definite

Release 1.0.1-dev

Daniel Lindsley

Jul 07, 2022

GUIDES:

Simple finite state machines.

Perfect for representing workflows.

CHAPTER ONE

TUTORIAL

definite lets you create small finite state machines. These are great for tracking state, as well as triggering transition behavior when the states change.

1.1 The Most Basic Example

To start with, you'll need to import the `FSM` class from `definite`, then subclass it.

We'll define the tiniest fully-functional subclass we can, allowing just a `start` & `end` state. Beyond the subclassing `FSM`, the only requirements are defining the `allowed_transitions` and `default_state` attributes on the subclass.

```
from definite import FSM

class Tiny(FSM):
    allowed_transitions = {
        "start": ["end"],
        "end": None,
    }
    default_state = "start"
```

Note: You may be wondering why `default_state` is also required. A natural assumption would be that the first mentioned state in `allowed_transitions` would be the default.

Only relatively recently in Python's history did dictionary ordering become guaranteed (documented feature in 3.7). And even now, not all third-party tooling respects this.

This could be solved in a variety of clever ways (special syntax, changing from a `dict` to a two-`tuple` structure, using `collections.OrderedDict`, etc.).

But in the end, manually specifying it this way is clear, easy-to-read, & unsurprising/doesn't require special knowledge. Hence, having to specify it.

Now you can instantiate the new `Tiny` class. Each instance starts in the provided default state of `start`.

```
>>> tiny = Tiny()
>>> tiny.current_state()
"start"
```

There are also a variety of methods for inspecting what the `FSM` can do, such as checking what states are available, if a given state is a recognized/valid state, and if a given state can be transitioned to.

```
>>> tiny.all_states()
["end", "start"]
>>> tiny.is_valid("start")
True
>>> tiny.is_valid("nopenopenope")
False
>>> tiny.is_allowed("end")
True
```

From there, we can trigger a state change by tell it to transition to the end state.

```
>>> tiny.transition_to("end")
>>> tiny.current_state()
"end"
```

Just two states isn't very useful, so let's move on to a more complex example.

1.2 More States/Transitions

Enforcing workflows are a great use of finite state machines, so let's model a possible workflow for a small publisher. We'll assume there are writers, editors, and an editor-in-chief.

- Writers can only create drafts, then submit them for review.
- Editors perform the review, then decide whether to send it back for changes (back to draft), to mark it as reviewed for the editor-in-chief to look at, or to reject the story.
- Editor-in-chief looks at reviewed stories, then either publishes them or can reject them.

We can model all the states a news story could be in this:

```
from definite import FSM

class Workflow(FSM):
    allowed_transitions = {
        "draft": ["awaiting_review", "rejected"],
        "awaiting_review": ["draft", "reviewed", "rejected"],
        "reviewed": ["published", "rejected"],
        "published": None,
        "rejected": ["draft"],
    }
    default_state = "draft"
```

Once this is defined, we can immediately start using it to guide what states our business logic.

```
>>> workflow = Workflow()
>>> workflow.current_state() # "draft"

>>> workflow.transition_to("awaiting_review")
>>> workflow.transition_to("reviewed")

>>> workflow.is_allowed("published") # True
```

(continues on next page)

(continued from previous page)

```
# Invalid/disallowed transitions will throw an exception.
>>> workflow.current_state() # "reviewed"
# ...which can only go to "published" or "rejected", but...
>>> workflow.transition_to("awaiting_review")
# Traceback (most recent call last):
# ...
# workflow.TransitionNotAllowed: "reviewed" cannot transition to "awaiting_review"
```

1.3 Adding Transition Behavior

You can build your own logic around the FSMs anywhere, but `definite` also supports adding your own transition logic directly to the state machine, keeping the state & behavior together.

For instance, let's say our previous example should send emails when a story is either waiting on review, or when it's available for the editor-in-chief to publish.

We can expand on transition behavior by adding "handlers" to specific states. In `definite`, any method prefixed with `handle_` followed by the desired state name will **automatically** be called when changing to that state.

