event_processor

Release 2.4.0

Nicolas Marier

May 19, 2021

CONTENTS

This library aims to simplify the common pattern of event processing. It simplifies the process of filtering, dispatching and pre-processing events as well as injecting dependencies in event processors.

The only requirement is that your events are regular python dictionaries.

Take a look at the following examples to get an overview of the features available! Of course, you can mix and combine them in any way you like to create more complex scenarios.

CHAPTER

ONE

DOCUMENTATION

1.1 Core Concepts

The core idea of this library is really simple. All you have are processors, filters and dependencies. Read on to learn more about each concept.

1.1.1 Processors

Processors are simply functions that you create to process a certain event. Which processor gets called for which event depends on the filters you use for that processor. More info on the *processors* page.

1.1.2 Filters

Filters are how you tell the library which processor to invoke for each event you want to process. There's a few different kinds of filters, the most common being the `Exists` filter and the `Eq` filter. More info on the *filters* page.

Note: It's possible to have ambiguous filters (see *Filters* for details). To resolve them, take a look at *Ranking Processors* and *Invocation Strategies*.

1.1.3 Dependencies

Dependencies are a pretty key concept, because they allow your processors to depend on values obtained dynamically (which is super important if you want to use external APIs in your processors). It's also possible to depend on the event (so you can have it injected into your processor). More info on the *dependencies* page.

1.1.4 Example

This example shows how these three concepts click together to make event processing easy. For the example, we just use a stub for the SSM client and assume that the `admin_email` parameter has a value of `admin@example.com`.

```
from event_processor import EventProcessor, Event, Depends
from event_processor.filters import Exists

event_processor = EventProcessor()
```

(continues on next page)

(continued from previous page)

```
def get_ssm():
    return FakeSSMClient()


@event_processor.processor(Exists("user.email"))
def user_is_admin(raw_event: Event, ssm_client: FakeSSMClient = Depends(get_ssm)) ->␣
↪bool:
    ssm_response = ssm_client.get_parameter(Name="admin_email")
    admin_email = ssm_response["Parameter"]["Value"]
    return raw_event["user"]["email"] == admin_email

print("admin@example.com is admin:", event_processor.invoke({"user": {"email":
↪"admin@example.com"}}))
print("user@example.com is admin:", event_processor.invoke({"user": {"email":
↪"user@example.com"}}))
```

```
admin@example.com is admin: True
user@example.com is admin: False
```

You can see that because the event contains a value at `user.email` (i.e. this path `Exists` in the event), the processor was invoked. It also received the event by specifying a parameter with the `Event` type and received an SSM client by depending on the value returned by `get_ssm`.

1.2 Processors

Processors are the least involved part of the library. All you have to do is register your processors into an event processor so that events can be dispatched to it.

1.2.1 Parameters

You can't specify just any random parameters for your processors, event-processor needs to know what to do with them when invoking your processor. The parameters that your processor can accept are documented in the *Dependencies* section!

1.2.2 Multiple Event Processors

Note that when you register a processor, it will be invoked only by the event processor for which it is registered. For example,

```
from event_processor import EventProcessor, InvocationError
from event_processor.filters import Accept

event_processor = EventProcessor()
other_event_processor = EventProcessor()


@event_processor.processor(Accept())
def my_processor():
```

(continues on next page)

(continued from previous page)

```
    pass


event_processor.invoke({})  # This is fine, a processor exists for the event

try:
    other_event_processor.invoke({})  # This will raise
except InvocationError:
    print("Raised!")
```

```
Raised!
```

1.2.3 Sub-Processors

In a big application, you might not want to have all your processors in the same module, so it's possible to setup sub-processors which get merged with a main processor.

my_module.py

```
from event_processor import EventProcessor
from event_processor.filters import Accept

sub_processor = EventProcessor()


@sub_processor.processor(Accept())
def my_processor():
    pass
```

main.py

```
from event_processor import EventProcessor
from event_processor.filters import Accept

# from my_module.py import sub_processor

main_processor = EventProcessor()
main_processor.add_subprocessor(sub_processor)

# Note that we are invoking on the main processor,
# but the event will be dispatched to the sub-processor.
result = main_processor.invoke({})

print(result)
```

```
sub_processing!
```

1.2.4 Ranking Processors

Note: It's not always necessary to use ranking. Take a look at the warning on the *Filters* page to learn more and see if it's something you need to be concerned about.

