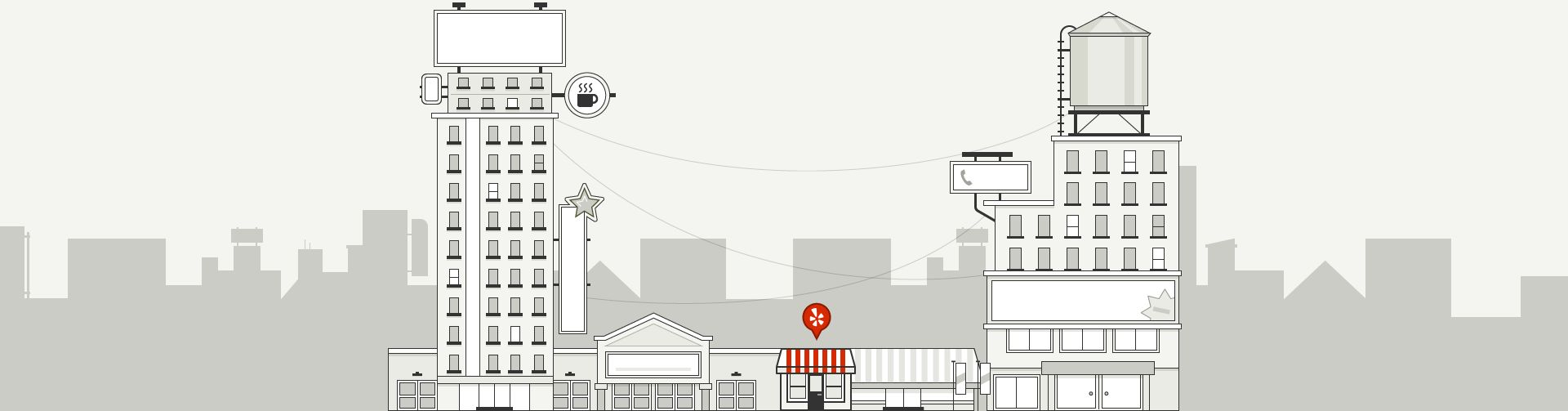
write more decorators

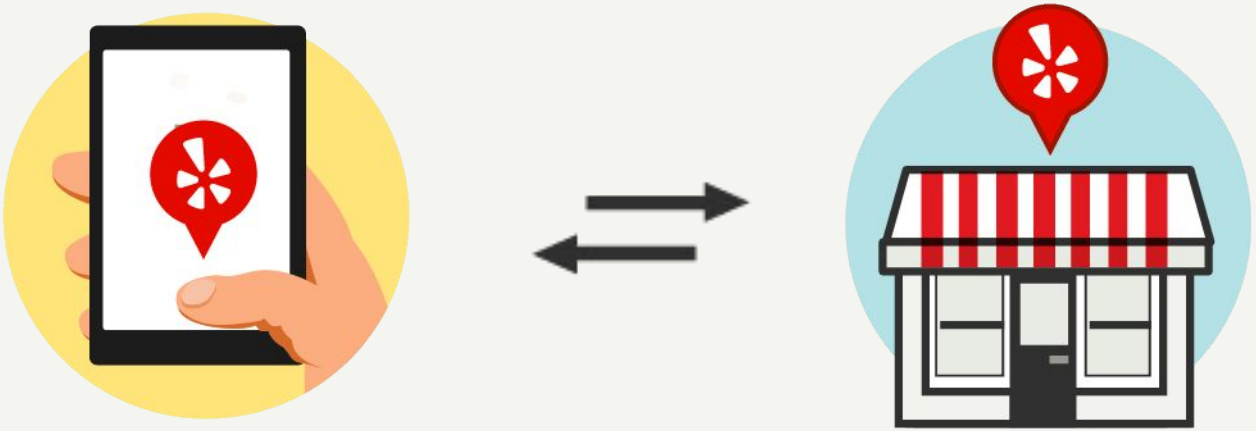
(and fewer classes)

Antonio Verardi
@porosVII
poros.github.io



Yelp's Mission

Connecting people with great local businesses.



this talk has been inspired by

Stop Writing Classes by Jack Diederich

The controller pattern is awful (and other OO heresy) by Lexy Munroe aka Eevee

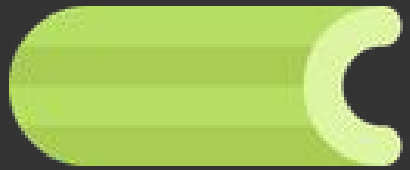
all code is available at
https://goo.gl/vNYczj

let users utilize **all python features**
instead of
just inheritance

go for **decorators**
when your **classes**
have only one method
and
are instantiated only once

what’s this all about?

```python
from celery import Celery

app = Celery('tasks', broker='pyamqp://guest@localhost//')

@app.task
def add(x, y):
    return x + y
```



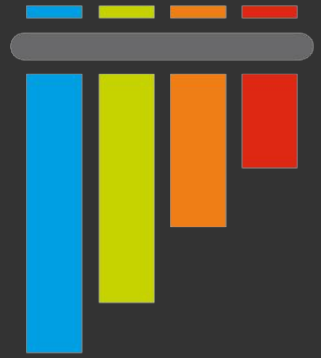
```python
from flask import Flask

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

```python
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.view import view_config

@view_config(route_name='hello', renderer='string')
def hello_world(request):
    return 'Hello World'

if __name__ == '__main__':
    config = Configurator()
    config.add_route('hello', '/')
    config.scan()
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

but what about classes?

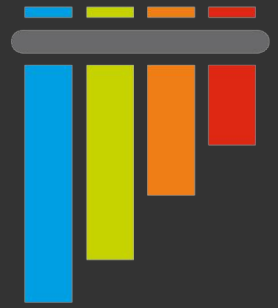
```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50, 50))

    def test_widget_resize(self):
        self.widget.resize(100, 150)
        self.assertEqual(self.widget.size(), (100, 150))
```

```python
import pytest

@pytest.fixture
def widget():
    return Widget('The widget')

def test_default_widget_size(widget):
    assert widget.size() == (50, 50)

def test_widget_resize(widget):
    widget.resize(100, 150)
    assert widget.size() == (100, 150)
```