So, to send emails when `awaiting_review` & `reviewed` are met, we'd implement the following handlers.

```
from email.message import EmailMessage
import smtplib

from definite import FSM


FROM_EMAIL = "no-reply@example.com"
EDITORS_EMAIL = "ed-staff@example.com"
CHIEF_EMAIL = "chief@example.com"


# To keep with the standard library theme, here's a very basic email-sending
# function. Probably not very production-friendly.
def send_mail(from_addr, to_addr, subject, message):
    msg = EmailMessage()
    msg["From"] = from_addr
    msg["To"] = to_addr
    msg["Subject"] = subject
    msg.set_content(message)

    smtp = smtplib.SMTP('localhost')
    smtp.send_message(msg)
    smtp.quit()


# Here's our already-written FSM...
class Workflow(FSM):
    allowed_transitions = {
        "draft": ["awaiting_review", "rejected"],
        "awaiting_review": ["draft", "reviewed", "rejected"],
```

(continues on next page)

(continued from previous page)

```
        "reviewed": ["published", "rejected"],
        "published": None,
        "rejected": ["draft"],
    }
    default_state = "draft"

    # ...but here we add our new handlers!
    def handle_awaiting_review(self, state_name):
        msg = "There's a story awaiting review. Go log in & check it out!"
        send_mail(
            FROM_EMAIL,
            EDITORS_EMAIL,
            "Story ready for review!",
            msg
        )

    def handle_reviewed(self, state_name):
        msg = "There's a story ready for publishing. Please have a look!"
        send_mail(
            FROM_EMAIL,
            CHIEF_EMAIL,
            "Story ready for publishing!",
            msg
        )
```

These handlers (`handle_awaiting_review` & `handle_reviewed`) will automatically be called upon transition. For example:

```
>>> workflow = Workflow()
# Some work happens, then...

>>> workflow.transition_to("awaiting_review")
# During this transition, the ``handle_awaiting_review`` method gets called
# and the email is sent to the editors!

>>> workflow.transition_to("reviewed")
# And similarly with the editor-in-chief!
```

1.4 Using/Affecting External Objects

We've got better encapsulization, but there are a couple shortcomings of the last example.

1. The emails don't include any information about *which* story got changed.
2. **Because they're stored in the database, we're presumably having to manually**
   manage the status of each story.

To improve on this, we'll introduce two more concepts: the ability for a FSM to be specific to an external object, and the special `handle_any` transition handler.

Note: For brevity, we're going to omit that same email code in all the future examples. Assume it's still defined, or that you've put it in its own module & are importing it.

First, external objects. By passing **ANY** Python object in during initialization, you can enable the FSM to use it during transition handlers. We'll make a couple small tweaks to our existing handlers.

```
class Workflow(FSM):
    allowed_transitions = {
        "draft": ["awaiting_review", "rejected"],
        "awaiting_review": ["draft", "reviewed", "rejected"],
        "reviewed": ["published", "rejected"],
        "published": None,
        "rejected": ["draft"],
    }
    default_state = "draft"

    def handle_awaiting_review(self, state_name):
        # Note that we're now using a format string & `self.obj` here!
        msg = f"'{self.obj.title}' is awaiting review. Go log in & check it out!"
        send_mail(
            FROM_EMAIL,
            EDITORS_EMAIL,
            "Story ready for review!",
            msg
        )

    def handle_reviewed(self, state_name):
        # Note that we're now using a format string & `self.obj` here!
        msg = f"'{self.obj.title}' is ready for publishing. Please have a look!"
        send_mail(
            FROM_EMAIL,
            CHIEF_EMAIL,
            "Story ready for publishing!",
            msg
        )
```

Then, when we go to use the workflow, we pass the news story to the constructor. The FSM will save a reference to it & exposes it as `self.obj` to the handlers.

Note: For convenience, we'll use a Django model here. But there's nothing stopping you from using whatever else, like SQLAlchemy's ORM, a Redis key/value, flat files, even built-in Python objects like `dict`!

```
>>> from news.models import NewsStory

>>> story = NewsStory.objects.create(
...     title="Hello, world!",
...     content="This is our very first story!",
...     author=some_writer,
...     state="draft",
... )

# We pass it in here via the `obj=...` kwarg!
>>> workflow = Workflow(obj=story)
```

(continues on next page)

(continued from previous page)

```
# Now when we make the transition to the new state, the editors will get
# a customized email telling them the title of the story that's ready for
# review!
>>> workflow.transition_to("awaiting_review")
```

Another improvement we can make is to persist the `Workflow` state in the database. So if a different server loads the story, the correct state will be preserved there. We'll implement this using the `handle_any` method.

The special `handle_any` method fires on **ANY/ALL** state changes, making it easy to add behavior that should happen with any change of state.

```
class Workflow(FSM):
    allowed_transitions = {
        "draft": ["awaiting_review", "rejected"],
        "awaiting_review": ["draft", "reviewed", "rejected"],
        "reviewed": ["published", "rejected"],
        "published": None,
        "rejected": ["draft"],
    }
    default_state = "draft"

    # Here's the new code!
    def handle_any(self, state_name):
        # The `state` field on the model isn't special, just a plain old
        # `CharField`. But we can push all the FSM's changes right onto it.
        self.obj.state = self.current_state()
        self.obj.save()

    def handle_awaiting_review(self, state_name):
        # Same as before...

    def handle_reviewed(self, state_name):
        # Same as before...
```

Now when we work with the story, we're also persisting the state to the DB.

```
# Same as before.
>>> from news.models import NewsStory
>>> story = NewsStory.objects.create(
...     title="Hello, world!",
...     content="This is our very first story!",
...     author=some_writer,
...     state="draft",
... )
>>> workflow = Workflow(obj=story)

# First, show that nothing has changed yet & no handlers have fired.
>>> story.state
"draft"
>>> workflow.current_state()
"draft"
```

(continues on next page)

(continued from previous page)

```
# But now, when we trigger the transition, both the `handle_any` & the
# `handle_awaiting_review` will fire!
>>> workflow.transition_to("awaiting_review")
# Email sent!

# Proof that `handle_any` fired!
>>> story.state
"awaiting_review"
```

This is great for generic things like adding logging, persisting to long-term storage, or performing integrity checks.

The final change to make is that we can pass the story's current state to the `Workflow` when creating it, making it so that no matter what server loads the story, the FSM is always in the matching state.

```
>>> from news.models import NewsStory
# We'll assume there's already some stories in the database.
>>> story = NewsStory.objects.get(title="Hello, world!")

# Here, we pass in the `initial_state` from the model, to synchronize the
# FSM to the correct state.
>>> workflow = Workflow(obj=story, initial_state=obj.state)

# Note that there's nothing special/magical about the `state` field name
# on the model. Hence the explicit use of `initial_state=...`.
```

1.5 Conclusion

We've learned how to define simple finite state machines, ones with complex state interactions, how to use the everyday parts of the API, and how to build in behaviors!

However, there's more that can be done with `definite`:

- You can store your states/transitions in external JSON files
- You can implement logic that only happens on certain transitions
- You can auto-create constants for your states

You can find examples of these within the *Advanced Usage* guide.

Alternatively, you can dive into the API references, such as the *definite.fsm* reference.

Enjoy!

CHAPTER

TWO

INSTALLATION

You must be running Python 3.6 or greater.

definite only requires the Python built-in standard library & has no other dependencies.

2.1 Standard Installation

For everyday usage, simply run:

```
$ pip install definite
```

It's recommended that you use a *virtualenv* (or *poetry*, or *pipx*, or whatever) to isolate your install from the system Python.

2.2 Development Installation

If you'd like to work on *definite*'s code, run tests or generate the docs locally, the setup is a touch more complex. Here's the recommended approach.

```
$ git clone git@github.com:toastdriven/definite.git
$ cd definite
$ poetry install
$ poetry shell

# Now you can run tests.
$ pytest .

# Or build the docs.
$ cd docs
$ make html && open _build/html/index.html
```

CHAPTER

THREE

ADVANCED USAGE

Here are some advanced ways you can use `definite`.

3.1 Use External JSON Files

`definite` has built-in support for loading states/transitions directly from JSON files. This allows you to share the states with other code, or allow non-technical users to edit them.

The JSON looks almost identical to the required attributes for a `FSM` subclass.

```
{
    allowed_transitions={
        "created": ["waiting"],
        "waiting": ["in_progress", "done"],
        "in_progress": ["done"],
        "done": null
    },
    default_state="created"
}
```

However, it can be loaded & the classes created at runtime.

```
>>> import json
>>> from definite import FSM

# First, we load the JSON up.
>>> state_data = json.loads("/path/to/above/file.json")

# Then we can dynamically create the class based on the JSON.
>>> JobFlow = FSM.from_json("JobFlow", state_data)

# And then instantiate it & use it.
>>> job_1 = JobFlow()
>>> job_1.current_state()
"created"
>>> job_1.all_states()
["created", "done", "in_progress", "waiting"]
```

3.2 Certain Transition Logic

When asked to perform a transition, the `FSM` class does things in a certain order:

1. Check for validity/allowed-ness new state
2. If present, call the `handle_any` handler with the new state
3. If present, call the `handle_<state_name>` handler with the new state
4. Finally update the current state to the new state

Because of this order, you can take special actions *only* for transitions between two certain states.

For example, the `Workflow` example from the *Tutorial* can reach the `rejected` state from a variety of other states (`draft`, `awaiting_review`, `reviewed`).

If you're feeling malicious, you could send the writer a scathing email only when the editor-in-chief rejects their story (the transition from `reviewed` to `rejected`).

```python
from .email import send_mail, FROM_EMAIL

# This is the same as the tutorial code.
class Workflow(FSM):
    allowed_transitions = {
        "draft": ["awaiting_review", "rejected"],
        "awaiting_review": ["draft", "reviewed", "rejected"],
        "reviewed": ["published", "rejected"],
        "published": None,
        "rejected": ["draft"],
    }
    default_state = "draft"

    # But here, we look for the `rejected` state.
    def handle_rejected(self, state_name):
        # The `state_name` here is "rejected".
        # But `self.current_state()` will tell you what the "old" state was!
        prev_state = self.current_state()

        # So if it was previously reviewed by the staff editors, it went
        # to the chief for publishing, but got rejected!
        if prev_state == "reviewed":
            # TIME TO BURN.
            msg = (
                f"The editors let '{self.obj.title}' through, but the Chief"
                "tossed it in the trash! Write better content!"
            )
            send_mail(
                FROM_EMAIL,
                self.obj.author.email,
                "The Chief rejected you!",
                msg
            )
```

Obviously, this is mean-spirited & would promote an unhealthy work environment. Don't do this per-se, but the utility to control behavior down to certain transitions has a lot of potential.

3.3 Auto-Create State Constants

`definite` automatically does a fair amount of checking of state names for validity. However, some programmers may prefer having constants for use instead of the simple strings shown throughout these docs.

Because `FSM` is designed to be subclassed, you could override/extend the built-in behavior to automatically create constants for use.

```
from definite import FSM


class AutoConstantsFSM(FSM):
    # We'll latch onto the `setup` method, which is called when the class
    # is instantiated.
    def setup(self):
        # Make sure you call `super()` first.
        super().setup()

        # Then we can automatically create the constants on the class.
        for state_name in self.allowed_transitions.keys():
            setattr(self, state_name.upper(), state_name)
```

Then you simply inherit from your new subclass instead of `FSM`.

```
class JobFlow(AutoConstantsFSM):
    allowed_transitions = {
        "created": ["waiting"],
        "waiting": ["in_progress", "done"],
        "in_progress": ["done"],
        "done": None,
    }
    default_state = "created"
```

Now all-caps versions of your states will be present on your instances.

```
>>> job_1 = JobFlow()
>>> job_1.transition_to(job_1.WAITING)
```

CHAPTER

FOUR

DEFINITE

DEFINITE.CONSTANTS

`definite.constants.`**`UNKNOWN_STATE: class`**

A sentinel value, used by the `FSM` class to detect when no `default_state` has been provided by the subclass.

It has no meaningful attributes/method, is never instantiated, & is strictly used for identity purposes.

`definite.constants.`**`ANY: str = ``"any"```**

A constant to represent any/all state names.

CHAPTER

SIX

DEFINITE.EXCEPTIONS

CHAPTER SEVEN

DEFINITE.FSM

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

INDEX