbreviations:
  - `{s|sec|second|seconds}` or empty for seconds,
  - `{m|min|mins|minute|minutes}` for minutes,
  - `{h|hr|hour|hours}` for hours,
  - `{d|day|days}` for days,
  - `{w|week|weeks}` for weeks.

  That allows things like `sunrise + 30m` to mean 30 minutes after sunrise, or `sunday sunset - 1.5 hour` to mean 1.5 hours before sunset on Sundays. The `number` can be floating point. (Note, there is no i18n support for those offset abbreviations - they are in English.)

In `@time_trigger`, each string specification `time_spec` can take one of five forms:

- `"startup"` triggers on HASS start and reload (ie, on function definition), and is equivalent to `"once(now)"`
- `"shutdown"` triggers on HASS shutdown and reload (ie, when the trigger function is no longer referenced)
- `"once(datetime)"` triggers once on the date and time. If the year is omitted, it triggers once per year on the date and time (eg, birthday). If the date is just a day of week, it triggers once on that day of the week. If the date is omitted, it triggers once each day at the indicated time. `once(now + 5 min)` means trigger once 5 minutes after startup.
- `"period(datetime_start, interval, datetime_end)"` or `"period(datetime_start, interval)"` triggers every interval starting at the starting datetime and finishing at the optional ending datetime. When there is no ending datetime, the periodic trigger runs forever. The interval has the form `number{sec|min|hours|days|weeks}` (the same as datetime offset without the leading sign), and the same abbreviations can be used. Period start and ends can also be based on `now`, for example `period(now + 10m, 5min, now + 30min)` will cause five triggers at 10, 15, 20, 25 and 30 minutes after startup.
- `"cron(min hr dom mon dow)"` triggers according to Linux-style crontab. Each of the five entries are separated by spaces and correspond to minutes, hours, day-of-month, month, day-of-week (0 = sunday):

field	allowed values
minute	0-59
hour	0-23
day of month	1-31
month	1-12
day of week	0-6 (0 is Sun)

  Each field can be a `*` (which means "all"), a single number, a range or comma-separated list of numbers or ranges (no spaces). Ranges are inclusive. For example, if you specify hours as `6,10-13` that means hours of 6,10,11,12,13. The trigger happens on the next minute, hour, day that matches the specification. See any Linux documentation for examples and more details (note: names for days of week and months are not supported; only their integer values are). The cron features use the `croniter` package, so check its documentation for additional specification formats that are supported (eg: `*/5` repeats every 5th unit, days of week can be specified with English abbreviations, and an optional 6th field allows seconds to be specified).

When the `@time_trigger` occurs and the function is called, the keyword argument `trigger_type` is set to `"time"`, and `trigger_time` is the exact `datetime` of the time specification that caused the trigger (it will be slightly before the current time), or `startup` or `shutdown` in the case of a `startup` or `shutdown` trigger.

Additional optional keyword parameters can be specified in the `kwargs` parameter to `@time_trigger`.

A final special form of `@time_trigger` has no arguments, which causes the function to run once automatically on startup or reload, which is the same as providing a single `"startup"` time specification:

```
@time_trigger
def run_on_startup_or_reload():
    """This function runs automatically once on startup or reload"""
    pass
```

The function is not re-started after it returns, unless a reload occurs. Startup occurs when the `EVENT_HOMEASSISTANT_STARTED` event is fired, which is after everything else is initialized and ready, so this function can call any services etc. A startup trigger can also occur after HASS is started when a new trigger function is defined (eg, on reload, defined at run-time using inner function/closure, or interactively in Jupyter).

Similarly, the `shutdown` trigger occurs when the `EVENT_HOMEASSISTANT_STOP` event is fired, meaning HASS is shutting down. It also occurs whenever a trigger function is no longer referenced (meaning it's being deleted), which happens during reload or redefined interactively in Jupyter. It's important that any trigger function based on `shutdown` runs as quickly as possible, since the shutdown of HASS (or reload completion) will be stalled until your function completes.

@event_trigger

```
@event_trigger(event_type, str_expr=None, kwargs=None)
```

`@event_trigger` triggers on the given `event_type`. Multiple `@event_trigger` decorators can be applied to a single function if you want to trigger the same function with different event types.

An optional `str_expr` can be used to match the event data, and the trigger will only occur if that expression evaluates to `True` or non-zero. This expression has available all the event parameters sent with the event, together with these two variables:

- `trigger_type` is set to "event"
- `event_type` is the string event type, which will be the same as the first argument to `@event_trigger`

Note unlike state variables, the event data values are not forced to be strings, so typically that data has its native type.

When the `@event_trigger` occurs, those same variables are passed as keyword arguments to the function in case it needs them. Additional keyword parameters can be specified by setting the optional `kwargs` argument to a `dict` with the keywords and values.

The `event_type` could be a user-defined string, or it could be one of the built-in events. You can access the names of those events by importing from `homeassistant.const`, eg:

```
from homeassistant.const import EVENT_CALL_SERVICE
```

To figure out what parameters are sent with an event and what objects (eg: `list`, `dict`) are used to represent them, you can look at the HASS source code, or initially use the `**kwargs` argument to capture all the parameters and log them. For example, you might want to trigger on certain service calls (not ones directed to pyscript), but you are unsure which one and what parameters it has. So initially you trigger on all service calls just to see them:

```
from homeassistant.const import EVENT_CALL_SERVICE

@event_trigger(EVENT_CALL_SERVICE)
def monitor_service_calls(**kwargs):
    log.info(f"got EVENT_CALL_SERVICE with kwargs={kwargs}")
```

After running that, you see that you are interested in the service call `lights.turn_on`, and you see that the `EVENT_CALL_SERVICE` event has parameters `domain` set to `lights` and `service` set to `turn_on`, and the service parameters are passed as a `dict` in `service_data`. So then you can narrow down the event trigger to that particular service call:

```
from homeassistant.const import EVENT_CALL_SERVICE

@event_trigger(EVENT_CALL_SERVICE, "domain == 'lights' and service == 'turn_on'")
def monitor_light_turn_on_service(service_data=None):
    log.info(f"lights.turn_on service called with service_data={service_data}")
```

This wiki page gives more examples of built-in and user events and how to create triggers for them.

@mqtt_trigger

```
@mqtt_trigger(topic, str_expr=None, kwargs=None)
```

`@mqtt_trigger` subscribes to the given MQTT `topic` and triggers whenever a message is received on that topic. Multiple `@mqtt_trigger` decorators can be applied to a single function if you want to trigger off different mqtt topics.

An optional `str_expr` can be used to match the MQTT message data, and the trigger will only occur if that expression evaluates to `True` or non-zero. This expression has available these four variables:

- `trigger_type` is set to "mqtt"
- `topic` is set to the topic the message was received on
- `payload` is set to the string payload of the message
- `payload_obj` if the payload was valid JSON, this will be set to the native python object representing that payload.
- `qos` is set to the message QoS.

When the `@mqtt_trigger` occurs, those same variables are passed as keyword arguments to the function in case it needs them. Additional keyword parameters can be specified by setting the optional `kwargs` argument to a `dict` with the keywords and values.

Wildcards in topics are supported. The `topic` variables will be set to the full expanded topic the message arrived on.

NOTE: The MQTT Integration in Home Assistant must be set up to use `@mqtt_trigger`.

