simpy-events Documentation

Release 0.0.1

Loïc Peron

Mar 18, 2019

documentation:

CHAPTER 1

documentation

1.1 TODO

todo

CHAPTER 2

source documentation

2.1 event

class simpy_events.event.**Callbacks**(*event*, *before*, *callbacks*, *after*)

Replace the 'callbacks' list in simpy.events.Event objects.

Internally used to replace the single list of callbacks in simpy.events.Event objects.

See also:

Event

It allows to add the *Event*'s hooks before, when and after the simpy.events.Event object is processed by simpy (that is when the items from its "callbacks" list are called).

Callbacks is intended to replace the original callbacks list of the simpy.events.Event object When iterated, it chains the functions attached to before, callbacks and after.

In order to behave as expected by simpy, adding or removing items from a *Callbacks* object works as expected by simpy: *Callbacks* is a collections.MutableSequence and callables added or removed from it will be called by simpy as regular callbacks, i.e *f(event)* where *event* is a simpy.events.Event object.

When used to replace the simpy.events.Event's callbacks attribute, it ensures the correct order is maintained if the original simpy.events.Event's callbacks attribute was itself a *Callbacks* object, example:

```
cross_red_light = Event(name='cross red light')
get_caught = Event(name='caught on camera')
evt = cross_red_light(env.timeout(1))
yield get_caught(evt)
```

In this example, the call order will be as follows

```
- cross_red_light's before
- get_caught's before
- cross_red_light's callbacks
```

(continues on next page)

(continued from previous page)

```
- get_caught's callbacks
- cross_red_light's after
- get_caught's after
```

__delitem__(*index*)
del callable item from 'callbacks' list

__getitem__(*index*)
return callable item from 'callbacks' list

__init__(*event*, *before*, *callbacks*, *after*)
Attach the *Callbacks* obj to a simpy.events.Event obj.

event is the simpy.events.Event object whose callbacks attribute is going to be replaced by this *Callbacks* object.

before, callbacks and after are callables which will be called respectively before, when and after the event is actually processed by simpy.

Note: the current event.callbacks attribute may already be a *Callbacks* object, see *Callbacks* description for details.

__len__()
return number of callable items in 'callbacks' list

__setitem__(*index*, *value*)
set callable item in 'callbacks' list

insert(*index*, *value*)
insert callable item in 'callbacks' list

class simpy_events.event.**Context**(***attributes*)
context object forwarded to event handlers by *EventDispatcher*

contains following attributes:

- event, the *Event* instance
- hook, the name of the hook

__init__(***attributes*)
initializes a new *Context* with keyword arguments

creates an attribute for each provided keyword arg.

class simpy_events.event.**Event**(***metadata*)
Event provides a node to access the event system.

an *Event* is an endpoint that allows to dispatch a hook to a set of handlers. A hook identifies a particular state for the *Event*, note *Event* is intended to be used to *wrapp* simpy.events.Event objects.

- **enable**: triggered when *Event.enabled* is set to True
- **disable**: triggered when *Event.enabled* is set to to False
- **before**: just before the simpy.events.Event is processed by simpy
- **callbacks**: when the simpy.events.Event is processed by simpy (i.e when callbacks are called)
- **after**: just after the simpy.events.Event is processed by simpy

Event provides two options to dispatch an event through the event system:

- immediately dispatch a `hook` with *Event.dispatch*: although this method is used internally it may be used to dispatch any arbitrary `hook` immediately.
- call the *Event* providing a `simpy.events.Event` object, so the 'before', 'callbacks' and 'after' hooks will be dispatched automatically when the event is processed by the `simpy` loop.

  See also:

  Event.__call__

Event is initialized with optional `metadata` attributes, provided as keyword args, which will be kept alltogather in `Event.metadata` attribute.

handlers:

Handlers are attached to an *Event* using the `Event.topics` list, which is expected to contain a sequence of mappings, each mapping holding itself a sequence of callable handlers for a given `hook`, for ex

```
evt = Event()

topic1 = {
    'before': [h1, h2, h3],
    'after': [h4, h5],
}

evt.topics.append(topic1)
```

Note: a topic is not expected to contain all the possible hook keys, it will be ignored if the hook is not found.

---

events dispatching:

`Event.dispatcher` holds a dispatcher object (such as *EventDispatcher*) that is called by the *Event* when dispatching a hook.

Note setting `Event.dispatcher` to `None` will prevent anything from being dispatched for the *Event* instance.

See also:

Event.dispatch

Event.enabled offers a switch to enable / disable dispatching. It also allows to notify handlers when the *Event* is enabled or disabled, for instance when adding / removing an *Event* in the simulation.

__call__(*event*)

Automatically trigger the *Event* when `event` is processed.

The *Event* will be attached to the provided `simpy.events.Event` object via its callbacks, and the following hooks will be dispatched when `event` is processed by `simpy` (i.e when its callbacks are called) :

- **before**: just before `event` is processed
- **callbacks**: when `event` is processed
- **after**: just after `event` is processed

Replaces the `simpy.events.Event` callbacks attribute by a *Callbacks* instance so the hooks subscribed to this *Event* will be called when the `simpy.events.Event` is processed by `simpy`.

When the `simpy.events.Event` is processed, then calls *Event.dispatch* respectively for 'before', 'callbacks' and 'after' hooks.

return the simpy.events.Event object.

example usage in a typical simpy process

```
something_happens = Event(name='important', context='test')

def my_process(env):
    [...]
    yield something_happens(env.timeout(1))
```

__init__(***metadata*)
Initialized a new *Event* object with optional metadata

metadata keyword args are kept in Event.metadata.

dispatch(*hook*, *data=None*)
immediately dispatch hook for this *Event*.

- hook is the name of the hook to dispatch, for instance 'before', 'after'... etc.
- data is an optional object to forward to the handlers. It will be None by default.

Does nothing if *Event.enabled* is False or Event.dispatcher is None.

calls the dispatcher.dispatch method with the following arguments:

- event: the *Event* instance
- hook
- data

enabled
enable / disable dispatching for the *Event*.

when the value of *Event.enabled* is changed the following hooks are dispatched:

- **enable** is dispatched just after the value is changed
- **disable** is dispatched just before the value is changed

See also:

Event.dispatch

class simpy_events.event.**EventDispatcher**
Responsible for dispatching an event to *Event*'s handlers

uses the *Event*'s sequence of topics to get all handlers for a given hook and call them sequentially.

dispatch(*event*, *hook*, *data*)
dispatch the event to each topic in Event.topics.

args:

- event, the *Event* instance
- hook, the name of the hook to dispatch
- data, data associated to the event

See also:

Event.dispatch

Each topic is expected to be a mapping containing a sequence of handlers for a given hook. The topic will be ignored if it doesn't contain the hook key.

For each sequence of handlers found for `hook`, a `tuple` is created to ensure consistency while iterating (it's likely handlers are removed / added while dispatching).

Handlers are then called sequentially with the following arguments:

- `context`, a *Context* object
- `data`

2.2 manager

class simpy_events.manager.**EventType**(*ns*, *name*)

Link a set of *simpy_events.event.Event* instances to a name.

EventType allows to define an *event type* identified by a name in a given *NameSpace*, and create *simpy_events.event.Event* instances from it, which will allow to manage those instances as a group and share common properties:

- *Topic* objects can be added to the *EventType* and then automatically linked to the *simpy_events.event.Event* instances.
- the *simpy_events.event.Event* instances are managed through the *NameSpace*/*EventType* hierarchy that allows to manage the `simpy_events.event.Event.dispatcher` and *simpy_events.event.Event.enabled* values either for a given *NameSpace* or a given *EventType*.
- the *NameSpace* instance and the name of the *EventType* will be given as metadata to the created events (see *EventType.create*).

Todo: remove event instance ?

__init__(*ns*, *name*)

initializes an *EventType* attached to *ns* by name *name*.

See also:

EventType are expected to be initialized automatically, see also *NameSpace.event_type*.

ns is the *NameSpace* instance that created and holds the *EventType*.

name is the name of the *EventType* and under which it's identified in its parent *ns*.

add_topic(*topic*)

add a *Topic* object to this *EventType*.

This will immediately link the *Topic* to the existing and future created *simpy_events.event.Event* instances for this *EventType*,

create(****metadata*)

create a *simpy_events.event.Event* instance

`metadata` are optional keyword args that will be forwarded as it is to initialize the event.

by default two keyword args are given to the *simpy_events.event.Event* class:

- *ns*: the *NameSpace* instance (*EventType.ns*)
- *name*: the name of the *EventType* (*EventType.name*)

those values will be overriden by custom values if corresponding keyword are contained in `metadata`.