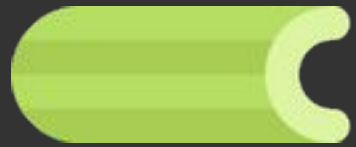
that’s cool, what’s the trick?

```python
def fixture(scope="function", params=None, autouse=False, ids=None, name=None):
    if callable(scope) and params is None and autouse == False:
            # direct decoration
            return FixtureFunctionMarker(
                    "function", params, autouse, name=name)(scope)
    if params is not None and not isinstance(params, (list, tuple)):
            params = list(params)
    return FixtureFunctionMarker(scope, params, autouse, ids=ids, name=name)


class FixtureFunctionMarker:
    def __init__(self, scope, params, autouse=False, ids=None, name=None):
        self.scope = scope
        self.params = params
        self.autouse = autouse
        self.ids = ids
        self.name = name

    def __call__(self, function):
        if isclass(function):
            raise ValueError(
                    "class fixtures not supported (may be in the future)")
        function._pytestfixturefunction = self
        return function
```



```python
class view_config(object):
    venusian = venusian
    def __init__(self, **settings):
        if 'for_' in settings:
            if settings.get('context') is None:
                settings['context'] = settings['for_']
        self.__dict__.update(settings)

    def __call__(self, wrapped):
        settings = self.__dict__.copy()
        depth = settings.pop('_depth', 0)

        def callback(context, name, ob):
            config = context.config.with_package(info.module)
            config.add_view(view=ob, **settings)

        info = self.venusian.attach(wrapped, callback, category='pyramid',
                                    depth=depth + 1)

        if info.scope == 'class':
            if settings.get('attr') is None:
                settings['attr'] = wrapped.__name__

        settings['_info'] = info.codeinfo
        return wrapped
```

```python
class Celery(object):
    def task(self, *args, **opts):
        if USING_EXECV and opts.get('lazy', True):
            from . import shared_task
            return shared_task(*args, lazy=False, **opts)

        def inner_create_task_cls(shared=True, filter=None, lazy=True, **opts):
            _filt = filter  # stupid 2to3

            def _create_task_cls(fun):
                if shared:
                    def cons(app):
                        return app._task_from_fun(fun, **opts)
                    cons.__name__ = fun.__name__
                    connect_on_app_finalize(cons)
                if not lazy or self.finalized:
                    ret = self._task_from_fun(fun, **opts)
                else:
                    ret = PromiseProxy(self._task_from_fun, (fun,), opts,
                                       __doc__=fun.__doc__)
                    self._pending.append(ret)
                if _filt:
                    return _filt(ret)
                return ret

            return _create_task_cls

        if len(args) == 1:
            if callable(args[0]):
                return inner_create_task_cls(**opts)(*args)
            raise TypeError('argument 1 to @task() must be a callable')
        if args:
            raise TypeError(
                '@task() takes exactly 1 argument ({0} given)'.format(
                    sum([len(args), len(opts)])))
        return inner_create_task_cls(**opts)
```




Reebok

decorators are hard

```python
@decorator
def func():
    pass

# is equivalent to

def func():
    pass

func = decorator(func)
```

```python
@decorator(arg)
def func():
    pass

# is equivalent to

def func():
    pass

func = decorator(arg)(func)
```

```
@dec1(arg)
@dec2
def func():
    pass

# is equivalent to

def func():
    pass

func = dec1(arg)(dec2(func))
```

decorators are hard

decorators are hard
to write

```
def decorator(fn):
    print("Decorating function")
    return fn


@decorator
def func(arg1, arg2):
    print("Running")
```

```
def decorator(fn):
    print("Decorating function")
    return fn


@decorator
def func(arg1, arg2):
    print("Running")


>>> from foo import func
Decorating function
>>> foo("A","B")
Running
```

decorators are hard
to write

decorators are ~~hard~~ easy to write

```python
@decorator(arg)
def func():
    pass

# is equivalent to

def func():
    pass

func = decorator(arg)(func)
```

```
def decorator(arg1, arg2):
    def actual_decorator(fn):
        return fn

    print(f"Decorating function with {arg1} and {arg2}")
    return actual_decorator


@decorator("bar", "baz")
def func(arg3, arg4):
    print("Running")
```

```
def decorator(arg1, arg2):
    def actual_decorator(fn):
        return fn

    print(f"Decorating function with {arg1} and {arg2}")
    return actual_decorator


@decorator("bar", "baz")
def func(arg3, arg4):
    print("Running")

>>> from foo import func
Decorating function with bar and baz
>>> foo("A","B")
Running
```

```
def decorator(fn):
    @functools.wraps(fn)  # preserve function's metadata
    def wrapper(*args, **kwargs):
        print("Before the function runs")
        return fn(*args, **kwargs)

    print("Decorating function")
    return wrapper


@decorator
def func(arg1, arg2):
    print("Running")
```

```
def decorator(fn):
    @functools.wraps(fn)   # preserve function's metadata
    def wrapper(*args, **kwargs):
        print("Before the function runs")
        return fn(*args, **kwargs)

    print("Decorating function")
    return wrapper


@decorator
def func(arg1, arg2):
    print("Running")
```

```
>>> from foo import func
Decorating function
>>> foo("A","B")
Before the function runs
Running
```

decorators are ~~hard~~ easy to write

a bit tedious

~~easy~~

decorators are ~~hard~~

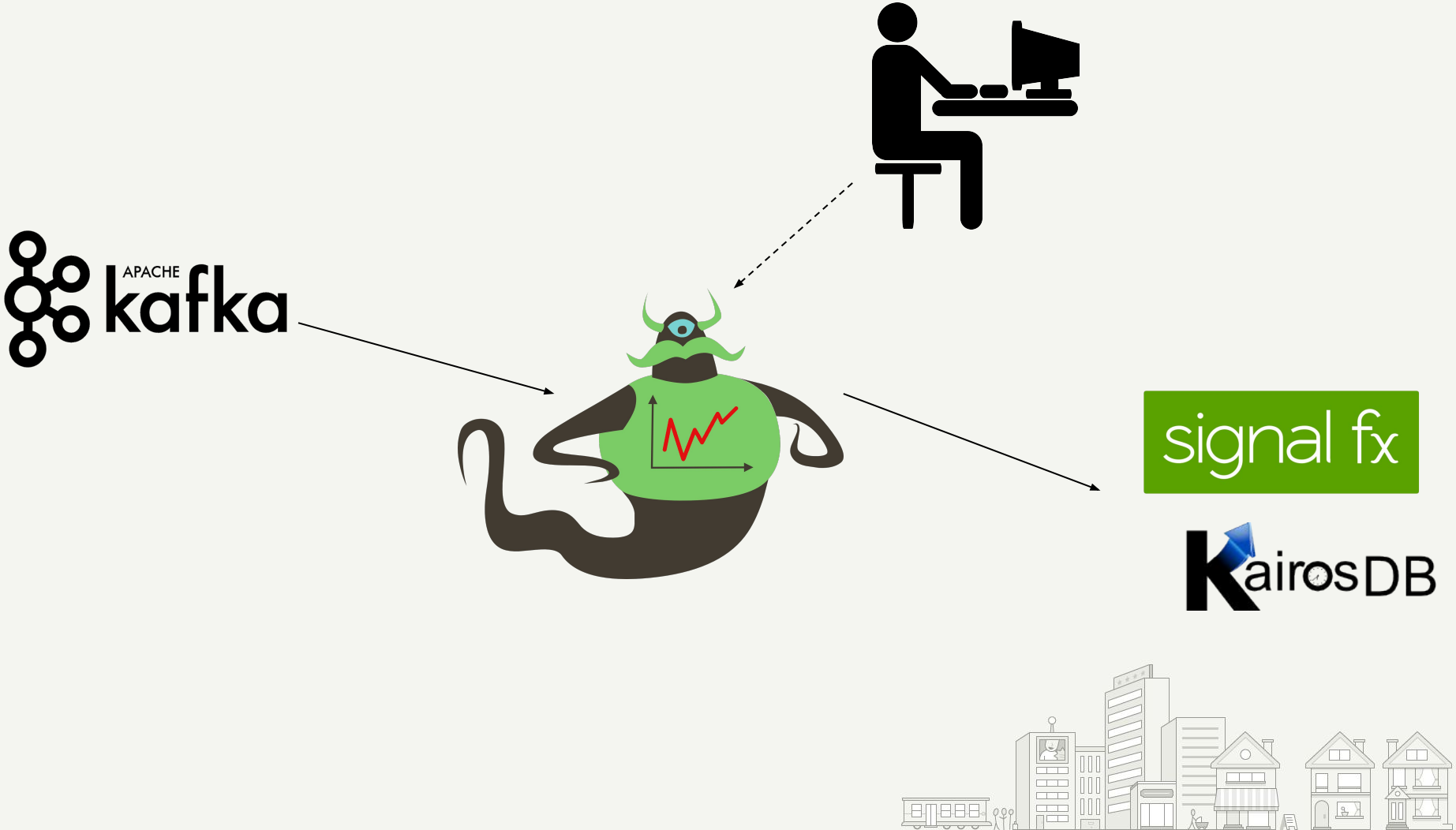
to write

```python
class Flask(_PackageBoundObject):
    def route(self, rule, **options):
        def decorator(f):
            endpoint = options.pop('endpoint', None)
            self.add_url_rule(rule, endpoint, f, **options)
            return f
        return decorator
```