@state_active

```
@state_active(str_expr)
```

When any trigger occurs (whether time, state or event), the `@state_active` expression is evaluated. If it evaluates to `False` (or zero), the trigger is ignored and the trigger function is not called. Only a single `@state_active` decorator can be used per function - you can combine multiple conditions into `str_expr` using `any`, `all`, or logical operators like `or` or `and`.

This decorator is roughly equivalent to starting the trigger function with an `if` statement with the `str_expr` (the minor difference is that this decorator uses the `@state_trigger` variable value, if present, when evaluating `str_expr`, whereas an `if` statement at the start of the function uses its current value, which might be different if the state variable was changed immediately after the trigger, and the `.old` value is not available).

If the trigger was caused by `@state_trigger`, the prior value of the state variable that caused the trigger is available to `str_expr` with a `.old` suffix.

Inside the `str_expr`, undefined state variables, undefined state attributes, and undefined `.old` variables evaluate to `None`, rather than throwing an exception. Any `.old` variable will be `None` if the trigger is not a state trigger, if a different state variable change caused the state trigger, or if the state variable that caused the trigger was set for the first time (so there is no prior value).

@time_active

```
@time_active(time_spec, ..., hold_off=None)
```

`@time_active` takes zero or more strings that specify time-based ranges. Only a single `@time_active` decorator can be used per function. When any trigger occurs (whether time, state or event), each time range specification is checked. If the current time doesn't fall within any range specified, the trigger is ignored and the trigger function is not called. The optional numeric `hold_off` setting in seconds will ignore any triggers that are within that amount of time from the last successful one. Think of this as making the trigger inactive for that number of seconds immediately following each successful trigger. This can be used for rate-limiting trigger events or debouncing a noisy sensor.

Each string specification `time_spec` can take two forms:

- `"range(datetime_start, datetime_end)"` is satisfied if the current time is in the indicated range, including the end points. As in `@time_trigger`, the year or date can be omitted to specify daily ranges. If the end is prior to the start, the range is satisfied if the current time is either greater than or equal to the start or less than or equal to the end. That allows a range like: `@time_active("range(sunset - 20min, sunrise + 15min)")` to mean at least 20 minutes before sunset, or at least 15 minutes after sunrise (note: at latitudes close to the polar circles, there can be cases where the sunset time is after midnight, so it is before the sunrise time, so this might not work correctly; at even greater latitudes sunset and sunrise will not be defined at all since there might not be daily sunrises or sunsets).
- `"cron(min hr dom mon dow)"` is satisfied if the current time matches the range specified by the `cron` parameters. For example, if `hr` is `6-10` that means hours between 6 and 10 inclusive. If additionally `min` is `*` (i.e., any), then that would mean a time interval from 6:00 to immediately prior to 11:00.

Each argument specification can optionally start with `not`, which inverts the meaning of that range or cron specification. If you specify multiple arguments without `not`, they are logically or'ed together, meaning the active check is true if any of the (positive) time ranges are met. If you have several `not` arguments, they are logically and'ed together, so the active check will be true if the current time doesn't match any of the "not" (negative) specifications. `@time_active` allows multiple arguments with and without `not`. The condition will be met if the current time matches any of the positive arguments, and none of the negative arguments.

1.5.7 Other Function Decorators

@pyscript_compile

By default in pyscript all functions are async, so they cannot be used in `task.executor`, as callbacks or methods in python packages that expect regular functions, or used with built-in functions like `filter`, `map` or special class methods that are called by python internals (eg, `__getattr__` or `__del__`).

The `@pyscript_compile` decorator causes the function to be treated as native Python and compiled, which results in a regular python function being defined, and it will run at full compiled speed. A `lambda` function is automatically

compiled so it behaves like a regular python `lambda` function (which means the lambda function body cannot contain pyscript features).

For example:

```
@pyscript_compile
def incr(x):
    return x + 1

x = list(map(incr, [0, 5, 10]))
```

sets `x` to `[1, 6, 11]`.

One use for `@pyscript_compile` is to encapsulate functions that block (eg, doing I/O), so they can be called from `task.executor`. This might be a more convenient way to create native python functions called by `task.executor`, instead of moving them all to an external module or package outside of `config/pyscript`. This example reads a file using a native compiled function called by `task.executor`:

```
@pyscript_compile
def read_file(file_name):
    try:
        with open(file_name, encoding="utf-8") as file_desc:
            return file_desc.read(), None
    except Exception as exc:
        return None, exc

contents, exception = task.executor(read_file, "config/configuration.yaml")
if exception:
    raise exception
log.info(f"contents = {contents}")
```

This is an experimental feature and might change in the future. Restrictions include:

- since it's native python, the function cannot use any pyscript-specific features; but since it's native python, all language features are available, including `open`, `yield` etc
- if you use `@pyscript_compile` on an inner function (ie, defined inside a pyscript function), then binding of variables defined outside the scope of the inner function does not work.

@pyscript_executor

The `@pyscript_executor` decorator does the same thing as `@pyscript_compile` and additionally wraps the compiled function with a call to `task.executor`. The resulting function is now a pyscript (async) function that can be called like any other pyscript function. This provides the cleanest way of defining a native python function that is executed in a new thread each time it is called, which is required for functions that does I/O or otherwise might block.

The file reading example above is simplified with the use of `@pyscript_executor`:

```
@pyscript_executor
def read_file(file_name):
    try:
        with open(file_name, encoding="utf-8") as file_desc:
            return file_desc.read(), None
    except Exception as exc:
        return None, exc

contents, exception = read_file("config/configuration.yaml")
if exception:
```

(continues on next page)

(continued from previous page)

```
    raise exception
log.info(f"contents = {contents}")
```

Notice that *read_file* is called like a regular function, and it automatically calls `task.executor`, which runs the compiled native python function in a new thread, and then returns the result.

@service(service_name, ..., supports_response="none")

The `@service` decorator causes the function to be registered as a service so it can be called externally. The string `service_name` argument is optional and defaults to `"pyscript.FUNC_NAME"`, where `FUNC_NAME` is the name of the function. You can override that default by specifying a string with a single period of the form `"DOMAIN.SERVICE"`. Multiple arguments and multiple `@service` decorators can be used to register multiple names (eg, aliases) for the same function.

The `supports_response` keyword argument can be set to one of three string values: `"none"` (the default), `"only"`, or `"optional"`, depending on whether the service provides no response, always provides a response, or sometimes provides a response. Services responses were added in HASS 2023.7. The service response is a dictionary returned by the function.

Other trigger decorators like `@state_active` and `@time_active` don't affect the service. Those still allow state, time or other triggers to be specified in addition.

The function is called with keyword parameters set to the service call parameters, plus `trigger_type` is set to `"service"`. The function definition should specify all the expected keyword arguments to match the service call parameters, or use the `**kwargs` argument declaration to capture all the keyword arguments.

The `doc_string` (the string immediately after the function declaration) is used as the service description that appears is in the Services tab of the Developer Tools page. The function argument names are used as the service parameter names, but there is no description.

