state$_chain.pyDocumentation$

Release 2.0rc1

Chad Whitacre et al.

May 30, 2021

Contents

The `state_chain` module helps define and run algorithms composed of multiple functions that operate on a shared state object.

CHAPTER 1

Installation

`state_chain` is available on GitHub and on PyPI:

```
$ pip install state_chain
```

The version of `state_chain` documented here has been tested against Python 3.6, 3.7, 3.8 and 3.9 on Ubuntu.

`state_chain` is MIT-licensed.

CHAPTER 2

Tutorial

This module provides an abstraction for implementing arbitrary algorithms as a list of functions that operate on a shared state object. Algorithms defined this way are easy to arbitrarily modify at run time, and they provide cascading exception handling.

To get started, create a `StateChain` object:

```
>>> from state_chain import StateChain
>>> chain = StateChain()
```

And add some functions to it:

```
>>> @chain.add
... def set_x(state):
...     state.x = 1
...
>>> @chain.add
... def set_y(state):
...     state.y = 2
...
>>> @chain.add
... def set_sum(state):
...     state.sum = state.x + state.y
...
```

As you can see, each function will receive the `state` object as its only argument. Moreover you may have noticed that the functions don't return anything. Returning a value isn't prohibited, but that value will be ignored by the `run` method.

Speaking of the run method, let's give it a go:

```
>>> chain.run().sum
3
```

Okay, we have the expected sum!

2.1 Modifying a State Chain

Let’s define three more functions to add to the state chain:

```
>>> def uh_oh(state):
...     if state.x == 0:
...         raise Exception('oops, state.x is zero')
...
>>> def deal_with_it(state):
...     print("I am dealing with it!")
...     state.exception = None
...
>>> def print_state(state):
...     print(state)
...
```

and make a copy of the chain that we’ll use later:

```
>>> chain_copy = chain.copy()
```

Now let’s interpolate the new functions into our state chain. Let’s put the `uh_oh` at the beginning:

```
>>> chain.add(uh_oh, position=0)
<function uh_oh ...>
>>> chain.functions
(<function uh_oh ...>, <function set_x ...>, <function set_y ...>,
 <function set_sum ...>)
```

Then let’s remove `set_y` and replace `set_sum` with `print_state`:

```
>>> chain.remove('set_y')
>>> chain.add(print_state, position=chain.before('set_sum'), exception='accepted')
<function print_state ...>
>>> chain.remove('set_sum')
>>> chain.functions
(<function uh_oh ...>, <function set_x ...>, <function print_state ...>)
```

Finally, let’s add our exception handler after `print_state`:

```
>>> chain.add(deal_with_it, position=chain.after('print_state'), exception='required')
<function deal_with_it ...>
>>> chain.functions
(<function uh_oh ...>, <function set_x ...>, <function print_state ...>,
 <function deal_with_it ...>)
```

Note: when making extensive changes to a state chain, you can use the `modify` method to rebuild the entire chain in a safe way. We could have achieved the same result as above like so:

```
>>> chain = (
...     chain_copy.modify()
...     .add(uh_oh)
...     .keep('set_x')
...     .drop('set_y')
...     .replace('set_sum', print_state, exception='accepted')
...     .add(deal_with_it, exception='required')
...     .end()
... )
```

(continues on next page)

(continued from previous page)

```
>>> chain.functions
(<function uh_oh ...>, <function set_x ...>, <function print_state ...>,
 <function deal_with_it ...>)
```

This allows you to see exactly what your chain does and how it differs from the original chain.

Either way, what happens when we run it?

```
>>> from state_chain import Object
>>> state = chain.run(Object(x=0))
Object(x=0, exception=Exception('oops, state.x is zero'))
I am dealing with it!
```

2.2 Exception Handling

Whenever a function raises an exception, like `uh_oh` did in the example above, `run` captures the exception and assigns it to `state.exception`. As long as this state attribute is not `None`, any normal function is skipped, and only exception handling functions get called. It's like a fast-forward. So in our example `print_state` and `deal_with_it` got called, but `set_x` didn't.

