automate Documentation

Release 0.10.8

Tuomas Airaksinen

Jun 24, 2017

Contents

CHAPTER 1

Table of Contents

Introduction

What is Automate?

Automate is a general purpose automatization library for Python. Its objective is to offer convenient and robust object-oriented programming framework for complex state machine systems. Automate can be used to design complex automation systems, yet it is easy to learn and fun to use. It was originally developed with home robotics/automatization projects in mind, but is quite general in nature and one could find applications from various fields that could take advantage of Automate. Automate can be embedded in other Python software as a component, which runs its operations in its own threads.

Highlights

- Supported hardware:
  - Raspberry Pi GPIO input/output ports (*Raspberry Pi GPIO Support for Automate* via RPIO library)
  - Arduino analog and digital input/output ports (*Arduino Support for Automate* via pyFirmata library)
  - Easy to write extensions to support other hardware, see *Making your own Automate Extensions*
- *System State Saving and Restoring via Serialization*
- Intelligent design:
  - Comprehensively tested via `py.test` unit/integration tests
  - Takes advantage of Traits library, especially its notification system.
  - IPython console to monitor, modify and control system on-the-fly
  - Versatile function/callable library to write state program logic
- RPC and Websocket interfaces (provided by *Remote Procedure Call Support for Automate* and *Web User Interface for Automate*) to connect between other applications or other Automate systems.

- Comprehensive and customizable Web User Interface via *Web User Interface for Automate*
- UML graphs can be drawn automaticlaly of the system (as can be seen in the examples of this documentation)

"Hello World" in Automate

Let us consider following short Automate program as a first example:

```
from automate import *

class MySystem(System):
    # HW swtich connected Raspberry Pi GPIO port 1
    hardware_switch = RpioSensor(port=1)
    # Switch that is controllable, for example, from WEB interface
    web_switch = UserBoolSensor()
    # Lamp relay that switches lamp on/off, connected to GPIO port 2
    lamp = RpioActuator(port=2)
    # Program that controls the system behaviour
    program = Program(
        active_condition=Or('web_switch', 'hardware_switch'),
        on_activate=SetStatus('lamp', True)
    )


my_system = MySystem(
    services=[WebService()]
)
```

This simple example has two sensors `hardware_switch`, `web_switch`, actuator (`lamp`) and a `program` that contains logic what to do and when. Here, `lamp` is switched on if either `web_switch` or `hardware_switch` has status True. `WebService` with default settings is enabled so that user can monitor system and set status of `web_switch`. The following figure (generated via `WebService` interface) illustrates the system in UML graph:

Original application

Automate was originally developed in order to enable simple and robust way of programming home automatization with Raspberry Pi minicomputer, to obtain automatization and automatic monitoring of rather complex planted aquarium safety/controlling system.

How to Install Automate?

Automate can be installed like ordinary python package. I recommend installation in within virtual environment (see virtualenv).

1. (optional): Create and start using virtualenv:

   ```
   mkvirtualenv automate
   workon automate
   ```

2. Install from pypi:

```
pip install automate
```

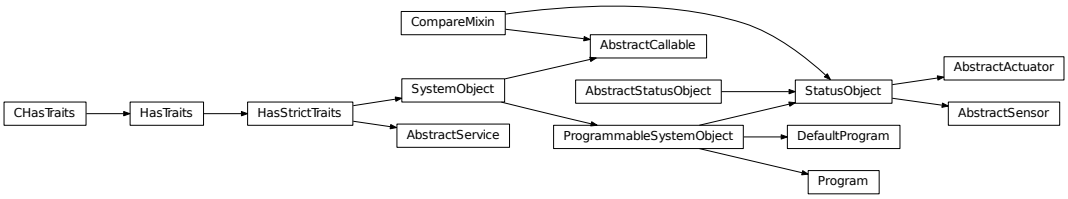
Optionally, you can specify some of the extras, i.e. web, rpc, raspberrypi, arduino:

```
pip install automate[web,rpc,raspberrypi,arduino]
```

or if you want them all:

```
pip install automate[all]
```

Automate Components



Automate system is built of the following components:

- **System** (derived by user from *System*) binds all parts together into a single state machine
- **Services** (subclassed of *AbstractService*) provide programming interfaces with user and devices that can be used by **SystemObjects**.
- **SystemObjects** (subclassed of *SystemObject* or *ProgrammableSystemObject*):
  - **Sensors** (subclassed on *AbstractSensor*) are used as an interface to the (usually read-only) state of device or software.
  - **Actuators** (subclassed on *AbstractActuator*) are used as an interface to set/write the state of device or software.
  - **Programs** (subclassed on *ProgrammableSystemObject*) define the logic between Sensors and Actuators. They are used to control statuses of Actuators, by rules that are programmed by using special **Callables** (subclasses of *AbstractCallable*) objects that depend on statuses of Sensors and other components. Also Sensors and Actuators are often subclassed from *ProgrammableSystemObject* so they also have similar features by themselves. Depending on the application, however, it might (or might not) improve readability if plain `Program` component is used.

All Automate components are derived from `HasTraits`, provided by Traits library, which provides automatic notification of attribute changes, which is used extensively in Automate. Due to traits, all Automate components are configured by passing attribute names as keyword arguments in object initialization (see for example attributes *pin* and *dev* traits of *ArduinoDigitalActuator* in the example below).

Automate system is written by subclassing *System* and adding there desired *SystemObject* as its attributes, such as in the following example:

```
from automate import *
class MySystem(System):
  mysensor = FloatSensor()
  myactuator = ArduinoDigitalActuator(pin=13, dev=0)
  myprogram = Program()
  ...
```

After defining the system, it can be instantiated. There, services with their necessary arguments can be explicitly defined as follows:

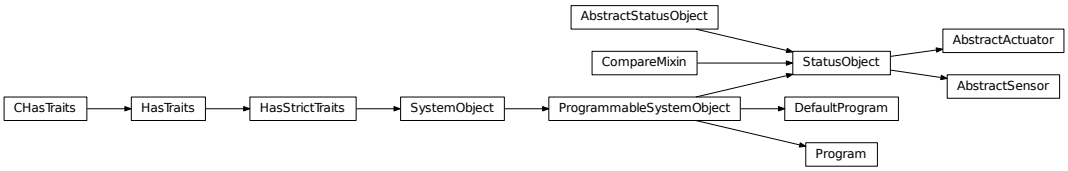
```
mysys = MySystem(services=[WebService(http_port=8080), ArduinoService(dev='/dev/ttyS0
↪')])
```

Some services (those that have *autoload* atribute set to True) do not need to be explicitly defined. For example, ArduinoService would be used automatically loaded because of the usage of ArduinoDigitalActuator, with default settings (dev='/dev/ttyUSB0'). Instantiating System will launch IPython shell to access the system internals from the command line. This can be prevented, if necessary, by defining keyword argument *exclude_services* as ['TextUIService'], which disables autoloading of *TextUIService*. For further information about services, see *Services*.

Programming Automate Objects

Programs

Program features are defined in *ProgrammableSystemObject* class. Program, DefaultProgram and *StatusObject* classes are subclassed from *ProgrammableSystemObject*, as can be seen in the following inheritance diagram.



Programs are used to define the logic on which system operates. Program behavior is determined by the conditions (*active_condition*, *update_condition*) and actions (*on_activate*, *on_update*, *on_deactivate*), that are of *AbstractCallable* type. Callables are special objects that are used to implement the actual programming of Automate program objects (see *Callables*). There are many special Callable classes to perform different operations (see *Builtin Callables*) and it is also easy to develop your own Callables (see *Deriving Custom Callables*).

All Sensors and Actuators that affect the return value of a condition callable, are *triggers* of a Callable. All actuators (and writeable sensors) that a callable may change, are *targets*. Whenever any of the triggers status change, programs conditions are automatically updated and actions are taken if appropriate condition evaluates as True.

Actions and conditions are used as follows. Programs can be either active or inactive depending on *active_condition*. When program actives (i.e. active_condition changes to True), *on_activate* action

is called. When program deactivates, *on_deactivate*, action is called, correspondingly. When program is active, its targets can be continuously manipulated by *on_update* callable, which is called whenever update_condition evaluates as `True`.

Actuator Status Manipulation

Program can control status of one or more actuators. Programs manipulate Actuator statuses the following way:

- One or more programs can control state of the same Actuator. Each program has *priority* (floating point number), so that the actual status of Actuator is determined by program with highest priority
- If highest priority program deactivates, the control of Actuator status is moved to the the second-highest priority active program.
- If there are no other Program, each Actuator has also one DefaultProgram, which then takes over Actuator control.

The following example application illustrates the priorities:

```
from automate import *
class MySystem(System):
    low_prio_prg = UserBoolSensor(priority=-5,
                                  active_condition=Value('low_prio_prg'),
                                  on_activate=SetStatus('actuator', 1.0),
                                  default=True,
                                  )
    med_prio_prg = UserBoolSensor(priority=1,
                                  active_condition=Value('med_prio_prg'),
                                  on_activate=SetStatus('actuator', 2.0),
                                  default=True,
                                  )
    high_prio_prg = UserBoolSensor(priority=5,
                                  active_condition=Value('high_prio_prg'),
                                  on_activate=SetStatus('actuator', 3.0),
                                  default=True,
                                  )
    inactive_high_prio_prg = UserBoolSensor(priority=6,
                                  active_condition=Value('inactive_high_prio_prg'),
                                  on_activate=SetStatus('actuator', 4.0),
                                  default=False,
                                  )

    actuator = FloatActuator()

ms = MySystem(services=[WebService()])
```

In this application, four programs (three manually defined programs and `DefaultProgram dp_actuator`) are active for actuator. The actual status of actuator (now: `3.0`) is determined by highest priority program. If `high_prio_prog` goes inactive (i.e. if its status is changed to `False`):

```
high_prio_prg.status = False
```

the status is then determined by `med_prio_prg` (=> `2.0`). And so on. All the active programs for actuator are visible in UML diagram. Red arrow shows the dominating program, blue arrows show the other non-dominating active programs and gray arrows show the inactive programs that have the actuator as a target (i.e. if they are activated,

they will manipulate the status of the actuator). `low_prio_prg` can never manipulate actuator status as its priority is lower than default program `dp_actuator` priority.

