statechain.pyDocumentation

Release 1.3.0-dev

Chad Whitacre et al.

Jun 01, 2019

Contents

Model algorithms as a list of functions operating on a shared state dict.

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 2.7, 3.4, and 3.5 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 dictionary. Algorithms defined this way are easy to arbitrarily modify at run time, and they provide cascading exception handling.

To get started, define some functions:

```
>>> def foo():
...     return {'baz': 1}
...
>>> def bar():
...     return {'buz': 2}
...
>>> def bloo(baz, buz):
...     return {'sum': baz + buz}
...
```

Each function returns a `dict`, which is used to update the state of the current run of the algorithm. Names from the state dictionary are made available to downstream functions via `dependency_injection`. Now make an *`StateChain`* object:

```
>>> from state_chain import StateChain
>>> blah = StateChain(foo, bar, bloo)
```

The functions you passed to the constructor are loaded into a list:

```
>>> blah.functions          #doctest: +ELLIPSIS
[<function foo ...>, <function bar ...>, <function bloo ...>]
```

Now you can use *`run`* to run the functions. You'll get back a dictionary representing the algorithm's final state:

```
>>> state = blah.run()
>>> state['sum']
3
```

Okay!

2.1 Modifying a State Chain

Let's add two functions to the state chain. First let's define the functions:

```
>>> def uh_oh(baz):
...     if baz == 2:
...         raise heck
...
>>> def deal_with_it(exception):
...     print("I am dealing with it!")
...     return {'exception': None}
...
```

Now let's interpolate them into our state chain. Let's put the `uh_oh` function between `bar` and `bloo`:

```
>>> blah.insert_before('bloo', uh_oh)
>>> blah.functions      #doctest: +ELLIPSIS
[<function foo ...>, <function bar ...>, <function uh_oh ...>, <function bloo ...>]
```

Then let's add our exception handler at the end:

```
>>> blah.insert_after('bloo', deal_with_it)
>>> blah.functions      #doctest: +ELLIPSIS
[<function foo ...>, <function bar ...>, <function uh_oh ...>, <function bloo ...>,
↪<function deal_with_it ...>]
```

Just for kicks, let's remove the `foo` function while we're at it:

```
>>> blah.remove('foo')
>>> blah.functions      #doctest: +ELLIPSIS
[<function bar ...>, <function uh_oh ...>, <function bloo ...>, <function deal_with_
↪it ...>]
```

If you're making extensive changes to a state chain, you should feel free to directly manipulate the list of functions, rather than using the more cumbersome *insert_before*, *insert_after*, and *remove* methods. We could have achieved the same result like so:

```
>>> blah.functions = [ blah['bar']
...                  , uh_oh
...                  , blah['bloo']
...                  , deal_with_it
...                   ]
>>> blah.functions      #doctest: +ELLIPSIS
[<function bar ...>, <function uh_oh ...>, <function bloo ...>, <function deal_with_
↪it ...>]
```

Either way, what happens when we run it? Since we no longer have the `foo` function providing a value for `bar`, we'll need to supply that using a keyword argument to *run*:

```
>>> state = blah.run(baz=2)
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 populates an `exception` key in the current run's state dictionary. While `exception` is not `None`, any normal function is skipped, and only functions that ask for `exception` get called. It's like a fast-forward. So in our example `deal_with_it` got called, but `bloo` didn't, which is why there is no `sum`:

```
>>> 'sum' in state
False
```

If we run without tripping the exception in `uh_oh` then we have `sum` at the end:

```
>>> blah.run(baz=5)['sum']
7
```

CHAPTER 3

API Reference

exception state_chain.**FunctionNotFound**
Used when a function is not found in a state_chain function list (subclasses `KeyError`).

__str__()
Return str(self).

class state_chain.**StateChain**(**functions*, ***kw*)
Model an algorithm as a list of functions operating on a shared state dictionary.

Parameters

- **functions** – a sequence of functions in the order they are to be run
- **raise_immediately** (*bool*) – Whether to re-raise exceptions immediately. `False` by default, this can only be set as a keyword argument

Each function in the state chain must return a mapping or `None`. If it returns a mapping, the mapping will be used to update a state dictionary for the current run of the algorithm. Functions in the state chain can use any name from the current state dictionary as a parameter, and the value will then be supplied dynamically via `dependency_injection`. See the *run* method for details on exception handling.

__init__(**functions*, ***kw*)
Initialize self. See help(type(self)) for accurate signature.

functions = None
A list of functions comprising the algorithm.

run(*_raise_immediately=None*, *_return_after=None*, ***state*)
Run through the functions in the *functions* list.

Parameters

- **_raise_immediately** (*bool*) – if not `None`, will override any default for `raise_immediately` that was set in the constructor
- **_return_after** (*str*) – if not `None`, return after calling the function with this name
- **state** (*dict*) – remaining keyword arguments are used for the initial state dictionary for this run of the state chain

Raises *FunctionNotFound*, if there is no function named `_return_after`

Returns a dictionary representing the final state

The state dictionary is initialized with three items (their default values can be overriden using keyword arguments to *run*):

- `chain` - a reference to the parent *StateChain* instance
- `state` - a circular reference to the state dictionary
- `exception` - `None`

For each function in the *functions* list, we look at the function signature and compare it to the current value of `exception` in the state dictionary. If `exception` is `None` then we skip any function that asks for `exception`, and if `exception` is *not* `None` then we only call functions that *do* ask for it. The upshot is that any function that raises an exception will cause us to fast-forward to the next exception-handling function in the list.

Here are some further notes on exception handling:

- If a function provides a default value for `exception`, then that function will be called whether or not there is an exception being handled.
- You should return `{'exception': None}` to reset exception handling. Under Python 2 we will call `sys.exc_clear` for you (under Python 3 exceptions are cleared automatically at the end of except blocks).
- If an exception is raised by a function handling another exception, then `exception` is set to the new one and we look for the next exception handler.
- If `exception` is not `None` after all functions have been run, then we re-raise it.
- If `raise_immediately` evaluates to `True` (looking first at any per-call `_raise_immediately` and then at the instance default), then we re-raise any exception immediately instead of fast-forwarding to the next exception handler.
- When an exception occurs, the functions that accept an `exception` argument will be called from inside the `except:` block, so you can access `sys.exc_info` (which contains the traceback) even under Python 3.

__getitem__(*name*)

Return the function in the *functions* list named `name`, or raise *FunctionNotFound*.

```
>>> def foo(): pass
>>> algo = StateChain(foo)
>>> algo['foo'] is foo
True
>>> algo['bar']
Traceback (most recent call last):
  ...
FunctionNotFound: The function 'bar' isn't in this state chain.
```

get_names()

Returns a list of the names of the functions in the *functions* list.

insert_before(*name*, **newfuncs*)

Insert `newfuncs` in the *functions* list before the function named `name`, or raise *FunctionNotFound*.

```
>>> def foo(): pass
>>> algo = StateChain(foo)
>>> def bar(): pass
>>> algo.insert_before('foo', bar)
>>> algo.get_names()
['bar', 'foo']
>>> def baz(): pass
>>> algo.insert_before('foo', baz)
>>> algo.get_names()
['bar', 'baz', 'foo']
>>> def bal(): pass
>>> algo.insert_before(StateChain.START, bal)
>>> algo.get_names()
['bal', 'bar', 'baz', 'foo']
>>> def bah(): pass
>>> algo.insert_before(StateChain.END, bah)
>>> algo.get_names()
['bal', 'bar', 'baz', 'foo', 'bah']
```

insert_after(*name*, **newfuncs*)

Insert `newfuncs` in the *`functions`* list after the function named `name`, or raise *`FunctionNotFound`*.

```
>>> def foo(): pass
>>> algo = StateChain(foo)
>>> def bar(): pass
>>> algo.insert_after('foo', bar)
>>> algo.get_names()
['foo', 'bar']
>>> def baz(): pass
>>> algo.insert_after('bar', baz)
>>> algo.get_names()
['foo', 'bar', 'baz']
>>> def bal(): pass
>>> algo.insert_after(StateChain.START, bal)
>>> algo.get_names()
['bal', 'foo', 'bar', 'baz']
>>> def bah(): pass
>>> algo.insert_before(StateChain.END, bah)
>>> algo.get_names()
['bal', 'foo', 'bar', 'baz', 'bah']
```

remove(**names*)

Remove the functions named `name` from the *`functions`* list, or raise *`FunctionNotFound`*.

classmethod from_dotted_name(*dotted_name*, ***kw*)

Construct a new instance from functions defined in a Python module.

Parameters

- **dotted_name** – the dotted name of a Python module that contains functions that will be added to a state chain in the order of appearance.
- **kw** – keyword arguments are passed through to the default constructor

This is a convenience constructor to instantiate a state chain based on functions defined in a regular Python file. For example, create a file named `blah_state_chain.py` on your `PYTHONPATH`:

```
def foo():
    return {'baz': 1}

def bar():
    return {'buz': 2}

def bloo(baz, buz):
    return {'sum': baz + buz}
```

Then pass the dotted name of the file to this constructor:

```
>>> blah = StateChain.from_dotted_name('blah_state_chain')
```

All functions defined in the file whose name doesn't begin with _ are loaded into a list in the order they're defined in the file, and this list is passed to the default class constructor.

```
>>> blah.functions #doctest: +ELLIPSIS
[<function foo ...>, <function bar ...>, <function bloo ...>]
```

For this specific module, the code above is equivalent to:

```
>>> from blah_state_chain import foo, bar, bloo
>>> blah = StateChain(foo, bar, bloo)
```

debug (*function*)

Given a function, return a copy of the function with a breakpoint immediately inside it.

Parameters **function** (*function*) – a function object

This method wraps the module-level function *state_chain.debug*, adding three conveniences.

First, calling this method not only returns a copy of the function with a breakpoint installed, it actually replaces the old function in the state chain with the copy. So you can do:

```
>>> def foo():
...     pass
...
>>> algo = StateChain(foo)
>>> algo.debug(foo)              #doctest: +ELLIPSIS
<function foo at ...>
>>> algo.run()                   #doctest: +SKIP
(Pdb)
```

Second, it provides a method on itself to install via function name instead of function object:

```
>>> algo = StateChain(foo)
>>> algo.debug.by_name('foo')    #doctest: +ELLIPSIS
<function foo at ...>
>>> algo.run()                   #doctest: +SKIP
(Pdb)
```

Third, it aliases the `by_name` method as `__getitem__` so you can use mapping access as well:

```
>>> algo = StateChain(foo)
>>> algo.debug['foo']            #doctest: +ELLIPSIS
<function foo at ...>
>>> algo.run()                   #doctest: +SKIP
(Pdb)
```

Why would you want to do that? Well, let's say you've written a library that includes a state chain:

```
>>> def foo(): pass
...
>>> def bar(): pass
...
>>> def baz(): pass
...
>>> blah = StateChain(foo, bar, baz)
```

And now some user of your library ends up rebuilding the functions list using some of the original functions and some of their own:

```
>>> def mine(): pass
...
>>> def precious(): pass
...
>>> blah.functions = [ blah['foo']
...                   , mine
...                   , blah['bar']
...                   , precious
...                   , blah['baz']
...                    ]
```

Now the user of your library wants to debug `blah['bar']`, but since they're using your code as a library it's inconvenient for them to drop a breakpoint in your source code. With this feature, they can just insert `.debug` in their own source code like so:

```
>>> blah.functions = [ blah['foo']
...                   , mine
...                   , blah.debug['bar']
...                   , precious
...                   , blah['baz']
...                    ]
```

Now when they run the state chain they'll hit a pdb breakpoint just inside your `bar` function:

```
>>> blah.run()              #doctest: +SKIP
(Pdb)
```

state_chain.**debug** (*function*)

Given a function, return a copy of the function with a breakpoint immediately inside it.

Parameters **function** (*function*) – a function object

Okay! This is fun. :-)

This is a decorator, because it takes a function and returns a function. But it would be useless in situations where you could actually decorate a function using the normal decorator syntax, because then you have the source code in front of you and you could just insert the breakpoint yourself. It's also pretty useless when you have a function object that you're about to call, because you can simply add a `set_trace` before the function call and then step into the function. No: this helper is only useful when you've got a function object that you want to debug, and you have neither the definition nor the call conveniently at hand. See the method *StateChain.debug* for an explanation of how this situation arises with the *state_chain* module.

For our purposes here, it's enough to know that you can wrap any function:

```
>>> def foo(bar, baz):
...     return bar + baz
...
>>> func = debug(foo)
```

And then calling the function will drop you into pdb:

```
>>> func(1, 2)                  #doctest: +SKIP
(Pdb)
```

The fun part is how this is implemented: we dynamically modify the function's bytecode to insert the statements `import pdb; pdb.set_trace()`. Neat, huh? :-)

Python Module Index

S

Index

Symbols

D

F

G

I

R

S