Pykka

Release 4.0.2

Stein Magnus Jodal and contributors

Feb 18, 2024

USAGE

Pykka is a Python implementation of the actor model. The actor model introduces some simple rules to control the sharing of state and cooperation between execution units, which makes it easier to build concurrent applications.

For details and code examples, see the Pykka documentation.

Pykka is available from PyPI. To install it, run:

```
pip install pykka
```

Pykka works with Python 3.8 or newer.

CHAPTER

ONE

INSPIRATION

Much of the naming of concepts and methods in Pykka is taken from the Akka project which implements actors on the JVM. Though, Pykka does not aim to be a Python port of Akka, and supports far fewer features.

Notably, Pykka **does not** support the following features:

- Supervision: Linking actors, supervisors, or supervisor groups.
- Remoting: Communicating with actors running on other hosts.
- Routers: Pykka does not come with a set of predefined message routers, though you may make your own actors for routing messages.

CHAPTER

TWO

PROJECT RESOURCES

- Documentation
- Source code
- Releases
- Issue tracker
- Contributors
- Users

2.1 Quickstart

Pykka is a Python implementation of the actor model. The actor model introduces some simple rules to control the sharing of state and cooperation between execution units, which makes it easier to build concurrent applications.

2.1.1 Rules of the actor model

- An actor is an execution unit that executes concurrently with other actors.
- An actor does not share state with anybody else, but it can have its own state.
- An actor can only communicate with other actors by sending and receiving messages. It can only send messages to actors whose address it has.
- When an actor receives a message it may take actions like:
  - altering its own state, e.g. so that it can react differently to a future message,
  - sending messages to other actors, or
  - starting new actors.

  None of the actions are required, and they may be applied in any order.
- An actor only processes one message at a time. In other words, a single actor does not give you any concurrency, and it does not need to use locks internally to protect its own state.

2.1.2 The actor implementations

Pykka's actor API comes with the following implementations:

- Threads: Each *ThreadingActor* is executed by a regular thread, i.e. `threading.Thread`. As handles for future results, it uses *ThreadingFuture* which is a thin wrapper around a `queue.Queue`. It has no dependencies outside Python itself. *ThreadingActor* plays well together with non-actor threads.

Pykka 2 and earlier shipped with some alternative implementations that were removed in Pykka 3:

- gevent: Each actor was executed by a gevent greenlet.
- Eventlet: Each actor was executed by an Eventlet greenlet.

2.1.3 A basic actor

In its most basic form, a Pykka actor is a class with an *on_receive()* method:

```
import pykka

class Greeter(pykka.ThreadingActor):
    def on_receive(self, message):
        print('Hi there!')
```

To start an actor, you call the class' method *start()*, which starts the actor and returns an actor reference which can be used to communicate with the running actor:

```
actor_ref = Greeter.start()
```

If you need to pass arguments to the actor upon creation, you can pass them to the *start()* method, and receive them using the regular `__init__()` method:

```
import pykka

class Greeter(pykka.ThreadingActor):
    def __init__(self, greeting='Hi there!'):
        super().__init__()
        self.greeting = greeting

    def on_receive(self, message):
        print(self.greeting)

actor_ref = Greeter.start(greeting='Hi you!')
```

It can be useful to know that the init method is run in the execution context that starts the actor. There are also hooks for running code in the actor's own execution context when the actor starts, when it stops, and when an unhandled exception is raised. Check out the full API docs for the details.

To stop an actor, you can either call *stop()* on the *ActorRef*:

```
actor_ref.stop()
```

Or, if an actor wants to stop itself, it can simply do so:

```
self.stop()
```

Once an actor has been stopped, it cannot be restarted.

Sending messages

To send a message to the actor, you can either use the *tell()* method or the *ask()* method on the `actor_ref` object. *tell()* will fire off a message without waiting for an answer. In other words, it will never block. *ask()* will by default block until an answer is returned, potentially forever. If you provide a `timeout` keyword argument to *ask()*, you can specify for how long it should wait for an answer. If you want an answer, but don't need it right away because you have other stuff you can do first, you can pass `block=False`, and *ask()* will immediately return a "future" object.

The message itself can be of any type, for example a dict or your own message class type.

Summarized in code:

```
actor_ref.tell('Hi!')
# => Returns nothing. Will never block.

answer = actor_ref.ask('Hi?')
# => May block forever waiting for an answer

answer = actor_ref.ask('Hi?', timeout=3)
# => May wait 3s for an answer, then raises exception if no answer.

future = actor_ref.ask('Hi?', block=False)
# => Will return a future object immediately.
answer = future.get()
# => May block forever waiting for an answer
answer = future.get(timeout=0.1)
# => May wait 0.1s for an answer, then raises exception if no answer.
```

Warning: For performance reasons, Pykka **does not** clone the message you send before delivering it to the receiver. You are yourself responsible for either using immutable data structures or to `copy.deepcopy()` the data you're sending off to other actors.

Replying to messages

If a message is sent using `actor_ref.ask()` you can reply to the sender of the message by simply returning a value from the *on_receive()* method:

```
import pykka

class Greeter(pykka.ThreadingActor):
    def on_receive(self, message):
        return 'Hi there!'

actor_ref = Greeter.start()

answer = actor_ref.ask('Hi?')
print(answer)
# => 'Hi there!'
```

`None` is a valid response so if you return `None` explicitly, or don't return at all, a response containing `None` will be returned to the sender.

From the point of view of the actor it doesn't matter whether the message was sent using *tell()* or *ask()*. When the sender doesn't expect a response the *on_receive()* return value will be ignored.

The situation is similar in regard to exceptions: when *ask()* is used and you raise an exception from within *on_receive()* method, the exception will propagate to the sender:

```
import pykka

class Raiser(pykka.ThreadingActor):
    def on_receive(self, message):
        raise Exception('Oops')

actor_ref = Raiser.start()

try:
    actor_ref.ask('How are you?')
except Exception as e:
    print(repr(e))
    # => Exception('Oops')
```

2.1.4 Actor proxies

With the basic building blocks provided by actors and futures, we got everything we need to build more advanced abstractions. Pykka provides a single abstraction on top of the basic actor model, named "actor proxies". You can use Pykka without proxies, but we've found it to be a very convenient abstraction when building Mopidy.

Let's create an actor and start it:

```
import pykka

class Calculator(pykka.ThreadingActor):
    def __init__(self):
        super().__init__()
        self.last_result = None

    def add(self, a, b=None):
        if b is not None:
            self.last_result = a + b
        else:
            self.last_result += a
        return self.last_result

    def sub(self, a, b=None):
        if b is not None:
            self.last_result = a - b
        else:
            self.last_result -= a
        return self.last_result

actor_ref = Calculator.start()
```

You can create a proxy from any reference to a running actor:

```
proxy = actor_ref.proxy()
```

The proxy object will use introspection to figure out what public attributes and methods the actor has, and then mirror the full API of the actor. Any attribute or method prefixed with underscore will be ignored, which is the convention for keeping stuff private in Python.