Program Features

Program features are defined in `ProgrammableSystemObject` class. Its definition is as follows:

Note: Unfortunately, due to current Sphinx autodoc limitation, all trait types are displayed in this documentation as `None`. For the real trait types, please see the source fode.

---

class `automate.program.`**`ProgrammableSystemObject`**(**args*, ***kwargs*)

System object with standard program features (i.e. conditions & actions).

`active_condition` = **None**

A condition Callable which determines the condition, when the program is activated. Program deactivates, when condition turns to False. When program is activated, on_activate action is executed. When program deactivates. on_deactivate is executed.

`on_activate` = **None**

An action Callable to be executed when Program actives.

`on_deactivate` = **None**

An action Callable to be executed when Program deactivates.

`update_condition` = **None**

When program is active, this is the condition Callable that must equal to `True` in order to on_update action to be executed. Whenever a trigger is changed, this condition is checked and if `True`, on_update is executed.

`on_update` = **None**

Action Callable to be executed if Program is active and update_condition is `True`.

`priority` = **None**

When programs sets Actuator status, the actual status of Actuator is determined by a program that has highest priority. Lower priority programs are stacked and used only if higher priority programs are deactivated.

`active` = **None**

Is program active? Automatically changed. In UIs you can fake the program active status by changing this. Normally do not change manually.

`status` = **None**

Status property is introduced to have interface compability with Status objects. For plain Programs, status equals to the result of its active condition Callable.

`actual_triggers` = **None**

(read-only property) Set of triggers, that cause this Program conditions to be checked (and actions to be executed). This data is updated from custom triggers list, conditions and actions.

`actual_targets` = **None**

(read-only property) Set of targets that this Program might touch. This data is updated from custom targets list and actions.

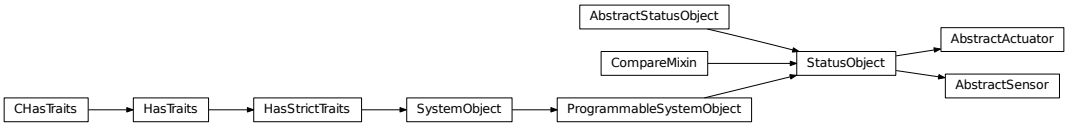
`triggers` = **None**

Custom set of additional triggers, whose status change will trigger this Program conditions/actions

exclude_triggers = None
Triggers in this set do not trigger the program actions/conditions even if they are introduced by Callables etc.

targets = None
Additional targets. Not usually needed, but if you want to set status for some reason by some custom function, for example, then you need to use this.

StatusObjects



Actuators (*AbstractActuator*) and sensors (*AbstractSensor*) are subclassed of *StatusObject*. The most important property is *status*, which may be of various data types, depending of the implementation defined in subclasses. Type of status is determined by `_status` trait.

There are couple of useful features in StatusObjects that may be used to affect when status is really changed. These are accessible via the following attributes:

- *safety_delay* and *safety_mode* can be used to define a minimum delay between status changes ("safety" ~ some devices might break if changed with big frequency)
- *change_delay* and *change_mode* can be used to define a delay which (always) takes place before status is changed.

Here, modes are one of `'rising'`, `'falling'`, `'both'`, default being `'rising'`. To disable functionality completely, set corresponding delay parameter to zero. Functions are described below.

Creating Custom Sensors and Actuators

Custom actuators and sensors can be easiliy written based on *AbstractActuator* and *AbstractSensor* classes, respectively.

As an example, we will define one of each:

```
# imports from your own library that you are using to define your sensor & actuator
from mylibrary import (setup_data_changed_callback,
                       fetch_data_from_my_datasource,
                       initialize_my_actuator_device,
                       change_status_in_my_actuator_device)

class MySensor(AbstractSensor):
    """
    Let us assume that you have your own library which has a status that you
    want to track in your Automate program.
    """
    # define your status data type
```

```
    _status = CBool
    def setup(self):
        setup_my_datasource()
        # we tell our library that update_status need to be called when status is
        # changed. We could use self.set_status directly, if library can pass
        # new status as an argument.
        setup_data_changed_callback(self.update_status)
    def update_status(self):
        # fetch new status from your datasource (this function is called by
        # your library)
        self.status = fetch_data_from_your_datasource()
    def cleanup(self):
        # define this if you need to clean things up when program is stopped
        pass

class MyActuator(AbstractActuator):
    # define your status data type. Transient=True is a good idea because
    # actuator status is normally determined by other values (sensors & programs etc)
    _status = CFloat(transient=True)
    def setup(self):
        initialize_my_actuator_device()
    def _status_changed(self):
        chagnge_status_in_my_actuator_device(self.status)
```

For more examples, look *builtin_sensors* and *builtin_actuators*. For more examples, see also *Extensions*, especially support modules for Arduino and Raspberry Pi IO devices)

StatusObject Definition

class automate.statusobject.**StatusObject**(**args*, ***kwargs*)

Baseclass for Sensors and Actuators

safety_delay = None

Determines minimum time required for switching. State change is then delayed if necessary.

safety_mode = None

Determines when *safety_delay* needs to be taken into account: when status is rising, falling or both.

change_delay = None

Similar to *safety_delay*, but just delays change to make sure that events shorter than change_delay are not taken into account

change_mode = None

As *safety_mode*, but for *change_delay*

silent = None

Do not emit actuator status changes into logs

debug = None

Print more debugging information into logs

changing = None

(property) Is delayed change taking place at the moment?

is_program

A property which can be used to check if StatusObject uses program features or not.

status = None

Status of the object.

get_status_display(***kwargs*)
Define how status is displayed in UIs (add units etc.).

get_as_datadict()
Get data of this object as a data dictionary. Used by websocket service.

set_status(*new_status*, *origin=None*, *force=False*)
For sensors, this is synonymous to:

```
sensor.status = new_status
```

For (non-slave) actuators, origin argument (i.e. is the program that is changing the status) need to be given,

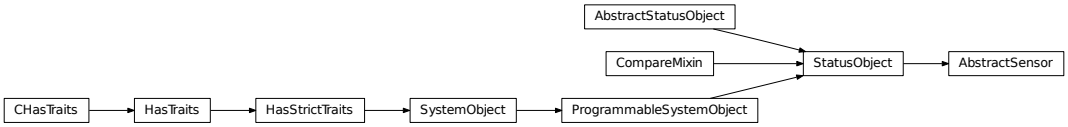
update_status()
In sensors: implement particular value reading from device etc. here (this calls set_status(value)). In actuators: set value in particular device. Implement in subclasses.

activate_program(*program*)
When program controlling this object activates, it calls this function.

deactivate_program(*program*)
When program controlling this object deactivates, it calls this function.

get_program_status(*program*)
Determine status of this object set by a particular program. Useful only for Actuators but defined here for interface compatibility.

Sensor Baseclass Definition



class automate.statusobject.**AbstractSensor**(**args*, ***kwargs*)
Base class for all sensors

user_editable = None
Is sensor user-editable in UIs. This variable is meant for per-instance tuning for Sensors, whereas *editable* is for per-class adjustment.

default = None
Default value for status

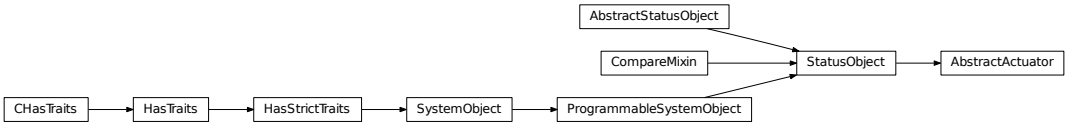
reset_delay = None
If non-zero, Sensor status will be reset to default after defined time (in seconds).

silent = None
Do not log status changes

set_status(*status*, *origin=None*, *force=False*)
Compatibility to actuator class. Also *SetStatus* callable can be used for sensors too, if so desired.

update_status()
A method to read and update actual status. Implement it in subclasses, if necessary

Actuator Baseclass Definition



class automate.statusobject.**AbstractActuator**(**args*, ***kwargs*)
Base class for all actuators.

default = None
Default value for status. For actuators, this is set by automatically created DefaultProgram dp_actuatorname

slave = None
If True, actual status can be set by any program anytime without restrictions.

program = None
A property giving current program governing the status of this actuator (program that has the highest priority)

priorities = None
This dictionary can be used to override program priorities.

Note: Keys here must be program names, (not Program instances).

---

default_program = None
Reference to actuators DefaultProgram

set_status(*status*, *origin=None*, *force=False*)
For programs, to set current status of the actuator. Each active program has its status in *program_stack* dictionary and the highest priority is realized in the actuator

activate_program(*program*)
Called by program which desires to manipulate this actuator, when it is activated.

deactivate_program(*program*)
Called by program, when it is deactivated.

update_program_stack()
Update *program_stack*. Used by programs _priority_changed attribute to reset ordering.

get_program_status(*prog*)
Give status defined by given program prog

program_stack = None
Program stack of current programs, sorted automatically by program priority.

program_status = None
Dictionary containing statuses set by each active program

Builtin Statusobject Types

Here are the definitions for the builtin statusobject types (sensors and actuators). More types are available in *Extensions*.

Builtin Sensors

Module for various Sensor classes.

class automate.sensors.builtin_sensors.**UserAnySensor**(**args*, ***kwargs*)
User editable sensor type that accepts values of any types

class automate.sensors.builtin_sensors.**UserBoolSensor**(**args*, ***kwargs*)
Boolean-valued user-editable sensor

class automate.sensors.builtin_sensors.**UserEventSensor**(**args*, ***kwargs*)
Boolean-valued user-editable sensor suitable for using for singular events.

After status has been changed to `True`, it changes automatically its status back to `False`.

class automate.sensors.builtin_sensors.**AbstractNumericSensor**(**args*, ***kwargs*)
Abstract class for numeric sensor types, that allows limiting value within a specific range.