Since it's not possible for the library to guess what should happen to a particular event matching multiple filters, figuring that out is left up to the user. In most cases, it's as simple as not worrying about it, but sometimes, dealing with ambiguous filters is just unavoidable.

This is when you should use processor ranking. A processor's rank is basically an indicator of how much priority it has with regards to other processors. It's what helps the library call the right processor for an event that might match multiple processors.

Here's an example of how you can use ranking :

Note: The default rank for processors is `0`. The matching processor with the highest rank will be called. **To learn how to specify what to do when multiple processors match with the same rank, see** *Invocation Strategy*.

Another useful thing to think about is that you can use the `-1` rank to make a processor be called last when there are multiple matches. This is especially useful when coupled with the *Accept* filter.

```
from event_processor import EventProcessor
from event_processor.filters import Exists, Eq

event_processor = EventProcessor()


@event_processor.processor(Exists("a"))
def processor_a():
    print("Processor a!")


@event_processor.processor(Eq("a", "b"), rank=1)
def processor_b():
    print("Processor b!")


event_processor.invoke({"a": "b"})
event_processor.invoke({"a": "not b"})
```

```
Processor b!
Processor a!
```

1.2.5 Invocation Strategy

To choose how to invoke your processor(s) in the case that multiple processors with the same rank all match a given event, you have to choose an invocation strategy.

Note: The default invocation strategy is the *First Match* strategy.

First Match

This strategy calls the first matching processor (among those with the highest rank). It returns the processor's return value as-is.

All Matches

This strategy calls all the matching processors (that have the highest rank). It returns a tuple of results for all the processors (even if only a single match occurred).

No Matches

This strategy calls none of the matching processors if there are more than one (and returns none). Otherwise, it calls the single matching processor and returns its value as-is.

No Matches Strict

This strategy calls none of the matching processors if there are more than one, and it raises an exception. Otherwise, it calls the single matching processors and returns its value as-is.

Example

To use a non-default invocation strategy, use the provided `InvocationStrategies` enum like so :

```
from event_processor import EventProcessor, InvocationStrategies
from event_processor.filters import Exists, Eq

event_processor = EventProcessor(invocation_strategy=InvocationStrategies.ALL_MATCHES)


@event_processor.processor(Exists("a"))
def processor_a():
    print("Processor a!")


@event_processor.processor(Eq("a", "b"))
def processor_b():
    print("Processor b!")


event_processor.invoke({"a": "b"})
```

```
Processor a!
Processor b!
```

1.2.6 Caveats

The main things to keep in mind for processors are :

- The same filter can only be used by one processor.
- It's possible to have ambiguous filters and those should be resolved with ranking.
- Invocation strategies are used when the rank doesn't resolve ambiguous filters.

1.3 Filters

There are a few available filters to help you make sure the correct processor is invoked for the correct event. To see how to use filters in practice, see the *core concepts*.

> **Warning:** It's possible to create different filters that will match the same event. For example, when using the *Exists* and *Eq* filters on the same key, if the *Eq* filter matches, then the *Exists* filter is guaranteed to match.
>
> Have a look at *Ranking Processors* to learn how to resolve these ambiguities. Also, note that these issues may not apply to your context. You only have to worry about this if you have ambiguous filters.

1.3.1 Accept

This filter will always match any event it is presented with. It will even match things that are not dictionaries. Use this if you need to take a default action whenever no processor exists for an event, or if an unexpected event was sent to your system.

```
from event_processor.filters import Accept

accept = Accept()

print(accept.matches({}))
print(accept.matches(None))
print(accept.matches({"Hello", "World"}))
```

```
True
True
True
```

1.3.2 Exists

This filter matches events that contain a certain key (which can be nested), but the value can be anything.

```
from event_processor.filters import Exists

a_exists = Exists("a")
nested = Exists("a.b.c")

print(a_exists.matches({"a": None}))
print(a_exists.matches({"a": 2}))
print(a_exists.matches({}))
print(nested.matches({"a": {"b": {"c": None}}}))
print(nested.matches({"a": {"b": {"c": 0}}}))
```

```
True
True
False
True
True
```

1.3.3 Eq

This filter matches a subset of the events matched by *Exists*. It only matches the events where a specific value is found at the specified key (as opposed to just existing).

```
from event_processor.filters import Eq

a_is_b = Eq("a", "b")
a_b_c_is_none = Eq("a.b.c", None)

print(a_is_b.matches({"a": "b"}))
print(a_is_b.matches({"a": 2}))
print(a_b_c_is_none.matches({"a": {"b": {"c": None}}}))
print(a_b_c_is_none.matches({"a": {"b": {"c": 0}}}))
```

```
True
False
True
False
```

1.3.4 NumCmp

This filter matches numbers that satisfy a comparison function with a given target.