Alternatively, if the `doc_string` starts with `yaml`, the rest of the string is used as a `yaml` service description. Here's the first example above, with a more detailed `doc_string` (for a more complete example and explanation of the service description, check the Home Assistant developer documentation):

```
@service
def hello_world(action=None, id=None):
    """yaml
name: Service example
description: hello_world service example using pyscript.
fields:
  action:
    description: turn_on turns on the light, fire fires an event
    example: turn_on
    required: true
    selector:
      select:
        options:
          - turn_on
          - fire
  id:
    description: id of light, or name of event to fire
    example: kitchen.light
    required: true
    selector:
      text:
"""
```

(continues on next page)

(continued from previous page)

```
    log.info(f"hello world: got action {action}")
    if action == "turn_on" and id is not None:
        light.turn_on(entity_id=id, brightness=255)
    elif action == "fire" and id is not None:
        event.fire(id)
```

@task_unique

```
@task_unique(task_name, kill_me=False)
```

This decorator is equivalent to calling `task.unique()` at the start of the function when that function is triggered. Like all the decorators, if the function is called directly from another Python function, this decorator has no effect. See *this section* for more details.

1.5.8 Functions

Most of these have been mentioned already, but here is the complete list of additional functions made available by `pyscript`.

Note that even though the function names contain a period, the left portion is not a class (e.g., `state` is not a class, and in fact isn't even defined). These are simply functions whose name includes a period. This is one aspect where the interpreter behaves slightly differently from real Python.

However, if you set a variable like `state`, `log` or `task` to some value, then the functions defined with that prefix will no longer be available, since the portion after the period will now be interpreted as a method or class function acting on that variable. That's the same behavior as Python - for example if you set `bytes` to some value, then the `bytes.fromhex()` class method is no longer available in the current scope.

State variables

State variables can be used and set just by using them as normal Python variables. However, there could be cases where you want to dynamically generate the variable name (eg, in a function or loop where the state variable name is computed dynamically). These functions allow you to get and set a variable using its string name. The set function also allows you to optionally set the attributes, which you can't do if you are directly assigning to the variable:

`state.delete(name)` Deletes the given state variable or attribute. The Python `del` statement can also be used to delete a state variable or attribute.

`state.get(name)` Returns the value of the state variable given its string `name`. A `NameError` exception is thrown if the name doesn't exist. If `name` is a string of the form `DOMAIN.entity.attr` then the attribute `attr` of the state variable `DOMAIN.entity` is returned; an `AttributeError` exception is thrown if that attribute doesn't exist.

`state.getattr(name)` Returns a `dict` of attribute values for the state variable `name` string, or `None` if it doesn't exist. Alternatively, `name` can be a state variable. In pyscript prior to 1.0.0, this function was `state.get_attr()`. That deprecated name is still supported, but it logs a warning message and will be removed in a future version.

`state.names(domain=None)` Returns a list of all state variable names (ie, `entity_ids`) of a domain. If `domain` is not specified, it returns all HASS state variable (`entity_id`) names.

`state.persist(entity_id, default_value=None, default_attributes=None)` Indicates that the entity `entity_id` should be persisted. Optionally, a default value and default attributes (a `dict`) can be specified, which are applied to the entity if it doesn't exist or doesn't have any attributes respectively.

"Persist" mean its value and attributes are preserved across HASS restarts. This only applies to entities in the `pyscript` domain (ie, name starts with `pyscript.`). See *this section* for more information

`state.set(name, value=None, new_attributes=None, **kwargs)` Sets the state variable to the given value, with the optional attributes. The optional 3rd argument, `new_attributes`, should be a `dict` and it will overwrite all the existing attributes if specified. If instead attributes are specified using keyword arguments, then just those attributes will be set and other attributes will not be affected. If no optional arguments are provided, just the state variable value is set and the attributes are not changed. If no value is provided, just the state attributes are set and the value isn't changed. To clear all the attributes, set `new_attributes={}`.

`state.setattr(name, value)` Sets a state variable attribute to the given value. The `name` should fully specify the state variable and attribute as a string in the form `DOMAIN.entity.attr`.

Note that in HASS, all state variable values are coerced into strings. For example, if a state variable has a numeric value, you might want to convert it to a numeric type (eg, using `int()` or `float()`). Attributes keep their native type.

Service Calls

`service.call(domain, name, blocking=False, return_response=False, **kwargs)` calls the service `domain.name` with the given keyword arguments as parameters. If `blocking` is `True`, pyscript will wait for the service to finish executing before continuing the current routine. If the service returns a response, set `return_response` to `True` (which also causes blocking), and the response will be returned by the function call. Note that starting in HASS 2023.7, the blocking timeout parameter `limit` is no longer supported.

`service.has_service(domain, name)` returns whether the service `domain.name` exists.

Event Firing

`event.fire(event_type, **kwargs)` sends an event with the given `event_type` string and the keyword parameters as the event data.

Logging

Five logging functions are provided, with increasing levels of severity:

`log.debug(str)` log a message at debug level

`log.info(str)` log a message at info level

`log.warning(str)` log a message at warning level

`log.error(str)` log a message at error level

`print(str)` same as `log.debug(str)`; currently `print` doesn't support other arguments.

The `Logger` component can be used to specify the logging level. Log messages below the configured level will not appear in the log. Each log message function uses a log name of the form:

```
custom_components.pyscript.file.SCRIPTNAME.FUNCNAME
```

where `FUNCNAME` is the name of the top-level Python function (e.g., the one called by a trigger or service), defined in the script file `SCRIPTNAME.py`. See the *Global Context* section for the logging paths for other cases.

That allows you to set the log level for each Python top-level script or function separately if necessary. That setting also applies to any other Python functions that the top-level Python function calls. For example, these settings:

```
logger:
  default: info
  logs:
    custom_components.pyscript.file: info
    custom_components.pyscript.file.my_script.my_function: debug
```

will log all messages at `info` or higher (ie: `log.info()`, `log.warning()` and `log.error()`), and inside `my_function` defined in the script file `my_script.py` (and any other functions it calls) will log all messages at `debug` or higher.

Note that in Jupyter, all the `log` functions will display output in your session, independent of the `logger` configuration settings.

Changing the log level via the main configuration yaml file will typically require a restart of HASS. To avoid that, you can set the log level for any specific logging path by calling the `logger.set_level` service from Jupyter, VSC, or in a `pyscript` script; for example:

```
logger.set_level(**{"custom_components.pyscript.file.my_script": "debug"})
```

will change the log level for the `my_script.py` script to `debug`, while:

```
logger.set_level(**{"custom_components.pyscript.file.my_script.func1": "debug
↪"})
```

will change the log level for the `func1()` function in `my_script.py` script to `debug`. You can change multiple components in a single call just by adding them to the `**kwargs` dict.

Alternatively, you can use the Developer Tools to call the `logger.set_level` service with the logging path and desired level as parameters.

Tasks

A task is a lightweight execution context that runs a function inside an event loop, effectively providing asynchronous (actually collaborative serial) execution. They are part of Python's `asyncio` package and are central to how HASS and pyscript handle multiple activities. Tasks can run any Python code, provided it does not block (eg, for I/O) or run without giving up control for too long (which will prevent other tasks from running).

Task Management

These functions allow new tasks to be created and managed. Normally you won't need to create your own tasks, since trigger functions automatically create a new task to run the function every time the trigger occurs.

`task.create(func, *args, **kwargs)` Creates a new task and calls the given function `func` with the positional and keyword arguments. Returns the task id of the newly created task. The task id is an asyncio task object, which supports several methods. The function `func` runs asynchronously until it returns or the task is cancelled.