If limiting values (*value_min*, *value_max*) are used, value that exceeds these limits, is clipped to the range.

value_min = None
Minimum allowed value for status

value_max = None
Maximum allowed value for status

class automate.sensors.builtin_sensors.**UserIntSensor**(**args*, ***kwargs*)
Integer-valued user-editable sensor

class automate.sensors.builtin_sensors.**UserFloatSensor**(**args*, ***kwargs*)
Float-valued user-editable sensor

class automate.sensors.builtin_sensors.**UserStrSensor**(**args*, ***kwargs*)
String-valued user-editable sensor

class automate.sensors.builtin_sensors.**CronTimerSensor**(**args*, ***kwargs*)
Scheduled start/stop timer. Both start and stop times are configured by cron-type string (see man 5 crontab for description of the definition format).

timer_on = None
Semicolon separated lists of cron-compatible strings that indicate when to switch status to `True`

timer_off = None
Semicolon separated lists of cron-compatible strings that indicate when to switch status to `False`

class automate.sensors.builtin_sensors.**FileChangeSensor**(**args*, ***kwargs*)
Sensor that detects file changes on filesystem. Integer valued status is incremented by each change.

filename = None
Name of file or directory to monitor

watch_flags = None
PyInotify flags to configure what file change events to monitor

class automate.sensors.builtin_sensors.**AbstractPollingSensor**(**args*, ***kwargs*)
Abstract baseclass for sensor that polls periodically its status

interval = None
How often to do polling

poll_active = None
This can be used to enable/disable polling

class automate.sensors.builtin_sensors.**PollingSensor**(**args*, ***kwargs*)
Polling sensor that uses a Callable when setting the status of the sensor.

status_updater = None
Return value of this Callable is used to set the status of the sensor when polling

type = None
If set, typeconversion to this is used. Can be any function or type.

class automate.sensors.builtin_sensors.**IntervalTimerSensor**(**args*, ***kwargs*)
Sensor that switches status between True and False periodically.

class automate.sensors.builtin_sensors.**SocketSensor**(**args*, ***kwargs*)
Sensor that reads a TCP socket.

Over TCP port, it reads data per lines and tries to set the status of the sensor to the value specified by the line. If content of the line is 'close', then connection is dropped.

host = None
Hostname/IP to listen. Use `'0.0.0.0'` to listen all interfaces.

port = None
Port to listen

stop = None
set to `True` to tell SocketSensor to stop listening to port

class automate.sensors.builtin_sensors.**ShellSensor**(**args*, ***kwargs*)
Run a shell command and follow its output. Status is set according to output, which is filtered through custome filter function.

cmd = None
Command can be, for example, 'tail -f logfile.log', which is convenient approach to follow log files.

caller = None
If this is set to true, caller object is passed to the filter function as second argument

filter = None
Filter function, which must be a generator, such as for example:

```
def filter(queue):
    while True:
        line = queue.get()
        if line == 'EOF':
            break
        yield line
```

or a simple line-by-line filter:

```
def filter(line):
    return processed(line)
```

Builtin Actuators

Module for builtin Actuator classes

class automate.actuators.builtin_actuators.**BoolActuator**(**args*, ***kwargs*)
Boolean valued actuator

class automate.actuators.builtin_actuators.**IntActuator**(**args*, ***kwargs*)
Integer valued actuator

class automate.actuators.builtin_actuators.**FloatActuator**(**args*, ***kwargs*)
Floating point valued actuator

class automate.actuators.builtin_actuators.**AbstractInterpolatingActuator**(**args*, ***kwargs*)
Abstract base class for interpolating actuators.

change_frequency = None
How often to update status (as frequency)

slave_actuator = None
Slave actuator, that does the actual work (set .slave attribute to True in slave actuator)

class automate.actuators.builtin_actuators.**ConstantSpeedActuator**(**args*, ***kwargs*)
Change slave status with constant speed

speed = None
Status change speed (change / second)

class automate.actuators.builtin_actuators.**ConstantTimeActuator**(**args*, ***kwargs*)
Change slave status in constant time

change_time = None
Time that is needed for change

Callables

Introduction

Callables are used like a small *programming language* to define the programming logic within the Automate system. All classes derived from *ProgrammableSystemObject* have five attributes that accept Callable type objects:

- Conditions
  - *active_condition*
  - *update_condition*
- Actions
  - *on_activate*
  - *on_update*
  - *on_deactivate*

Conditions determine *when* and actions, correspondingly, *what* to do.

Actions are triggered by *triggers* that are Sensors and Actuators. Triggers are collected from Callables (conditions and actions) automatically, and their status changes are subscribed and followed automatically by a

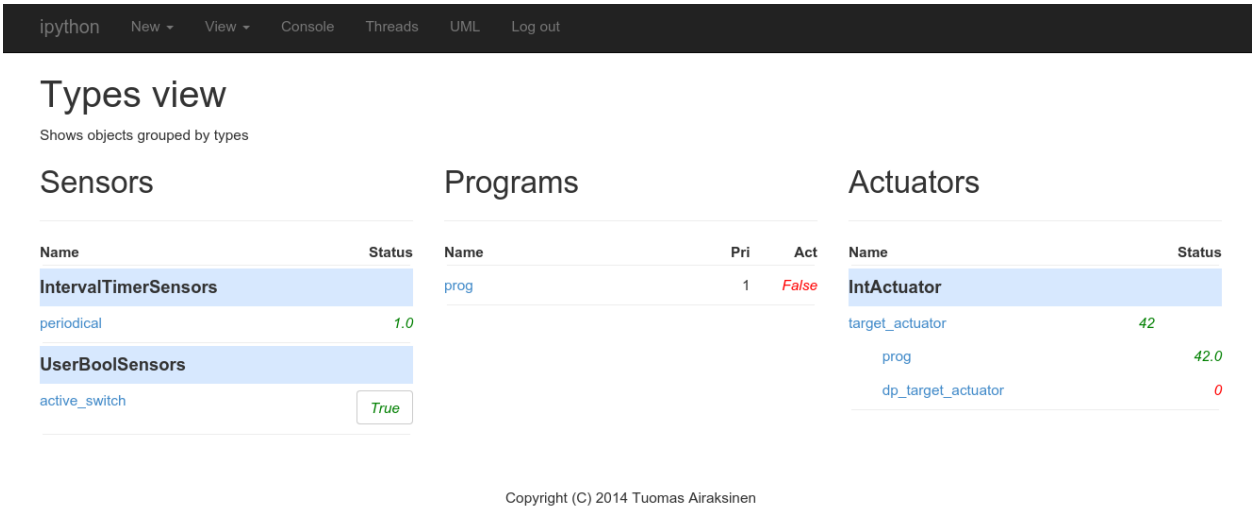
ProgrammableSystemObject. Thus, condition statuses are evaluated automatically, and actions are executed based on condition statuses.

Let us take a look at a small example that uses conditions and actions:

```
from automate import *

class CounterClock(System):
   active_switch = UserBoolSensor()
   periodical = IntervalTimerSensor(interval=1)

   target_actuator = IntActuator()

   prog = Program(
               active_condition = Value(active_switch),
               on_activate = SetStatus(target_actuator, 0),
               on_update = SetStatus(target_actuator,
                                     target_actuator + 1),
               triggers = [periodical],
               exclude_triggers = [target_actuator],
               )

s = CounterClock(services=[WebService(read_only=False)])
```

When user has switched `active_switch` sensor to `True`, this simple program will start adding +1 to target_actuator value every second. Because `periodical` is not used as a trigger in any action/condition, we need to explicitly define it as a trigger with triggers attribute. Correspondingly, `target_actuator` is automatically collected as prog's trigger (because it is the second argument of SetStatus), so we need to explicitly exclude it with `exclude_triggers` attribute.

Tip: Try the code yourself! Just cpaste the code into your IPython shell and go to http://localhost:8080 in your browser! Screenshot:



Deriving Custom Callables

A collection of useful Callables is provided by *builtin_callables* module. It is also easy to derive custom callables from *AbstractCallable* baseclass. For most cases it is enough to re-define *call()* method.

If Callable utilizes threads (like *Delay*, *WaitUntil* and *While*) and continues as an background process after returning from call method, it is also necessary to define *cancel()* that notifies threads that their processing must be stopped. These threaded Callables can store their threads and other information in *state* dictionary, which stores information per caller Program. Per-caller state information is fetched via *get_state()*. After data is no longer needed, it must be cleared with *del_state()* method.

Arguments given to Callable are stored in `_args` and keyword arguments in `_kwargs`. There are the following shortcuts that may be used: *obj*, *value* and *objects*. When accessing these, it is necessary (almost) always to use *call_eval()* method, which evaluates concurrent status value out of Callable, StatusObject, or string that represents name of an object residing in System namespace. See more in the following section.

Trigger and Target Collection

Triggers and targets are automatically collected from Callables recursively. All Callable types can specify which arguments are considered as triggers and which are considered as targets, by defining *_give_triggers()* and *_give_targets()*, correspondingly.

As a general rule, Callable should not consider criteria conditions as triggers (for example the conditions of *If*, *Switch* etc).

Referring to Other Objects in Callables

Various system objects can be referred either by name (string), or by object references. Name is preferred, because it allows to refer to objects that are defined in different scopes (i.e. those that are defined either in *Groups* or later in the code).

If desired, automatic name referencing can be also disabled by setting *allow_name_referencing* False. Then it is possible to refer to other objects by using special construct Object('name').

All variables passed to Callables are/must be evaluated through *call_eval()* method, i.e. if Callables are used as arguments, they are evaluated by their *call()* method and *StatusObject*'s status attribute is used, respectively.

Callable Abstract Base Class definition

Callable classes are are subclassed of *AbstractCallable*.

class `automate.callable.`**AbstractCallable**(**args*, ***kwargs*)

A base class for subclassing Callables that are used in Program conditions and action attributes.

Callables are configured by giving them arguments and keyword arguments.They must always define *call()* method which defines their functionality.

triggers = None

Property that gives set of all *triggers* of this callable and it's children callables. Triggers are all those StatusObjects that alter the status (return value of *call()*) of Callable.

targets = None

Property that gives set of all *targets* of this callable and it's children callables. Targets are all those StatusObjects of which status the callable might alter in *call()*.

status = None
Read-only status property of the callable. Usefull only when callable is used as a condition. This automatically depends on all the StatusObjects below the Callable tree.

state = None
State dictionary that is used by *call()* and *cancel()* if some state variables are needed to be saved Remember to clean data in subclasses when it is no longer needed.

get_state(*caller*)
Get per-program state.

del_state(*caller*)
Delete per-program state.

on_setup_callable = None
Event that can be used to execute code right after callable setup. See *OfType*. Something that needs to be done manually this way, because Traits does not allow defining the order of subscribed function calls.

call_eval(*value*, *caller*, *return_value=True*, ***kwargs*)
Value might be either name registered in System namespace, or object, either StatusObject or Callable. If Callable, evaluate *call()* method. If StatusObject, return status.

setup_callable_system(*system*, *init=False*)
This function basically sets up *system*, if it is not yet set up. After that, other Callable initialization actions are performed.