real life example

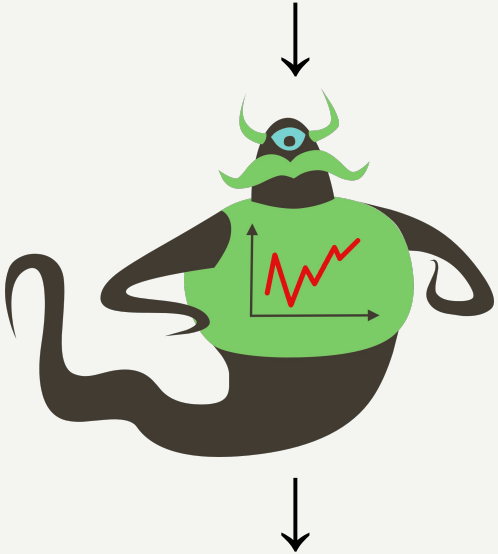
statmonster

statmonster is a framework to extract real-time metrics out of logs


APACHE
kafka
signal fx
KairosDB

logs → statmonster → metrics

```
{"start": 1000, "duration": 42, "method": "GET", "code": 200}
```



```
Timer("request", 1042, 42, {"method": "GET"})
```

logs → statmonster → metrics

```
from enum import Enum
from collections import namedtuple
from functools import partial

MetricType = Enum("MetricType", ("COUNTER", "TIMER"))

Metric = namedtuple(
    "Metric",
    ("name", "ts", "value", "dims", "type")
)

Counter = partial(Metric, type=MetricType.COUNTER)
Timer = partial(Metric, type=MetricType.TIMER)
```

```
from enum import Enum
from collections import namedtuple
from functools import partial

MetricType = Enum("MetricType", ("COUNTER", "TIMER"))

Metric = namedtuple(
    "Metric",
    ("name", "ts", "value", "dims", "type")
)

Counter = partial(Metric, type=MetricType.COUNTER)
Timer = partial(Metric, type=MetricType.TIMER)
```

```
from enum import Enum
from collections import namedtuple
from functools import partial

MetricType = Enum("MetricType", ("COUNTER", "TIMER"))

Metric = namedtuple(
    "Metric",
    ("name", "ts", "value", "dims", "type")
)

Counter = partial(Metric, type=MetricType.COUNTER)
Timer = partial(Metric, type=MetricType.TIMER)
```

```
from enum import Enum
from collections import namedtuple
from functools import partial

MetricType = Enum("MetricType", ("COUNTER", "TIMER"))

Metric = namedtuple(
    "Metric",
    ("name", "ts", "value", "dims", "type")
)

Counter = partial(Metric, type=MetricType.COUNTER)
Timer = partial(Metric, type=MetricType.TIMER)
```

logs → statmonster → metrics

```
import json

def decode_json(line):
    return json.loads(line)

class Log:
    name = None
    decoder = decode_json

    @classmethod
    def decode(cls, line):
        return cls.decoder(line)

    def __init__(self):
        assert self.name, "log name must be specified"
```

```
import json

def decode_json(line):
    return json.loads(line)

class Log:
    name = None
    decoder = decode_json

    @classmethod
    def decode(cls, line):
        return cls.decoder(line)

    def __init__(self):
        assert self.name, "log name must be specified"
```

```
import json

def decode_json(line):
    return json.loads(line)

class Log:
    name = None
    decoder = decode_json

    @classmethod
    def decode(cls, line):
        return cls.decoder(line)

    def __init__(self):
        assert self.name, "log name must be specified"
```

```
from statmonster import Log

class EventsLog(Log):
    name = "events"
```

```
from statmonster import Log

def decode_text(line):
    time, request = line.split()
    return {'time': time, 'request': request}

class RequestsLog(Log):
    name = "tmp_requests"
    decoder = decode_text
```

logs → statmonster → metrics

a trigger is a class which encapsulate the logic to extract metrics from a log

```
class Trigger:
    owners = None

    def __init__(self):
        assert self.owners

    def digest(self, entry):
        raise NotImplementedError
```

```
class Trigger:
    owners = None

    def __init__(self):
        assert self.owners

    def digest(self, entry):
        raise NotImplementedError
```

```
class Trigger:
    owners = None

    def __init__(self):
        assert self.owners

    def digest(self, entry):
        raise NotImplementedError
```

```
def process(log, triggers, line):
    entry = log.decode(line)
    for trigger in triggers:
        try:
            yield from trigger.digest(entry)
        except Exception as e:
            send_email(trigger.owners, e)
```

```
from statmonster import Log, Trigger, Counter

class EventsLog(Log):
    name = "events"

class CountEventsTrigger(Trigger):
    owners = ["antonio@yelp.com"]

    def digest(self, entry):
        yield Counter(
          "events",
          entry["time"], 1, {"type": entry["type"]}
        )
```

```
from statmonster import Log, Trigger, Counter

class EventsLog(Log):
    name = "events"

class CountEventsTrigger(Trigger):
    owners = ["antonio@yelp.com"]

    def digest(self, entry):
        yield Counter(
          "events",
          entry["time"], 1, {"type": entry["type"]}
        )
```

```
from statmonster import Log, Trigger, Counter

class EventsLog(Log):
    name = "events"

class CountEventsTrigger(Trigger):
    owners = ["antonio@yelp.com"]

    def digest(self, entry):
        yield Counter(
          "events",
          entry["time"], 1, {"type": entry["type"]}
        )
```

```
from statmonster import Log, Trigger, Counter

class EventsLog(Log):
    name = "events"

class CountEventsTrigger(Trigger):
    owners = ["antonio@yelp.com"]

    def digest(self, entry):
        yield Counter(
          "events",
          entry["time"], 1, {"type": entry["type"]}
        )
```

```python
from statmonster import Log, Trigger, Timer

class RusageLog(Log):
    name = "rusage"

class TimeCpuTrigger(Trigger):
    owners = ["antonio@yelp.com", "metrics@yelp.com"]

    def digest(self, entry):
        for metric in ("stime", "utime"):
            yield Timer(
                f"cpu.{metric}",
                entry["start_time"],
                entry[metric],
                {}
            )
```

```
from statmonster import Log, Trigger, Timer

class RusageLog(Log):
    name = "rusage"

class TimeCpuTrigger(Trigger):
    owners = ["antonio@yelp.com", "metrics@yelp.com"]

    def digest(self, entry):
        for metric in ("stime", "utime"):
            yield Timer(
                f"cpu.{metric}",
                entry["start_time"],
                entry[metric],
                {}
            )
```

```
from statmonster import Log, Trigger, Timer

class RusageLog(Log):
    name = "rusage"

class TimeCpuTrigger(Trigger):
    owners = ["antonio@yelp.com", "metrics@yelp.com"]

    def digest(self, entry):
        for metric in ("stime", "utime"):
            yield Timer(
                f"cpu.{metric}",
                entry["start_time"],
                entry[metric],
                {}
            )
```

```
from statmonster import Log, Trigger, Timer

class RusageLog(Log):
    name = "rusage"

class TimeCpuTrigger(Trigger):
    owners = ["antonio@yelp.com", "metrics@yelp.com"]

    def digest(self, entry):
        for metric in ("stime", "utime"):
            yield Timer(
                f"cpu.{metric}",
                entry["start_time"],
                entry[metric],
                {}
            )
```

USERS


ENGINEERS

how do I inherit from a base log class?