`task.cancel(task_id=None)` Cancels (kills) the task specified by the `task_id` returned by `task.create`. This is a simpler alternative to the `task_id.cancel()` method, which also requires waiting for the task to cancel. With no argument, *task.cancel* cancels the current task, which you might prefer to use in a trigger function on error, instead of a *return* or raising an exception.

`task.current_task()` Returns the task id of the current task.

task.name2id(name=None) Returns the task id given a name that task previously passed to `task.unique`. A `NameError` exception is raised if the task name is unknown. With no arguments it returns a dict mapping all names to task ids. The names are specific to the current global context.

task.wait(task_set) Waits until the given set of tasks complete. This function calls `asyncio.wait`, so it takes the same arguments. `task_set` is a set or list of task ids, and it returns two sets of done and pending task ids. An example:

```
done, pending = task.wait({task_id1, task_id2})
```

This waits until both tasks complete, and sets `done` to `{task_id1, task_id2}` and `pending` to an empty set (`pending` might be non-empty if you specify a timeout or `return_when`).

You can check if a task is done with `task_id.done()`. After a task has finished, the function's return value is available as `task_id.result()`, which raises an exception if the task is not finished or was cancelled.

task.add_done_callback(task_id, func, *args, **kwargs) This is a more convenient alternative to the `task_id.add_done_callback` method that supports both pyscript function (coroutine) and regular function callbacks. The function `func` with the specified arguments is called when the task completes. Multiple callbacks (with different functions) can be added to one task. If you use the same `func` argument it replaces the prior callback for that function (ie, only one done callback per function is supported).

task.remove_done_callback(task_id, func) This is a more convenient alternative to the `task_id.remove_done_callback` method that supports both pyscript function (coroutine) and regular function callbacks. This removes a previously added done callback, and the callback to that function will no longer occur when the task completes.

Task executor

If you call any Python functions that do I/O or otherwise block, they need to be run outside the main event loop using `task.executor`:

task.executor(func, *args, **kwargs) Run the given function in a separate thread. The first argument is the function to be called, followed by each of the positional or keyword arguments that function expects. The `func` argument can only be a regular Python function (eg, defined in an imported module), not a function defined in pyscript. `task.executor` waits for the function to complete in the other thread, and it returns the return value from the function `func`.

See *this section* for more information.

Task sleep

task.sleep(seconds) sleeps for the indicated number of seconds, which can be floating point. Do not import `time` and use `time.sleep()` - that will block lots of other activity.

Task unique

task.unique(task_name, kill_me=False) kills any currently running task that previously called `task.unique` with the same `task_name`. The name can be any string. If `kill_me=True` then the current task is killed if another task that is running previously called `task.unique` with the same `task_name`.

Note that `task.unique` is specific to the current global context, so names used in one global context will not affect another.

Once a task calls `task.unique` with a name, that name can be used to look up the task id by name using `task.name2id`. So even if `task.unique` is not used to kill a prior task, it can be used to associate that task with a name which might be helpful if you need to manage specific tasks (eg, cancel them, wait for them to complete, or get their return value when they finish).

A task can call `task.unique` multiple times with different names, which will kill any tasks that previously called `task.unique` with each of those names. The task continues to "own" all of those names, so any one of them could subsequently be used by a different task to kill the original task. Any of those names can be used to find the task id by name using `task.name2id`.

`task.unique` can also be called outside a function, for example in the preamble of a script file or interactively using Jupyter. That causes any currently running functions (ie, functions that have already been triggered and are running Python code) that previously called `task.unique` with the same name to be terminated. Since any currently running functions are not terminated on reload, this is the mechanism you can use should you wish to terminate specific functions on reload. If used outside a function or interactively with Jupyter, calling `task.unique` with `kill_me=True` causes `task.unique` to do nothing.

The `task.unique` functionality is also provided via a decorator `@task_unique`. If your function immediately and always calls `task.unique`, you could choose instead to use the equivalent function decorator form.

Task waiting

`task.wait_until(**kwargs)` allows functions to wait for events, using identical syntax to the decorators. This can be helpful if at some point during execution of some logic you want to wait for some additional triggers.

It takes the following keyword arguments (all are optional):

- `state_trigger=None` can be set to a string just like `@state_trigger`, or it can be a list of strings that are logically "or"ed together.
- `time_trigger=None` can be set to a string or list of strings with datetime specifications, just like `@time_trigger`.
- `event_trigger=None` can be set to a string or list of two strings, just like `@event_trigger`. The first string is the name of the event, and the second string (when the setting is a two-element list) is an expression based on the event parameters.
- `mqtt_trigger=None` can be set to a string or list of two strings, just like `@mqtt_trigger`. The first string is the MQTT topic, and the second string (when the setting is a two-element list) is an expression based on the message variables.
- `timeout=None` an overall timeout in seconds, which can be floating point.
- `state_check_now=True` if set, `task.wait_until()` checks any `state_trigger` immediately to see if it is already `True`, and will return immediately if so. If `state_check_now=False`, `task.wait_until()` waits until a state variable change occurs, before checking the expression. Using `True` is safer to help avoid race conditions, although `False` makes `task.wait_until()` behave like `@state_trigger`, which by default doesn't check at startup. However, if you use the default of `True`, and your function will call `task.wait_until()` again, it's recommended you set that state variable to some other value immediately after `task.wait_until()` returns. Otherwise the next call will also return immediately. Note that entries in `state_trigger` that are plain state variable names (which mean trigger on any change) are ignored during this initial check; only expressions are evaluated.
- `state_hold=None` is an optional numeric duration is seconds. If specified, any `state_trigger` delays returning for this amount of time. If the state trigger expression changes to `False` during that time, the trigger is canceled and a wait for a new trigger begins. If the state trigger expression changes, but is still `True` then the `state_hold` time is not restarted - `task.wait_until()` will return that number of seconds after the first

state trigger (unless a different trigger type or a `timeout` occurs first). This setting also applies to the initial check when `state_check_now=True`.

- `state_hold_false=None` requires the expression evaluate to `False` for this duration in seconds before a subsequent state trigger occurs. The default value of `None` means that the trigger can occur without the trigger expression having to be `False`. A value of `0` requires the expression become `False` before the trigger, but with no minimum time in that state. When `state_hold_false` is set, the state trigger expression is evaluated immediately. If `False` the `state_hold_false` period begins. If `True`, a wait for the next `False` value begins. If `state_check_now` is also set, `task.wait_until()` will still return immediately if the expression is initially `True`.

When a trigger occurs, the return value is a `dict` containing the same keyword values that are passed into the function when the corresponding decorator trigger occurs. There will always be a key `trigger_type` that will be set to:

- `"state"`, `"time"` or `"event"` when each of those triggers occur.
- `"timeout"` if there is a timeout after `timeout` seconds (the `dict` has no other values)
- `"none"` if you specify only `time_trigger` and no `timeout`, and there is no future next time that satisfies the trigger condition (e.g., a `range` or `once` is now in the past). Otherwise, `task.wait_until()` would never return.

In the special case that `state_check_now=True` and `task.wait_until()` returns immediately, the other return variables that capture the variable name and value that just caused the trigger are not included in the `dict` - it will just contain `trigger_type="state"`.