Parameters **init** – value `True` is given when running this at the initialization phase. Then system attribute is set already, but callable needs to be initialized otherwise.

call(**args*, ***kwargs*)
The basic functionality of the Callable is implemented in this function. Needs to be defined in derived subclasses.

If callable is used as a Program condition, this must return the value of the condition (see for example conditions *And*, *Sum* etc.), otherwise return value is optional.

objects
Shortcut to `_args`.

obj
Shortcut property to the first stored object.

value
Shortcut property to the second stored object.

name_to_system_object(*value*)
Return object for given name registered in System namespace.

collect(*target*)
Recursively collect all potential triggers/targets in this node and its children. Define targets and triggers of this particular callable in `_give_triggers()` and `_give_targets()`.

Parameters **target** (*str*) – valid values: `'targets'` and `'triggers'`

children
A property giving a generator that goes through all the children of this Callable (not recursive)

_give_triggers()
Give all triggers of this object (non-recursive)

_give_targets()
Give all targets of this object (non-recursive)

cancel(*caller*)
Recursively cancel all threaded background processes of this Callable. This is called automatically for actions if program deactivates.

give_str()
Give string representation of the callable.

give_str_indented(*tags=False*)
Give indented string representation of the callable. This is used in *Web User Interface for Automate*.

Builtin Callables

Module *builtin_callables* provides classes that may be used to various purposes in Automate `Program`, in condition and action attributes. They are loaded automatically into `automate.callables` along with callables from possible installed extensions.

Builtin Callable Types

class automate.callables.builtin_callables.**Empty**(**args*, ***kwargs*)
Do nothing but return None. Default action in Programs.

Usage:

```
Empty()
```

class automate.callables.builtin_callables.**AbstractAction**(**args*, ***kwargs*)
Abstract base class for actions (i.e. callables that do something but do not necessarily return anything.

class automate.callables.builtin_callables.**Attrib**(**args*, ***kwargs*)
Give specified attribute of a object.

Parameters **bool** (*no_eval*) – if True, evaluation of object is skipped – use this to access attributes of SystemObjects

Usage & example:

```
Attrib(obj, 'attributename')
Attrib(sensor_name, 'status', no_eval=True)
```

class automate.callables.builtin_callables.**Method**(**args*, ***kwargs*)
Call method in an object with specified args

Usage:

```
Method(obj, 'methodname')
```

class automate.callables.builtin_callables.**Func**(**args*, ***kwargs*)
Call function with given arguments.

Usage & example:

```
Func(function, *args, **kwargs)
Func(time.sleep, 2)
```

Parameters **add_caller** (*bool*) – if True, then caller program is passed as first argument.

class automate.callables.builtin_callables.**OnlyTriggers**(**args*, ***kwargs*)
Baseclass for actions that do not have any targets (i.e. almost all actions).

class automate.callables.builtin_callables.**Log**(**args*, ***kwargs*)
Print callable argument outputs / other arguments to the log.

Usage:

```
Log(object1, object2, 'string1'...)
```

Parameters **log_level** (*str*) – Log level (i.e. logging function name) (default 'info')

class automate.callables.builtin_callables.**Debug**(**args*, ***kwargs*)
Same as *Log* but with debug logging level.

class automate.callables.builtin_callables.**ToStr**(**args*, ***kwargs*)
Return string representation of given arguments evaluated. Usage:

```
ToStr('formatstring {} {}', callable1, statusobject1)
```

Parameters **no_sub** (*bool*) – if True, removes format string from argument list. Then usage is simply:

```
ToStr(callable1, statusobject1, no_sub=True)
```

class automate.callables.builtin_callables.**Eval**(**args*, ***kwargs*)
Execute python command given as a string with eval (or exec).

Usage:

```
Eval("print time.{param}()", pre_exec="import time", param="time")
```

First argument: python command to be evaluated. If it can be evaluated by eval() then return value is the evaluated value. Otherwise, exec() is used and True is returned.

Parameters

- **pre_exec** (*str*) – pre-execution string. For example necessary import commands.
- **namespace** (*dict*) – Namespace. Defaults to locals() in *builtin_callables*.

Optionally, other keyword arguments can be given, and they are replaced in the first argument by format().

See also (and prefer using): *Func*

class automate.callables.builtin_callables.**Exec**(**args*, ***kwargs*)
Synonym to *Eval*

class automate.callables.builtin_callables.**GetService**(**args*, ***kwargs*)
Get service by name and number.

Usage:

```
GetService(name)
GetService(name, number)
```

Usage examples:

```
GetService('WebService')
GetService('WebService', 1)
```

class automate.callables.builtin_callables.**ReloadService**(**args*, ***kwargs*)
Reload given service.

Usage:

```
ReloadService(name, number)
ReloadService(name)
```

Usage examples:

```
ReloadService('WebService', 0)
ReloadService('ArduinoService')
```

class automate.callables.builtin_callables.**Shell**(**args*, ***kwargs*)
Execute shell command and return string value

Parameters

- **no_wait** (*bool*) – if True, execute shell command in new thread and return pid
- **output** (*bool*) – if True, execute will return the output written to stdout by shell command. By default, execution status (integer) is returned.
- **input** (*str*) – if given, input is passed to stdin of the given shell command.

Usage examples:

```
Shell('/bin/echo test', output=True) # returns 'test'
Shell('mplayer something.mp3', no_wait=True) # returns PID of mplayer
                                             # process that keeps running
Shell('/bin/cat', input='test', output=True) # returns 'test'.
```

class automate.callables.builtin_callables.**SetStatus**(**args*, ***kwargs*)
Set sensor or actuator value

Usage:

```
SetStatus(target, source)
# sets status of target to the status of source.
SetStatus(target, source, Force=True)
# sets status to hardware level even if it is not changed
SetStatus([actuator1, actuator2], [sensor1, sensor2])
# sets status of actuator 1 to status of sensor1 and
# status of actuator2 to status of sensor2.
```

class automate.callables.builtin_callables.**SetAttr**(*obj*, ***kwargs*)
Set object's attributes

Usage:

```
SetAttr(obj, attr=value, attr2=value2)
# performs setattr(obj, attr, value) and setattr(obj, attr2, value2).
```

class automate.callables.builtin_callables.**Changed**(**args*, ***kwargs*)
Is value changed since evaluated last time? If this is the first time this Callable is called (i.e. comparison to last value cannot be made), return True.

Usage:

```
Changed(sensor1)
```

class automate.callables.builtin_callables.**Swap**(**args*, ***kwargs*)
Swap sensor or BinaryActuator status (False to True and True to False)

Usage:

```
Swap(actuator1)
```

class automate.callables.builtin_callables.**AbstractRunner**(**args*, ***kwargs*)
Abstract baseclass for Callables that are used primarily to run other Actions

class automate.callables.builtin_callables.**Run**(**args*, ***kwargs*)
Run specified Callables one at time. Return always True.

Usage:

```
Run(callable1, callable2, ...)
```

class automate.callables.builtin_callables.**Delay**(**args*, ***kwargs*)
Execute commands delayed by time (in seconds) in separate thread

Usage:

```
Delay(delay_in_seconds, action)
```

class automate.callables.builtin_callables.**Threaded**(**args*, ***kwargs*)
Execute commands in a single thread (in order)

Usage:

```
Threaded(action)
```

class automate.callables.builtin_callables.**If**(**args*, ***kwargs*)
Basic If statement

Usage:

```
If(x, y, z) # if x, then run y, z, where x, y, and z are Callables or␣
↪StatusObjects
If(x, y)
```

class automate.callables.builtin_callables.**IfElse**(**args*, ***kwargs*)
Basic if - then - else statement

Usage:

```
IfElse(x, y, z) # if x, then run y, else run z, where x, y,
                # and z are Callables or StatusObjects
IfElse(x, y)
```

class automate.callables.builtin_callables.**Switch**(**args*, ***kwargs*)
Basic switch - case statement.

Two alternative usages:

- First argument switch criterion (integer-valued) and others are cases **OR**
- First argument is switch criterion and second argument is dictionary that contains all possible cases as keys and related actions as their values.

Usage:

```
Switch(criterion, choice1, choice2...) # where criteria is integer-valued
                                       # (Callable or StatusObject etc.)
                                       # and choice1, 2... are Callables.

Switch(criterion, {'value1': callable1, 'value2': 'callable2'})
```

class automate.callables.builtin_callables.**TryExcept**(**args*, ***kwargs*)
Try returning *x*, but if exception occurs in the value evaluation, then return *y*.

Usage:

```
Try(x, y) # where x and y are Callables or StatusObjects etc.
```

class automate.callables.builtin_callables.**Min**(**args*, ***kwargs*)
Give minimum number of given objects.

