automat Documentation

Release 0.6.0

Glyph

Sep 15, 2017

Contents:

Automat is a library for concise, idiomatic Python expression of finite-state automata (particularly deterministic finite-state transducers).

CHAPTER 1

Why use state machines?

Sometimes you have to create an object whose behavior varies with its state, but still wishes to present a consistent interface to its callers.

For example, let's say you're writing the software for a coffee machine. It has a lid that can be opened or closed, a chamber for water, a chamber for coffee beans, and a button for "brew".

There are a number of possible states for the coffee machine. It might or might not have water. It might or might not have beans. The lid might be open or closed. The "brew" button should only actually attempt to brew coffee in one of these configurations, and the "open lid" button should only work if the coffee is not, in fact, brewing.

With diligence and attention to detail, you can implement this correctly using a collection of attributes on an object; has_water, has_beans, is_lid_open and so on. However, you have to keep all these attributes consistent. As the coffee maker becomes more complex - perhaps you add an additional chamber for flavorings so you can make hazelnut coffee, for example - you have to keep adding more and more checks and more and more reasoning about which combinations of states are allowed.

Rather than adding tedious 'if' checks to every single method to make sure that each of these flags are exactly what you expect, you can use a state machine to ensure that if your code runs at all, it will be run with all the required values initialized, because they have to be called in the order you declare them.

You can read about state machines and their advantages for Python programmers in considerably more detail in this excellent series of articles from ClusterHQ.

Quick Start

What makes Automat different?

There are dozens of libraries on PyPI implementing state machines. So it behooves me to say why yet another one would be a good idea.

Automat is designed around this principle: while organizing your code around state machines is a good idea, your callers don't, and shouldn't have to, care that you've done so. In Python, the "input" to a stateful system is a method call; the "output" may be a method call, if you need to invoke a side effect, or a return value, if you are just performing

a computation in memory. Most other state-machine libraries require you to explicitly create an input object, provide that object to a generic "input" method, and then receive results, sometimes in terms of that library's interfaces and sometimes in terms of classes you define yourself.

For example, a snippet of the coffee-machine example above might be implemented as follows in naive Python:

```
class CoffeeMachine(object):
    def brew_button(self):
        if self.has_water and self.has_beans and not self.is_lid_open:
            self.heat_the_heating_element()
            # ...
```

With Automat, you'd create a class with a *automat.MethodicalMachine* attribute:

```
from automat import MethodicalMachine

class CoffeeBrewer(object):
    _machine = MethodicalMachine()
```

and then you would break the above logic into two pieces - the *brew_button input*, declared like so:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    @_machine.input()
    def brew_button(self):
        "The user pressed the 'brew' button."
```

It wouldn't do any good to declare a method *body* on this, however, because input methods don't actually execute their bodies when called; doing actual work is the *output*'s job:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.output()
    def _heat_the_heating_element(self):
        "Heat up the heating element, which should cause coffee to happen."
        self._heating_element.turn_on()
```

As well as a couple of *states* - and for simplicity's sake let's say that the only two states are *have_beans* and *dont_have_beans*:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.state()
    def have_beans(self):
        "In this state, you have some beans."

    @_machine.state(initial=True)
        def dont_have_beans(self):
            "In this state, you don't have any beans."
```

dont_have_beans is the *initial* state because *CoffeeBrewer* starts without beans in it.

(And another input to put some beans in:)

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.input()
    def put_in_beans(self):
        "The user put in some beans."
```

Finally, you hook everything together with the *upon()* method of the functions decorated with *_machine.state*:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    # When we don't have beans, upon putting in beans, we will then have beans
    # (and produce no output)
    dont_have_beans.upon(put_in_beans, enter=have_beans, outputs=[])

    # When we have beans, upon pressing the brew button, we will then not have
    # beans any more (as they have been entered into the brewing chamber) and
    # our output will be heating the heating element.
    have_beans.upon(brew_button, enter=dont_have_beans,
                    outputs=[_heat_the_heating_element])
```

To *users* of this coffee machine class though, it still looks like a POPO (Plain Old Python Object):