Reebok

```python
from statmonster import Log
from utils import decode_apache

class ApacheBaseLog(Log):
    decoder = decode_apache
```

_apache.py

```python
from statmonster import Log
from utils import decode_apache

class ApacheBaseLog(Log):
    decoder = decode_apache
```

_apache.py

```python
from ._apache import ApacheBaseLog

class AccessLog(ApacheBaseLog):
    name = "access"
```

access.py

```python
from ._apache import ApacheBaseLog

class AdminAccessLog(ApacheBaseLog):
    name = "admin_access"
```

admin_access.py

how do I inherit from a base trigger class?


Reebok

```python
from statmonster import Trigger, Counter

class EndpointsTimingBaseTrigger(Trigger):
    metric_name = None
    endpoints = None

    def __init__(self):
        super().__init__()
        assert self.metric_name
        assert self.endpoints

    def get_additional_dimensions(self, entry):
        return {}
```

_endpoints.py

```python
from statmonster import Trigger, Counter

class EndpointsTimingBaseTrigger(Trigger):
    metric_name = None
    endpoints = None

    def __init__(self):
        super().__init__()
        assert self.metric_name
        assert self.endpoints

    def get_additional_dimensions(self, entry):
        return {}
```

_endpoints.py

```python
from statmonster import Trigger, Counter

class EndpointsTimingBaseTrigger(Trigger):
    metric_name = None
    endpoints = None

    def __init__(self):
        super().__init__()
        assert self.metric_name
        assert self.endpoints

    def get_additional_dimensions(self, entry):
        return {}
```

_endpoints.py

```
def digest(self, entry):
    ts = entry['start_time']
    add_dims = self.get_additional_dimensions(entry)
    for endpoint in self.endpoints:
        if endpoint == entry["endpoint"]:
            timing = entry["endpoint"]["time"]
            dims = {'endpoint': endpoint}.update(add_dims)
            yield Counter(
                self.metric_name,
                ts,
                timing,
                dims
            )
```

_endpoints.py

```
def digest(self, entry):
    ts = entry['start_time']
    add_dims = self.get_additional_dimensions(entry)
    for endpoint in self.endpoints:
        if endpoint == entry["endpoint"]:
            timing = entry["endpoint"]["time"]
            dims = {'endpoint': endpoint}.update(add_dims)
            yield Counter(
                self.metric_name,
                ts,
                timing,
                dims
            )
```

_endpoints.py

```
def digest(self, entry):
    ts = entry['start_time']
    add_dims = self.get_additional_dimensions(entry)
    for endpoint in self.endpoints:
        if endpoint == entry["endpoint"]:
            timing = entry["endpoint"]["time"]
            dims = {'endpoint': endpoint}.update(add_dims)
            yield Counter(
                self.metric_name,
                ts,
                timing,
                dims
            )
```

_endpoints.py

```
from statmonster import Log
from ._endpoints import EndpointsTimingBaseTrigger

class HomePageLog(Log):
    name = "homepage"

class HomePageEndpointsTrigger(EndpointsTimingBaseTrigger):
    owners = ["consumer@yelp.com"]
    endpoints = ["best_of_yelp", "suggestions", "nearby"]
    metric_name = "home"

    def get_additional_dimensions(self, entry):
        return {
            "mode": entry.get("mode", "sync"),
            "user": "loggedin" if entry["user"] else "anon",
        }
```

```
from statmonster import Log
from ._endpoints import EndpointsTimingBaseTrigger

class HomePageLog(Log):
    name = "homepage"

class HomePageEndpointsTrigger(EndpointsTimingBaseTrigger):
    owners = ["consumer@yelp.com"]
    endpoints = ["best_of_yelp", "suggestions", "nearby"]
    metric_name = "home"

    def get_additional_dimensions(self, entry):
        return {
            "mode": entry.get("mode", "sync"),
            "user": "loggedin" if entry["user"] else "anon",
        }
```

```
from statmonster import Log
from ._endpoints import EndpointsTimingBaseTrigger

class HomePageLog(Log):
    name = "homepage"

class HomePageEndpointsTrigger(EndpointsTimingBaseTrigger):
    owners = ["consumer@yelp.com"]
    endpoints = ["best_of_yelp", "suggestions", "nearby"]
    metric_name = "home"

    def get_additional_dimensions(self, entry):
        return {
            "mode": entry.get("mode", "sync"),
            "user": "loggedin" if entry["user"] else "anon",
        }
```

how do I inherit from **two** trigger classes?


Reebok
Reebok

you…

you…
just…

you…
just…
don’t…

```python
from statmonster import Trigger, Timer, Counter

class ServiceBaseTrigger(Trigger):
    metric_name = None

    def __init__(self):
        super().__init__()
        assert self.metric_name

    def make_key(self, stat_name):
        return f"{self.metric_name}.{stat_name}"
```

_service.py

```python
class ServiceTimingBaseTrigger(ServiceBaseTrigger):

    def digest(self, entry):
        key = self.make_key("request_latency")
        dimensions = {'method': entry['method_name']}

        yield Timer(
            key,
            entry['time'],
            entry['time_elapsed'],
            dimensions
        )
```

_service.py

```
class ServiceTimingBaseTrigger(ServiceBaseTrigger):

    def digest(self, entry):
        key = self.make_key("request_latency")
        dimensions = {'method': entry['method_name']}

        yield Timer(
            key,
            entry['time'],
            entry['time_elapsed'],
            dimensions
        )
```

_service.py

```
class ServiceCountBaseTrigger(ServiceBaseTrigger):

  def digest(self, entry):
    try:
      bytes_written = int(entry['headers']['Content-Length'])
    except KeyError:
        bytes_written = 0
    dimensions = {'method': entry['method_name']}
    request_count_key = self.make_key("request_count")
    bytes_written_key = self.make_key("total_written_bytes")
    ts = entry['time']

    yield Counter(request_count_key, ts, 1, dimensions)
    yield Counter(
      bytes_written_key,
      ts,
      bytes_written,
      dimensions
    )
```

_service.py

```
class ServiceCountBaseTrigger(ServiceBaseTrigger):

  def digest(self, entry):
    try:
      bytes_written = int(entry['headers']['Content-Length'])
    except KeyError:
        bytes_written = 0
    dimensions = {'method': entry['method_name']}
    request_count_key = self.make_key("request_count")
    bytes_written_key = self.make_key("total_written_bytes")
    ts = entry['time']

    yield Counter(request_count_key, ts, 1, dimensions)
    yield Counter(
      bytes_written_key,
      ts,
      bytes_written,
      dimensions
    )
```

_service.py

```python
from statmonster import Log
from ._service import ServiceTimingBaseTrigger
from ._service import ServiceCountBaseTrigger

class SuggestionsLog(Log):
    name = "suggest"

class SuggestionsTimings(ServiceTimingBaseTrigger):
    owners = ["search@yelp.com"]
    metric_name = 'suggestions_timings'

class SuggestionsCount(ServiceCountBaseTrigger):
    owners = ["search@yelp.com"]
    metric_name = 'suggestions_count'
```

suggestions_service.py

```
from statmonster import Log
from ._service import ServiceTimingBaseTrigger
from ._service import ServiceCountBaseTrigger

class SuggestionsLog(Log):
    name = "suggest"

class SuggestionsTimings(ServiceTimingBaseTrigger):
    owners = ["search@yelp.com"]
    metric_name = 'suggestions_timings'

class SuggestionsCount(ServiceCountBaseTrigger):
    owners = ["search@yelp.com"]
    metric_name = 'suggestions_count'
```

suggestions_service.py

```
from statmonster import Log
from ._service import ServiceTimingBaseTrigger
from ._service import ServiceCountBaseTrigger

class SuggestionsLog(Log):
    name = "suggest"

class SuggestionsTimings(ServiceTimingBaseTrigger):
    owners = ["search@yelp.com"]
    metric_name = 'suggestions_timings'

class SuggestionsCount(ServiceCountBaseTrigger):
    owners = ["search@yelp.com"]
    metric_name = 'suggestions_count'
```

suggestions_service.py

how do I test my trigger class?

```python
from metrics.rusage import TimeCpuTrigger
from statmonster import Timer
import pytest

@pytest.fixture
def trigger():
    return TimeCpuTrigger()

def test_time_cpu(trigger):
    entry = {"start_time": 1000, "stime": 42, "utime": 33}
    assert Timer(
        "cpu.stime",
        1000,
        42,
        {}
    ) in list(trigger.digest(entry))
```

```python
from metrics.rusage import TimeCpuTrigger
from statmonster import Timer
import pytest

@pytest.fixture
def trigger():
    return TimeCpuTrigger()

def test_time_cpu(trigger):
    entry = {"start_time": 1000, "stime": 42, "utime": 33}
    assert Timer(
        "cpu.stime",
        1000,
        42,
        {}
    ) in list(trigger.digest(entry))
```

```python
from metrics.rusage import TimeCpuTrigger
from statmonster import Timer
import pytest

@pytest.fixture
def trigger():
    return TimeCpuTrigger()

def test_time_cpu(trigger):
    entry = {"start_time": 1000, "stime": 42, "utime": 33}
    assert Timer(
        "cpu.stime",
        1000,
        42,
        {}
    ) in list(trigger.digest(entry))
```

but how do I test my **base** trigger class?


Reebok

```
class ServiceCountBaseTrigger(ServiceBaseTrigger):

  def digest(self, entry):
    try:
      bytes_written = int(entry['headers']['Content-Length'])
    except KeyError:
        bytes_written = 0
    dimensions = {'method': entry['method_name']}
    request_count_key = self.make_key("request_count")
    bytes_written_key = self.make_key("total_written_bytes")
    ts = entry['time']

    yield Counter(request_count_key, ts, 1, dimensions)
    yield Counter(
      bytes_written_key,
      ts,
      bytes_written,
      dimensions
    )
```

_service.py

```
class ServiceCountBaseTrigger(ServiceBaseTrigger):

  def digest(self, entry):
    try:
      bytes_written = int(entry['headers']['Content-Length'])
    except KeyError:
        bytes_written = 0
    dimensions = {'method': entry['method_name']}
    request_count_key = self.make_key("request_count")
    bytes_written_key = self.make_key("total_written_bytes")
    ts = entry['time']

    yield Counter(request_count_key, ts, 1, dimensions)
    yield Counter(
      bytes_written_key,
      ts,
      bytes_written,
      dimensions
    )
```

_service.py

```python
from statmonster import Trigger, Timer, Counter

class ServiceBaseTrigger(Trigger):
    metric_name = None

    def __init__(self):
        super().__init__()
        assert self.metric_name

    def make_key(self, stat_name):
        return f"{self.metric_name}.{stat_name}"
```

_service.py

```python
from statmonster import Trigger, Timer, Counter

class ServiceBaseTrigger(Trigger):
    metric_name = None

    def __init__(self):
        super().__init__()
        assert self.metric_name

    def make_key(self, stat_name):
        return f"{self.metric_name}.{stat_name}"
```

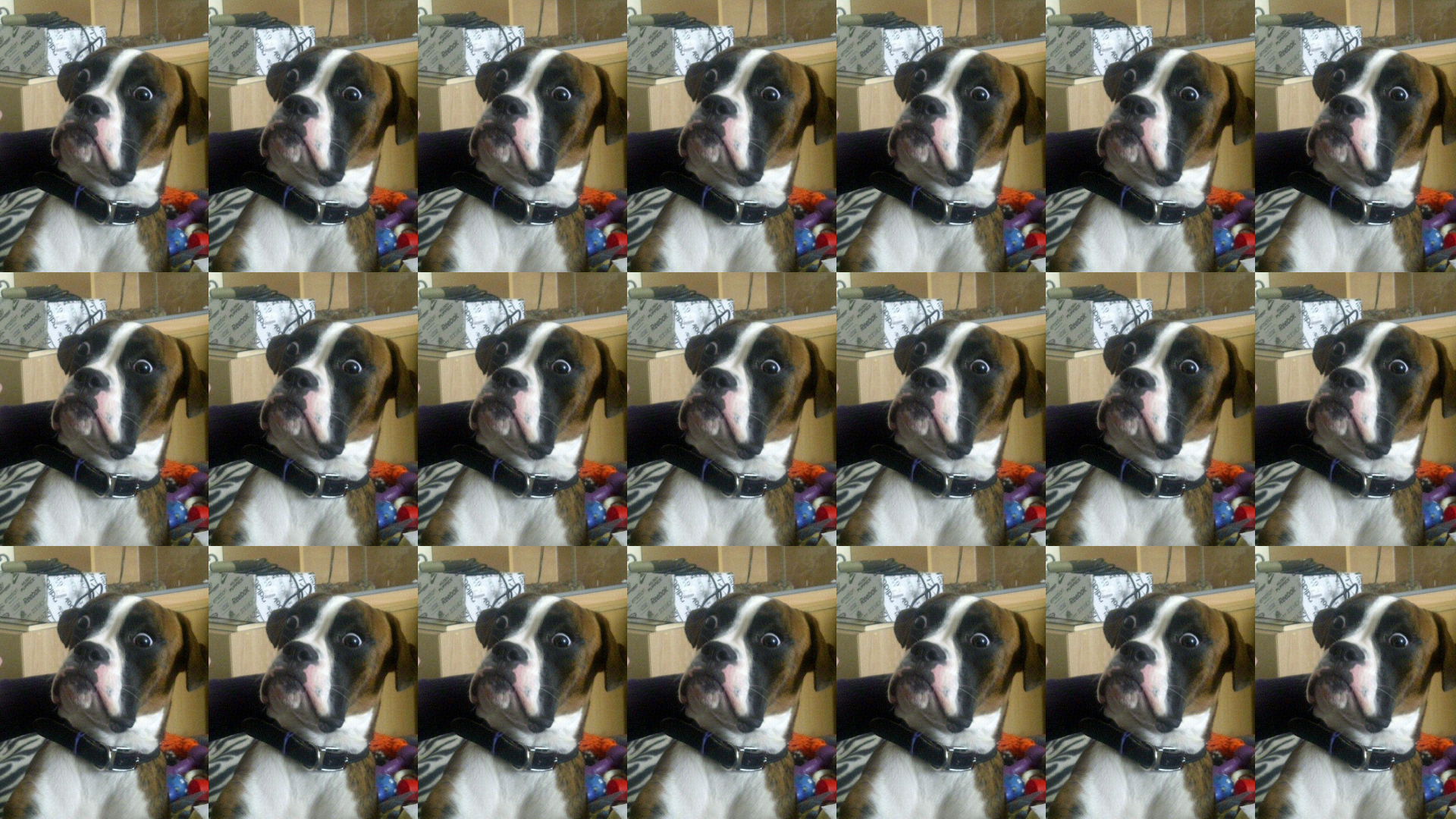
_service.py

```python
from statmonster import Trigger, Timer, Counter

class ServiceBaseTrigger(Trigger):
    metric_name = None

    def __init__(self):
        super().__init__()
        assert self.metric_name

    def make_key(self, stat_name):
        return f"{self.metric_name}.{stat_name}"
```

_service.py

```
class Trigger:
    owners = None

    def __init__(self):
        assert self.owners

    def digest(self, line):
        raise NotImplementedError
```

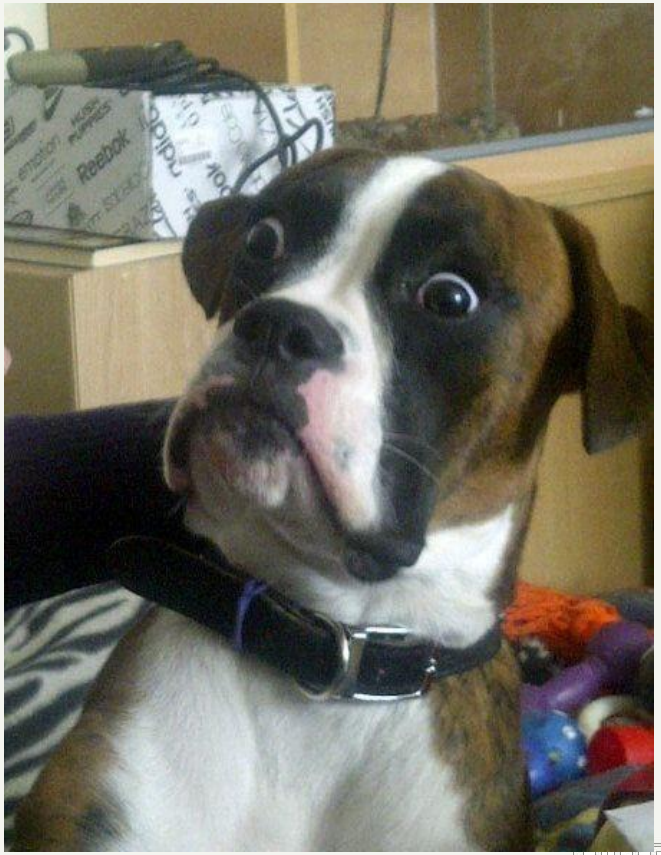
```
class Trigger:
    owners = None

    def __init__(self):
        assert self.owners

    def digest(self, line):
        raise NotImplementedError
```



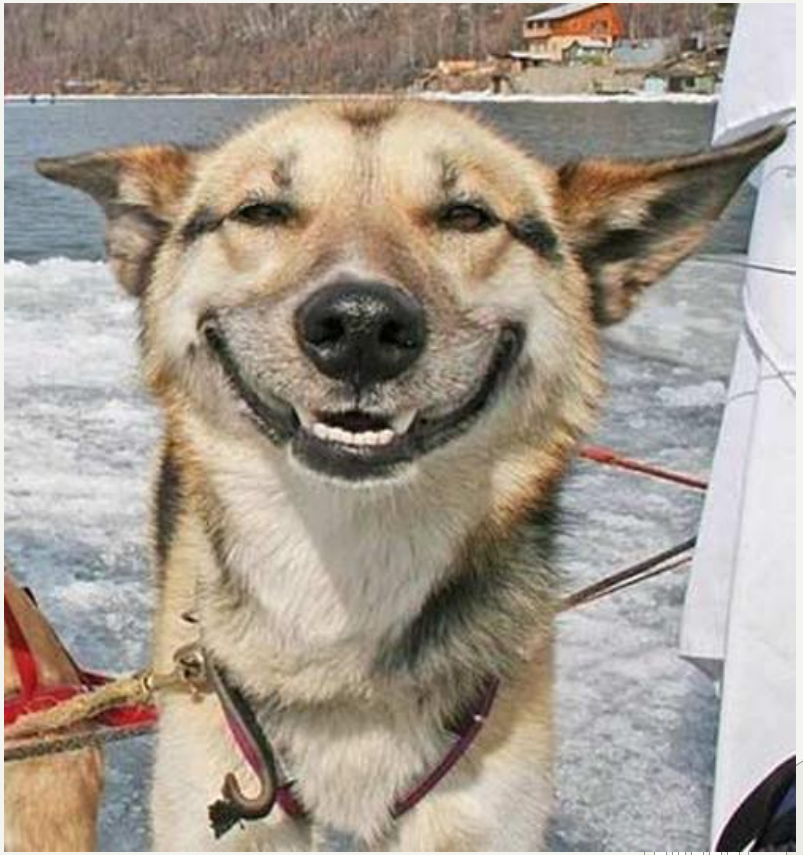
```python
import pytest
from metrics._service import ServiceCountBaseTrigger

@pytest.fixture
def trigger(self):
    return type(
        "TestTrigger",
        (ServiceCountBaseTrigger,),
        {
            "metric_name": "test",
            "owners": ["foo@yelp.com"]
        },
    )()
```

let’s make things better

before decorators



after decorators



logs → statmonster → metrics

what is a trigger?

what is a trigger?

it’s a class that...

what is a trigger?

it's a class that...

let users utilize **all python features**
instead of
just inheritance

logs → statmonster → metrics

```
from enum import Enum
from collections import namedtuple
from functools import partial

MetricType = Enum("MetricType", ("COUNTER", "TIMER"))

Metric = namedtuple(
    "Metric",
    ("name", "ts", "value", "dims", "type")
)

Counter = partial(Metric, type=MetricType.COUNTER)
Timer = partial(Metric, type=MetricType.TIMER)
```

logs → statmonster → metrics

```
class Log:
    def __init__(self, name, decoder=decode_json):
        self.name = name
        self.decoder = decoder
        self.fns = set()

    def decode(self, line):
        return self.decoder(line)
```

```
from statmonster import Log

events = Log("events")
rusage = Log("rusage")
suggestions_service = Log("suggest")
home = Log("homepage")
```

```
from statmonster import Log

def decode_text(line):
    time, request = line.split()
    return {'time': time, 'request': request}

requests = Log("tmp_requests", decoder=decode_text)
```

logs → statmonster → metrics

```
from emails import send_email

class Log:
    def process(self, line):
        entry = self.decode(line)
        for fn in self.fns:
            try:
                yield from fn(entry)
            except Exception as e:
                send_email(fn.owners, e)
```

```
from .logs import events
from statmonster import owners, register, Counter

@events.register
@owners("antonio@yelp.com")
def count_events(entry):
    yield Counter(
        "events",
        entry["time"],
        1,
        {"type": entry["type"]}
)
```

```
from emails import send_email

class Log:
    def register(self, fn):
        self.fns.add(fn)
        return fn
```

```python
class Flask(_PackageBoundObject):
    def route(self, rule, **options):
        def decorator(f):
            endpoint = options.pop('endpoint', None)
            self.add_url_rule(rule, endpoint, f, **options)
            return f
        return decorator
```

```
def register(*logs):
    def decorator(fn):
        for log in logs:
            log.register(fn)
        return fn

    return decorator
```

```
def owners(*handlers):
    def decorator(fn):
        fn.owners = handlers
        return fn

    return decorator
```

```
from .logs import rusage, aux_rusage
from statmonster import owners, Timer

@register(rusage, aux_rusage)
@owners("antonio@yelp.com", "metrics@yelp.com")
def time_cpu(entry):
    for metric in ("stime", "utime"):
        yield Timer(
            f"cpu.{metric}",
            entry["start_time"],
            entry[metric],
            {}
        )
```

how do I inherit from a base log class?

```
from statmonster import Log
from utils import decode_apache

ApacheLog = partial(Log, decoder=decode_apache)
access = ApacheLog("access")
admin_access = ApacheLog("admin_access")
```

how do I inherit from a base trigger class?

```
from statmonster import Counter

def emit_endpoints_timings(
  metric_name,
  endpoints,
  additional_dims,
  entry
):
  ts = entry['start_time']
  for endpoint in endpoints:
    if endpoint == entry["endpoint"]:
      timing = entry["endpoint"]["time"]
      dims = {'endpoint': endpoint}.update(additional_dims)
      yield Counter(metric_name, ts, timing, dims)
```

```
from .logs import home
from .endpoints import emit_endpoints_timings
from statmonster import owners, register

ENDPOINTS = ["best_of_yelp", "suggestions", "nearby"]

@register(home)
@owners("consumer@yelp.com")
def time_home_endpoints(entry):
    dims = {
        "mode": entry.get("mode", "sync"),
        "user": "loggedin" if entry["username"] else "anon",
    }
    yield from emit_endpoints_timings(
        "home",
        ENDPOINTS,
        dims,
        entry
    )
```

```
from .logs import home
from .endpoints import emit_endpoints_timings
from statmonster import owners, register

ENDPOINTS = ["best_of_yelp", "suggestions", "nearby"]

@register(home)
@owners("consumer@yelp.com")
def time_home_endpoints(entry):
    dims = {
        "mode": entry.get("mode", "sync"),
        "user": "loggedin" if entry["username"] else "anon",
    }
    yield from emit_endpoints_timings(
        "home",
        ENDPOINTS,
        dims,
        entry
    )
```

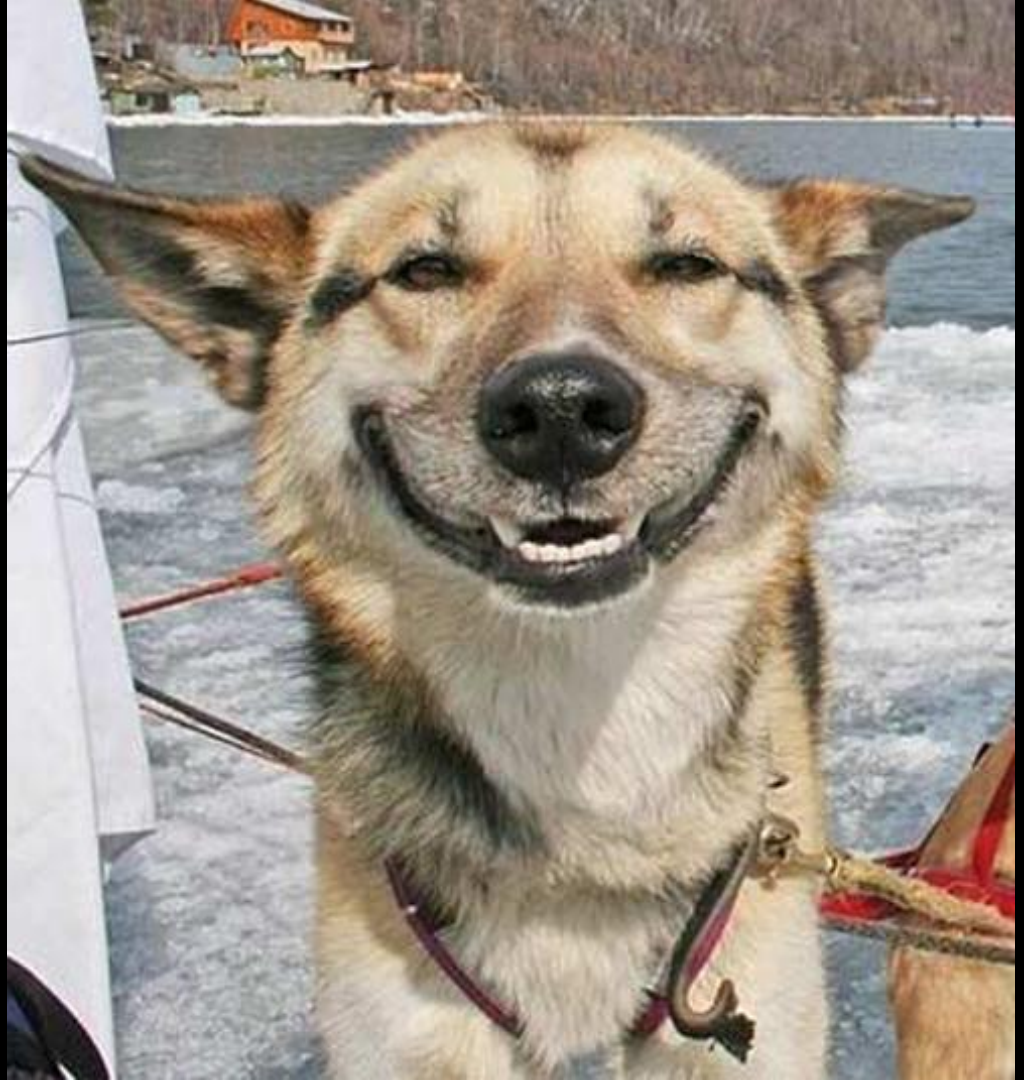
how do I inherit from **two** trigger classes?

```
def make_key(metric_name, stat_name):
    return f"{metric_name}.{stat_name}"

def emit_service_timings(prefix, entry):
    key = make_key(prefix, "requestLatency")
    dimensions = {"method": entry["method_name"]}

    yield Timer(
        key,
        entry["time"],
        entry["time_elapsed"],
        dimensions
    )
```

```
def emit_service_counters(prefix, entry):
  try:
    bytes_written = int(entry["headers"]["Content-Length"])
  except KeyError:
    bytes_written = 0
    dimensions = {"method": entry["method_name"]}
    request_count_key = make_key(prefix, "request_count")
    bytes_written_key = make_key(prefix, "total_written_bytes")
    ts = entry["time"]

    yield Counter(request_count_key, ts, 1, dimensions)
    yield Counter(
      bytes_written_key,
      ts,
      bytes_written,
      dimensions
    )
```

```python
from statmonster import owners
from metrics.logs import suggestions_service
from metrics.service import emit_service_timings
from metrics.service import emit_service_counters

@suggestions_service.register
@owners("search@yelp.com")
def emit_suggestions_metrics(entry):
    yield from emit_service_timings("suggestion_timings", entry)
    yield from emit_service_counters("suggestion_count", entry)
```