Usage:

```
Min(x, y, z...)
# where x,y,z are anything that can be
# evaluated as number (Callables, Statusobjects etc).
```

class automate.callables.builtin_callables.**Max**(**args*, ***kwargs*)
Give maximum number of given objects

Usage:

```
Max(x, y, z...)
# where x,y,z are anything that can be
# evaluated as number (Callables, Statusobjects etc).
```

class automate.callables.builtin_callables.**Sum**(**args*, ***kwargs*)
Give sum of given objects

Usage:

```
Sum(x, y, z...)
# where x,y,z are anything that can be
# evaluated as number (Callables, Statusobjects etc).
```

class automate.callables.builtin_callables.**Product**(**args*, ***kwargs*)
Give product of given objects

Usage:

```
Product(x, y, z...)
# where x,y,z are anything that can be
# evaluated as number (Callables, Statusobjects etc).
```

class automate.callables.builtin_callables.**Mult**(**args*, ***kwargs*)
Synonym of Product

class automate.callables.builtin_callables.**Division**(**args*, ***kwargs*)
Give division of given objects

Usage:

```
Division(x, y)
# where x,y are anything that can be
# evaluated as number (Callables, Statusobjects etc).
```

class automate.callables.builtin_callables.**Div**(**args*, ***kwargs*)
Synonym of Division

class automate.callables.builtin_callables.**Add**(**args*, ***kwargs*)
Synonym of Sum

class automate.callables.builtin_callables.**AbstractLogical**(**args*, ***kwargs*)
Abstract class for logic operations (*And*, *Or* etc.)

class automate.callables.builtin_callables.**Anything**(**args*, ***kwargs*)
Condition which gives *True* always

Usage:

```
Anything(x,y,z...)
```

class automate.callables.builtin_callables.**Or**(**args*, ***kwargs*)
Or condition

Usage:

```
Or(x,y,z...) # gives truth value of x or y or z or ,,,
```

class automate.callables.builtin_callables.**And**(**args*, ***kwargs*)
And condition

Usage:

```
And(x,y,z...) # gives truth value of x and y and z and ...
```

class automate.callables.builtin_callables.**Neg**(**args*, ***kwargs*)
Give negative of specified callable (minus sign)

Usage:

```
Neg(x) # returns -x
```

class automate.callables.builtin_callables.**Inverse**(**args*, ***kwargs*)
Give inverse of specified callable (1/something)

Usage:

```
Inv(x) # returns 1/x
```

class automate.callables.builtin_callables.**Inv**(**args*, ***kwargs*)
Synonym of Inverse

class automate.callables.builtin_callables.**Not**(**args*, ***kwargs*)
Give negation of specified object

Usage:

```
Not(x) # returns not x
```

class automate.callables.builtin_callables.**Equal**(**args*, ***kwargs*)
Equality condition, i.e. is x == y

Usage:

```
Equal(x, y) # returns truth value of x == y
```

class automate.callables.builtin_callables.**Less**(**args*, ***kwargs*)
Condition: is x < y

Usage:

```
Less(x,y) # returns truth value of x < y
```

class automate.callables.builtin_callables.**More**(**args*, ***kwargs*)
Condition: is x > y

Usage:

```
More(x,y) # returns truth value of x > y
```

class automate.callables.builtin_callables.**Value**(**args*, ***kwargs*)
Give specified value

Usage:

```
Value(x) # returns value of x. Used to convert StatusObject into Callable,
         # for example, if StatusObject status needs to be used directly
         # as a condition of Program condition attributes.
```

class automate.callables.builtin_callables.**AbstractQuery**(**args*, ***kwargs*)
Baseclass for query type of Callables, i.e. those that return set of objects from system based on given conditions.

class automate.callables.builtin_callables.**OfType**(**args*, ***kwargs*)
Gives all objects of given type that are found in System

Usage & example:

```
OfType(type, **kwargs)
OfType(AbstractActuator, exclude=['actuator1', 'actuator2'])
# returns all actuators in system, except those named 'actuator1' and 'actuator2'.
```

Parameters **exclude** (*list*) – list of instances to be excluded from the returned list.

class automate.callables.builtin_callables.**RegexSearch**(**args*, ***kwargs*)
Scan through string looking for a match to the pattern. Return matched parts of string by `re.search()`.

Parameters **group** (*int*) – Match group can be chosen by group number.

Usage & examples:

```
RegexSearch(match_string, content_to_search, **kwargs)

RegexSearch(r'(\d*)(\w*)', '12test')          # returns '12'
RegexSearch(r'(\d*)(\w*)', '12test', group=2) # returns 'test'
RegexSearch(r'testasfd', 'test')              # returns ''
```

Tip: More examples in unit tests

class automate.callables.builtin_callables.**RegexMatch**(**args*, ***kwargs*)
Try to apply the pattern at the start of the string. Return matched parts of string by `re.match()`.

Parameters **group** (*int*) – Match group can be chosen by group number.

Usage & examples from unit tests:

```
RegexMatch(match_string, content_to_search, **kwargs)

RegexMatch(r'heptest', 'heptest')                # returns 'heptest'
RegexMatch(r'heptest1', 'heptest')               # returns ''
RegexMatch(r'(hep)te(st1)', 'heptest1', group=1) # returns 'hep'
RegexMatch(r'(hep)te(st1)', 'heptest1', group=2) # returns 'st1'
```

Tip: More examples in unit tests

class automate.callables.builtin_callables.**RemoteFunc**(**args*, ***kwargs*)
Evaluate remote function via XMLRPC.

Usage:

```
RemoteFunc('host', 'funcname', *args, **kwargs)
```

class automate.callables.builtin_callables.**WaitUntil**(**args*, ***kwargs*)
Wait until sensor/actuator/callable status changes to True and then execute commands. WaitUntil will return immediately and only execute specified actions after criteria is fullfilled.

Usage:

```
WaitUntil(sensor_or_callable, Action1, Action2, etc)
```

Note: No triggers are collected from WaitUntil

class automate.callables.builtin_callables.**While**(**args*, ***kwargs*)
Executes commands (in thread) as long as criteria (sensor, actuator, callable status) remains true. Flushes worker queue between each iteration such that criteria is updated, if executed actions alter it.

Parameters **do_after** (*Callable*) – given Callable is executed after while loop is finished.

Usage & example:

```
While(criteria, action1, action2, do_after=action3)

# Example loop that runs actions 10 times. Assumes s=UserIntSensor()
Run(
    SetStatus(s, 0),
    While(s < 10,
        SetStatus(s, s+1),
        other_actions
    )
)
```

Note: While execution is performed in separate thread

Note: No triggers are collected from While

class automate.callables.builtin_callables.**TriggeredBy**(**args*, ***kwargs*)
Return whether action was triggered by one of specified triggers or not

If no arguments, return the trigger.

Usage:

```
TriggeredBy() # -> returns the trigger
TriggeredBy(trig1, trig2...) #-> Returns if trigger is one of arguments
```

Automate System

Introduction

automate.system.System encapsulates the state machine parts into single object. It has already been explained how to use System. Here we will go further into some details.

Groups

In Automate system, it is possible to group objects by putting them to Groups. Grouping helps organizing objects in code level as well as in GUIs (*Web User Interface for Automate* etc.).

Here is an example:

```
class MySystem(System):
    class group1(Group):
        sensor1 = UserBoolSensor()
        sensor2 = UserBoolSensor()

    class group2(Group):
        sensor3 = UserBoolSensor()
        sensor4 = UserBoolSensor()
```

By adding SystemObject to a group, will assign it a tag corresponding to its groups class name. I.e. here, `sensor1` and `sensor2` will get tag *group:group1* and `sensor3` and `sensor4` will get tag *group:group2*.

System has single namespace dictionary that contains names of all objects. That implies that objects in different groups may not have same name.

System State Saving and Restoring via Serialization

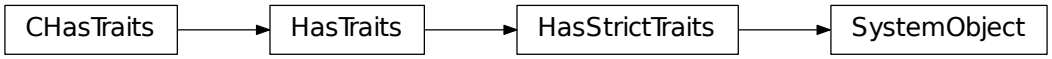
If System state is desired to be loaded later from periodically auto-saved state dumps, system can be instantiated via *load_or_create()* as follows:

```
my_system_instance = MySystem.load_or_create('my_statedump.dmp')
```

Then system state will be saved periodically (by default, once per 30 minutes) by *StatusSaverService*, which is automatically loaded service (see *Services*). If you desire to change interval, you need to explicitly define *dump_interval* as follows:

```
status_saver = StatusSaverService(dump_interval=10) # interval in seconds
my_system_instance = MySystem.load_or_create('my_statedump.dmp', services=[status_
↪saver])
```

SystemObject



SystemObject is baseclass for all objects that may be used within *System* (most importantly, Sensors, Actuators and Programs).

Due to multiple inheritance, many SystemObjects, such as Sensors (*AbstractSensor*), Actuators (*AbstractActuator*), and Programs (Program) can act in *multiple roles*, in addition to their primary role, as follows:

- Sensors and actuators can always be used also as a program i.e. they may have conditions and action callables defined, because they derive from *ProgrammableSystemObject*.
- Both Actuators and Sensors can be used as *triggers* in Callables and via them in Programs
- Also plain Programs can be used as a Sensor. Then its activation status (boolean) serves as Sensor status.

Sensors and Programs do not have Actuator properties (i.e. per-program statuses), but Sensor status can still be set/written by a Program, similarly to actuators with *slave* attribute set to True.

System Class Definition

class automate.system.**System**(*loadstate=None*, ***traits*)

allow_name_referencing = None
Allow referencing objects by their names in Callables. If disabled, you can still refer to objects by names by Object('name')

filename = None
Filename to where to dump the system state

logfile = None
Name of the file where logs are stored

print_level = None
Log level for the handler that writes to stdout

logger = None
Reference to logger instance (read-only)

log_handler = None
Instance to the log handler that writes to stdout

log_format = None
Format string of the log handler that writes to stdout

print_handler = None
Instance to the log handler that writes to logfile (read-only)

logfile_format = None
Format string of the log handler that writes to logfile

log_level = None
Log level of the handler that writes to logfile

default_services = None
Add here services that you want to be added automatically. This is meant to be re-defined in subclass.

services = None
List of services that are loaded in the initialization of the System.

exclude_services = None
List of servicenames that are desired to be avoided (even if normally autoloaded).

namespace = None
System namespace (read-only)

objects_sorted = None
Property giving objects sorted alphabetically (read-only)

sensors = None
Read-only property giving all sensors of the system

actuators = None
Read-only property giving all actuator of the system

programs = None
Read-only property giving all objects that have program features in use

ordinary_programs = None
Read-only property giving all `Program` objects

worker_autostart = None
Start worker thread automatically after system is initialized

pre_exit_trigger = None
Trigger which is triggered before quiting (used by Services)

all_tags = None
Read-only property that gives list of all object tags

two_phase_queue = None
Enable experimental two-phase queue handling technique (not recommended)

classmethod load_or_create(*filename=None*, ***kwargs*)
Load system from a dump, if dump file exists, or create a new system if it does not exist.

save_state()
Save state of the system to a dump file `System.filename`

cmd_namespace
A read-only property that gives the namespace of the system for evaluating commands.

get_unique_name(*obj*, *name=''*, *name_from_system=''*)
Give unique name for an Sensor/Program/Actuator object

services_by_name
A property that gives a dictionary that contains services as values and their names as keys.

service_names
A property that gives the names of services as a list

flush()
Flush the worker queue. Usefull in unit tests.