Here's an example. Whenever a door is opened, we want to do something if the door closes within 30 seconds. If a timeout of more than 30 seconds elapses (ie, the door is still open), we want to do some other action. We use a decorator trigger when the door is opened, and we use `task.wait_until` to wait for either the door to close, or a timeout of 30 seconds to elapse. The return value tells which of the two events happened:

```
@state_trigger("security.rear_door == 'open'")
def rear_door_open_too_long():
    """send alert if door is open for more than 30 seconds"""
    trig_info = task.wait_until(
                    state_trigger="security.rear_door == 'closed'",
                    timeout=30
                )
    if trig_info["trigger_type"] == "timeout":
        # 30 seconds elapsed without the door closing; do some actions
        pass
    else:
        # the door closed within 30 seconds; do some other actions
        pass
```

`task.wait_until()` is logically equivalent to using the corresponding decorators, with some important differences. Consider these two alternatives, which each run some code whenever there is an event `test_event3` with parameters `args == 20` and `arg2 == 30`:

```
@event_trigger("test_event3", "arg1 == 20 and arg2 == 30")
def process_test_event3(**trig_info):
    # do some things, including waiting a while
    task.sleep(5)
    # do some more things
```

versus:

```
@time_trigger    # empty @time_trigger means run the function on startup
def wait_for_then_process_test_event3():
```

(continues on next page)

(continued from previous page)

```
    while 1:
        trig_info = task.wait_until(
                        event_trigger=["test_event3", "arg1 == 20 and arg2 == 30"]
                    )
        # do some things, including waiting a while
        task.sleep(5)
        # do some more things
```

Logically they are the similar, but the important differences are:

- `task.wait_until()` only looks for the trigger conditions when it is called, and it stops monitoring them as soon as it returns. That means the trigger (especially an event trigger) could occur before or after `task.wait_until()` is called, and you will miss the event. In contrast, the decorator triggers monitor the trigger conditions continuously, so they will not miss state changes or events once they are initialized. The reason for the `state_check_now` argument, and its default value of `True` is to help avoid this race condition for state triggers. Time triggers should generally be safe.
- The decorators run each trigger function as a new independent task, and don't wait for it to finish. So a function will be run for every matching event. In contrast, if your code runs for a while before calling `task.wait_until()` again (e.g., `task.sleep()` or any code), or even if there is no other code in the `while` loop, some events or state changes of interest will be potentially missed.

Summary: use trigger decorators whenever you can. Be especially cautious using `task.wait_until()` to wait for events; you must make sure your logic is robust to missing events that happen before or after `task.wait_until()` runs.

Global Context

Each pyscript file that is loaded, and each Jupyter session, runs inside its own global context, which means its global variables and functions are isolated from each other (unless they are a module or package that is explicitly imported). In normal use you don't need to worry about global contexts. But for interactive debugging and development, you might want your Jupyter session to access variables and functions defined in a script file.

Here is the naming convention for each file's global context (upper case mean any value; lower case are actual fixed names):

pyscript file path	global context name
`pyscript/FILE.py`	`file.FILE`
`pyscript/modules/MODULE.py`	`modules.MODULE`
`pyscript/modules/MODULE/__init__.py`	`modules.MODULE`
`pyscript/modules/MODULE/FILE.py`	`modules.MODULE.FILE`
`pyscript/apps/APP.py`	`apps.APP`
`pyscript/apps/APP/__init__.py`	`apps.APP`
`pyscript/apps/APP/FILE.py`	`apps.APP.FILE`
`pyscript/scripts/FILE.py`	`scripts.FILE`
`pyscript/scripts/DIR1/FILE.py`	`scripts.DIR1.FILE`
`pyscript/scripts/DIR1/DIR2/FILE.py`	`scripts.DIR1.DIR2.FILE`

Note that if the package form of an app or module (`pyscript/apps/APP/__init__.py` or `pyscript/modules/MODULE/__init__.py`) exists, the module form (`pyscript/apps/APP.py` or `pyscript/modules/MODULE.py`) is ignored.

The logging path uses the global context name, so you can customize logging verbosity for each global context, to the granularity of specific functions eg:

```
logger:
  default: info
  logs:
    custom_components.pyscript.file: info
    custom_components.pyscript.file.my_script.my_function: debug
    custom_components.pyscript.apps.my_app: debug
    custom_components.pyscript.apps.my_app.my_function: debug
```

Each Jupyter global context name is `jupyter_NNN` where `NNN` is a unique integer starting at 0.

You can set the log level for one or more logging paths by calling the `logger.set_level` service from Jupyter, VSC, or in a `pyscript` script; for example:

```
logger.set_level(**{"custom_components.pyscript.file.my_script": "debug"})
```

will change the log level for the `my_script.py` script to `debug`.

When a script file has changed (or an app's configuration has changed, provided the `yaml` file is below the `pyscript` directory), a reload is triggered, and the corresponding global context whose names starts with `file.`, `modules.`, `apps.` or `scripts.` is removed. As each file is reloaded, the corresponding global context is created.

Three functions are provided for getting, setting and listing the global contexts. That allows you to interactively change the global context during a Jupyter session. You could also use these functions in your script files, but that is strongly discouraged because it violates the name space isolation among the script files. Here are the functions:

`pyscript.get_global_ctx()` returns the current global context name.

`pyscript.list_global_ctx()` lists all the global contexts, with the current global context listed first.

`pyscript.set_global_ctx(new_ctx_name)` sets the current global context to the given name.

When you exit a Jupyter session, its global context is deleted, which means any triggers, functions, services and variables you created are deleted (HASS state variables survive). If you switch to a script file's context, then any triggers, functions, services or variables you interactively create there will persist after you exit the Jupyter session. However, if you don't update the corresponding script file, whenever the script is modified and automatically reloaded, or upon HASS restart, those interactive changes will be lost, since reloading a script file recreates a new global context.

1.5.9 Advanced Topics

Workflow

Without Jupyter, the pyscript workflow involves editing scripts in the `<config>/pyscript` folder. Each time a file is changed, it is automatically reloaded. You will need to look at the log file for error messages (eg, syntax errors), or log output from your code.

If a module or app file (or any `yaml` files in the `pyscript` directory) has changed, all files in that module or app get reloaded too, and also any other files that import that module. If you want to reload all the files, call the `pyscript.reload` service with the optional parameter `global_ctx` set to `*`.

An excellent alternative to repeatedly modifying a script file is to use Jupyter notebook to interactively develop and test functions, triggers and services.

Jupyter auto-completion (with `<TAB>`) is supported in Jupyter notebook, console and lab. It should work after you have typed at least the first character. After you hit `<TAB>` you should see a list of potential completions from which you can select. It's a great way to easily see available state variables, functions or services.

In a Jupyter session, one or more functions can be defined in each code cell. Every time that cell is executed (eg, `<Shift>Return`), those functions are redefined, and any existing trigger decorators with the same function name are canceled and replaced by the new definition. You might have other function and trigger definitions in another

cell - they won't be affected (assuming those function names are different), and they will only be replaced when you re-execute that other cell.

When the Jupyter session is terminated, its global context is deleted, which means any trigger rules, functions, services and variables you created are deleted. The pyscript Jupyter kernel is intended as an interactive sandbox. As you finalize specific functions, triggers and automation logic, you should copy them to a pyscript script file, which will automatically reloaded once that file is written. That ensures they will be loaded and run each time you re-start HASS.