When we access attributes or call methods on the proxy, it will ask the actor to access the given attribute or call the given method, and return the result to us. All results are wrapped in "future" objects, so you must use the *get()* method to get the actual data:

```
future = proxy.add(1, 3)
future.get()
# => 4

proxy.last_result.get()
# => 4
```

Since an actor only processes one message at the time and all messages are kept in order, you don't need to add the call to *get()* just to block processing until the actor has completed processing your last message:

```
proxy.sub(5)
proxy.add(3)
proxy.last_result.get()
# => 2
```

Since assignment doesn't return anything, it works just like on regular objects:

```
proxy.last_result = 17
proxy.last_result.get()
# => 17
```

Under the hood, the proxy does everything by sending messages to the actor using the regular *ask()* method we talked about previously. By doing so, it maintains the actor model restrictions. The only "magic" happening here is some basic introspection and automatic building of three different message types; one for method calls, one for attribute reads, and one for attribute writes.

Traversable attributes on proxies

Sometimes you'll want to access an actor attribute's methods or attributes through a proxy. For this case, Pykka supports "traversable attributes". By marking an actor attribute as traversable, Pykka will not return the attribute when accessed, but wrap it in a new proxy which is returned instead.

To mark an attribute as traversable, simply mark it with the *traversable()* function:

```
import pykka

class AnActor(pykka.ThreadingActor):
    playback = pykka.traversable(Playback())

class Playback(object):
    def play(self):
        return True
```

(continues on next page)

(continued from previous page)

```
proxy = AnActor.start().proxy()
play_success = proxy.playback.play().get()
```

You can access methods and attributes nested as deep as you like, as long as all attributes on the path between the actor and the method or attribute on the end are marked as traversable.

2.2 Examples

The `examples/` dir in Pykka's Git repo includes some runnable examples of Pykka usage.

2.2.1 Plain actor

```
#!/usr/bin/env python3

import pykka

GetMessages = object()


class PlainActor(pykka.ThreadingActor):
    def __init__(self):
        super().__init__()
        self.stored_messages = []

    def on_receive(self, message):
        if message is GetMessages:
            return self.stored_messages
        self.stored_messages.append(message)
        return None


if __name__ == "__main__":
    actor = PlainActor.start()
    actor.tell({"no": "Norway", "se": "Sweden"})
    actor.tell({"a": 3, "b": 4, "c": 5})
    print(actor.ask(GetMessages))
    actor.stop()
```

Output:

```
[{'no': 'Norway', 'se': 'Sweden'}, {'a': 3, 'b': 4, 'c': 5}]
```

2.2.2 Actor with proxy

```
#!/usr/bin/env python3

import threading
import time

import pykka


class AnActor(pykka.ThreadingActor):
    field = "this is the value of AnActor.field"

    def proc(self):
        log("this was printed by AnActor.proc()")

    def func(self):
        time.sleep(0.5)  # Block a bit to make it realistic
        return "this was returned by AnActor.func() after a delay"


def log(msg):
    thread_name = threading.current_thread().name
    print(f"{thread_name}: {msg}")


if __name__ == "__main__":
    actor = AnActor.start().proxy()
    for _ in range(3):
        # Method with side effect
        log("calling AnActor.proc() ...")
        actor.proc()

        # Method with return value
        log("calling AnActor.func() ...")
        result = actor.func()  # Does not block, returns a future
        log("printing result ... (blocking)")
        log(result.get())  # Blocks until ready

        # Field reading
        log("reading AnActor.field ...")
        result = actor.field  # Does not block, returns a future
        log("printing result ... (blocking)")
        log(result.get())  # Blocks until ready

        # Field writing
        log("writing AnActor.field ...")
        actor.field = "new value"  # Assignment does not block
        result = actor.field  # Does not block, returns a future
        log("printing new field value ... (blocking)")
        log(result.get())  # Blocks until ready
    actor.stop()
```

Output:

```
MainThread: calling AnActor.proc() ...
MainThread: calling AnActor.func() ...
MainThread: printing result ... (blocking)
AnActor-1: this was printed by AnActor.proc()
MainThread: this was returned by AnActor.func() after a delay
MainThread: reading AnActor.field ...
MainThread: printing result ... (blocking)
MainThread: this is the value of AnActor.field
MainThread: writing AnActor.field ...
MainThread: printing new field value ... (blocking)
MainThread: new value
MainThread: calling AnActor.proc() ...
MainThread: calling AnActor.func() ...
MainThread: printing result ... (blocking)
AnActor-1: this was printed by AnActor.proc()
MainThread: this was returned by AnActor.func() after a delay
MainThread: reading AnActor.field ...
MainThread: printing result ... (blocking)
MainThread: new value
MainThread: writing AnActor.field ...
MainThread: printing new field value ... (blocking)
MainThread: new value
MainThread: calling AnActor.proc() ...
MainThread: calling AnActor.func() ...
AnActor-1: this was printed by AnActor.proc()
MainThread: printing result ... (blocking)
MainThread: this was returned by AnActor.func() after a delay
MainThread: reading AnActor.field ...
MainThread: printing result ... (blocking)
MainThread: new value
MainThread: writing AnActor.field ...
MainThread: printing new field value ... (blocking)
MainThread: new value
```

2.2.3 Multiple cooperating actors

```
#!/usr/bin/env python3

import pykka


class Adder(pykka.ThreadingActor):
    def add_one(self, i):
        print(f"{self} is increasing {i}")
        return i + 1


class Bookkeeper(pykka.ThreadingActor):
    def __init__(self, adder):
        super().__init__()
        self.adder = adder
```

(continues on next page)

(continued from previous page)

```
    def count_to(self, target):
        i = 0
        while i < target:
            i = self.adder.add_one(i).get()
            print(f"{self} got {i} back")


if __name__ == "__main__":
    adder = Adder.start().proxy()
    bookkeeper = Bookkeeper.start(adder).proxy()
    bookkeeper.count_to(10).get()
    pykka.ActorRegistry.stop_all()
```

Output:

```
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 0
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 1 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 1
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 2 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 2
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 3 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 3
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 4 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 4
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 5 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 5
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 6 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 6
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 7 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 7
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 8 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 8
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 9 back
Adder (urn:uuid:f50029eb-7cea-4ab9-98bf-a5bf65af8b8f) is increasing 9
Bookkeeper (urn:uuid:4f2d4e78-7a33-4c4f-86ac-7c415a7205f4) got 10 back
```