```
>>> coffee_machine = CoffeeMachine()
>>> coffee_machine.put_in_beans()
>>> coffee_machine.brew_button()
```

All of the *inputs* are provided by calling them like methods, all of the *outputs* are automatically invoked when they are produced according to the outputs specified to `automat.MethodicalState.upon()` and all of the states are simply opaque tokens - although the fact that they're defined as methods like inputs and outputs allows you to put docstrings on them easily to document them.

How do I get the current state of a state machine?

Don't do that.

One major reason for having a state machine is that you want the callers of the state machine to just provide the appropriate input to the machine at the appropriate time, and *not have to check themselves* what state the machine is in. So if you are tempted to write some code like this:

```
if connection_state_machine.state == "CONNECTED":
    connection_state_machine.send_message()
else:
    print("not connected")
```

Instead, just make your calling code do this:

```
connection_state_machine.send_message()
```

and then change your state machine to look like this:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.state()
    def connected(self):
        "connected"
    @_machine.state()
    def not_connected(self):
        "not connected"
    @_machine.input()
    def send_message(self):
        "send a message"
    @_machine.output()
    def _actually_send_message(self):
        self._transport.send(b"message")
    @_machine.output()
    def _report_sending_failure(self):
        print("not connected")
    connected.upon(send_message, enter=connected, [_actually_send_message])
    not_connected.upon(send_message, enter=not_connected, [_report_sending_failure])
```

so that the responsibility for knowing which state the state machine is in remains within the state machine itself.

Input for Inputs and Output for Outputs

Quite often you want to be able to pass parameters to your methods, as well as inspecting their results. For example, when you brew the coffee, you might expect a cup of coffee to result, and you would like to see what kind of coffee it is. And if you were to put delicious hand-roasted small-batch artisanal beans into the machine, you would expect a *better* cup of coffee than if you were to use mass-produced beans. You would do this in plain old Python by adding a parameter, so that's how you do it in Automat as well.

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.input()
    def put_in_beans(self, beans):
        "The user put in some beans."
```

However, one important difference here is that *we can't add any implementation code to the input method*. Inputs are purely a declaration of the interface; the behavior must all come from outputs. Therefore, the change in the state of the coffee machine must be represented as an output. We can add an output method like this:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.output()
    def _save_beans(self, beans):
        "The beans are now in the machine; save them."
        self._beans = beans
```

and then connect it to the *put_in_beans* by changing the transition from *dont_have_beans* to *have_beans* like so:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    dont_have_beans.upon(put_in_beans, enter=have_beans,
                         outputs=[_save_beans])
```

Now, when you call:

```
coffee_machine.put_in_beans("real good beans")
```

the machine will remember the beans for later.

So how do we get the beans back out again? One of our outputs needs to have a return value. It would make sense if our *brew_button* method returned the cup of coffee that it made, so we should add an output. So, in addition to heating the heating element, let's add a return value that describes the coffee. First a new output:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.output()
    def _describe_coffee(self):
        return "A cup of coffee made with {}.".format(self._beans)
```

Note that we don't need to check first whether *self._beans* exists or not, because we can only reach this output method if the state machine says we've gone through a set of states that sets this attribute.

Now, we need to hook up *_describe_coffee* to the process of brewing, so change the brewing transition to:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    have_beans.upon(brew_button, enter=dont_have_beans,
                    outputs=[_heat_the_heating_element,
                             _describe_coffee])
```

Now, we can call it:

```
>>> coffee_machine.brew_button()
[None, 'A cup of coffee made with real good beans.']
```

Except... wait a second, what's that *None* doing there?

Since every input can produce multiple outputs, in automat, the default return value from every input invocation is a *list*. In this case, we have both *_heat_the_heating_element* and *_describe_coffee* outputs, so we're seeing both of their return values. However, this can be customized, with the *collector* argument to `upon()`; the *collector* is a callable which takes an iterable of all the outputs' return values and "collects" a single return value to return to the caller of the state machine.

In this case, we only care about the last output, so we can adjust the call to `upon()` like this:

```
class CoffeeBrewer(object):
    _machine = MethodicalMachine()

    # ...

    have_beans.upon(brew_button, enter=dont_have_beans,
                    outputs=[_heat_the_heating_element,
                             _describe_coffee],
                    collector=lambda iterable: list(iterable)[-1]
    )
```

And now, we'll get just the return value we want:

```
>>> coffee_machine.brew_button()
'A cup of coffee made with real good beans.'
```

If I can't get the state of the state machine, how can I save it to (a database, an API response, a file on disk...)

There are APIs for serializing the state machine.

First, you have to decide on a persistent representation of each state, via the *serialized*= argument to the *MethodicalMachine.state()* decorator.

Let's take this very simple "light switch" state machine, which can be on or off, and flipped to reverse its state:

```
class LightSwitch(object):
    _machine = MethodicalMachine()

    @_machine.state(serialized="on")
    def on_state(self):
        "the switch is on"

    @_machine.state(serialized="off", initial=True)
    def off_state(self):
        "the switch is off"

    @_machine.input()
    def flip(self):
        "flip the switch"

    on_state.upon(flip, enter=off_state, outputs=[])
    off_state.upon(flip, enter=on_state, outputs=[])
```

In this case, we've chosen a serialized representation for each state via the *serialized* argument. The on state is represented by the string *"on"*, and the off state is represented by the string *"off"*.

Now, let's just add an input that lets us tell if the switch is on or not.

```
class LightSwitch(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.input()
    def query_power(self):
        "return True if powered, False otherwise"
```

```
    @_machine.output()
    def _is_powered(self):
        return True
    @_machine.output()
    def _not_powered(self):
        return False
    on_state.upon(query_power, enter=on_state, outputs=[_is_powered],
                  collector=next)
    off_state.upon(query_power, enter=off_state, outputs=[_not_powered],
                   collector=next)
```

To save the state, we have the *MethodicalMachine.serializer()* method. A method decorated with *@serializer()* gets an extra argument injected at the beginning of its argument list: the serialized identifier for the state. In this case, either *"on"* or *"off"*. Since state machine output methods can also affect other state on the object, a serializer method is expected to return *all* relevant state for serialization.

For our simple light switch, such a method might look like this:

```
class LightSwitch(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.serializer()
    def save(self, state):
        return {"is-it-on": state}
```

Serializers can be public methods, and they can return whatever you like. If necessary, you can have different serializers - just multiple methods decorated with *@_machine.serializer()* - for different formats; return one data-structure for JSON, one for XML, one for a database row, and so on.

When it comes time to unserialize, though, you generally want a private method, because an unserializer has to take a not-fully-initialized instance and populate it with state. It is expected to *return* the serialized machine state token that was passed to the serializer, but it can take whatever arguments you like. Of course, in order to return that, it probably has to take it somewhere in its arguments, so it will generally take whatever a paired serializer has returned as an argument.

So our unserializer would look like this:

```
class LightSwitch(object):
    _machine = MethodicalMachine()

    # ...

    @_machine.unserializer()
    def _restore(self, blob):
        return blob["is-it-on"]
```

Generally you will want a classmethod deserialization constructor which you write yourself to call this, so that you know how to create an instance of your own object, like so:

```
class LightSwitch(object):
    _machine = MethodicalMachine()

    # ...

    @classmethod
    def from_blob(cls, blob):
```

```
        self = cls()
        self._restore(blob)
        return self
```

Saving and loading our *LightSwitch* along with its state-machine state can now be accomplished as follows:

```
>>> switch1 = LightSwitch()
>>> switch1.query_power()
False
>>> switch1.flip()
[]
>>> switch1.query_power()
True
>>> blob = switch1.save()
>>> switch2 = LightSwitch.from_blob(blob)
>>> switch2.query_power()
True
```

More comprehensive (tested, working) examples are present in *docs/examples*.

Go forth and machine all the state!

Visualizations

Installation

To create state machine graphs you must install *automat* with the graphing dependencies.