name_to_system_object(*name*)
Give SystemObject instance corresponding to the name

eval_in_system_namespace(*exec_str*)
Get Callable for specified string (for GUI-based editing)

register_service_functions(**funcs*)
Register function in the system namespace. Called by Services.

register_service(*service*)
Register service into the system. Called by Services.

request_service(*type*, *id=0*)
Used by Sensors/Actuators/other services that need to use other services for their operations.

cleanup()
Clean up before quitting

cmd_exec(*cmd*)
Execute commands in automate namespace

name = None
Name of the system (shown in WEB UI for example)

worker_thread = None
Reference to the worker thread (read-only)

post_init_trigger = None
Trigger which is triggered after initialization is ready (used by Services)

SystemObjects Class Definition

class automate.systemobject.**SystemObject**(*name=''*, ***traits*)
Baseclass for Programs, Sensor, Actuators

callables = []
Names of attributes that accept Callables. If there are custom callables being used, they must be added here. The purpose of this list is that these Callables will be initialized properly. *ProgrammableSystemObject* introduces 5 basic callables (see also *Programs*).

get_default_callables()
Get a dictionary of default callables, in form {name:callable}. Re-defined in subclasses.

system = None
Reference to System object

description = None
Description of the object (shown in WEB interface)

tags = None
Tags are used for (for example) grouping objects. See *Groups*.

name = None
Name property is determined by System namespace. Can be read/written.

hide_in_uml = None
If set to *True*, current SystemObject is hidden in the UML diagram of WEB interface.

view = ['hide_in_uml']
Attributes that can be edited by user in WEB interface

data_type = ''
The data type name (as string) of the object. This is written in the initialization, and is used by WEB interface Django templates.

editable = False
If editable=True, a quick edit widget will appear in the web interface. Define in subclasses.

object_type
A read-only property that gives the object type as string; sensor, actuator, program, other. Used by WEB interface templates.

logger = None
Python Logger instance for this object. System creates each object its own logger instance.

get_status_display(***kwargs*)
Redefine this in subclasses if status can be represented in human-readable way (units etc.)

get_as_datadict()
Get information about this object as a dictionary. Used by WebSocket interface to pass some relevant information to client applications.

setup(**args*, ***kwargs*)
Initialize necessary services etc. here. Define this in subclasses.

setup_system(*system*, *name_from_system=''*, ***kwargs*)
Set system attribute and do some initialization. Used by System.

setup_callables()
Setup Callable attributes that belong to this object.

cleanup()
Write here whatever cleanup actions are needed when object is no longer used.

Services

Introduction

There are two kinds of *Services* in Automate: *UserServices* and *SystemServices*.

SystemServices are mainly designed to implement a practical way of writing an interface between your custom SystemObjects and their corresponding resources (devices for example). For example, `RpioService` provide access to Raspberry Pi GPIO pins for `RpioActuator` and `RpioSensor` objects, and ArduinoService, correspondingly, provides access to Arduino devices for ArduinoActuator and ArduinoSensors. (Arduino and RPIO support are provided by extensions, see *Extensions*).

UserServices, on the other hand, provide user interfaces to the system. For example, `WebService` provides access to the system via web browser, `TextUIService` via *IPython* shell and `RpcService` via XmlRPC (remote procedure call) interface for other applications.

If not automatically loaded (services with `autoload` set to `True`), they need to be instantiated (contrary to `SystemObject`) outside the System, and given in the initialization of the system (`services`). For example of initialization and configuring of `WebService`, see *"Hello World" in Automate*.

Services Class Definitions

class automate.service.**AbstractService**
Base class for System and UserServices

autoload = False
If set to *True*, service is loaded automatically (if not explicitly prevented in *automate.system.System.exclude_services*). Overwrite this in subclasses,

setup()
Initialize service here. Define in subclasses.

cleanup()
Cleanup actions must be performed here. This must be blocking until service is fully cleaned up.

Define in subclasses.

class automate.service.**AbstractUserService**
Baseclass for UserServices. These are set up on startup. They provide usually user interaction services.

class automate.service.**AbstractSystemService**
Baseclass for SystemServices. These are set up by when first requested by Sensor or other object.

Builtin Services

class automate.services.logstore.**LogStoreService**
Provides interface to log output. Used by WebService.

log_level = None
Log level

log_length = None
Log length

most_recent_line = None
The most recent log line is always updated here. t Subscription to this attribute can be used to follow new log entries.

class automate.services.statussaver.**StatusSaverService**
Service which is responsible for scheduling dumping system into file periodically.

dump_interval = None
Dump saving interval, in seconds. Default 30 minutes.

class automate.services.plantumlserv.**PlantUMLService**
Provides UML diagrams of the system as SVG images. Used by WebService.

PLantUMLService requires either PlantUML software (which is opensource software written in Java) to be installed locally (see http://plantuml.sourceforge.net/) or it is possible to use online service of plantuml.com In addition you need python package `plantuml` (available via PYPI).

url = None
URL of PlantUML Java Service. To use PlantUML online service, set this to 'http://www.plantuml.com/plantuml/svg/'

arrow_colors = None
Arrow colors as HTML codes stored as a dictionary with keys: controlled_target, active_target, inactive_target, trigger

background_colors = None
Background colors as HTML codes, stored as a dictionary with keys: program, actuator, sensor

write_puml(*filename=''*)
Writes PUML from the system. If filename is given, stores result in the file. Otherwise returns result as a string.

write_svg()
Returns PUML from the system as a SVG image. Requires plantuml library.

class automate.services.textui.**TextUIService**
Provides interactive Python shell frontend to the System. Uses IPython if it is installed. Provides couple of functions to the System namespace.

ls(*what*)
List actuators, programs or sensors (what is string)

lsa()
List actuators

lsp()
List programs

lss()
List sensors

help(**args*, ***kwargs*)
Print Automate help if no parameter is given. Otherwise, act as pydoc.help()

text_ui()
Start Text UI main loop

Extensions

Automate functionality can be easily extended by various extension modules, and it is also possible to make your own Automate extensions, for details see *Making your own Automate Extensions*. The following extensions are included in Automate:

Web User Interface for Automate

Introduction

Automate Web UI extension provides easy to use approach to monitoring and modifying Automate system and its components. Features:

- Displayed data updated in real time via Websocket
- Responsive design (using Bootstrap CSS & JS library), suitable for mobile and desktop use.
- Optional authentication
- Read-only and read-write modes.
  - Read-only mode allows only monitoring (default)
  - Read-write mode allows modifying the System by:
    - adding new Actuators / Sensors / Programs
    - modifying existing Actuators / Sensors / Programs
    - Quick editing of user-editable sensors from main views
- HTTP and secured HTTPS servers supported (powered by built in Tornado Web Server)

Main features are illustrated with a few screenshots:

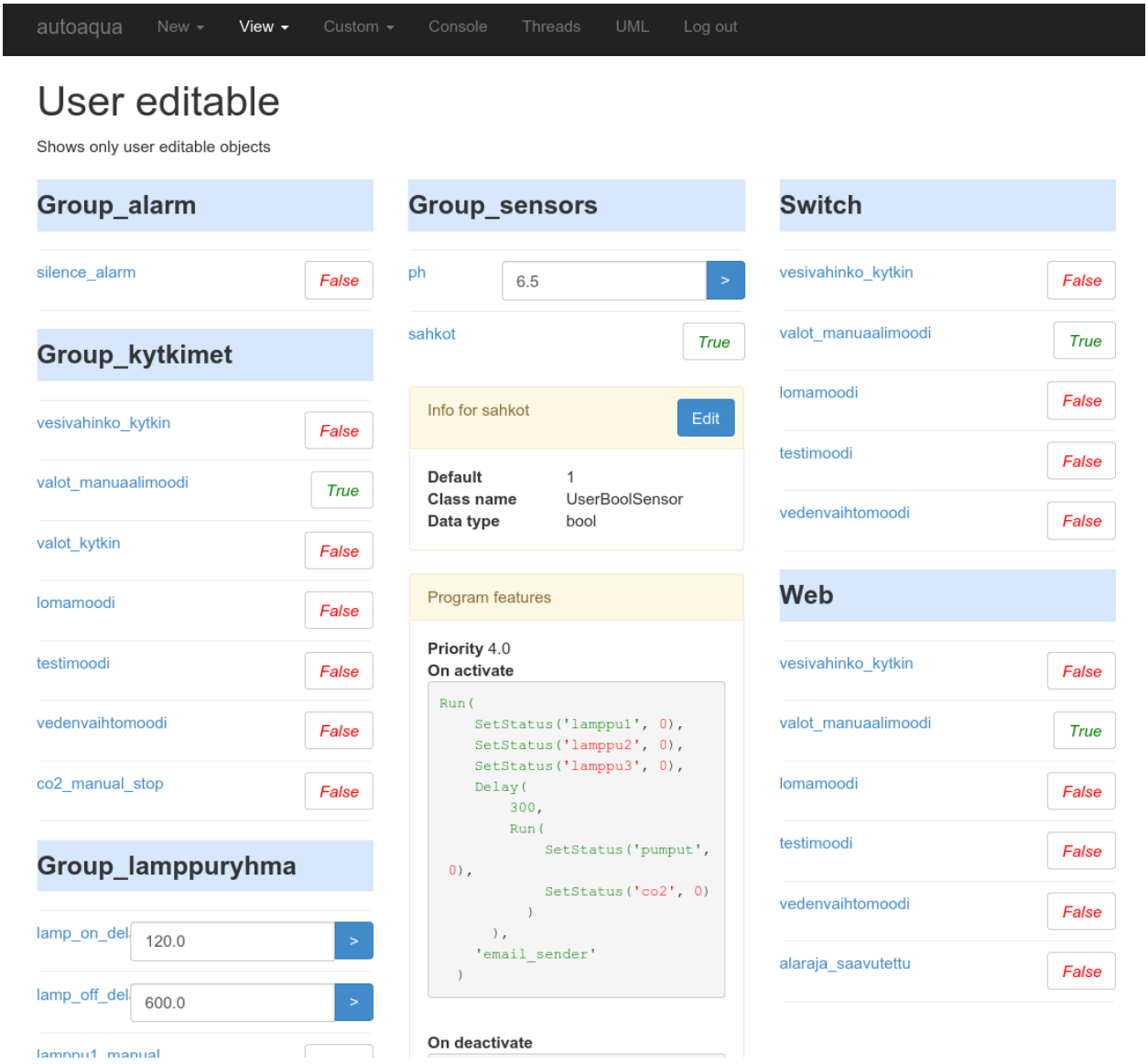
Installation

Install extras:

```
pip install automate[web]
```

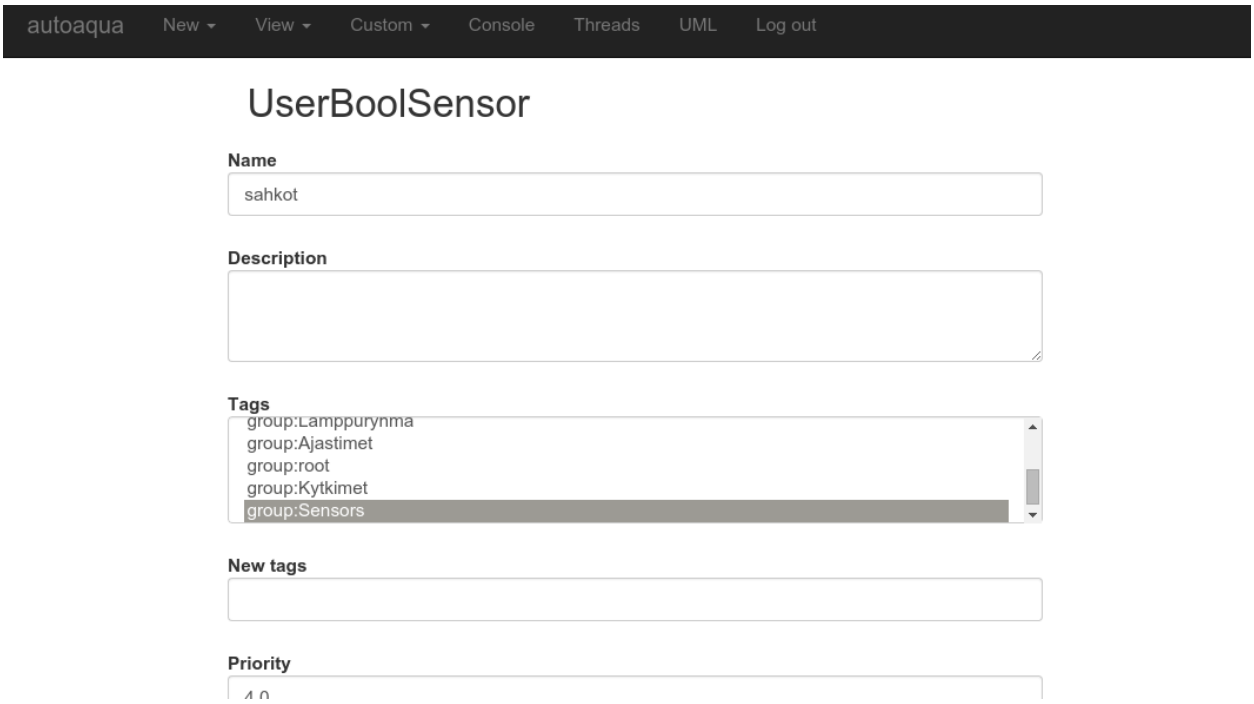
Main view

In main view you can observe actuator and sensor statuses in real time, and also easily access user-editable sensor statuses. Clicking object name will give more details of the selected item as well as 'edit' button. (only in read-write mode).



Edit view

In edit view you can edit almost all the attributes of the objects that are in the system. You can also create new ones.



UML view

In UML view you can see nice UML diagram of the whole system. To enable UML diagram, you need to set up `PlantUMLService`.

Console view

In console view you can see the log as well as type commands same way as in IPython shell.

Example application using Web UI

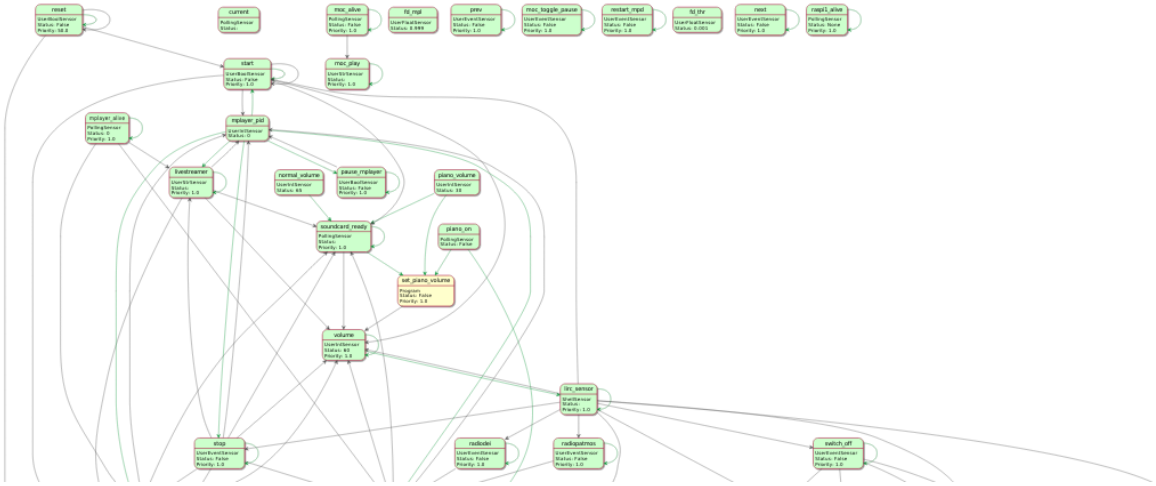
This is extended example of the *"Hello World" in Automate*, that opens two web services, one for port 8085 and another in 8086. It will go to UML view by default and in "User defined" -view you will see only `web_switch`.

```
from automate import *

class MySystem(System):
    # HW swtich connected Raspberry Pi GPIO port 1
    hardware_switch = RpioSensor(port=1)
    # Switch that is controllable, for example, from WEB interface
    web_switch = UserBoolSensor(tags=['user'])
    # Lamp relay that switches lamp on/off, connected to GPIO port 2
    lamp = RpioActuator(port=2)
    # Program that controls the system behaviour
```

musicserver New View Console Threads UML Log out

UML

UML graph powered by PlantUML. Does not refresh automatically! Get plain text. Show/hide legend



autoaqua New View Custom Console Threads UML Log out

Log console

```
16:42:13 autoaqua.TextUIService  working model.
16:42:13 autoaqua.TextUIService
16:42:13 autoaqua.TextUIService  Any Python commands are accepted. You may refer to all your
16:42:13 autoaqua.TextUIService  automate objects by their name attribute. You may do anything to
16:42:13 autoaqua.TextUIService  modify the state on fly by this field, for example, create new programs,
16:42:13 autoaqua.TextUIService  actuators, sensors, etc. Some practical commands:
16:42:13 autoaqua.TextUIService
16:42:13 autoaqua.TextUIService   help -- show this help message (or with parameter, normal pydoc)
16:42:13 autoaqua.TextUIService   lsa, lsp, lss -- list actuators, programs, sensors
16:42:13 autoaqua.TextUIService   get_statusmsg -- print current system status
16:42:13 autoaqua.TextUIService   gui -- show GUI
16:42:13 autoaqua.TextUIService   quit -- quit cleanly
16:42:13 autoaqua.TextUIService
16:42:13 autoaqua.TextUIService   python_str(filename) -- print all objects in a python file
16:42:13 autoaqua.TextUIService   obj.set_value(value) -- set object value for obj
16:42:13 autoaqua.TextUIService   obj.print_traits() -- print all traits attributes for obj
16:42:13 autoaqua.TextUIService
16:42:13 autoaqua.TextUIService
16:42:13 autoaqua.RelayActuator.lamppu1 RelayActuator lamppu1 (by lamput) setting status to True
16:42:42 autoaqua.RelayActuator.uvc RelayActuator uvc (by ajastinohjelma) setting status to True
16:43:39 autoaqua.WebService WebSocket opened for session d8nx9ydo5z13msneyqe7jtw25pb0ctm2
16:44:13 autoaqua.RelayActuator.lamppu3 RelayActuator lamppu3 (by lamput) setting status to True
16:44:25 autoaqua.UserBoolSensor.valot_manuaalimoodi status changed to True
16:44:25 autoaqua.BoolActuator.lamput BoolActuator lamput (by valot_manuaalimoodi) setting status to False
16:44:25 autoaqua.RelayActuator.lamppu1 RelayActuator lamppu1 (by lamput) setting status to False
16:44:54 autoaqua.WebService WebSocket closed for session d8nx9ydo5z13msneyqe7jtw25pb0ctm2
16:44:54 autoaqua.WebService WebSocket opened for session d8nx9ydo5z13msneyqe7jtw25pb0ctm2
```

Command

Run

Switch to multiline

```
    program = Program(
        active_condition = Or('web_switch', 'hardware_switch'),
        on_activate = SetStatus('lamp', True)
    )

# To view UML diagram, we need to set up PlantUMLService. Here we will use
# plantuml.com online service to render the UML graphics.
plantuml_service = PlantUMLService(url='http://www.plantuml.com/plantuml/svg/')
web_service = WebService(
                    read_only=False,
                    default_view='plantuml',
                    http_port=8085,
                    http_auth = ('myusername', 'mypassword'),
                    user_tags = ['user'],
                    )

# Just to give example of slave feature, let's open another server instance
# at port 8086.
slave = WebService(
                   http_port=8086,
                   slave=True,
                   )

my_system = MySystem(services=[plantuml_service, web_service, slave])
```

Tip: Try to run the code in your IPython shell by copying & pasting it with `cpaste` command!

WebService class definition

class `automate.extensions.webui.`**`WebService`**

Web User Interface Service for Automate

`read_only` = None
: Restrict usage to only monitoring statuses (default: `True`). If WebService is not in read_only mode, it is possible to run arbitrary Python commands through eval/exec via web browser. This is, of course, a severe security issue. Secure SSL configuration HIGHLY recommended, if not operating in `read_only` mode.

`default_view` = None
: Default view that is displayed when entering the server. Can be the name of any view in views.py

`show_actuator_details` = None
: Below Actuator row, show active Programs that are controlling Actuator

`http_port` = None
: HTTP port to listen

`http_auth` = None
: Authentication for logging into web server. (user,password) pairs in a tuple.