If a function you define has been triggered and is currently executing Python code, then re-running the cell in which the function is defined, or exiting the Jupyter session, will not stop or cancel the already running function. This is the same behavior as reload. In pyscript, each triggered function (ie, a trigger has occurred, the trigger conditions are met, and the function is actually executing Python code) runs as an independent task until it finishes. So if you are testing triggers of a long-running function (eg, one that uses `task.sleep()` or `task.wait_until()`) you could end up with many running instances. It's strongly recommended that you use `task.unique()` to make sure old running function tasks are terminated when a new one is triggered. Then you can manually call `task.unique()` to terminate that last running function before exiting the Jupyter session.

If you switch global contexts to a script file's context, and create some new variables, triggers, functions or services there, then those objects will survive the termination of your Jupyter session. However, if you reload that script, those newly-created objects will no longer exist. To make any additions or changes permanent (meaning they will be re-created on each reload or each time your restart HASS) then you should copy the changes or additions to one of your pyscript script files.

Importing

Pyscript supports importing two types of packages or modules:

- Pyscript code can be put into modules or packages and stored in the `<config>/pyscript/modules` folder. Any pyscript code can import and use these modules or packages. These modules are not autoloaded on startup; they are only loaded when another script imports them. Any changes to a module's files will cause all of the module files to be unloaded, and any scripts or apps that import that module will be reloaded. Imports of pyscript modules and packages are not affected by the `allow_all_imports` setting - if a file is in the `<config>/pyscript/modules` folder then it can be imported.

  Package-style layout is also supported where a PACKAGE is defined in `<config>/pyscript/modules/PACKAGE/__init__.py`, and that file can, in turn, do relative imports of other files in that same directory. This form is most convenient for sharing useful pyscript libraries, since all the files for one package are stored in its own directory.

- Installed Python packages can be imported. By default, pyscript only allows a short list of Python packages to be imported, for both security reasons and to reduce the risk that package functions that block doing I/O are called.

The rest of this section discusses the second style - importing installed Python modules and packages.

If you set the `allow_all_imports` configuration parameter, any available Python package can be imported. You should be cautious about setting this if you are going to install community pyscript code without inspecting it, since it could, for example, `import os` and call `os.remove()`. However, if you are developing your own code then there is no issue with enabling all imports.

Pyscript code is run using an asynchronous interpreter, which allows it to run in the HASS main event loop. That allows many of the "magic" features to be implemented without the user having to worry about the details. However, the performance will be much slower that regular Python code, which is typically compiled. Any Python packages you import will run at native, compiled speed.

So if you plan to run large chunks of code in pyscript without needing any of the pyscript-specific features, or you want access to native Python features that aren't supported in pyscript (like `yield`, `open`, `read` or `write`), you

might consider putting them in a package and importing it instead. That way it will run at native compiled speed and have full access to the native Python language.

One way to do that is in one of your pyscript script files, add this code:

```
import sys

if "/config/pyscript_modules" not in sys.path:
    sys.path.append("/config/pyscript_modules")
```

This adds the directory `/config/pyscript_modules` to Python's module search path (you should use the correct full path specific to your installation). You will need to set the `allow_all_imports` configuration parameter to `true` to allow importing of `sys`. You can then add modules (files ending in `.py`) to that folder, which will contain native python that is compiled when imported (note that none of the pyscript-specific features are available in those modules).

Pyscript can install required Python packages if they are missing. Depending on how you run HASS (eg, using a minimal Docker container) it might not be convenient to manually install Python packages using `pip`. If your pyscript code requires particular Python packages that are not already installed by HASS, add a `requirements.txt` file the `<config>/pyscript` directory. This file lists each required package one per line, with an optional version if you require a specific version or minimum version of that package, eg:

```
# this is a comment
aiohttp
amazing_stuff==3.1
another_package==5.1.2
```

When a specific version of a package is required, the == specifier must be used. Unpinned packages (no version specified) are also accepted, but the highest pinned version will always take precedence when a package has been specified as a requirement multiple times.

Each app's or module's directory (assuming they use the directory-form of a package) can also contain an optional `requirements.txt` file:

- `<config>/pyscript/modules/my-package-name/requirements.txt`
- `<config>/pyscript/apps/APP_NAME/requirements.txt`

That allows you to specify the specific requirements for each pyscript module or app. If you release or share your module or app, all its code and requirements are self-contained, and any user can simply install the files in that directory and the requirements will be checked on the next start of HASS or reload.

If a required package version differs from the installed one, no change is made since it's likely HASS has a requirement that pyscript should not change. In that case a warning message will be logged and the requirement will be skipped.

Trigger Closures

Pyscript supports trigger functions that are defined as closures, ie: functions defined inside another function. This allows you to easily create many similar trigger functions that might differ only in a couple of parameters (eg, a common function in different rooms or for each media setup). The trigger will be stopped when the function is no longer referenced in any scope. Typically the closure function is returned, and the return value is assigned to a variable. If that variable is re-assigned or deleted, the trigger function will be destroyed.

Here's an example:

```
def state_trigger_factory(sensor_name, trig_value):

    @state_trigger(f"input_boolean.{sensor_name} == '{trig_value}'")
```

(continues on next page)

(continued from previous page)

```
    def func_trig(value=None):
        log.info(f"func_trig: {sensor_name} is {value}")

    return func_trig

f1 = state_trigger_factory("test1", "on")
f2 = state_trigger_factory("test2", "on")
f3 = state_trigger_factory("test3", "on")
```

This creates three trigger functions that fire when the given sensor `input_boolean.testN` is `on`. If you re-assign or delete `f1` then that trigger will be destroyed, and the other two will not be affected. If you repeatedly re-run this block of code in Jupyter the right thing will happen - each time it runs the old triggers are destroyed when the variables are re-assigned.

Any data type could be used to maintain a reference to the trigger function. For example a list could be manually built:

```
input_boolean_test_triggers = [
    state_trigger_factory("test1", "on"),
    state_trigger_factory("test2", "on"),
    state_trigger_factory("test3", "on")
]
```

or dynamically in a loop:

```
input_boolean_test_triggers = []
for i in range(1, 4):
    input_boolean_test_triggers.append(state_trigger_factory(f"test{i}", "on"))
```

If you are writing a factory function and you prefer the caller not to bother with maintaining variables with the closure functions, you could move the appending into the function and use a global variable (a class could also be used):

```
input_boolean_test_triggers = []

def state_trigger_factory(sensor_name, trig_value):

    @state_trigger(f"input_boolean.{sensor_name} == '{trig_value}'")
    def func_trig(value=None):
        log.info(f"func_trig: {sensor_name} is {value}")

    input_boolean_test_triggers.append(func_trig)

state_trigger_factory("test1", "on")
state_trigger_factory("test2", "on")
state_trigger_factory("test3", "on")
```

Notice there is no return value from the factory function.

A `dict` could be used instead of a list, with a key that combines the unique parameters of the trigger. That way a new trigger with the same parameters will replace an old one when the `dict` entry is set, if that's the behavior you want.