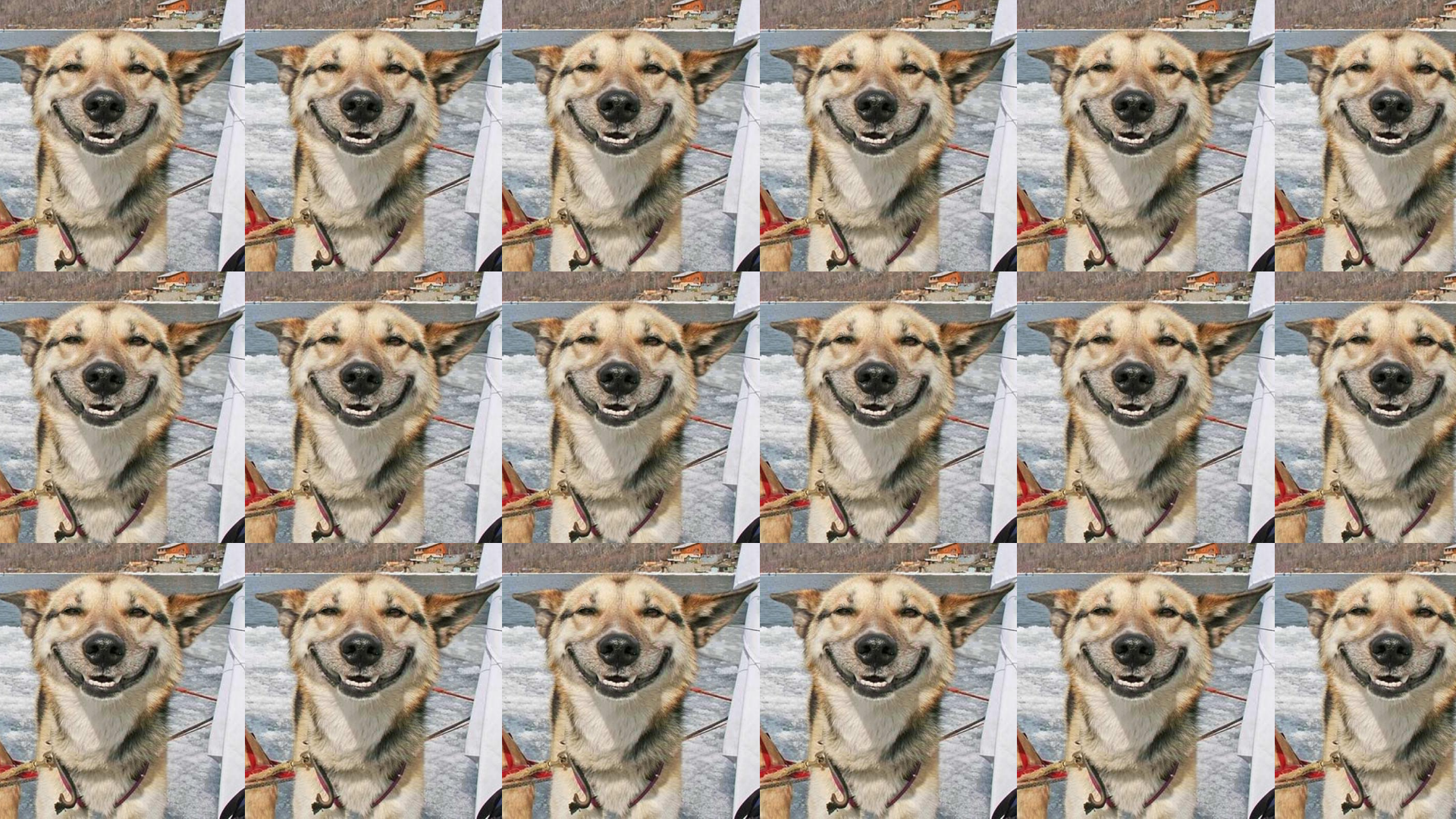
how do I test my trigger class?



```python
from metrics.rusage import time_cpu
from statmonster import Timer

def test_time_cpu():
    entry = {"start_time": 1000, "stime": 42, "utime": 33}
    assert Timer(
        "cpu.stime",
        1000,
        42,
        {}
    ) in list(time_cpu(entry))
```

```
from metrics.rusage import time_cpu
from statmonster import Timer

def test_time_cpu():
    entry = {"start_time": 1000, "stime": 42, "utime": 33}
    assert Timer(
        "cpu.stime",
        1000,
        42,
        {}
    ) in list(time_cpu(entry))
```

but how do I test my **base** trigger class?



```
from metrics.service import emit_service_counters

def test_emit_service_counters():
    entry = {
        "time": 123456,
        "status": 200,
        "method_name": "GET",
        "blob": "foo",
    }
    expected = Counter(
        "test.request_count",
        123456,
        1,
        {"method": "GET"}
    )
    assert expected in list(emit_service_counters("test", entry))
```

```
from metrics.service import emit_service_counters

def test_emit_service_counters():
    entry = {
        "time": 123456,
        "status": 200,
        "method_name": "GET",
        "blob": "foo",
    }
    expected = Counter(
        "test.request_count",
        123456,
        1,
        {"method": "GET"}
    )
    assert expected in list(emit_service_counters("test", entry))
```

but how do I make sure
all functions have owners?

```
from collect import collect

def test_owners():
    for name, log in collect():
        for fn in log.fns:
            assert hasattr(fn, "owners")
```

all of this only for users’ sake?

import system

```python
from importlib import import_module
import glob
import os
import inspect
from statmonster import Log

def collect():
    for file_path in glob.iglob(os.path.join("metrics", "*.py")):
        module_name = '.'.join(file_path[:-3].split('/'))
        import_module(module_name)

    module = import_module("metrics.logs")
    for name, obj in inspect.getmembers(module):
        if isinstance(obj, Log):
            yield name, obj

if __name__ == "__main__":
    for name, log in collect():
        print(f"{name}:{log}")
```

```python
from importlib import import_module
import glob
import os
import inspect
from statmonster import Log, Trigger


DEFAULT_DIR = "metrics"


def collect():
    all_triggers = {}
    for module_name, module in get_triggers_modules():
        log, triggers = get_log_and_triggers(module_name, module)
        if log:
            all_triggers[log] = triggers
    return all_triggers


def get_log_and_triggers(module_name, module):
    log = None
    triggers = []
    for name, cls in inspect.getmembers(module, inspect.isclass):
        if is_valid_log(module_name, cls):
            assert not log, "Multiple logs in the same module: %s" % module
            log = cls()
        elif is_valid_trigger(module_name, cls):
            triggers.append(cls())
    return log, set(triggers)


def is_valid_log(module_name, log):
    return (issubclass(log, Log) and
            log.__module__ == f"{DEFAULT_DIR}.{module_name}")


def is_valid_trigger(module_name, trigger):
    return (issubclass(trigger, Trigger) and
            trigger.__module__ == "%s.%s" % (DEFAULT_DIR, module_name))


def get_triggers_modules(triggers_folder=DEFAULT_DIR):
    for file_path in glob.iglob(os.path.join(triggers_folder, "*.py")):
        name = os.path.basename(file_path)[:-3]
        # Filter eg. _service and __init__
        if not name.startswith('_'):
            yield name, dynamic_import_module(file_path)


def dynamic_import_module(module_path):
    module_name = '.'.join(module_path[:-3].split('/'))
    return import_module(module_name)


if __name__ == "__main__":
    for log, triggers in collect().items():
        print(f"{log}:{triggers}")
```

let users utilize **all python features**
instead of
just inheritance

go for **decorators**

go for **decorators**
when your **classes**

go for **decorators**
when your **classes**
have only one method

go for **decorators**
when your **classes**
have only one method
and
are instantiated only once


We're Hiring!
www.yelp.com/careers/

fb.com/YelpEngineers

@YelpEngineering

engineeringblog.yelp.com

github.com/yelp

QUESTIONS?

backup slides

some syntax

```
yield from iterable

# shortened form of

for item in iterable:
    yield item
```

```python
from functools import partial

# convert base 2 string to an int
basetwo = partial(int, base=2)
assert basetwo('10010') == 18
```

```
from enum import Enum

Color = Enum("Color", ("RED", "GREEN", "BLUE"))

>>> print(repr(Color.RED))
<Color.RED: 1>
```

putting all together

```python
def decorator(arg1, arg2):
    def actual_decorator(fn):
        @functools.wraps(fn)  # preserve function's metadata
        def wrapper(*args, **kwargs):
            print("Before the function runs")
            return fn(*args, **kwargs)
        return wrapper

    print(f"Decorating function with {arg1} and {arg2}")
    return actual_decorator

@decorator("bar", "baz")
def func(arg3, arg4):
    pass
```