`websocket_timeout` = None
: Let websocket connection die after `websocket_timeout` time of no ping reply from client.

`user_tags` = None
: Tags that are shown in user defined view

`debug` = None
: Django debugging mode (slower, more info shown when error occurs)

custom_pages = None
User-defined custom pages as a dictionary of form `{name: template_content}`

slave = None
set to True, if you want to launch multiple servers with same system. Authentication and other settings are then taken from master service. Only web server settings (http host/port) are used from slave.

django_settings = None
In this dictionary you can define your custom Django settings which will override the default ones

WSGI Support for Automate

This extension provides Web server for WSGI-aware extensions, such as *Remote Procedure Call Support for Automate*, *Web User Interface for Automate*. It is of no use alone.

Class definition

class `automate.extensions.wsgi.`**TornadoService**
Abstract service that provides HTTP server for WSGI applications.

http_ipaddr = None
Which ip address to listen. Use `0.0.0.0` (default) to listen to all local networking interfaces.

http_port = None
HTTP (or HTTPS if using SSL) port to listen

ssl_certificate = None
Path to ssl certificate file. If set, SSL will be used.

Tip: You may use script scripts/generate_selfsigned_certificate.sh to generate a self-signed openssl certificate.

---

ssl_private_key = None
Path to ssl private key file

num_threads = None
Number of listener threads to spawn

static_dirs = None
Extra static dirs you want to serve. Example:

```
static_dirs = {'/my_static/(.*)': '/path/to/my_static'}
```

get_wsgi_application()
Get WSGI function. Implement this in subclasses.

Remote Procedure Call Support for Automate

Introduction

This extension provides XmlRPC API for external applications. Exported API is by default defined by *automate.extensions.rpc.rpc.ExternalApi*.

Installation

Install extras:

```
pip install automate[rpc]
```

Class definitions

class automate.extensions.rpc.**RpcService**

view_tags = None
Tags that are displayed via get_websensors RPC function

api = None
If you want to define custom api (similar to, or derived from *ExternalApi*, it can be given here.

class automate.extensions.rpc.rpc.**ExternalApi**(*system*, *tag*)

set_status(*name*, *status*)
Set sensor name status to status.

get_status(*name*)
Get status of object with name name.

set_object_status(*statusdict*)
Set statuses from a dictionary of format {name: status}

toggle_object_status(*objname*)
Toggle boolean-valued sensor status between True and False.

get_sensors()
Get sensors as a dictionary of format {name: status}

get_websensors()
Get sensors with defined tag as a dictionary of format {name: status}

get_actuators()
Get actuators as a dictionary of format {name: status}

flush()
Flush the system queue. If you have just set a status and then read a value, it might be necessary to flush queue first, such that related changes have been applied.

is_alive()
Simple RPC command that returns always True.

log()
Return recent log entries as a string.

Arduino Support for Automate

Introduction

This extension provides interface to Arduino devices via pyFirmata library.

Installation

Install extras:

```
pip install automate[arduino]
```

Example application

This example application sets up couple of analog and digital Arduino Sensors and Actuators. It also introduces Servo actuator with *ConstantTimeActuator*, which functions such a way that if value of `a1` changes, the value of `servo` will change smoothly within given time interval.

```
from automate import *


class MySystem(System):
    a1 = ArduinoAnalogSensor(dev=0, pin=0)
    d12 = ArduinoDigitalSensor(dev=0, pin=12)

    d13 = ArduinoDigitalActuator(dev=0, pin=13)  # LED on Arduino board
    servo = ArduinoServoActuator(min_pulse=200,
                                 max_pulse=8000,
                                 dev=0,
                                 pin=3,
                                 default=50,
                                 slave=True)

    interp = ConstantTimeActuator(change_time=2.,
                                  change_frequency=20.,
                                  slave_actuator=servo)

    prog = Program(
        on_update=Run(Log("Value: %s", Value(a1)),
                      SetStatus(d13, d12),
                      SetStatus(interp, Value(180) * Value(a1)))
    )

my_arduino = ArduinoService(
    arduino_devs=["/dev/ttyUSB0"],
    arduino_dev_types=["Arduino"],
    arduino_dev_sampling=[500])

s = MySystem(services=[my_arduino, WebService()])
```

Class definitions

Service

class automate.extensions.arduino.**ArduinoService**

Service that provides interface to Arduino devices via pyFirmata library.

arduino_devs = None

Arduino devices to use, as a list

arduino_dev_types = None
Arduino device board types, as a list of strings. Choices are defined by pyFirmata board class names, i.e. allowed values are "Arduino", "ArduinoMega", "ArduinoDue".

arduino_dev_sampling = None
Arduino device sampling rates, as a list (in milliseconds).

change_digital(*dev*, *pin_nr*, *value*)
Change digital Pin value (boolean). Also PWM supported(float)

Sensors

class automate.extensions.arduino.**AbstractArduinoSensor**(**args*, ***kwargs*)
Abstract base class for Arduino sensors

dev = None
Arduino device number (specify, if more than 1 devices configured in ArduinoService)

pin = None
Arduino pin number

class automate.extensions.arduino.**ArduinoDigitalSensor**(**args*, ***kwargs*)
Boolean-valued sensor object for digital Arduino input pins

class automate.extensions.arduino.**ArduinoAnalogSensor**(**args*, ***kwargs*)
Float-valued sensor object for analog Arduino input pins

Actuators

class automate.extensions.arduino.**AbstractArduinoActuator**(**args*, ***kwargs*)
Abstract base class for Arduino actuators

dev = None
Arduino device number (specify, if more than 1 devices configured in ArduinoService)

pin = None
Arduino pin number

class automate.extensions.arduino.**ArduinoDigitalActuator**(**args*, ***kwargs*)
Boolean-valued actuator object for digital Arduino output pins

class automate.extensions.arduino.**ArduinoPWMActuator**(**args*, ***kwargs*)
Float-valued actuator object for Arduino output pins that can be configured in PWM mode Status is float between 0.0 and 1.0.

class automate.extensions.arduino.**ArduinoServoActuator**(**args*, ***kwargs*)
Float-valued actuator object for Arduino output pins that can be configured in Servo mode Status is servo angle (0-360).

min_pulse = None
Minimum pulse time (in microseconds)

max_pulse = None
Maximum pulse time (in microseconds)

Raspberry Pi GPIO Support for Automate

Introduction

This extension provides interface to Raspberry Pi GPIO via RPIO library. RPIO library.

Installation

Install extras:

```
pip install automate[raspberrypi]
```

Example application

This simple example application sets up simple relation between input pin `button` in port 22 and output pin `light` in port 23. If for a button is attached in `button`, pushing it down will light the led, that is attached to `light`.

```
from automate import *
from automate.extensions.rpio import RpioSensor, RpioActuator


class MySystem(System):
    button = RpioSensor(port=22, button_type='down')
    light = RpioActuator(port=23, change_delay=2)
    myprog = Program(active_condition=Value('mysensor'),
                     on_activate=SetStatus('myactuator', True))

mysystem = MySystem()
```

Class definitions

Service

class automate.extensions.rpio.**RpioService**
Service that provides interface to Raspberry Pi GPIO via RPIO library.

gpio_cleanup = None
Perform GPIO cleanup when exiting (default: False).

rpio = None
Use RPIO instead of RPI.GPIO

Sensors

class automate.extensions.rpio.**RpioSensor**(**args*, ***kwargs*)
Boolean-valued sensor object that reads Raspberry Pi GPIO input pins.

port = None
GPIO port

inverted = None
Set to True to have inversed status value

button_type = None
Button setup: "down": pushdown resistor, "up": pushup resistor, or "none": no resistor set up.

class automate.extensions.rpio.**RpioSensor**(**args*, ***kwargs*)
Boolean-valued sensor object that reads Raspberry Pi GPIO input pins.

port = None
GPIO port

inverted = None
Set to True to have inversed status value

button_type = None
Button setup: "down": pushdown resistor, "up": pushup resistor, or "none": no resistor set up.

Actuators

class automate.extensions.rpio.**RpioActuator**(**args*, ***kwargs*)
Boolean-valued actuator for setting Raspberry Pi GPIO port statuses (on/off).

port = None
GPIO port id

inverted = None
Set to True to have inversed status value

class automate.extensions.rpio.**TemperatureSensor**(**args*, ***kwargs*)
W1 interface (on Raspberry Pi board) that polls polling temperature. (kernel modules w1-gpio and w1-therm required). Not using RPIO, but placed this here, since this is also Raspberry Pi related sensor.

addr = None
Address of W1 temperature sensor (something like `"28-00000558263c"`), see what you have in `/sys/bus/w1/devices/`

max_jump = None
Maximum jump in temperature, between measurements. These temperature sensors tend to give sometimes erroneous results.

max_errors = None
Maximum number of erroneous measurements, until value is really set

class automate.extensions.rpio.**RpioPWMActuator**(**args*, ***kwargs*)
Actuator to control PWM (pulse-width-modulation) ports on Raspberry pi GPIO.

Status range 0...1

This is not recommended to be used because RPIO PWM implementation is not very well behaving. I recommend to use ArduinoPWMActuator with an Arduino loaded with StandardFirmata. It's much more stable and robust solution.

port = None
GPIO port number

dma_channel = None
RPIO PWM DMA channel

frequency = None
PWM frequency (Hz)

Making your own Automate Extensions

Extension Development

Automate extensions allow extending Automate functionalities by writing external libraries that may consist of new Service, Sensor, Actuator, or Callable classes.

To start developing automate extensions, it is recommended to use cookiecutter template. This is how it works:

1. Install `cookiecutter` 1.0.0 or newer:

```
pip install cookiecutter
```

2. Generate a Automate extension project:

```
cookiecutter https://github.com/tuomas2/cookiecutter-automate-ext-template.git
```

Cookiecutter asks few questions and you have great basis for starting your template development. There will be created Python files where you may add your new custom Automate classes.

For your classes to be exported to the Automate, make sure that they are listed in `extension_classes` list in `__init__.py` of the extension module.

All installed Automate Extensions are available from Automate applications and are imported to automate namespace.

Tip: You can install your extension in *editable* mode by running `pip install -e .` in your extension root directory.

Tip: You can look at *Extensions* for examples.

- genindex

Python Module Index

a

Index

Symbols

A

B

C

D

E

M

N

O

P

R

S

T

U

V

W