Accessing YAML Configuration

Pyscript binds all of its `yaml` configuration to the variable `pyscript.config`. That allows you to add configuration settings that can be processed by your pyscript code. Additionally, an application's configuration (eg, for an application `app_name`, all the settings in `app_name` below `apps`) are available in the variable `pyscript.app_config` in the global scope of the application's main file (eg, `apps/app_name.py` or `apps/app_name/`

__init__.py). Note that pyscript.app_config is not defined in regular scripts - only in each application's main file.

One motivation is to allow pyscript apps to be developed and shared that can instantiate triggers and logic based on yaml configuration. That allows other users to use and configure your pyscript code without needing to edit or even understand it - they just need to add the corresponding yaml configuration.

The configuration for an application should be placed below that application's name under the apps configuration key. For example, the settings for a pyscript application called auto_lights below an entry apps. That entry could contain a list of settings (eg, for handling multiple rooms or locations).

Here's an example yaml configuration with settings for two applications, auto_lights and motion_light:

```
pyscript:
  allow_all_imports: true
  apps:
    auto_lights:
      - room: living
        level: 60
        some_list:
         - 1
         - 20
      - room: dining
        level: 80
        some_list:
         - 1
         - 20
    motion_light:
      - sensor: rear_left
        light: rear_flood
      - sensor: side_yard
        light: side_flood
      - sensor: front_patio
        light: front_porch
  global:
    setting1: 10
    setting2: true
```

For the auto_lights application, those settings are available to that application's main source file (eg, apps/auto_lights.py or apps/auto_lights/__init__.py) in the global variable pyscript.app_config, which will be set to:

```
[
    {"room": "living", "level": 60, "some_list": [1, 20]},
    {"room": "dining", "level": 80, "some_list": [1, 20]},
],
```

The corresponding global pyscript.config variable value will be:

```
{
    "allow_all_imports": True,
    "apps": {
        "auto_lights": [
            {"room": "living", "level": 60, "some_list": [1, 20]},
            {"room": "dining", "level": 80, "some_list": [1, 20]},
        ],
        "motion_light": [
            {"sensor": "rear_left", "light": "rear_flood"},
```

(continues on next page)

(continued from previous page)

```
            {"sensor": "side_yard", "light": "side_flood"},
            {"sensor": "front_patio", "light": "front_porch"},
        ],
    },
    "global": {
        "setting1": 10,
        "setting2": True,
    },
}
```

Note that accessing `pyscript.config["apps"]` is deprecated. The `apps` entry will be removed in a future release so that apps do not have access to another app's configuration (which might include token, passwords or keys).

Your application code for `auto_lights` would be in either

- `<config>/pyscript/apps/auto_lights.py`
- `<config>/pyscript/apps/auto_lights/__init__.py`

It can simply iterate over `pyscript.app_config` settings up the necessary triggers and application logic, eg:

```
def setup_triggers(room=None, level=None, some_list=None):
    #
    # define some trigger functions etc
    #
    pass

for inst in pyscript.app_config:
    setup_triggers(**inst)
```

Validating the configuration can be done either manually or with the `voluptuous` package.

Secrets can also be bound to variables by substituting the value with `!secret my_secret`. This is useful for API keys or passwords.

Here is an example config for `service_checker` which needs a name (not a secret) and a URL and an API key (which are secret):

```
pyscript:
  allow_all_imports: true
  apps:
    service_checker:
      - service_name: my_service
        url: !secret my_secret_url
        api_key: !secret my_secret_api_key
```

Access to Hass

If the `hass_is_global` configuration setting is set (default is off), then the variable `hass` is available as a global variable in all pyscript contexts. That provides significant flexibility in accessing HASS internals for cases where pyscript doesn't provide some binding or access.

Ideally you should only use `hass` for read-only access. However, you do need a good understanding of `hass` internals and objects if you try to call functions or update anything. With great power comes great responsibility!

For example, you can access configuration settings like `hass.config.latitude` or `hass.config.time_zone`.

You can use `hass` to compute sunrise and sunset times using the same method HASS does, eg:

```
import homeassistant.helpers.sun as sun
import datetime

location = sun.get_astral_location(hass)
sunrise = location[0].sunrise(datetime.datetime.today()).replace(tzinfo=None)
sunset = location[0].sunset(datetime.datetime.today()).replace(tzinfo=None)
print(f"today sunrise = {sunrise}, sunset = {sunset}")
```

(Note that the `sun.sun` attributes already provide times for the next sunrise and sunset, so this example is a bit contrived. Also note this only applies to HA 2021.5 and later; prior to that, `sun.get_astral_location()` doesn't return the elevation, so replace `location[0]` with `location` in the two expressions if you have an older version of HA.)

Here's another method that uses the installed version of `astral` directly, rather than the HASS helper function. HA 2021.5 upgraded to `astral 2.2`, and here's one way of getting sunrise using this version (prior to this HA used a very old version of `astral`).

```
import astral
import astral.location
import datetime

here = astral.location.Location(astral.LocationInfo("City", "Country", str(hass.
↪config.time_zone),
                                                    hass.config.latitude, hass.config.
↪longitude))
sunrise = here.sunrise(datetime.datetime.today()).replace(tzinfo=None)
sunset = here.sunset(datetime.datetime.today()).replace(tzinfo=None)
print(f"today sunrise = {sunrise}, sunset = {sunset}")
```

If there are particular HASS internals that you think many pyscript users would find useful, consider making a feature request or PR so it becomes a built-in feature in pyscript, rather than requiring users to always have to delve into `hass`.

Avoiding Event Loop I/O

All pyscript code runs in the HASS main event loop. That means if you execute code that blocks, for example doing I/O like reading or writing files or fetching a URL, then the main loop in HASS will be blocked, which will delay all other tasks.

All the built-in functionality in pyscript is written using asynchronous code, which runs seamlessly together with all the other tasks in the main event loop. However, if you import Python packages and call functions that block (eg, file or network I/O) then you need to run those functions outside the main event loop. That can be accomplished wrapping those function calls with the `task.executor` function, which runs the function in a separate thread:

`task.executor(func, *args, **kwargs)` Run the given function in a separate thread. The first argument is the function to be called, followed by each of the positional or keyword arguments that function expects. The `func` argument can only be a regular Python function, not a function defined in pyscript.

If you forget to use `task.executor`, you might get this warning from HASS:

```
WARNING (MainThread) [homeassistant.util.async_] Detected I/O inside the event loop.␣
↪This is
causing stability issues. Please report issue to the custom component author for␣
↪pyscript doing
I/O at custom_components/pyscript/eval.py, line 1583: return func(*args, **kwargs)
```

Currently the built-in functions that do I/O, such as `open`, `read` and `write` are not supported to avoid I/O in the main event loop, and also to avoid security issues if people share pyscripts. Also, the `print` function only logs a message, rather than implements the real `print` features, such as specifying an output file handle. If you want to do file I/O from pyscript, you have two choices:

- put the code in a separate native Python module, so that functions like `open`, `read` and `write` are available, and call the function in that module from pyscript using `task.executor`. See *Importing* for how to set Python's `sys.path` to import a local Python module.
- you could use the `os` package (which can be imported by setting `allow_all_imports`) and calling the low-level functions like `os.open` and `os.read` using `task.executor` to wrap every function.