2.2.4 Pool of actors sharing work

```
#!/usr/bin/env python3

"""Resolve a bunch of IP addresses using a pool of resolver actors.

Based on example contributed by Kristian Klette <klette@klette.us>.

Either run without arguments:

    ./resolver.py

Or specify pool size and IPs to resolve:
```

(continues on next page)

(continued from previous page)

```
    ./resolver.py 3 193.35.52.{1,2,3,4,5,6,7,8,9}
"""

import pprint
import socket
import sys

import pykka


class Resolver(pykka.ThreadingActor):
    def resolve(self, ip):
        try:
            info = socket.gethostbyaddr(ip)
            print(f"Finished resolving {ip}")
            return info[0]
        except Exception:
            print(f"Failed resolving {ip}")
            return None


def run(pool_size, *ips):
    # Start resolvers
    resolvers = [Resolver.start().proxy() for _ in range(pool_size)]

    # Distribute work by mapping IPs to resolvers (not blocking)
    hosts = []
    for i, ip in enumerate(ips):
        hosts.append(resolvers[i % len(resolvers)].resolve(ip))

    # Gather results (blocking)
    ip_to_host = zip(ips, pykka.get_all(hosts))
    pprint.pprint(list(ip_to_host))

    # Clean up
    pykka.ActorRegistry.stop_all()


if __name__ == "__main__":
    if len(sys.argv[1:]) >= 2:
        run(int(sys.argv[1]), *sys.argv[2:])
    else:
        ips = [f"193.35.52.{i}" for i in range(1, 50)]
        run(10, *ips)
```

2.2.5 Mopidy music server

Pykka was originally created back in 2011 as a formalization of concurrency patterns that emerged in the Mopidy music server. The original Pykka source code wasn't extracted from Mopidy, but it built and improved on the concepts from Mopidy. Mopidy was later ported to build on Pykka instead of its own concurrency abstractions.

Mopidy still use Pykka extensively to keep independent parts, like the MPD and HTTP frontend servers or the Spotify and Google Music integrations, running independently. Every one of Mopidy's more than 100 extensions has at least one Pykka actor. By running each extension as an independent actor, errors and bugs in one extension is attempted isolated, to reduce the effect on the rest of the system.

You can browse the Mopidy source code to find many real life examples of Pykka usage.

2.3 Runtimes

By default, Pykka builds on top of Python's regular threading concurrency model, via the standard library modules `threading` and `queue`.

Pykka 2 and earlier shipped with some alternative implementations that ran on top of `gevent` or `eventlet`. These alternative implementations were removed in Pykka 3.

Note that Pykka does no attempt at supporting a mix of concurrency runtimes. Such a future feature has briefly been discussed in issue #11.

2.3.1 Threading

Installation

The default threading runtime has no dependencies other than Pykka itself and the Python standard library.

API

class pykka.**ThreadingFuture**

Implementation of *Future* for use with regular Python threads`.

The future is implemented using a `queue.Queue`.

The future does *not* make a copy of the object which is *set()* on it. It is the setters responsibility to only pass immutable objects or make a copy of the object before setting it on the future.

Changed in version 0.14: Previously, the encapsulated value was a copy made with `copy.deepcopy()`, unless the encapsulated value was a future, in which case the original future was encapsulated.

get(*, *timeout: float | None = None*) → Any

Get the value encapsulated by the future.

If the encapsulated value is an exception, it is raised instead of returned.

If `timeout` is `None`, as default, the method will block until it gets a reply, potentially forever. If `timeout` is an integer or float, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

The encapsulated value can be retrieved multiple times. The future will only block the first time the value is accessed.

Parameters

timeout (float or `None`) – seconds to wait before timeout

Raise
pykka.Timeout if timeout is reached

Raise
encapsulated value if it is an exception

Returns
encapsulated value if it is not an exception

`set`(*value: Any | None = None*) → None

Set the encapsulated value.

Parameters
value (any object or `None`) – the encapsulated value or nothing

Raise
an exception if set is called multiple times

`set_exception`(*exc_info: OptExcInfo | None = None*) → None

Set an exception as the encapsulated value.

You can pass an `exc_info` three-tuple, as returned by `sys.exc_info()`. If you don't pass `exc_info`, `sys.exc_info()` will be called and the value returned by it used.

In other words, if you're calling *set_exception()*, without any arguments, from an except block, the exception you're currently handling will automatically be set on the future.

Parameters
exc_info (*three-tuple of (exc_class, exc_instance, traceback)*) – the encapsulated exception

`class` pykka.`ThreadingActor`(**_args: Any*, ***_kwargs: Any*)

Implementation of *Actor* using regular Python threads.

`use_daemon_thread: ClassVar[bool] = False`

A boolean value indicating whether this actor is executed on a thread that is a daemon thread (`True`) or not (`False`). This must be set before *pykka.Actor.start()* is called, otherwise `RuntimeError` is raised.

The entire Python program exits when no alive non-daemon threads are left. This means that an actor running on a daemon thread may be interrupted at any time, and there is no guarantee that cleanup will be done or that *pykka.Actor.on_stop()* will be called.

Actors do not inherit the daemon flag from the actor that made it. It always has to be set explicitly for the actor to run on a daemonic thread.

2.4 Testing

Pykka actors can be tested using the regular Python testing tools like pytest, `unittest`, and `unittest.mock`.

To test actors in a setting as close to production as possible, a typical pattern is the following:

1. In the test setup, start an actor together with any actors/collaborators it depends on. The dependencies will often be replaced by mocks to control their behavior.
2. In the test, *ask()* or *tell()* the actor something.
3. In the test, assert on the actor's state or the return value from the *ask()*.
4. In the test teardown, stop the actor to properly clean up before the next test.

2.4.1 An example

Let's look at an example actor that we want to test:

```
import pykka


class ProducerActor(pykka.ThreadingActor):
    def __init__(self, consumer):
        super().__init__()
        self.consumer = consumer

    def produce(self):
        new_item = {"item": 1, "new": True}
        self.consumer.consume(new_item)
```

We can test this actor with pytest by mocking the consumer and asserting that it receives a newly produced item:

```
import pytest
from producer import ProducerActor


@pytest.fixture()
def consumer_mock(mocker):
    return mocker.Mock()


@pytest.fixture()
def producer(consumer_mock):
    # Step 1: The actor under test is wired up with
    # its dependencies and is started.
    proxy = ProducerActor.start(consumer_mock).proxy()

    yield proxy

    # Step 4: The actor is stopped to clean up before the next test.
    proxy.stop()


def test_producer_actor(consumer_mock, producer):
    # Step 2: Interact with the actor.
    # We call .get() on the last future returned by the actor to wait
    # for the actor to process all messages before asserting anything.
    producer.produce().get()

    # Step 3: Assert that the return values or actor state is as expected.
    consumer_mock.consume.assert_called_once_with({"item": 1, "new": True})
```

If this way of setting up and tearing down test resources is unfamiliar to you, it is strongly recommended to read up on pytest's great fixture feature.

2.5 Module

pykka.**__version__**

Pykka's **PEP 386** and **PEP 396** compatible version number

2.6 Actors

class pykka.**Actor**(**_args: Any*, ***_kwargs: Any*)

An actor is an execution unit that executes concurrently with other actors.

To create an actor:

1. subclass one of the *Actor* implementations:
   - *ThreadingActor*
2. implement your methods, including `__init__()`, as usual,
3. call *Actor.start()* on your actor class, passing the method any arguments for your constructor.

To stop an actor, call *Actor.stop()* or *ActorRef.stop()*.

For example:

```
import pykka

class MyActor(pykka.ThreadingActor):
    def __init__(self, my_arg=None):
        super().__init__()
        ... # My optional init code with access to start() arguments

    def on_start(self):
        ... # My optional setup code in same context as on_receive()

    def on_stop(self):
        ... # My optional cleanup code in same context as on_receive()

    def on_failure(self, exception_type, exception_value, traceback):
        ... # My optional cleanup code in same context as on_receive()

    def on_receive(self, message):
        ... # My optional message handling code for a plain actor

    def a_method(self, ...):
        ... # My regular method to be used through an ActorProxy

my_actor_ref = MyActor.start(my_arg=...)
my_actor_ref.stop()
```

classmethod start(**args: Any*, ***kwargs: Any*) → *ActorRef*[A]

Start an actor.

Starting an actor also registers it in the *ActorRegistry*.

Any arguments passed to *start()* will be passed on to the class constructor.

Behind the scenes, the following is happening when you call *start()*:

1. The actor is created:
    1. *actor_urn* is initialized with the assigned URN.
    2. *actor_inbox* is initialized with a new actor inbox.
    3. *actor_ref* is initialized with a *pykka.ActorRef* object for safely communicating with the actor.
    4. At this point, your `__init__()` code can run.
2. The actor is registered in *pykka.ActorRegistry*.
3. The actor receive loop is started by the actor's associated thread/greenlet.

Returns
a *ActorRef* which can be used to access the actor in a safe manner

property actor_ref: *ActorRef*[A]

The actor's *ActorRef* instance.

actor_urn: str

The actor URN string is a universally unique identifier for the actor. It may be used for looking up a specific actor using *ActorRegistry.get_by_urn()*.

actor_inbox: ActorInbox

The actor's inbox. Use *ActorRef.tell()*, *ActorRef.ask()*, and friends to put messages in the inbox.

actor_stopped: Event

A `threading.Event` representing whether or not the actor should continue processing messages. Use *stop()* to change it.

stop() → None

Stop the actor.

It's equivalent to calling *ActorRef.stop()* with `block=False`.

on_start() → None

Run code at the beginning of the actor's life.

Hook for doing any setup that should be done *after* the actor is started, but *before* it starts processing messages.

For *ThreadingActor*, this method is executed in the actor's own thread, while `__init__()` is executed in the thread that created the actor.

If an exception is raised by this method the stack trace will be logged, and the actor will stop.

on_stop() → None

Run code at the end of the actor's life.

Hook for doing any cleanup that should be done *after* the actor has processed the last message, and *before* the actor stops.

This hook is *not* called when the actor stops because of an unhandled exception. In that case, the *on_failure()* hook is called instead.

For *ThreadingActor* this method is executed in the actor's own thread, immediately before the thread exits.

If an exception is raised by this method the stack trace will be logged, and the actor will stop.

on_failure(*exception_type: type[BaseException] | None*, *exception_value: BaseException | None*, *traceback: TracebackType | None*) → None

Run code when an unhandled exception is raised.

Hook for doing any cleanup *after* an unhandled exception is raised, and *before* the actor stops.

For *ThreadingActor* this method is executed in the actor's own thread, immediately before the thread exits.

The method's arguments are the relevant information from `sys.exc_info()`.

If an exception is raised by this method the stack trace will be logged, and the actor will stop.

on_receive(*message: Any*) → Any

May be implemented for the actor to handle regular non-proxy messages.

Parameters
message (*any*) – the message to handle

Returns
anything that should be sent as a reply to the sender

class pykka.**ActorRef**(*actor: A*)

Reference to a running actor which may safely be passed around.

ActorRef instances are returned by *Actor.start()* and the lookup methods in *ActorRegistry*. You should never need to create *ActorRef* instances yourself.

Parameters
actor (*Actor*) – the actor to wrap

actor_class: `type[A]`

The class of the referenced actor.

actor_urn: `str`

See *Actor.actor_urn*.

actor_inbox: **ActorInbox**

See *Actor.actor_inbox*.

actor_stopped: **Event**

See *Actor.actor_stopped*.

is_alive() → bool

Check if actor is alive.

This is based on the actor's stopped flag. The actor is not guaranteed to be alive and responding even though *is_alive()* returns `True`.

Returns
Returns `True` if actor is alive, `False` otherwise.

tell(*message: Any*) → None

Send message to actor without waiting for any response.

Will generally not block, but if the underlying queue is full it will block until a free slot is available.

Parameters
message (*any*) – message to send

Raise
pykka.ActorDeadError if actor is not available

Returns
nothing

ask(*message: Any*, *, *block: Literal[False]*, *timeout: float | None = None*) → *Future*[Any]

ask(*message: Any*, *, *block: Literal[True]*, *timeout: float | None = None*) → Any

ask(*message: Any*, *, *block: bool = True*, *timeout: float | None = None*) → Any | *Future*[Any]

Send message to actor and wait for the reply.

The message can be of any type. If `block` is `False`, it will immediately return a *Future* instead of blocking.

If `block` is `True`, and `timeout` is `None`, as default, the method will block until it gets a reply, potentially forever. If `timeout` is an integer or float, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

Parameters

- **message** (*any*) – message to send
- **block** (*boolean*) – whether to block while waiting for a reply
- **timeout** (float or `None`) – seconds to wait before timeout if blocking

Raise
pykka.Timeout if timeout is reached if blocking

Raise
any exception returned by the receiving actor if blocking

Returns
pykka.Future, or response if blocking

stop(*, *block: Literal[True]*, *timeout: float | None = None*) → bool

stop(*, *block: Literal[False]*, *timeout: float | None = None*) → *Future*[bool]

stop(*, *block: bool = True*, *timeout: float | None = None*) → Any | *Future*[Any]

Send a message to the actor, asking it to stop.

Returns `True` if actor is stopped or was being stopped at the time of the call. `False` if actor was already dead. If `block` is `False`, it returns a future wrapping the result.

Messages sent to the actor before the actor is asked to stop will be processed normally before it stops.

Messages sent to the actor after the actor is asked to stop will be replied to with *pykka.ActorDeadError* after it stops.

The actor may not be restarted.

`block` and `timeout` works as for *ask()*.

Returns
pykka.Future, or a boolean result if blocking

proxy() → *ActorProxy*[A]

Wrap the *ActorRef* in an *ActorProxy*.

Using this method like this:

```
proxy = AnActor.start().proxy()
```

is analogous to:

```
proxy = ActorProxy(AnActor.start())
```

Raise
pykka.ActorDeadError if actor is not available

Returns
pykka.ActorProxy

2.7 Proxies

class pykka.**ActorProxy**(*, *actor_ref:* ActorRef*[A]*, *attr_path: AttrPath | None = None*)

An *ActorProxy* wraps an *ActorRef* instance.

The proxy allows the referenced actor to be used through regular method calls and field access.

You can create an *ActorProxy* from any *ActorRef*:

```
actor_ref = MyActor.start()
actor_proxy = ActorProxy(actor_ref)
```

You can also get an *ActorProxy* by using *proxy()*:

```
actor_proxy = MyActor.start().proxy()
```

Attributes and method calls

When reading an attribute or getting a return value from a method, you get a *Future* object back. To get the enclosed value from the future, you must call *get()* on the returned future:

```
print(actor_proxy.string_attribute.get())
print(actor_proxy.count().get() + 1)
```

If you call a method just for it's side effects and do not care about the return value, you do not need to accept the returned future or call *get()* on the future. Simply call the method, and it will be executed concurrently with your own code:

```
actor_proxy.method_with_side_effect()
```

If you want to block your own code from continuing while the other method is processing, you can use *get()* to block until it completes:

```
actor_proxy.method_with_side_effect().get()
```

You can also use the `await` keyword to block until the method completes:

```
await actor_proxy.method_with_side_effect()
```

If you access a proxied method as an attribute, without calling it, you get an *CallableProxy*.

Proxy to itself

An actor can use a proxy to itself to schedule work for itself. The scheduled work will only be done after the current message and all messages already in the inbox are processed.

For example, if an actor can split a time consuming task into multiple parts, and after completing each part can ask itself to start on the next part using proxied calls or messages to itself, it can react faster to other incoming messages as they will be interleaved with the parts of the time consuming task. This is especially useful for being able to stop the actor in the middle of a time consuming task.

To create a proxy to yourself, use the actor's *actor_ref* attribute:

```
proxy_to_myself_in_the_future = self.actor_ref.proxy()
```

If you create a proxy in your actor's constructor or *on_start* method, you can create a nice API for deferring work to yourself in the future:

```
def __init__(self):
    ...
    self._in_future = self.actor_ref.proxy()
    ...

def do_work(self):
    ...
    self._in_future.do_more_work()
    ...

def do_more_work(self):
    ...
```

To avoid infinite loops during proxy introspection, proxies to self should be kept as private instance attributes by prefixing the attribute name with _.

Examples

An example of *ActorProxy* usage:

```
#!/usr/bin/env python3

import pykka


class Adder(pykka.ThreadingActor):
    def add_one(self, i):
        print(f"{self} is increasing {i}")
        return i + 1


class Bookkeeper(pykka.ThreadingActor):
    def __init__(self, adder):
        super().__init__()
        self.adder = adder

    def count_to(self, target):
        i = 0
        while i < target:
            i = self.adder.add_one(i).get()
            print(f"{self} got {i} back")


if __name__ == "__main__":
    adder = Adder.start().proxy()
    bookkeeper = Bookkeeper.start(adder).proxy()
    bookkeeper.count_to(10).get()
    pykka.ActorRegistry.stop_all()
```

Parameters
actor_ref (*pykka.ActorRef*) – reference to the actor to proxy

Raise
pykka.ActorDeadError if actor is not available

actor_ref: *ActorRef*[A]

The actor's *pykka.ActorRef* instance.

class pykka.**CallableProxy**(*, *actor_ref:* ActorRef*[A]*, *attr_path: AttrPath*)

Proxy to a single method.

CallableProxy instances are returned when accessing methods on a *ActorProxy* without calling them.

Example:

```
proxy = AnActor.start().proxy()

# Ask semantics returns a future. See `__call__()` docs.
future = proxy.do_work()

# Tell semantics are fire and forget. See `defer()` docs.
proxy.do_work.defer()
```

__call__(**args: Any*, ***kwargs: Any*) → *Future*[Any]

Call with *ask()* semantics.

Returns a future which will yield the called method's return value.

If the call raises an exception is set on the future, and will be reraised by *get()*. If the future is left unused, the exception will not be reraised. Either way, the exception will also be logged. See *Logging* for details.

defer(**args: Any*, ***kwargs: Any*) → None

Call with *tell()* semantics.

Does not create or return a future.

If the call raises an exception, there is no future to set the exception on. Thus, the actor's *on_failure()* hook is called instead.

New in version 2.0.

pykka.**traversable**(*obj: T*) → T

Mark an actor attribute as traversable.

The traversable marker makes the actor attribute's own methods and attributes available to users of the actor through an *ActorProxy*.

Used as a function to mark a single attribute:

```
class AnActor(pykka.ThreadingActor):
    playback = pykka.traversable(Playback())

class Playback(object):
    def play(self):
        return True
```

This function can also be used as a class decorator, making all instances of the class traversable:

```
class AnActor(pykka.ThreadingActor):
    playback = Playback()

@pykka.traversable
class Playback(object):
    def play(self):
        return True
```

The third alternative, and the only way in Pykka < 2.0, is to manually mark a class as traversable by setting the `pykka_traversable` attribute to `True`:

```
class AnActor(pykka.ThreadingActor):
    playback = Playback()

class Playback(object):
    pykka_traversable = True

    def play(self):
        return True
```

When the attribute is marked as traversable, its methods can be executed in the context of the actor through an actor proxy:

```
proxy = AnActor.start().proxy()
assert proxy.playback.play().get() is True
```

New in version 2.0.

2.8 Futures

class pykka.**Future**

A handle to a value which is available now or in the future.

Typically returned by calls to actor methods or accesses to actor fields.

To get hold of the encapsulated value, call *Future.get()* or `await` the future.

get(*, *timeout: float | None = None*) → T

Get the value encapsulated by the future.

If the encapsulated value is an exception, it is raised instead of returned.

If `timeout` is `None`, as default, the method will block until it gets a reply, potentially forever. If `timeout` is an integer or float, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

The encapsulated value can be retrieved multiple times. The future will only block the first time the value is accessed.

Parameters
timeout (float or `None`) – seconds to wait before timeout

Raise
pykka.Timeout if timeout is reached

Raise
encapsulated value if it is an exception

Returns
encapsulated value if it is not an exception

set(*value: T | None = None*) → None

Set the encapsulated value.

Parameters
value (any object or `None`) – the encapsulated value or nothing

Raise
an exception if set is called multiple times

set_exception(*exc_info: OptExcInfo | None = None*) → None

Set an exception as the encapsulated value.

You can pass an `exc_info` three-tuple, as returned by `sys.exc_info()`. If you don't pass `exc_info`, `sys.exc_info()` will be called and the value returned by it used.

In other words, if you're calling *set_exception()*, without any arguments, from an except block, the exception you're currently handling will automatically be set on the future.

Parameters
exc_info (*three-tuple of (exc_class, exc_instance, traceback)*) – the encapsulated exception

set_get_hook(*func: Callable[[float | None], T]*) → None

Set a function to be executed when *get()* is called.

The function will be called when *get()* is called, with the `timeout` value as the only argument. The function's return value will be returned from *get()*.

New in version 1.2.

Parameters
func (*function accepting a timeout value*) – called to produce return value of *get()*

filter(*func: Callable[[J], bool]*) → *Future*[Iterable[J]]

Return a new future with only the items passing the predicate function.

If the future's value is an iterable, *filter()* will return a new future whose value is another iterable with only the items from the first iterable for which `func(item)` is true. If the future's value isn't an iterable, a `TypeError` will be raised when *get()* is called.

Example:

```
>>> import pykka
>>> f = pykka.ThreadingFuture()
>>> g = f.filter(lambda x: x > 10)
>>> g
<pykka.future.ThreadingFuture at ...>
>>> f.set(range(5, 15))
>>> f.get()
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> g.get()
[11, 12, 13, 14]
```

New in version 1.2.

join(**futures:* Future*[Any]*) → *Future*[Iterable[Any]]

Return a new future with a list of the result of multiple futures.

One or more futures can be passed as arguments to *join()*. The new future returns a list with the results from all the joined futures.

Example:

```
>>> import pykka
>>> a = pykka.ThreadingFuture()
>>> b = pykka.ThreadingFuture()
>>> c = pykka.ThreadingFuture()
>>> f = a.join(b, c)
>>> a.set('def')
>>> b.set(123)
>>> c.set(False)
>>> f.get()
['def', 123, False]
```

New in version 1.2.

map(*func: Callable[[T], M]*) → *Future*[M]

Pass the result of the future through a function.

Example:

```
>>> import pykka
>>> f = pykka.ThreadingFuture()
>>> g = f.map(lambda x: x + 10)
>>> f.set(30)
>>> g.get()
40

>>> f = pykka.ThreadingFuture()
>>> g = f.map(lambda x: x['foo'])
>>> f.set({'foo': 'bar'})
>>> g.get()
'bar'
```

New in version 1.2.

Changed in version 2.0: Previously, if the future's result was an iterable (except a string), the function was applied to each item in the iterable. This behavior is unpredictable and makes regular use cases like extracting a single field from a dict difficult, thus the behavior has been simplified. Now, the entire result value is passed to the function.

reduce(*func: Callable[[R, J], R]*, **args: R*) → *Future*[R]

Reduce a future's iterable result to a single value.

The function of two arguments is applied cumulatively to the items of the iterable, from left to right. The result of the first function call is used as the first argument to the second function call, and so on, until the end of the iterable. If the future's value isn't an iterable, a `TypeError` is raised.

reduce() accepts an optional second argument, which will be used as an initial value in the first function call. If the iterable is empty, the initial value is returned.

Example:

```
>>> import pykka
>>> f = pykka.ThreadingFuture()
>>> g = f.reduce(lambda x, y: x + y)
>>> f.set(['a', 'b', 'c'])
>>> g.get()
'abc'

>>> f = pykka.ThreadingFuture()
>>> g = f.reduce(lambda x, y: x + y)
>>> f.set([1, 2, 3])
>>> (1 + 2) + 3
6
>>> g.get()
6

>>> f = pykka.ThreadingFuture()
>>> g = f.reduce(lambda x, y: x + y, 5)
>>> f.set([1, 2, 3])
>>> ((5 + 1) + 2) + 3
11
>>> g.get()
11

>>> f = pykka.ThreadingFuture()
>>> g = f.reduce(lambda x, y: x + y, 5)
>>> f.set([])
>>> g.get()
5
```

New in version 1.2.

pykka.**get_all**(*futures: Iterable[*Future*[T]]*, *, *timeout: float | None = None*) → Iterable[T]

Collect all values encapsulated in the list of futures.

If `timeout` is not `None`, the method will wait for a reply for `timeout` seconds, and then raise *pykka.Timeout*.

Parameters

- **futures** (list of *pykka.Future*) – futures for the results to collect
- **timeout** (float or `None`) – seconds to wait before timeout

Raise

pykka.Timeout if timeout is reached

Returns

list of results

2.9 Registry

class pykka.**ActorRegistry**

Registry which provides easy access to all running actors.

Contains global state, but should be thread-safe.

classmethod broadcast(*message: Any*, *target_class: str | type[*Actor*] | None = None*) → None

Broadcast `message` to all actors of the specified `target_class`.

If no `target_class` is specified, the message is broadcasted to all actors.

Parameters

- **message** (*any*) – the message to send
- **target_class** (*class or class name*) – optional actor class to broadcast the message to

classmethod get_all() → list[*ActorRef*[Any]]

Get all running actors.

Returns
list of *pykka.ActorRef*

classmethod get_by_class(*actor_class: type[A]*) → list[*ActorRef*[A]]

Get all running actors of the given class or a subclass.

Parameters
actor_class (*class*) – actor class, or any superclass of the actor

Returns
list of *pykka.ActorRef*

classmethod get_by_class_name(*actor_class_name: str*) → list[*ActorRef*[Any]]

Get all running actors of the given class name.

Parameters
actor_class_name (*string*) – actor class name

Returns
list of *pykka.ActorRef*

classmethod get_by_urn(*actor_urn: str*) → *ActorRef*[Any] | None

Get an actor by its universally unique URN.

Parameters
actor_urn (*string*) – actor URN

Returns
pykka.ActorRef or `None` if not found

classmethod register(*actor_ref:* ActorRef*[Any]*) → None

Register an *ActorRef* in the registry.

This is done automatically when an actor is started, e.g. by calling *Actor.start()*.

Parameters
actor_ref (*pykka.ActorRef*) – reference to the actor to register

classmethod stop_all(*, *block: Literal[True]*, *timeout: float | None = None*) → list[bool]

classmethod stop_all(*, *block: Literal[False]*, *timeout: float | None = None*) → list[*Future*[bool]]

classmethod stop_all(*, *block: bool = True*, *timeout: float | None = None*) → list[bool] | list[*Future*[bool]]

Stop all running actors.

`block` and `timeout` works as for *ActorRef.stop()*.

If `block` is `True`, the actors are guaranteed to be stopped in the reverse of the order they were started in. This is helpful if you have simple dependencies in between your actors, where it is sufficient to shut down actors in a LIFO manner: last started, first stopped.

If you have more complex dependencies in between your actors, you should take care to shut them down in the required order yourself, e.g. by stopping dependees from a dependency's *on_stop()* method.

Returns

If not blocking, a list with a future for each stop action. If blocking, a list of return values from *pykka.ActorRef.stop()*.

classmethod unregister(*actor_ref:* ActorRef*[A]*) → None

Remove an *ActorRef* from the registry.

This is done automatically when an actor is stopped, e.g. by calling *Actor.stop()*.

Parameters

actor_ref (*pykka.ActorRef*) – reference to the actor to unregister

2.10 Exceptions

exception pykka.**ActorDeadError**

Exception raised when trying to use a dead or unavailable actor.

exception pykka.**Timeout**

Exception raised at future timeout.

2.11 Messages

The *pykka.messages* module contains Pykka's own actor messages.

In general, you should not need to use any of these classes. However, they have been made part of the public API so that certain optimizations can be done without touching Pykka's internals.

An example is to combine *ask()* and *ProxyCall* to call a method on an actor without having to spend any resources on creating a proxy object:

```
reply = actor_ref.ask(
    ProxyCall(
        attr_path=['my_method'],
        args=['foo'],
        kwargs={'bar': 'baz'}
    )
)
```

Another example is to use *tell()* instead of *ask()* for the proxy method call, and thus avoid the creation of a future for the return value if you don't need it.

It should be noted that these optimizations should only be necessary in very special circumstances.

New in version 2.0.

class pykka.messages.**ProxyCall**(*attr_path: AttrPath*, *args: Tuple[Any, ...]*, *kwargs: Dict[str, Any]*)

Message to ask the actor to call the method with the arguments.

attr_path: AttrPath

List with the path from the actor to the method.

args: Tuple[Any, ...]

List with positional arguments.

kwargs: Dict[str, Any]

Dict with keyword arguments.

class pykka.messages.**ProxyGetAttr**(*attr_path: AttrPath*)

Message to ask the actor to return the value of the attribute.

attr_path: AttrPath

List with the path from the actor to the attribute.

class pykka.messages.**ProxySetAttr**(*attr_path: AttrPath*, *value: Any*)

Message to ask the actor to set the attribute to the value.

attr_path: AttrPath

List with the path from the actor to the attribute.

value: Any

The value to set the attribute to.

2.12 Logging

Pykka uses Python's standard `logging` module for logging debug messages and any unhandled exceptions in the actors. All log messages emitted by Pykka are issued to the logger named `pykka`, or a sub-logger of it.

2.12.1 Log levels

Pykka logs at several different log levels, so that you can filter out the parts you're not interested in:

CRITICAL (highest)
This level is only used by the debug helpers in *pykka.debug*.

ERROR
Exceptions raised by an actor that are not captured into a reply future are logged at this level.

WARNING
Unhandled messages and other potential programming errors are logged at this level.

INFO
Exceptions raised by an actor that are captured into a reply future are logged at this level. If the future result is used elsewhere, the exceptions is reraised there too. If the future result isn't used, the log message is the only trace of the exception happening.

To catch bugs earlier, it is recommended to show log messages this level during development.

DEBUG (lowest)
Every time an actor is started or stopped, and registered or unregistered in the actor registry, a message is logged at this level.

In summary, you probably want to always let log messages at `WARNING` and higher through, while `INFO` should also be kept on during development.

2.12.2 Log handlers

Out of the box, Pykka is set up with `logging.NullHandler` as the only log record handler. This is the recommended approach for logging in libraries, so that the application developer using the library will have full control over how the log messages from the library will be exposed to the application's users.

In other words, if you want to see the log messages from Pykka anywhere, you need to add a useful handler to the root logger or the logger named `pykka` to get any log output from Pykka.

The defaults provided by `logging.basicConfig()` is enough to get debug log messages from Pykka:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

2.12.3 Recommended setup

If your application is already using `logging`, and you want debug log output from your own application, but not from Pykka, you can ignore debug log messages from Pykka by increasing the threshold on the Pykka logger to `INFO` level or higher:

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('pykka').setLevel(logging.INFO)
```

Given that you've fixed all unhandled exceptions logged at the `INFO` level during development, you probably want to disable logging from Pykka at the `INFO` level in production to avoid logging exceptions that are properly handled:

```
import logging
logging.basicConfig(level=logging.DEBUG)
logging.getLogger('pykka').setLevel(logging.WARNING)
```

For more details on how to use `logging`, please refer to the Python standard library documentation.

2.13 Debug helpers

Debug helpers.

pykka.debug.**log_thread_tracebacks**(**_args: Any*, ***_kwargs: Any*) → None

Log a traceback for each running thread at `logging.CRITICAL` level.

This can be a convenient tool for debugging deadlocks.

The function accepts any arguments so that it can easily be used as e.g. a signal handler, but it does not use the arguments for anything.

To use this function as a signal handler, setup logging with a `logging.CRITICAL` threshold or lower and make your main thread register this with the `signal` module:

```
import logging
import signal

import pykka.debug

logging.basicConfig(level=logging.DEBUG)
signal.signal(signal.SIGUSR1, pykka.debug.log_thread_tracebacks)
```

If your application deadlocks, send the *SIGUSR1* signal to the process:

```
kill -SIGUSR1 <pid of your process>
```

Signal handler caveats:

- The function *must* be registered as a signal handler by your main thread. If not, `signal.signal()` will raise a `ValueError`.
- All signals in Python are handled by the main thread. Thus, the signal will only be handled, and the tracebacks logged, if your main thread is available to do some work. Making your main thread idle using `time.sleep()` is OK. The signal will awaken your main thread. Blocking your main thread on e.g. `queue.Queue.get()` or *pykka.Future.get()* will break signal handling, and thus you won't be able to signal your process to print the thread tracebacks.

The morale is: setup signals using your main thread, start your actors, then let your main thread relax for the rest of your application's life cycle.

New in version 1.1.

2.13.1 Deadlock debugging

This is a complete example of how to use *log_thread_tracebacks()* to debug deadlocks:

```
#!/usr/bin/env python3

import logging
import os
import signal
import time

import pykka
import pykka.debug


class DeadlockActorA(pykka.ThreadingActor):
    def foo(self, b):
        logging.debug("This is foo calling bar")
        return b.bar().get()


class DeadlockActorB(pykka.ThreadingActor):
    def __init__(self, a):
        super().__init__()
        self.a = a
```

(continues on next page)

(continued from previous page)

```
    def bar(self):
        logging.debug("This is bar calling foo; BOOM!")
        return self.a.foo().get()


if __name__ == "__main__":
    print("Setting up logging to get output from signal handler...")
    logging.basicConfig(level=logging.DEBUG)

    print("Registering signal handler...")
    signal.signal(signal.SIGUSR1, pykka.debug.log_thread_tracebacks)

    print("Starting actors...")
    a = DeadlockActorA.start().proxy()
    b = DeadlockActorB.start(a).proxy()

    print("Now doing something stupid that will deadlock the actors...")
    a.foo(b)

    time.sleep(0.01)  # Yield to actors, so we get output in a readable order

    pid = os.getpid()
    print("Making main thread relax; not block, not quit")
    print(f"1) Use `kill -SIGUSR1 {pid:d}` to log thread tracebacks")
    print(f"2) Then `kill {pid:d}` to terminate the process")
    while True:
        time.sleep(1)
```

Running the script outputs the following:

```
Setting up logging to get output from signal handler...
Registering signal handler...
Starting actors...
DEBUG:pykka:Registered DeadlockActorA (urn:uuid:60803d09-cf5a-46cc-afdc-0c813e2e6647)
DEBUG:pykka:Starting DeadlockActorA (urn:uuid:60803d09-cf5a-46cc-afdc-0c813e2e6647)
DEBUG:pykka:Registered DeadlockActorB (urn:uuid:626adc83-ae35-439c-866a-85a3e29fd42c)
DEBUG:pykka:Starting DeadlockActorB (urn:uuid:626adc83-ae35-439c-866a-85a3e29fd42c)
Now doing something stupid that will deadlock the actors...
DEBUG:root:This is foo calling bar
DEBUG:root:This is bar calling foo; BOOM!
Making main thread relax; not block, not quit
1) Use `kill -SIGUSR1 2284` to log thread tracebacks
2) Then `kill 2284` to terminate the process
```

The two actors are now deadlocked waiting for each other while the main thread is idling, ready to process any signals.

To debug the deadlock, send the SIGUSR1 signal to the process, which has PID 2284 in this example:

```
kill -SIGUSR1 2284
```

This makes the main thread log the current traceback for each thread. The logging output shows that the two actors are both waiting for data from the other actor:

```
CRITICAL:pykka:Current state of DeadlockActorB-2 (ident: 140151493752576):
File "/usr/lib/python3.6/threading.py", line 884, in _bootstrap
    self._bootstrap_inner()
File "/usr/lib/python3.6/threading.py", line 916, in _bootstrap_inner
    self.run()
File "/usr/lib/python3.6/threading.py", line 864, in run
    self._target(*self._args, **self._kwargs)
File ".../pykka/actor.py", line 195, in _actor_loop
    response = self._handle_receive(message)
File ".../pykka/actor.py", line 297, in _handle_receive
    return callee(*message['args'], **message['kwargs'])
File "examples/deadlock_debugging.py", line 25, in bar
    return self.a.foo().get()
File ".../pykka/threading.py", line 47, in get
    self._data = self._queue.get(True, timeout)
File "/usr/lib/python3.6/queue.py", line 164, in get
    self.not_empty.wait()
File "/usr/lib/python3.6/threading.py", line 295, in wait
    waiter.acquire()

CRITICAL:pykka:Current state of DeadlockActorA-1 (ident: 140151572883200):
File "/usr/lib/python3.6/threading.py", line 884, in _bootstrap
    self._bootstrap_inner()
File "/usr/lib/python3.6/threading.py", line 916, in _bootstrap_inner
    self.run()
File "/usr/lib/python3.6/threading.py", line 864, in run
    self._target(*self._args, **self._kwargs)
File ".../pykka/actor.py", line 195, in _actor_loop
    response = self._handle_receive(message)
File ".../pykka/actor.py", line 297, in _handle_receive
    return callee(*message['args'], **message['kwargs'])
File "examples/deadlock_debugging.py", line 15, in foo
    return b.bar().get()
File ".../pykka/threading.py", line 47, in get
    self._data = self._queue.get(True, timeout)
File "/usr/lib/python3.6/queue.py", line 164, in get
    self.not_empty.wait()
File "/usr/lib/python3.6/threading.py", line 295, in wait
    waiter.acquire()

CRITICAL:pykka:Current state of MainThread (ident: 140151593330496):
File ".../examples/deadlock_debugging.py", line 49, in <module>
    time.sleep(1)
File ".../pykka/debug.py", line 63, in log_thread_tracebacks
    stack = ''.join(traceback.format_stack(frame))
```

2.14 Type hints

The *pykka.typing* module contains helpers to improve type hints.

Since Pykka 4.0, Pykka has complete type hints for the public API, tested using both Mypy and Pyright.

Due to the dynamic nature of *ActorProxy* objects, it is not possible to automatically type them correctly. This module contains helpers to manually create additional classes that correctly describe the type hints for the proxy objects. In cases where a proxy objects is used a lot, this might be worth the extra effort to increase development speed and catch bugs earlier.

Example usage:

```
from typing import cast

from pykka import ActorProxy, ThreadingActor
from pykka.typing import ActorMemberMixin, proxy_field, proxy_method


# 1) The actor class to be proxied is defined as usual:

class CircleActor(ThreadingActor):
    pi = 3.14

    def area(self, radius: float) -> float:
        return self.pi * radius**2


# 2) In addition, a proxy class is defined, which inherits from ActorMemberMixin
# to get the correct type hints for the actor methods:

class CircleProxy(ActorMemberMixin, ActorProxy[CircleActor]):

    # For each field on the proxy, a proxy_field is defined:
    pi = proxy_field(CircleActor.pi)

    # For each method on the proxy, a proxy_method is defined:
    area = proxy_method(CircleActor.area)


# 3) The actor is started like usual, and a proxy is created as usual, but the
# proxy is casted to the recently defined proxy class:
proxy = cast(CircleProxy, CircleActor.start().proxy())


# Now, the type hints for the proxy are correct:

reveal_type(proxy.stop)
# Revealed type is 'Callable[[], pykka.Future[None]]'

reveal_type(proxy.pi)
# Revealed type is 'pykka.Future[float]'

reveal_type(proxy.area))
```

(continues on next page)

(continued from previous page)

```
# Revealed type is 'Callable[[float], pykka.Future[float]]'
```

New in version 4.0.

pykka.typing.**proxy_field**(*field: T*) → *Future*[T]

Type a field on an actor proxy.

New in version 4.0.

pykka.typing.**proxy_method**(*field: Callable[Concatenate[Any, P], T]*) → Method[P, *Future*[T]]

Type a method on an actor proxy.

New in version 4.0.

class pykka.typing.**ActorMemberMixin**

Mixin class for typing Actor methods which are accessible via proxy instances.

New in version 4.0.

CHAPTER

THREE

LICENSE

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

F

G

I

J

K

L

M

O

P

R

S

T

U

V