Note: You should try to avoid using this filter directly and instead use *Lt, Leq, Gt, Geq* when possible.

The reason for this advisory is that in python, callables with the same code will compare as not being equal, which means that if you start using lambdas as the comparator (and more critically, if you use different lambdas that have the same behavior as comparators), then the equality checks for this filter will be inaccurate. This leads to duplicate processors not raising exceptions at import time.

The tl;dr: if you use this filter, don't use lambdas as comparators and don't use different functions that do the same thing either.

```
from event_processor.filters import NumCmp

def y_greater_than_twice_x(x, y):
    return (2 * x) < y

# Note that the comparator is the same here, this is important.
# You can use different comparators, but only if they do different things.
twice_a_less_than_four = NumCmp("a", y_greater_than_twice_x, 4)
twice_a_less_than_eight = NumCmp("a", y_greater_than_twice_x, 8)

print(twice_a_less_than_four.matches({"a": 1}))
print(twice_a_less_than_four.matches({"a": 2}))
print(twice_a_less_than_eight.matches({"a": 3}))
print(twice_a_less_than_eight.matches({"a": 4}))
print(twice_a_less_than_eight.matches({"not-a": 2}))
```

```
True
False
True
False
False
```

1.3.5 Lt, Leq, Gt, Geq

These filters all work in the same way in that they match when a value is present at the given path and it satisfies a comparison operation.

- `Lt` means <
- `Leq` means <=
- `Gt` means >
- `Geq` means >=

```
from event_processor.filters import Lt, Leq, Gt, Geq

a_lt_0 = Lt("a", 0)
a_leq_0 = Leq("a", 0)
a_gt_0 = Gt("a", 0)
a_geq_0 = Geq("a", 0)

print(a_lt_0.matches({"a": 0}))
print(a_leq_0.matches({"a": 0}))
print(a_gt_0.matches({"a": 0}))
print(a_geq_0.matches({"a": 0}))
```

```
False
True
```

(continues on next page)

(continued from previous page)

```
False
True
```

1.3.6 Dyn

This filter accepts a resolver parameter, which is any callable. Whether or not it matches a given event depends on the return value of the resolver. If the resolver returns a truthy value, then the filter matches. Otherwise, it doesn't. This is useful when your events have a more complex structure that can't really be handled by other existing filters.

> **Warning:** When using a dynamic filter, it's your job to make sure the functions you supply won't match the same events (and if they do, to specify a *rank* or an *invocation strategy*).

With the Dyn filter, it's useful to use lambda functions because they fit nicely in one line and won't clutter your code. If you use lambda functions, the functions you create **must** accept a single argument (which will be the event).

```
from event_processor.filters import Dyn

a_len_is_0 = Dyn(lambda e: len(e.get("a", [])) == 0)
a_len_is_bigger = Dyn(lambda e: len(e.get("a", [])) >= 1)

print(a_len_is_0.matches({"a": []}))
print(a_len_is_0.matches({"a": [0]}))
print(a_len_is_bigger.matches({"a": []}))
print(a_len_is_bigger.matches({"a": [0, 1]}))
```

```
True
False
False
True
```

It's also possible to use standard functions with the Dyn filter, in which case you can specify any argument that would be valid for a dependency (see *Dependencies* for details). For example :

```
from event_processor import Depends, Event
from event_processor.filters import Dyn

def my_dependency():
    return 0

def my_filter_resolver(event: Event, dep_value: int = Depends(my_dependency)):
    return event["key"] == dep_value

a_filter = Dyn(my_filter_resolver)

print(a_filter.matches({"key": 0}))
print(a_filter.matches({"key": 1}))
```

```
True
False
```