If we run without triggering the exception in `uh_oh`, then we have a different result:

```
>>> _ = chain.run(Object(x=5))
Object(x=1, exception=None)
```

If we remove the `deal_with_it` function, then the exception isn't handled, so it's reraised at the end of the chain:

```
>>> chain.remove('deal_with_it')
>>> chain.run(Object(x=0))
Traceback (most recent call last):
    ...
Exception: oops, state.x is zero
```

Whether a function is skipped or called is determined by its "exception preference" (the value of the *exception* argument of the `StateChain.add` method). If it's 'unwanted', then the function will be skipped when an exception has been raised. If it's 'accepted', then the function will always be called. If it's 'required', then the function will only be called when an exception has been raised.

The default value is 'unwanted', but you can change it when creating the chain:

```
>>> chain = StateChain(exception_preference='accepted')
```

In that case, the chain's functions are always called, unless they were explicitly added with a different exception preference:

```
>>> chain.add(uh_oh)
<function uh_oh at ...>
>>> @chain.add(exception='unwanted')
... def skipped(state):
...     raise Exception("this function should not be called")
...
>>> @chain.add
... def always_called(state):
...     state.x = -1
```

(continues on next page)

(continued from previous page)

```
...     state.exception = None
...
>>> chain.run().x
-1
```

2.3 Argument Injection

So far we've only used chain functions that expect the state object as their only argument, but the *run* method can also automatically pass the value of any attribute of the state object to a function as an argument.

For example:

```
>>> def print_sum(x, y):
...     print(f"x + y = {x + y}")
...
>>> chain = StateChain(functions=[set_x, set_y, print_sum])
>>> _ = chain.run()
x + y = 3
```

If a chain function has an *exception* argument, then its default exception preference is 'accepted' if the argument is optional (e.g. *exception=None*), and 'required' otherwise.

```
>>> def exception_handler(foo, exception):
...     "This function's default exception preference is 'required'"
...
>>> def exception_tolerant_function(foo, exception=None):
...     "This function's default exception preference is 'accepted'"
...
```

Argument injection is implemented in the `call` function and relies on the standard library function `inspect.signature` introduced in Python 3.3.

2.4 Static Typing

Since version 2.0, the *state_chain* module includes complete type annotations, and its API has been redesigned to facilitate type checking the applications that use it.

Here is an example of a statically typed chain:

```
>>> from dataclasses import dataclass
>>> from typing import Optional
>>> @dataclass
... class State:
...     request: str
...     response: Optional[str] = None
...     exception: Optional[Exception] = None
...
>>> def respond(state: State):
...     if state.request.startswith("I want chocolate"):
...         state.response = "Me too."
...     else:
...         state.response = "Sorry, I don't understand your request."
```

(continues on next page)

(continued from previous page)

```
...
>>> def match_punctuation(state: State):
...     if state.response and state.request.endswith('!'):
...         state.response = state.response.replace('.', '!')
...
>>> chain = StateChain(State, [respond, match_punctuation])
>>> chain.run(State("I want chocolate!")).response
'Me too!'
```

2.5 Aliases

If you're building a state chain that is meant to be used and modified by other programs, it can be useful to give friendly and stable aliases to some of your chain's functions.

```
>>> temp_chain = StateChain(State)
>>> temp_chain.add(set_y, alias='y_is_available')
<function set_y ...>
```

This makes it easy to insert a function before or after a specific function is run, even if that function is later renamed.

```
>>> temp_chain.add(print_state, position=temp_chain.after('y_is_available'))
<function print_state ...>
```

CHAPTER 3

API Reference

CHAPTER 4

Migrating from 1.x

Version 2.0 of the `state_chain` module includes several breaking changes.

1) The ways to create and initialize a state chain have changed. The `StateChain.from_dotted_name` constructor no longer exists, and the default `StateChain` constructor no longer takes a variable number of arguments.

```
-chain = StateChain.from_dotted_name(...)
+chain = StateChain()
+
+@chain.add
+def foo(...):
+    ...
+
+@chain.add
+def bar(...):
+    ...
```

```
-chain = StateChain(foo, bar)
+chain = StateChain(functions=[foo, bar])
```

2) The `StateChain.run` method no longer accepts a variable number of arguments.

```
-chain.run(x=0, _raise_immediately=True, _return_after='foo')
+from state_chain import Object
+chain.run(Object(x=0), raise_immediately=True, return_after='foo')
```

3) Modifying the state by returning dictionaries is no longer supported. You have to explicitly modify the `state` object instead.

```
-def foo():
-    return {'x': 0}
+def foo(state):
+    state.x = 0
```