```
pip install automat[visualize]
```

Example

Given the following project structure:

```
mystate/
- __init__.py
- machine.py
```

And the following state machine defined in *machine.py*

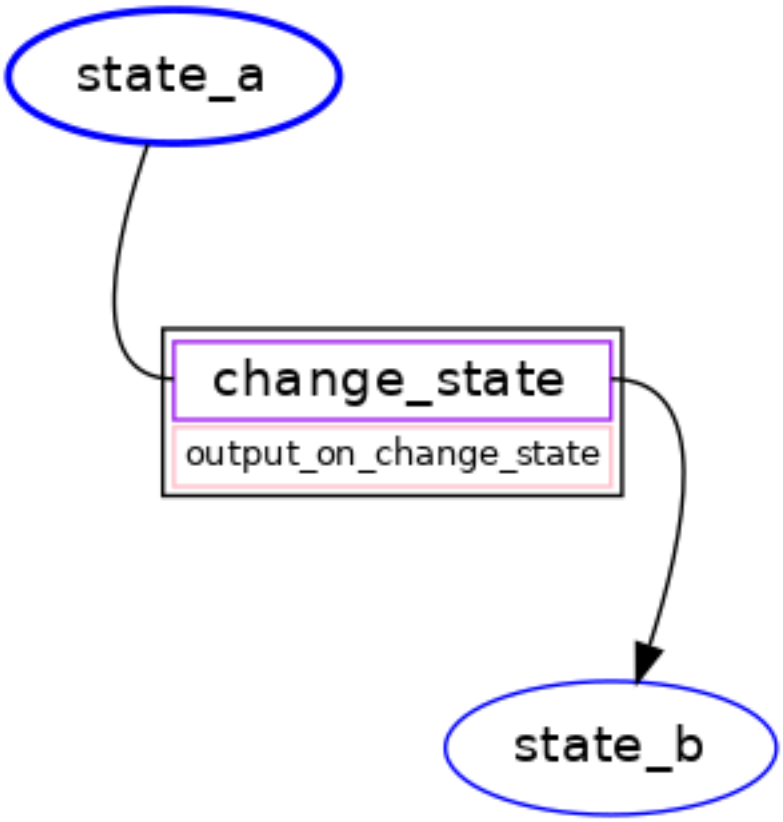
```
from automat import MethodicalMachine

class MyMachine(object):
    _machine = MethodicalMachine()

    @_machine.state(initial=True)
    def state_a(self):
        """
        State A
        """

    @_machine.state()
    def state_b(self):
        """
```

```
        State B
        """

    @_machine.input()
    def change_state(self):
        """
        Change state
        """

    @_machine.input()
    def output_on_change_state(self):
        """
        Change state
        """
        return "Changing state"

    state_a.upon(change_state, enter=state_b, outputs=[output_on_change_state])
```

You can generate a state machine visualization by running:

```
$ automat-visualize mystate
mystate.machine.MyMachine._machine ...discovered
mystate.machine.MyMachine._machine ...wrote image and dot into .automat_visualize
```

The *dot* file and *png* will be saved in the default output directory of *.automat_visualize/mystatemachine.MyMachine._machine.dot.png*



automat-visualize help

```
$ automat-visualize -h
usage: /home/tom/Envs/tmp-72fe664d2dc5cbf/bin/automat-visualize
       [-h] [--quiet] [--dot-directory DOT_DIRECTORY]
       [--image-directory IMAGE_DIRECTORY]
       [--image-type {gv,vml,dot_json,imap_np,pov,tiff,pic,canon,jpg,ismap,sgi,webp,
↪gd,json0,ps2,cmapx_np,plain-ext,wbmp,xdot_json,ps,cgimage,ico,gtk,pct,gif,json,fig,
↪xlib,xdot1.2,tif,tk,xdot1.4,svgz,gd2,jpe,psd,xdot,bmp,jpeg,x11,cmapx,jp2,imap,png,
↪tga,pict,plain,eps,vmlz,cmap,exr,svg,pdf,vrml,dot}]
       [--view]
       fqpn


Visualize automat.MethodicalMachines as graphviz graphs.

positional arguments:
  fqpn                  A Fully Qualified Path name representing where to find
                        machines.

optional arguments:
  -h, --help            show this help message and exit
  --quiet, -q           suppress output
  --dot-directory DOT_DIRECTORY, -d DOT_DIRECTORY
                        Where to write out .dot files.
  --image-directory IMAGE_DIRECTORY, -i IMAGE_DIRECTORY
                        Where to write out image files.
  --image-type {gv,vml,dot_json,imap_np,pov,tiff,pic,canon,jpg,ismap,sgi,webp,gd,
↪json0,ps2,cmapx_np,plain-ext,wbmp,xdot_json,ps,cgimage,ico,gtk,pct,gif,json,fig,
↪xlib,xdot1.2,tif,tk,xdot1.4,svgz,gd2,jpe,psd,xdot,bmp,jpeg,x11,cmapx,jp2,imap,png,
↪tga,pict,plain,eps,vmlz,cmap,exr,svg,pdf,vrml,dot}, -t {gv,vml,dot_json,imap_np,pov,
↪tiff,pic,canon,jpg,ismap,sgi,webp,gd,json0,ps2,cmapx_np,plain-ext,wbmp,xdot_json,ps,
↪cgimage,ico,gtk,pct,gif,json,fig,xlib,xdot1.2,tif,tk,xdot1.4,svgz,gd2,jpe,psd,xdot,
↪bmp,jpeg,x11,cmapx,jp2,imap,png,tga,pict,plain,eps,vmlz,cmap,exr,svg,pdf,vrml,dot}
                        The image format.
  --view, -v            View rendered graphs with default image viewer

You must have the graphviz tool suite installed. Please visit
http://www.graphviz.org for more information.
```

API Docs

class automat.**MethodicalMachine**

A MethodicalMachine is an interface to an *Automaton* that uses methods on a class.

input(***kw*)

Declare an input.

This is a decorator for methods.

output(***kw*)

Declare an output.

This is a decorator for methods.

This method will be called when the state machine transitions to this state as specified in the decorated *output* method.

state(***kw*)
Declare a state, possibly an initial state or a terminal state.

This is a decorator for methods, but it will modify the method so as not to be callable any more.

Parameters

- **initial** (*bool*) – is this state the initial state? Only one state on this *automat.MethodicalMachine* may be an initial state; more than one is an error.
- **terminal** (*bool*) – Is this state a terminal state? i.e. a state that the machine can end up in? (This is purely informational at this point.)
- **serialized** (*Hashable*) – a serializable value to be used to represent this state to external systems. This value should be hashable; `unicode()` is a good type to use.

`MethodicalState.`**upon**(*input*, *enter*, *outputs*, *collector=<type 'list'>*)
Declare a state transition within the *automat.MethodicalMachine* associated with this `automat.MethodicalState`: upon the receipt of the *input*, enter the *state*, emitting each output in *outputs*.

Parameters

- **input** (*MethodicalInput*) – The input triggering a state transition.
- **enter** (*MethodicalState*) – The resulting state.
- **outputs** (*Iterable[MethodicalOutput]*) – The outputs to be triggered as a result of the declared state transition.
- **collector** (*Callable*) – The function to be used when collecting output return values.

Raises

- **TypeError** – if any of the *outputs* signatures do not match the *inputs* signature.
- **ValueError** – if the state transition from *self* via *input* has already been defined.

Debugging

Tracing API

Warning: The Tracing API is currently private and unstable. Use it for local debugging, but if you think you need to commit code that references it, you should either pin your dependency on the current version of Automat, or at least be prepared for your application to break when this API is changed or removed.

The tracing API lets you assign a callback function that will be invoked each time an input event causes the state machine to move from one state to another. This can help you figure out problems caused by events occurring in the wrong order, or not happening at all. Your callback function can print a message to stdout, write something to a logfile, or deliver the information in any application-specific way you like. The only restriction is that the function must not touch the state machine at all.

To prepare the state machine for tracing, you must assign a name to the "_setTrace" method in your class. In this example, we use *setTheTracingFunction*, but the name can be anything you like:

```
class Sample(object):
    mm = MethodicalMachine()

    @mm.state(initial=True)
```

```
    def begin(self):
        "initial state"

    @mm.state()
    def end(self):
        "end state"

    @mm.input()
    def go(self):
        "event that moves us from begin to end"

    @mm.output()
    def doThing1(self):
        "first thing to do"

    @mm.output()
    def doThing2(self):
        "second thing to do"

    setTheTracingFunction = mm._setTrace

    begin.upon(go, enter=end, outputs=[doThing1, doThing2])
```

Later, after you instantiate the *Sample* object, you can set the tracing callback for that particular instance by calling the *setTheTracingFunction()* method on it:

```
s = Sample()
def tracer(oldState, input, newState):
    pass
s.setTheTracingFunction(tracer)
```

Note that you cannot shortcut the name-assignment step: *s.mm._setTrace(tracer)* will not work, because Automat goes to great lengths to hide that *mm* object from external access. And you cannot set the tracing function at class-definition time (e.g. a class-level *mm._setTrace(tracer)*) because the state machine has merely been *defined* at that point, not instantiated (you might eventually have multiple instances of the Sample class, each with their own independent state machine), and each one can be traced separately.

Since this is a private API, consider using a tolerant *getattr* when retrieving the *_getTrace* method. This way, if you do commit code which references it, but you only *call* that code during debugging, then at least your application or tests won't crash when the API is removed entirely:

```
mm = MethodicalMachine()
setTheTracingFunction = getattr(mm, "_setTrace", lambda self, f: None)
```

The Tracer Callback Function

When the input event is received, before any transitions are made, the tracer function is called with three positional arguments:

- *oldState*: a string with the name of the current state
- *input*: a string with the name of the input event
- *newState*: a string with the name of the new state

If your tracer function returns None, then you will only be notified about the input events. But, if your tracer function returns a callable, then just before each output function is executed (if any), that callable will be executed with a single

output argument (as a string).

So if you only care about the transitions, your tracing function can just do:

```
>>> s = Sample()
>>> def tracer(oldState, input, newState):
...     print("%s.%s -> %s" % (oldState, input, newState))
>>> s.setTheTracingFunction(tracer)
>>> s.go()
begin.go -> end
```

But if you want to know when each output is invoked (perhaps to compare against other log messages emitted from inside those output functions), you can do:

```
>>> s = Sample()
>>> def tracer(oldState, input, newState):
>>>     def traceOutputs(output):
...         print("%s.%s -> %s: %s()" % (oldState, input, newState, output))
...     print("%s.%s -> %s" % (oldState, input, newState))
...     return traceOutputs
>>> s.setTheTracingFunction(tracer)
>>> s.go()
begin.go -> end
begin.go -> end: doThing1()
begin.go -> end: doThing2()
```

Tracing Multiple State Machines

If you have multiple state machines in your application, you will probably want to pass a different tracing function to each, so your logs can distinguish between the transitions of MachineFoo vs those of MachineBar. This is particularly important if your application involves network communication, where an instance of MachineFoo (e.g. in a client) is in communication with a sibling instance of MachineFoo (in a server). When exercising both sides of this connection in a single process, perhaps in an automated test, you will need to clearly mark the first as "foo1" and the second as "foo2" to avoid confusion.

```
s1 = Sample()
s2 = Sample()
def tracer1(oldState, input, newState):
 print("S1: %s.%s -> %s" % (oldState, input, newState))
s1.setTheTracingFunction(tracer1)
def tracer2(oldState, input, newState):
 print("S2: %s.%s -> %s" % (oldState, input, newState))
s2.setTheTracingFunction(tracer2)
```

Index