1.3.7 And

This filter does exactly what you would expect, and matches when all the events supplied to it as arguments match. It acts as a logical AND between all its sub-filters.

```
from event_processor.filters import And, Exists

a_exists = Exists("a")
b_exists = Exists("b")
c_exists = Exists("c")

a_and_b_exist = And(a_exists, b_exists)
a_b_and_c_exist = And(a_exists, b_exists, c_exists)

print(a_and_b_exist.matches({"a": 0, "b": 0}))
print(a_and_b_exist.matches({"a": 0, "b": 0, "c": 0}))
print(a_b_and_c_exist.matches({"a": 0, "b": 0}))
print(a_b_and_c_exist.matches({"a": 0, "b": 0, "c": 0}))
```

```
True
True
False
True
```

You can also use & between processors instead of And explicitly to make your filters prettier.

```
from event_processor.filters import And, Exists

a_exists = Exists("a")
b_exists = Exists("b")
c_exists = Exists("c")

a_and_b_exist = a_exists & b_exists
a_b_and_c_exist = a_exists & b_exists & c_exists

print(a_and_b_exist.matches({"a": 0, "b": 0}))
print(a_and_b_exist.matches({"a": 0, "b": 0, "c": 0}))
print(a_b_and_c_exist.matches({"a": 0, "b": 0}))
print(a_b_and_c_exist.matches({"a": 0, "b": 0, "c": 0}))
```

```
True
True
False
True
```

1.3.8 Or

This filter is similar to the *And* filter, except that it will match if any of its sub-filters match.

```
from event_processor.filters import Or, Exists

a_exists = Exists("a")
b_exists = Exists("b")
c_exists = Exists("c")

a_b_or_c_exist = Or(a_exists, b_exists, c_exists)

print(a_b_or_c_exist.matches({"a": 0}))
print(a_b_or_c_exist.matches({"b": 0}))
print(a_b_or_c_exist.matches({"c": 0}))
print(a_b_or_c_exist.matches({"d": 0}))
```

```
True
True
True
False
```

Again, to make things more ergonomic, you can use `|` instead of `Or`.

```
from event_processor.filters import Or, Exists

a_exists = Exists("a")
b_exists = Exists("b")
c_exists = Exists("c")

a_b_or_c_exist = a_exists | b_exists | c_exists

print(a_b_or_c_exist.matches({"a": 0}))
print(a_b_or_c_exist.matches({"b": 0}))
print(a_b_or_c_exist.matches({"c": 0}))
print(a_b_or_c_exist.matches({"d": 0}))
```

```
True
True
True
False
```

1.4 Dependencies

Dependency injection is a useful tool that you can use to keep your code clean and testable, which is why this library offers simple dependency injection out of the box. The current offering was heavily inspired by the excellent FastAPI framework.

1.4.1 Functional Dependencies

This type of dependency is the most flexible and powerful. It essentially allows you to inject a value into your processor which will be computed from the result of another function of your choice.

Note: These dependencies are cached by default, so if that's something you don't want, be sure to specify `cache=False` in your dependency.

Simple Example

```
from event_processor import EventProcessor, Depends
from event_processor.filters import Accept

event_processor = EventProcessor()


def get_my_value():
    return 42


@event_processor.processor(Accept())
def my_processor(my_value : int = Depends(get_my_value)):
    print(my_value)


event_processor.invoke({})
```

```
42
```

Caching Example

If a value should always be dynamic, caching can easily be disabled. Note that two dependencies can refer to the same callable to get a value, and will still honor the caching decision. That is, one call to the callable may be cached, whereas another may not.

```
from event_processor import EventProcessor, Depends
from event_processor.filters import Accept, Exists

event_processor = EventProcessor()
numeric_value = 0


def get_my_value():
    global numeric_value
    numeric_value = numeric_value + 1
    return numeric_value


@event_processor.processor(Accept())
```

(continues on next page)

(continued from previous page)

```
def my_processor_with_caching(my_value : int = Depends(get_my_value)):
    print(my_value)


# Note the rank is required because otherwise Accept() will match anything
@event_processor.processor(Exists("a"), rank=1)
def my_processor_with_caching(my_value : int = Depends(get_my_value, cache=False)):
    print(my_value)


event_processor.invoke({})
event_processor.invoke({})
event_processor.invoke({"a": 0})
```

```
1
1
2
```