Once the event has been created the *Topic* objects linked to the *EventType* are linked to the *simpy_events.event.Event* instance.

Then *simpy_events.event.Event.enabled* and `simpy_events.event.Event.dispatcher` values for the created event are synchronized with the hierarchy (*NameSpace*/*EventType*).

instances
iter on created *simpy_events.event.Event* instances

name
(read only) The name of the *EventType*

ns
(read only) The *NameSpace* that holds the *EventType*

remove_topic(*topic*)
remove a *Topic* object from this *EventType*.

The *Topic* will immediately be unlinked from the existing *simpy_events.event.Event* instances for this *EventType*.

topics
iter on added *Topic* objects

class simpy_events.manager.**EventsPropertiesMixin**(*parent*, ***values*)
Internally used mixin class to add *EventsProperty* instances

This class add an *EventsProperty* instance for each attribute name in `EventsPropertiesMixin._props` :

- "dispatcher"
- "enabled"

This is used to ensure a hierarchically set value for the corresponding attribute of *simpy_events.event.Event* instances.

See also:

NameSpace, *EventType*

For each attribute:

- a `property` is used to set / get the value
- the *EventsProperty* object is stored in a private attribute using the name '_{attr_name}' (ex: "_dispatcher")

Then the *EventsPropertiesMixin._add_event_properties* and `EventsPropertiesMixin.remove_event_properties` methods can be used in subclasses to add / remove an event to / from the *EventsProperty* instances.

__init__(*parent*, ***values*)
`parent` is either `None` or a *EventsPropertiesMixin*.

`values` are optional extra keyword args to initialize the value of the *EventsProperty* objects (ex: dispatcher=...).

For each managed attribute, the *EventsProperty* object is stored in a private attribute using the name '_{attr_name}' (ex: "_dispatcher").

_add_event_properties(*event*)
used in subclasses to add a *simpy_events.event.Event*.

This add the event to each contained *EventsProperty* object, so the corresponding attribute is hierarchically set for the event.

_remove_event_properties(*event*)
used in subclasses to remove a *simpy_events.event.Event*.

This remove the event from each contained *EventsProperty* object.

class simpy_events.manager.**EventsProperty**(*name*, *value*, *parent*)
Set an attribue value for a hierarchy of parents/children

EventsProperty is used internally to automatically set the value of a specific attribute from a parent down to a hierarchy given the following rules:

- the value of the parent is set recursively to children until a child contains a not `None` value.
- if the value of a given node is set to `None` then the first parent whose value is not `None` will be used to replace the value recursively.

In other words *EventsProperty* ensures the a hierarchically set value that can be overriden by children nodes.

See also:

EventsPropertiesMixin

__init__(*name*, *value*, *parent*)
creates a new hierarchical attribute linked to `parent`

for each event added to this node its `name` attribute will be set every time the applicable value is updated (this *EventsProperty*'s value or a parent value depending on whether the value is `None` or not).

add_event(*event*)
add an event to this node

the corresponding attribute will be hierarchically set starting from this node in the hierarchy for the added event.

remove_event(*event*)
remove an event from the hierarchy.

This doesn't modify the corresponding attribute.

value
return the current value of this node in the hierarchy

class simpy_events.manager.**Handlers**(*lst=None*)
Holds a sequence of handlers.

Handlers is a sequence object which holds handlers for a specific hook in a topic.

See also:

simpy_events.event.Event

Handlers behave like a `list` expect it's also callable so it can be used as a decorator to append handlers to it.

__call__(*fct*)
append `fct` to the sequence.

Handlers object can be used as a decorator to append a handler to it.

__init__(*lst=None*)
Initialize self. See help(type(self)) for accurate signature.

insert(*index*, *value*)
S.insert(index, value) – insert value before index

class simpy_events.manager.**NameSpace**(*parent*, *name*, *root*, ***kwargs*)
Define a hierarchical name space to link events and handlers.

NameSpace provides a central node to automatically link *simpy_events.event.Event* objects and their handlers.

NameSpace allows to define *EventType* objects and create *simpy_events.event.Event* instances associated with those event types.

It also allows to define *Topic* objects and link them to event types. Handlers can then be attached to the *Topic* objects, which will automatically link them to the related *simpy_events.event.Event* instances.

Then, *NameSpace* and *EventType* also allow to set / override *simpy_events.event.Event.enabled* and simpy_events.event.Event.dispatcher attributes at a given point in the hierarchy.

See also:

RootNameSpace

__init__(*parent*, *name*, *root*, ***kwargs*)
NameSpace are expected to be initialized automatically

See also:

NameSpace.ns, *RootNameSpace*

- parent is the parent *NameSpace* that created it
- *name* is the name of the *NameSpace*
- root is the *RootNameSpace* for the hierarchy
- additional kwargs are forwarded to *EventsPropertiesMixin*

event(*name*, **args*, ***kwargs*)
create a *simpy_events.event.Event* instance

name is the name of the event type to use, it is either relative or absolute (see *NameSpace.event_type*).

additional args and kwargs are forwarded to *EventType.create*.

NameSpace.event is a convenience method, the following

```
ns.event('my event')
```

is equivalent to

```
ns.event_type('my event').create()
```

event_type(*name*)
find or create an *EventType*

name is either relative or absolute (see *NameSpace.ns* for details).

Note: the *EventType* objects have their own mapping within a given *NameSpace*, this means an *EventType* and a child *NameSpace* can have the same name, ex:

```
ns.event_type('domain')
ns.ns('domain')
```

---

will create the *EventType* instance if it doesn't exist.

handlers(*name*, *hook*)
return the handlers for the topic *name* and the hook `hook`

This is a convenience method that returns the *Handlers* sequence for a given `hook` in a given *Topic*.

See also:

NameSpace.topic, *Topic.handlers*

Then the following

```
ns.handlers('my topic', 'before')
```

is equivalent to

```
ns.topic('my topic').handlers('before')
```

Note: this method can be used as a decorator to register a handler, for ex

```
@ns.handlers('my topic', 'before')
def handler(context, data):
    pass
```

---

name
(read only) the name of the *NameSpace*

example:

```
root = RootNameSpace(dispatcher)
ns = root.ns('first::second::third')
assert ns.name == 'third'
```

ns(*name*)
return or create the child *NameSpace* for *name*

There is a unique *name*:*NameSpace* pair from a given *NameSpace* instance. It's automatically created when accessing it if it doesn't exist.

name is either a relative or absolute name. An absolute name begins with '::'.

If *name* is absolute the *NameSpace* is referenced from the *RootNameSpace* in the hierarchy, ex:

```
ns = root.ns('one')
assert ns.ns('::one::two') is root.ns('one::two')
```

On the other hand a relative name references a *NameSpace* from the node on which ns is called, ex:

```
ns = root.ns('one')
assert ns.ns('one::two') is not root.ns('one::two')
assert ns.ns('one::two') is ns.ns('one').ns('two')
assert ns.ns('one::two') is root.ns('one::one::two')
```

Note: *name* cannnot be empty (ValueError), and redundant separators ('::'), as well as trailing separators will be ignored, ex:

```
ns1 = ns.ns('::::one::::two::::::::three::')
assert ns1 is ns.ns('::one::two::three')
```

Note: ':' will be processed as a normal character, ex:

```
assert ns.ns(':one').name == ':one'
ns1 = ns.ns(':one::two:::::three:')
ns2 = ns.ns(':one').ns('two').ns(':').ns('three:')
assert ns1 is ns2
```

See also:

NameSpace.path

path
(read only) return the absolute path of in the hierarchy

example:

```
root = RootNameSpace(dispatcher)
ns = root.ns('first::second::third')
assert ns.path == '::first::second::third'
```

Note: **str(ns)** will return **ns.path**

topic(*name*)
find or create an *Topic*

name is either relative or absolute (see *NameSpace.ns* for details).

Note: the *Topic* objects have their own mapping within a given *NameSpace*, this means an *Topic* and a child *NameSpace* can have the same name, ex:

```
ns.topic('domain')
ns.ns('domain')
```

will create the *Topic* instance if it doesn't exist.

class simpy_events.manager.**RootNameSpace**(*dispatcher=None*, *enabled=False*)
The root *NameSpace* object in the hierarchy.

the *RootNameSpace* differs from *NameSpace* because it has no parent, as a consequence:

- *RootNameSpace.path* returns None
- RootNameSpace.name returns None
- RootNameSpace.dispatcher cannot be None (i.e unspecified)

  a value can be specified when creating the instance, otherwise a *simpy_events.event.EventDispatcher* will be created