Here's an example fetching a URL. Inside pyscript, this is the wrong way since it does I/O without using a separate thread:

```
import requests

url = "https://raw.githubusercontent.com/custom-components/pyscript/master/README.md"
resp = requests.get(url)
```

The correct way is:

```
import requests

url = "https://raw.githubusercontent.com/custom-components/pyscript/master/README.md"
resp = task.executor(requests.get, url)
```

An even better solution to fetch a URL is to use a Python package that uses asyncio, in which case there is no need for `task.executor`. In this case, `aiohttp` can be used (the `await` keyword is optional in pyscript):

```
import aiohttp

url = "https://raw.githubusercontent.com/custom-components/pyscript/master/README.md"
async with aiohttp.ClientSession() as session:
    async with session.get(url) as resp:
        print(resp.status)
        print(resp.text())
```

Here's another example that creates a client connection to a TCP server and exchanges messages in a manner that avoids event loop I/O by using `asyncio.open_connection`.

```
import asyncio

Reader, Writer = None, None

@time_trigger('startup')
def do_client_connection():
    global Reader, Writer
    Reader, Writer = asyncio.open_connection('127.0.0.1', 8956)

def client_send(message):
    if not Writer:
       raise("Client is not connected")
    Writer.write(message.encode())
    Writer.drain()
    return Reader.readline().decode()
```

This connects to the server (in this example at `127.0.0.1:8956`) at startup, and then you can call `client_send()` and it will send the message and return the reply.

This assumes the protocol is line-oriented; you could call `Reader.read()` instead if you want to read bytes instead of expecting a newline with `Reader.readline()`.

To test the code above, you can create a server by running `nc -l 8956` before you run the code. When you call `client_send("hello\n")` you should see the `hello` printed by `nc`. Then whatever you type back at `nc` will be returned by `client_send()`.

Persistent State

Pyscript has the ability to persist state variables in the `pyscript.` domain, meaning their values and attributes are preserved across HASS restarts. To specify that the value of a particular entity persists, you need to request persistence explicitly. This must be done in a code location that will be certain to run at startup.

```
state.persist('pyscript.last_light_on')

@state_trigger('binary_sensor.motion == "on"')
def turn_on_lights():
  light.turn_on('light.overhead')
  pyscript.last_light_on = "light.overhead"
```

With this in place, `state.persist()` will be called every time this script is parsed, ensuring the `pyscript.last_light_on` state variable state will persist between HASS restarts. If `state.persist` is not called on a particular state variable before HASS stops, then that state variable will not be preserved on the next start.

Language Limitations

Pyscript implements a Python interpreter in a fully-async manner, which means it can run safely in the main HASS event loop.

The language coverage is relatively complete, but it's quite possible there are discrepancies with Python in certain cases. If so, please report them.

Here are some areas where pyscript differs from real Python:

- The pyscript-specific function names and state names that contain a period are treated as plain identifiers that contain a period, rather than an attribute (to the right of the period) of an object (to the left of the period). For example, while `pyscript.reload` and `state.get` are functions, `pyscript` and `state` aren't defined. However, if you set `pyscript` or `state` to some value (ie: assign them as a variable), then `pyscript.reload` and `state.get` are now treated as accessing those attributes in the `pyscript` or `state` object, rather than calls to the built-in functions, which are no longer available. That's similar to regular Python, where if you set `bytes` to some value, the `bytes.fromhex` function is no longer available.
- Since pyscript is async, it detects whether functions are real or async, and calls them in the correct manner. So it's not necessary to use `async` and `await` in pyscript code - they are optional.
- All pyscript functions are async. So if you call a Python module that takes a pyscript function as a callback argument, that argument is an async function, not a normal function. So a Python module won't be able to call that pyscript function unless it uses `await`, which requires that function to be declared `async`. Unless the Python module is designed to support async callbacks, it is not currently possible to have Python modules and packages call pyscript functions. The workaround is to move your callbacks from pyscript and make them native Python functions; see *Importing*.
- Continuing that point, special methods (eg, *__eq__*) in a class created in *pyscript* will not work since they are async functions and Python will not be able to call them. The two workarounds are to use the `@pyscript_compile` decorator so the method is compiled to a native (non-ascync) Python function, or write your class in native Python and import it into `pyscript`; see *Importing*.

- The `import` function in `pyscript` fails to import certain complex packages. This is an open bug and it would be great if someone with some Python expertise could help fix it. In the meantime, the workaround is to import the module in a native Python file, and then import that shim module into pyscript. See *Importing*.
- pyscript and the HASS primitives that it uses are not thread safe - the code should only be executed in the main event loop. The `task.executor()` function is one way that regular Python code (not pyscript code) can safely be executed in a separate thread. The `threading` package can potentially be used (although that is untested), so long as any threads created only run regular Python code (and not call any pyscript functions, which are all async). Bad things will happen if you call pyscript functions from a thread you create; currently there isn't error checking for that case.

A handful of language features are not supported:

- generators and the `yield` statement; these are difficult to implement in an interpreter.
- built-in functions that do I/O, such as `open`, `read` and `write` are not supported to avoid I/O in the main event loop, and also to avoid security issues if people share pyscripts. The `print` function only logs a message, rather than implements the real `print` features, such as specifying an output file handle.
- The built-in function decorators (eg, `state_trigger`) aren't functions that can be called and used in-line. However, you can define your own function decorators that could include those decorators on the inner functions they define. Currently none of Python's built-in decorators are supported.

Pyscript can call Python modules and packages, so you can always write your own native Python code (eg, if you need a generator or other unsupported feature) that can be called by pyscript (see *Importing* for how to create and import native Python modules in pyscript).

1.6 Contributing

Contributions are welcome! You are encouraged to submit PRs, bug reports, feature requests or add to the Wiki with examples and tutorials. It would be fun to hear about unique and clever applications you develop. Please see this README for setting up a development environment and running tests.

Even if you aren't a developer, please participate in our discussions community. Helping other users is another great way to contribute to pyscript!

1.7 Releases and New Features

The releases and release notes are available on GitHub. Use HACS to install different versions of pyscript.

You can also install the master (head of tree) version from GitHub, either using HACS or manually. Because pyscript has quite a few unit tests, generally the master version should work ok. But it's not guaranteed to work at any random time, and newly-added features might change.

This is 1.5.0, released on July 30, 2023. Here is the documentation for that release. Here is the stable documentation for the latest release.

Over time, the master (head of tree) version in GitHub will include new features and bug fixes. Here is the latest documentation if you want to see the development version of the documentation.

If you want to see development progress since 1.5.0, see new features in the latest documentation, or look at the GitHub repository.

Planned new features post 1.5.0 include:

- Services defined in pyscript should support entity methods if they include an `entity_id` keyword argument.

- Consider supporting the built-in functions that do I/O, such as `open`, `read` and `write`, which are not currently supported to avoid I/O in the main event loop, and also to avoid security issues if people share pyscripts. The `print` function only logs a message, rather than implements the real `print` features, such as specifying an output file handle. Support might be added in the future using an executor job, perhaps enabled when `allow_all_imports` is set.

The new features since 1.5.0 in master include:

None yet.

Breaking changes since 1.5.0 include:

None yet.

Bug fixes since 1.5.0 include:

None yet.

1.8 Useful Links

- Documentation stable: latest release
- Documentation latest: current master in GitHub
- Discussion and help: community support and discussion
- GitHub repository (please add a star if you like pyscript!)
- Release notes: see what's changed
- Issues: report bugs or propose new features
- Wiki: share your pyscript apps and scripts
- Using Jupyter
- Jupyter notebook tutorial