Nesting Example

You can also nest dependencies as deep as you want to go, so you can easily re-use them.

```
from event_processor import EventProcessor, Depends
from event_processor.filters import Accept

event_processor = EventProcessor()


def get_zero():
    return 0


# This dependency can itself depend on another value
def get_my_value(zero: int = Depends(get_zero)):
    return zero + 1


@event_processor.processor(Accept())
def my_processor_with_caching(my_value : int = Depends(get_my_value)):
    print(my_value)


event_processor.invoke({})
```

```
1
```

Class Dependencies

Classes themselves are also callables. By default, their init method will be called when you call them, so you can use classes as dependencies as well.

```
from event_processor import EventProcessor, Depends, Event
from event_processor.filters import Exists

event_processor = EventProcessor()


class MyThing:
    def __init__(self, event: Event):
        self.username = event["username"]

    def get_username(self):
        return self.username


@event_processor.processor(Exists("username"))
def my_processor_with_caching(my_thing : MyThing = Depends(MyThing)):
    print(my_thing.get_username())


event_processor.invoke({"username": "someone"})
```

```
someone
```

1.4.2 Event Dependencies

Sometimes it's useful for processors to receive a copy of the event that triggered their invocation, so you can easily signal that it is required by your processor by having a parameter annotated with the `Event` type.

Note: Event dependencies follow the same rules as other dependencies in that other dependencies can depend on the event, allowing dynamic fetching of data or just creation of a convenient type for the event.

Here's an example of a simple event dependency :

```
from event_processor import EventProcessor, Event
from event_processor.filters import Accept

event_processor = EventProcessor()


@event_processor.processor(Accept())
def my_processor_with_caching(event: Event):
    print(event)


event_processor.invoke({"hello": "world"})
```

```
{'hello': 'world'}
```

And here's an example where a dependency depends on the event :

```
from event_processor import EventProcessor, Event
from event_processor.filters import Exists

event_processor = EventProcessor()


# This function could also query a database (in which case it might depend
# on another function that will return a connection from a connection pool).
def extract_email(event: Event):
    return event["email"]


@event_processor.processor(Exists("email"))
def my_processor_with_caching(email: str = Depends(extract_email)):
    print(email)


event_processor.invoke({"email": "someone@example.com"})
```

```
someone@example.com
```

1.4.3 Pydantic Dependencies

Pydantic is a library which helps with data validation and settings management using python type annotations. You can leverage it in event processors to benefit from both the convenience of automatically parsing an event into a given type and having it fully validated. Pydantic can also provide detailed and friendly error messages to users for validation errors.

Here's a simple example to illustrate how the event might be parsed for use in a processor :

```
from event_processor import EventProcessor
from event_processor.filters import Eq
from pydantic import BaseModel

event_processor = EventProcessor()


class CreateUserQuery(BaseModel):
    email: str
    password: str


@event_processor.processor(Eq("query", "create_user"))
def handle_user_creation(query: CreateUserQuery):
    print(query.email)
    print(query.password)
```

(continues on next page)

(continued from previous page)

```
event_processor.invoke(
    {"query": "create_user", "email": "someone@example.com", "password": "hunter2"}
)
```

```
someone@example.com
hunter2
```

You can also add custom validations for fields using validators as well as many other things. Take a look at the pydantic docs to learn more!

1.4.4 Scalar Dependencies

Sometimes, you don't need many parts of an input event, just one or two fields, so depending on the whole event or having to make a pydantic model just for a few fields might feel excessive. This is what scalar dependencies are good for.

> **Warning:** If you want to benefit from type validation for your scalar dependencies, you need to have pydantic installed. If you don't have pydantic, no types will be validated for scalar dependencies (really, not even basic ones).
>
> Also, if you *do* use pydantic, but don't specify a type annotation for a parameter, then `typing.Any` is assumed.

Here's a very basic example :

```
from event_processor import EventProcessor
from event_processor.filters import Exists

event_processor = EventProcessor()


@event_processor.processor(Exists("email"))
def handle_user(email: str):
    print(email)


event_processor.invoke({"email": "someone@example.com"})
```

```
someone@example.com
```

Here's an example with a pydantic field type :

```
from event_processor import EventProcessor
from event_processor.filters import Exists
from pydantic import ValidationError
from pydantic.color import Color

event_processor = EventProcessor()


@event_processor.processor(Exists("my_color"))
```

(continues on next page)

(continued from previous page)

```
def handle_user(my_color: Color):
    print(my_color.as_hex())


event_processor.invoke({"my_color": "white"})

try:
    event_processor.invoke({"my_color": "not-a-color"})
except ValidationError as e:
    print(e.errors()[0]["msg"])
```

```
#fff
value is not a valid color: string not recognised as a valid color
```

1.5 Testing Processors

Thanks to the separation between the definition and invocation of processors, it's really easy to test processors. Since the processor decorator returns the function as-is (and does not modify it), it's possible to test your processor the same way you would test any other function. Dependencies also make it easy to use mocks for your external service dependencies.