- `RootNameSpace.enabled` cannot be `None` (i.e unspecified)

  the value can be specifiied at creation (`False` by default)

__init__(*dispatcher=None*, *enabled=False*)
init the root *NameSpace* in the hierarchy

- **dispatcher: used (unless overriden in children) to set** `simpy_events.event.Event.dispatcher`

  if the value is not provided then a *simpy_events.event.EventDispatcher* is created

- **enabled: used (unless overriden in children) to set** *simpy_events.event.Event.enabled*

  Default value is `False`

path
(read only) return the absolute path of in the hierarchy

example:

```
root = RootNameSpace(dispatcher)
ns = root.ns('first::second::third')
assert ns.path == '::first::second::third'
```

Note: **str(ns)** will return **ns.path**

---

class simpy_events.manager.**Topic**(*ns*, *name*)
Holds a mapping of handlers to link to specific events.

Topic is a sequence that contains names of events to be linked automatically when they are created or the name of existing events is added.

When events are created they're registered by event type (*EventType*), identified by a name. If that name is contained in a *Topic* then the topic will be added to the *simpy_events.event.Event*'s `topcis` sequence and the handlers it contains will be called when the event is dispatched.

a *Topic* carries a `dict` containing sequences of handlers for specific hooks ('before', 'after'...), and this `dict` is added to *simpy_events.event.Event*'s topics. The topic's dict is added to an event's topics sequence either when the *simpy_events.event.Event* is created or when the corresponding event's type (name) is added to the *Topic*.

Topic's `dict` contains key:value pairs where keys are hook names ('before', 'after'...) and values are *Handlers* objects. The handler functions added to the *Topic* are added to the *Handlers* objects.

The topic is removed automatically from an *simpy_events.event.Event* if the corresponding event type (name) is removed from the *Topic*.

See also:

simpy_events.event.Event, *NameSpace.topic*, *NameSpace.event*

__delitem__(*index*)
remove an event name from the *Topic*

this will remove the topic from the events identified by the event name at the removed `index`.

Note: cannot use a `slice` as `index`, this will raise a `NotImplementedError`.

__getitem__(*index*)
return an event name added to the *Topic*

__init__(*ns*, *name*)
initializes a *Topic* attached to *ns* by its name *name*.

See also:

Topic are expected to be initialized automatically, see also *NameSpace.topic*.

ns is the *NameSpace* instance that created and holds the *Topic*.

name is the name of the *Topic* and under which it's identified in its parent *ns*.

__setitem__(*index*, *event*)
add an event name to the *Topic*

this will take care of removing the topic from the events identified by the current event name at the specified `index`

then the new event name will be added to the sequence and the corresponding events will be linked if instances exist.

Note: cannot use a `slice` as `index`, this will raise a `NotImplementedError`.

---

get_handlers(*hook*)
eq. to *Topic.handlers* but doesnt create the *Handlers*

return the *Handlers* sequence or `None`.

handlers(*hook*)
return the *Handlers* sequence for the hook `hook`.

the *Handlers* sequence for a given hook (i.e 'before', 'after'...) is created in a lazy way by the *Topic*.

See also:

simpy_events.event.Event for details about hooks.

Since *Handlers* can be used as a decorator itself to add a handler to it, this method can be used as a decorator to register a handler, for ex

```
@topic.handlers('before')
def handler(context, data):
    pass
```

See also:

- *Topic.get_handlers*
- `Topic.enable`
- `Topic.disable`
- `Topic.before`
- `Topic.callbacks`
- `Topic.after`

insert(*index*, *event*)
insert an event name into the *Topic*

The new event name is added to the sequence at the specified `index` and the corresponding events are linked if instances exist.

name
(read only) The name of the *Topic*

ns
(read only) The *NameSpace* that holds the *Topic*

topic
(read only) The `dict` that is added to event's `topics`

CHAPTER 3

simpy-events

event system with SimPy to decouple simulation code and increase reusability

(>>>>>>> **WORK IN PROGRESS** <<<<<<<)

CHAPTER 4

A basic example

Note: SimPy is a process-based discrete-event simulation framework based on standard Python.

- Our simplified scenario is composed of:
  - satellites emitting signals
  - receivers receiving and processing signals
- basic imports and creating the root namespace:

```
from simpy_events.manager import RootNameSpace
import simpy

root = RootNameSpace()
```

- implementing a satellite model:

```
sat = root.ns('satellite')

class Satellite:
    chunk = 4

    def __init__(self, name, data):
        self.signal = sat.event('signal', sat=name)
        self.data = tuple(map(str, data))

    def process(self, env):
        signal = self.signal
        data = self.data
        chunk = self.chunk
        # slice data in chunks
        for chunk in [data[chunk*i:chunk*i+chunk]
                      for i in range(int(len(data) / chunk))]:
```

(continues on next page)

(continued from previous page)

```
            event = env.timeout(1, ','.join(chunk))
            yield signal(event)
```

- implementing a receiver model:

```
receiver = root.ns('receiver')
signals = receiver.topic('signals')

@signals.after
def receive_signal(context, event):
    env = event.env
    metadata = context.event.metadata
    header = str({key: val for key, val in metadata.items()
                  if key not in ('name', 'ns')})
    env.process(process_signal(env, header, event.value))

def process_signal(env, header, signal):
    receive = receiver.event('process')
    for data in signal.split(','):
        yield receive(env.timeout(0, f'{header}: {data}'))
```

- creating code to analyse what's going on:

```
@root.enable('analyse')
def new_process(context, event):
    metadata = context.event.metadata
    context = {key: str(val) for key, val in metadata.items()}
    print(f'new signal process: {context}')

@root.after('analyse')
def signal(context, event):
    metadata = context.event.metadata
    ns = metadata['ns']
    print(f'signal: {ns.path}: {event.value}')
```

- setting up our simulation:

```
root.topic('receiver::signals').extend([
    '::satellite::signal',
])
root.topic('analyse').extend([
    '::satellite::signal',
    '::receiver::process',
])

def run(env):
    # create some actors
    s1 = Satellite('sat1', range(8))
    s2 = Satellite('sat2', range(100, 108))
    env.process(s1.process(env))
    env.process(s2.process(env))

    # execute
    root.enabled = True
    env.run()
```

- running the simulation

```
new signal process: {'ns': '::satellite', 'name': 'signal', 'sat': 'sat1'}
new signal process: {'ns': '::satellite', 'name': 'signal', 'sat': 'sat2'}
signal: ::satellite: 0,1,2,3
new signal process: {'ns': '::receiver', 'name': 'process'}
signal: ::satellite: 100,101,102,103
new signal process: {'ns': '::receiver', 'name': 'process'}
signal: ::receiver: {'sat': 'sat1'}: 0
signal: ::receiver: {'sat': 'sat2'}: 100
signal: ::receiver: {'sat': 'sat1'}: 1
signal: ::receiver: {'sat': 'sat2'}: 101
signal: ::receiver: {'sat': 'sat1'}: 2
signal: ::receiver: {'sat': 'sat2'}: 102
signal: ::receiver: {'sat': 'sat1'}: 3
signal: ::receiver: {'sat': 'sat2'}: 103
signal: ::satellite: 4,5,6,7
new signal process: {'ns': '::receiver', 'name': 'process'}
signal: ::satellite: 104,105,106,107
new signal process: {'ns': '::receiver', 'name': 'process'}
signal: ::receiver: {'sat': 'sat1'}: 4
signal: ::receiver: {'sat': 'sat2'}: 104
signal: ::receiver: {'sat': 'sat1'}: 5
signal: ::receiver: {'sat': 'sat2'}: 105
signal: ::receiver: {'sat': 'sat1'}: 6
signal: ::receiver: {'sat': 'sat2'}: 106
signal: ::receiver: {'sat': 'sat1'}: 7
signal: ::receiver: {'sat': 'sat2'}: 107
```

CHAPTER 5

install and test

5.1 install from pypi

using pip:

```
$ pip install simpy-events
```

5.2 dev install

There is a makefile in the project root directory:

```
$ make dev
```

Using pip, the above is equivalent to:

```
$ pip install -r requirements-dev.txt
$ pip install -e .
```

5.3 run the tests

Use the makefile in the project root directory:

```
$ make test
```

This runs the tests generating a coverage html report

5.4 build the doc

The documentation is made with sphinx, you can use the makefile in the project root directory to build html doc:

```
$ make doc
```

CHAPTER 6

Documentation

Documentation on Read The Docs.

CHAPTER 7

Meta

loicpw - peronloic.us@gmail.com

https://github.com/loicpw

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Symbols

A

C

D

E

G

H

I

N

P

R

S

T

V