Here's an example of how you might test a processor :

```
from event_processor import EventProcessor, Event, Depends
from event_processor.filters import Exists

event_processor = EventProcessor()


class FakeDatabase:
    values = {
        "users": [
            {"email": "admin@example.com", "role": "admin"},
            {"email": "user@example.com", "role": "user"},
        ]
    }

    def get_role_by_email(self, email: str) -> str:
        user = [user for user in self.values["users"] if user["email"] == email][0]
        return user["role"]


database_instance = None
def get_database() -> FakeDatabase:
    global database_instance
    if database_instance is None:
        database_instance = FakeDatabase()
    return database_instance
```

(continues on next page)

(continued from previous page)

```
def extract_email(event: Event):
    return event["email"]


@event_processor.processor(Exists("email"))
def user_is_admin(
    email: str = Depends(extract_email, cache=False),
    db_client: FakeDatabase = Depends(get_database),
):
    user_role = db_client.get_role_by_email(email)
    return user_role == "admin"


print(event_processor.invoke({"email": "user@example.com"}))
print(event_processor.invoke({"email": "admin@example.com"}))


##################### Tests #####################
from unittest.mock import Mock


def test_user_is_admin_returns_true_for_admin_user():
    mock_db = Mock()
    mock_db.get_role_by_email.return_value = "admin"

    result = user_is_admin("someone@example.com", mock_db)

    assert result is True


def test_user_is_admin_returns_false_for_non_admin_user():
    mock_db = Mock()
    mock_db.get_role_by_email.return_value = "user"

    result = user_is_admin("someone@example.com", mock_db)

    assert result is False


test_user_is_admin_returns_true_for_admin_user()
test_user_is_admin_returns_false_for_non_admin_user()
```

```
False
True
```

1.6 API Documentation

1.6.1 Event Processor

class src.event_processor.event_processor.**EventProcessor**(*invocation_strategy: src.event_processor.invocation_strategies.InvocationStrategi = <InvocationStrategies.FIRST_MATCH: <class 'src.event_processor.invocation_strategies.FirstMatch'>>*)

A self-contained event processor.

add_subprocessor(*subprocessor:* src.event_processor.event_processor.EventProcessor)

Add a subprocessor to this event processor

Parameters **subprocessor** – The other event processor to add

invoke(*event: Dict*) → Any

Invoke the correct processor for an event.

There may be multiple processors invoked, depending on the invocation strategy.

Parameters **event** – The event to find a processor for

Returns The return value of the processor

processor(*event_filter: src.event_processor.filters.Filter*, *rank: int = 0*)

Register a new processor with the given filter and rank.

Parameters

- **event_filter** – The filter for which to match events
- **rank** – This processor's rank (when there are multiple matches for a single event)

1.6.2 Exceptions

Exceptions for event processor.

exception DependencyError

Exceptions for failures while resolving dependencies.

exception EventProcessorError

General exception for the event-processor library.

exception FilterError

Exception for failures related to filters.

exception InvocationError

Exception for failures in invocation.

1.6.3 Filtering

1.6.4 Dependency Injection

CHAPTER

TWO

CHANGELOG

- v2.4.1: Fix scalar dependency resolution without pydantic (only raise on actual missing values and not none values)
- v2.4.0: Support scalar value dependencies in processor parameters
- v2.3.1: Raise the correct exception when processor parameters are invalid due to optional args
- v2.3.0: Support dynamic filters
- v2.2.0: Support pydantic models as processor dependencies
- v2.1.1: Fix negative ranks and document the -1 rank usage
- v2.1.0: Add number comparison filters
- v2.0.0: Refactor a lot of the internals, make filters more user friendly and dependency injection more intuitive
- v1.1.0: Add support for subprocessors
- v1.0.0: Move the state and decorators inside a class
- v0.0.1: Initial release

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX