Mode Documentation

Release 3.1.3

Robinhood Markets

Oct 30, 2019

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Copyright

Mode User Manual

by Ask Solem

Note: While the Mode *documentation* is offered under the Creative Commons *Attribution-ShareAlike 4.0 International* license the Mode *software* is offered under the BSD License (3 Clause)

1.2 Introduction to Mode

Version 3.1.3

Web http://mode.readthedocs.org/

Download http://pypi.org/project/mode

Source http://github.com/ask/mode

Keywords async, service, framework, actors, bootsteps, graph

1.2.1 What is Mode?

Mode is a very minimal Python library built-on top of AsyncIO that makes it much easier to use.

In Mode your program is built out of services that you can start, stop, restart and supervise.

A service is just a class:

```
class PageViewCache(Service):
    redis: Redis = None

    async def on_start(self) -> None:
        self.redis = connect_to_redis()

    async def update(self, url: str, n: int = 1) -> int:
        return await self.redis.incr(url, n)

    async def get(self, url: str) -> int:
        return await self.redis.get(url)
```

Services are started, stopped and restarted and have callbacks for those actions.

It can start another service:

```
class App(Service):
    page_view_cache: PageViewCache = None

    async def on_start(self) -> None:
        await self.add_runtime_dependency(self.page_view_cache)

    @cached_property
    def page_view_cache(self) -> PageViewCache:
        return PageViewCache()
```

It can include background tasks:

```
class PageViewCache(Service):

    @Service.timer(1.0)
    async def _update_cache(self) -> None:
        self.data = await cache.get('key')
```

Services that depends on other services actually form a graph that you can visualize.

Worker Mode optionally provides a worker that you can use to start the program, with support for logging, blocking detection, remote debugging and more.

To start a worker add this to your program:

```
if __name__ == '__main__':
    from mode import Worker
    Worker(Service(), loglevel="info").execute_from_commandline()
```

Then execute your program to start the worker:

```
$ python examples/tutorial.py
[2018-03-27 15:47:12,159: INFO]: [^Worker]: Starting...
[2018-03-27 15:47:12,160: INFO]: [^-AppService]: Starting...
[2018-03-27 15:47:12,160: INFO]: [^--Websockets]: Starting...
```

(continues on next page)

(continued from previous page)

```
STARTING WEBSOCKET SERVER
[2018-03-27 15:47:12,161: INFO]: [^--UserCache]: Starting...
[2018-03-27 15:47:12,161: INFO]: [^--Webserver]: Starting...
[2018-03-27 15:47:12,164: INFO]: [^--Webserver]: Serving on port 8000
REMOVING EXPIRED USERS
REMOVING EXPIRED USERS
```

To stop it hit `Control-c`:

```
[2018-03-27 15:55:08,084: INFO]: [^Worker]: Stopping on signal received...
[2018-03-27 15:55:08,084: INFO]: [^Worker]: Stopping...
[2018-03-27 15:55:08,084: INFO]: [^-AppService]: Stopping...
[2018-03-27 15:55:08,084: INFO]: [^--UserCache]: Stopping...
REMOVING EXPIRED USERS
[2018-03-27 15:55:08,085: INFO]: [^Worker]: Gathering service tasks...
[2018-03-27 15:55:08,085: INFO]: [^--UserCache]: -Stopped!
[2018-03-27 15:55:08,085: INFO]: [^--Webserver]: Stopping...
[2018-03-27 15:55:08,085: INFO]: [^Worker]: Gathering all futures...
[2018-03-27 15:55:08,085: INFO]: [^--Webserver]: Closing server
[2018-03-27 15:55:08,086: INFO]: [^--Webserver]: Waiting for server to close␣
↪handle
[2018-03-27 15:55:08,086: INFO]: [^--Webserver]: Shutting down web application
[2018-03-27 15:55:08,086: INFO]: [^--Webserver]: Waiting for handler to shut down
[2018-03-27 15:55:08,086: INFO]: [^--Webserver]: Cleanup
[2018-03-27 15:55:08,086: INFO]: [^--Webserver]: -Stopped!
[2018-03-27 15:55:08,086: INFO]: [^--Websockets]: Stopping...
[2018-03-27 15:55:08,086: INFO]: [^--Websockets]: -Stopped!
[2018-03-27 15:55:08,087: INFO]: [^-AppService]: -Stopped!
[2018-03-27 15:55:08,087: INFO]: [^Worker]: -Stopped!
```

Beacons The `beacon` object that we pass to services keeps track of the services in a graph.

They are not stricly required, but can be used to visualize a running system, for example we can render it as a pretty graph.

This requires you to have the `pydot` library and GraphViz installed:

```
$ pip install pydot
```

Let's change the app service class to dump the graph to an image at startup:

```
class AppService(Service):

    async def on_start(self) -> None:
        print('APP STARTING')
        import pydot
        import io
        o = io.StringIO()
        beacon = self.app.beacon.root or self.app.beacon
        beacon.as_graph().to_dot(o)
        graph, = pydot.graph_from_dot_data(o.getvalue())
        print('WRITING GRAPH TO image.png')
        with open('image.png', 'wb') as fh:
            fh.write(graph.create_png())
```

1.2.2 Creating a Service

To define a service, simply subclass and fill in the methods to do stuff as the service is started/stopped etc.:

```
class MyService(Service):

    async def on_start(self) -> None:
        print('Im starting now')

    async def on_started(self) -> None:
        print('Im ready')

    async def on_stop(self) -> None:
        print('Im stopping now')
```

To start the service, call `await service.start()`:

```
await service.start()
```

Or you can use `mode.Worker` (or a subclass of this) to start your services-based asyncio program from the console:

```
if __name__ == '__main__':
    import mode
    worker = mode.Worker(
        MyService(),
        loglevel='INFO',
        logfile=None,
        daemon=False,
    )
    worker.execute_from_commandline()
```

1.2.3 It's a Graph!

Services can start other services, coroutines, and background tasks.

1) Starting other services using `add_depenency`:

```
class MyService(Service):

    def __post_init__(self) -> None:
       self.add_dependency(OtherService(loop=self.loop))
```

2) Start a list of services using `on_init_dependencies`:

```
class MyService(Service):

    def on_init_dependencies(self) -> None:
        return [
            ServiceA(loop=self.loop),
            ServiceB(loop=self.loop),
            ServiceC(loop=self.loop),
        ]
```

3) Start a future/coroutine (that will be waited on to complete on stop):

```
class MyService(Service):

    async def on_start(self) -> None:
        self.add_future(self.my_coro())

    async def my_coro(self) -> None:
        print('Executing coroutine')
```

4) Start a background task:

```
class MyService(Service):

    @Service.task
    async def _my_coro(self) -> None:
        print('Executing coroutine')
```

5) Start a background task that keeps running:

```
class MyService(Service):

    @Service.task
    async def _my_coro(self) -> None:
        while not self.should_stop:
            # NOTE: self.sleep will wait for one second, or
            #       until service stopped/crashed.
            await self.sleep(1.0)
            print('Background thread waking up')
```

1.2.4 What do I need?

Version Requirements

Mode version 1.0 runs on

- Python 3.6.2

Mode requires Python 3.6.2 or later.

There's currently no plan to port Mode to earlier Python versions, please get in touch if this is something that you want to work on.

1.3 User Guide

Release 3.1

Date Oct 30, 2019

1.3.1 Services

Basics

The Service class manages the services and background tasks started by the async program, so that we can implement graceful shutdown and also helps us visualize the relationships between services in a dependency graph.

Anything that can be started/stopped and restarted should probably be a subclass of the *Service* class.

The Service API

A service can be started, and it may start other services and background tasks. Most actions in a service are asynchronous, so needs to be executed from within an async function.

This first section defines the public service API, as if used by the user, the next section will define the methods service authors write to define new services.

Methods

class mode.**Service**

set_shutdown() → None
Set the shutdown signal.

Notes

If *wait_for_shutdown* is set, stopping the service will wait for this flag to be set.

Attributes

class mode.**Service**

started
Return `True` if the service was started.

label
Label used for graphs.

shortlabel
Label used for logging.

beacon
Beacon used to track services in a dependency graph.

Defining new services

Adding child services

Child services can be added in three ways,

1) Using `add_dependency()` in `__post_init__`:

```
class MyService(Service):

    def __post_init__(self) -> None:
        self.add_dependency(OtherService())
```

2) Using `add_dependency()` in `on_start`:

```
class MyService(Service):

    async def on_start(self) -> None:
        self.add_dependency(OtherService())
```

3) Using `on_init_dependencies()`

   This is is a method that if customized should return an iterable of service instances:

```
from typing import Iterable
from mode import Service, ServiceT

class MyService(Service):

    def on_init_dependencies(self) -> Iterable[ServiceT]:
        return [ServiceA(), ServiceB()]
```

Ordering

Knowing exactly what is called, when it's called and in what order is important, and this table will help you understand that:

Order at start (`await Service.start()`)

1. The `on_first_start` callback is called.
2. Service logs: `"[Service] Starting..."`.
3. `on_start` callback is called.
4. All `@Service.task` background tasks are started (in definition order).
5. All child services added by `add_dependency()`, or `on_init_dependencies())` are started.
6. Service logs: `"[Service] Started"`.
7. The `on_started` callback is called.

Order when stopping (`await Service.stop()`)

1. Service logs; `"[Service] Stopping..."`.

2. The `on_stop()` callback is called.
3. All child services are stopped, in reverse order.
4. All asyncio futures added by `add_future()` are cancelled in reverse order.
5. Service logs: `"[Service] Stopped"`.
6. If `Service.wait_for_shutdown = True`, it will wait for the `Service.set_shutdown()` signal to be called.
7. All futures started by `add_future()` will be gathered (awaited).
8. The `on_shutdown()` callback is called.
9. The service logs: `"[Service] Shutdown complete!"`.

Order when restarting (`await Service.restart()`)

1. The service is stopped (`await service.stop()`).
2. The `__post_init__()` callback is called again.
3. The service is started (`await service.start()`).

Callbacks

class mode.**Service**

Handling Errors

class mode.**Service**

Utilities

class mode.**Service**

Logging

Your service may add logging to notify the user what is going on, and the Service class includes some shortcuts to include the service name etc. in logs.

The `self.log` delegate contains shortcuts for logging:

```
# examples/logging.py

from mode import Service


class MyService(Service):

    async def on_start(self) -> None:
        self.log.debug('This is a debug message')
        self.log.info('This is a info message')
```

(continues on next page)

(continued from previous page)

```
        self.log.warn('This is a warning message')
        self.log.error('This is a error message')
        self.log.exception('This is a error message with traceback')
        self.log.critical('This is a critical message')

        self.log.debug('I can also include templates: %r %d %s',
                       [1, 2, 3], 303, 'string')
```

The logs will be emitted by a logger with the same name as the module the Service class is defined in. It's similar to this setup, that you can do if you want to manually define the logger used by the service:

```
# examples/manual_service_logger.py

from mode import Service, get_logger

logger = get_logger(__name__)


class MyService(Service):
    logger = logger
```

1.4 FAQ: Frequently Asked Questions

1.4.1 FAQ

Can I use Mode with Django/Flask/etc.?

Yes! Use gevent/eventlet as a bridge to integrate with asyncio.

Using `gevent`

This works with any blocking Python library that can work with gevent.

Using gevent requires you to install the `aiogevent` module, and you can install this as a bundle with Mode:

```
$ pip install -U mode[gevent]
```

Then to actually use gevent as the event loop you have to execute the following in your entrypoint module (usually where you start the worker), before any other third party libraries are imported:

```
#!/usr/bin/env python3
import mode.loop
mode.loop.use('gevent')
# execute program
```

REMEMBER: This must be located at the very top of the module, in such a way that it executes before you import other libraries.

Using `eventlet`

This works with any blocking Python library that can work with eventlet.

Using eventlet requires you to install the `aioeventlet` module, and you can install this as a bundle with Mode:

```
$ pip install -U mode[eventlet]
```

Then to actually use eventlet as the event loop you have to execute the following in your entrypoint module (usually where you start the worker), before any other third party libraries are imported:

```
#!/usr/bin/env python3
import mode.loop
mode.loop.use('eventlet')
# execute program
```

REMEMBER: It's very important this is at the very top of the module, and that it executes before you import libraries.

Can I use Mode with Tornado?

Yes! Use the `tornado.platform.asyncio` bridge: http://www.tornadoweb.org/en/stable/asyncio.html

Can I use Mode with Twisted?

Yes! Use the asyncio reactor implementation: https://twistedmatrix.com/documents/17.1.0/api/twisted.internet.asyncioreactor.html

Will you support Python 3.5 or earlier?

There are no immediate plans to support Python 3.5, but you are welcome to contribute to the project.

Here are some of the steps required to accomplish this:

- Source code transformation to rewrite variable annotations to comments

  for example, the code:

```
     class Point:
         x: int = 0
         y: int = 0

must be rewritten into::

     class Point:
         x = 0  # type: int
         y = 0  # type: int
```

- Source code transformation to rewrite async functions

  for example, the code:

```
async def foo():
    await asyncio.sleep(1.0)
```

  must be rewritten into:

```
@coroutine
def foo():
    yield from asyncio.sleep(1.0)
```

Will you support Python 2?

There are no plans to support Python 2, but you are welcome to contribute to the project (details in question above is relevant also for Python 2).

At Shutdown I get lots of warnings, what is this about?

If you get warnings such as this at shutdown:

```
Task was destroyed but it is pending!
task: <Task pending coro=<Service._execute_task() running at /opt/devel/mode/mode/
↪services.py:643> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at␣
↪0x1100a7468>()]>>
Task was destroyed but it is pending!
task: <Task pending coro=<Service._execute_task() running at /opt/devel/mode/mode/
↪services.py:643> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at␣
↪0x1100a72e8>()]>>
Task was destroyed but it is pending!
task: <Task pending coro=<Service._execute_task() running at /opt/devel/mode/mode/
↪services.py:643> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at␣
↪0x1100a7678>()]>>
Task was destroyed but it is pending!
task: <Task pending coro=<Event.wait() running at /Library/Frameworks/Python.
↪framework/Versions/3.6/lib/python3.6/asyncio/locks.py:269> cb=[_release_waiter(
↪<Future pendi...1100a7468>()]>)() at /Library/Frameworks/Python.framework/Versions/
↪3.6/lib/python3.6/asyncio/tasks.py:316]>
Task was destroyed but it is pending!
    task: <Task pending coro=<Event.wait() running at /Library/Frameworks/Python.
↪framework/Versions/3.6/lib/python3.6/asyncio/locks.py:269> cb=[_release_waiter(
↪<Future pendi...1100a7678>()]>)() at /Library/Frameworks/Python.framework/Versions/
↪3.6/lib/python3.6/asyncio/tasks.py:316]>
```

It usually means you forgot to stop a service before the process exited.

1.5 API Reference

Release 3.1

Date Oct 30, 2019

1.5.1 Mode

mode

AsyncIO Service-based programming.

class mode.**Service**(*, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*)

An asyncio service that can be started/stopped/restarted.

Keyword Arguments

- **beacon** (NodeT) – Beacon used to track services in a graph.
- **loop** (*asyncio.AbstractEventLoop*) – Event loop object.

abstract = False

class Diag(*service: mode.types.services.ServiceT*)
Service diagnostics.

This can be used to track what your service is doing. For example if your service is a Kafka consumer with a background thread that commits the offset every 30 seconds, you may want to see when this happens:

```
DIAG_COMMITTING = 'committing'

class Consumer(Service):

    @Service.task
    async def _background_commit(self) -> None:
        while not self.should_stop:
            await self.sleep(30.0)
            self.diag.set_flag(DIAG_COMITTING)
            try:
                await self._consumer.commit()
            finally:
                self.diag.unset_flag(DIAG_COMMITTING)
```

The above code is setting the flag manually, but you can also use a decorator to accomplish the same thing:

```
@Service.timer(30.0)
async def _background_commit(self) -> None:
    await self.commit()

@Service.transitions_with(DIAG_COMITTING)
async def commit(self) -> None:
    await self._consumer.commit()
```

set_flag(*flag: str*) → None

unset_flag(*flag: str*) → None

wait_for_shutdown = False
Set to True if .stop must wait for the shutdown flag to be set.

shutdown_timeout = 60.0
Time to wait for shutdown flag set before we give up.

restart_count = 0
Current number of times this service instance has been restarted.

mundane_level = 'info'
The log level for mundane info such as *starting*, *stopping*, etc. Set this to `"debug"` for less information.

classmethod from_awaitable(*coro: Awaitable*, *, *name: str = None*, ***kwargs: Any*) → mode.types.services.ServiceT

classmethod task(*fun: Callable[Any, Awaitable[None]]*) → mode.services.ServiceTask
Decorate function to be used as background task.

Example

```
>>> class S(Service):
...
...     @Service.task
```

(continues on next page)

(continued from previous page)

```
...     async def background_task(self):
...         while not self.should_stop:
...             await self.sleep(1.0)
...             print('Waking up')
```

classmethod timer(*interval: Union[datetime.timedelta, float, str]*) → Callable[Callable[mode.types.services.ServiceT, Awaitable[None]], mode.services.ServiceTask]

Background timer executing every `n` seconds.

Example

```
>>> class S(Service):
...
...     @Service.timer(1.0)
...     async def background_timer(self):
...         print('Waking up')
```

classmethod transitions_to(*flag: str*) → Callable

Decorate function to set and reset diagnostic flag.

add_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

Add dependency to other service.

The service will be started/stopped with this service.

add_context(*context: ContextManager*) → Any

add_future(*coro: Awaitable*) → _asyncio.Future

Add relationship to asyncio.Future.

The future will be joined when this service is stopped.

on_init() → None

on_init_dependencies() → Iterable[mode.types.services.ServiceT]

Return list of service dependencies for this service.

service_reset() → None

set_shutdown() → None

Set the shutdown signal.

Notes

If *wait_for_shutdown* is set, stopping the service will wait for this flag to be set.

property started

Return `True` if the service was started.

property crashed

property should_stop

Return `True` if the service must stop.

property state

Service state - as a human readable string.

property label
Label used for graphs.

property shortlabel
Label used for logging.

property beacon
Beacon used to track services in a dependency graph.

logger = <Logger mode.services (WARNING)>

mode.**task**(*fun: Callable[Any, Awaitable[None]]*) → mode.services.ServiceTask
Decorate function to be used as background task.

Example

```
>>> class S(Service):
...
...     @Service.task
...     async def background_task(self):
...         while not self.should_stop:
...             await self.sleep(1.0)
...             print('Waking up')
```

mode.**timer**(*interval: Union[datetime.timedelta, float, str]*) → Callable[Callable[mode.types.services.ServiceT, Awaitable[None]], mode.services.ServiceTask]
Background timer executing every `n` seconds.

Example

```
>>> class S(Service):
...
...     @Service.timer(1.0)
...     async def background_timer(self):
...         print('Waking up')
```

class mode.**BaseSignal**(**, name: str = None, owner: Type = None, loop: asyncio.events.AbstractEventLoop = None, default_sender: Any = None, receivers: MutableSet[Any] = None, filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)
Base class for signal/observer pattern.

asdict() → Mapping[str, Any]

clone(***kwargs: Any*) → mode.types.signals.BaseSignalT

with_default_sender(*sender: Any = None*) → mode.types.signals.BaseSignalT

unpack_sender_from_args(**args: Any*) → Tuple[T, Tuple[Any, ...]]

connect(*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]] = None, **kwargs: Any*) → Callable

disconnect(*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]], *, weak: bool = False, sender: Any = None*) → None

iter_receivers(*sender: T_contra*) → Iterable[Union[Callable[[T, Any, mode.types.signals.BaseSignalT, Any], None], Callable[[T, Any, mode.types.signals.BaseSignalT, Any], Awaitable[None]]]]

property ident

property label

class mode.**Signal**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Asynchronous signal (using `async def` functions).

clone(***kwargs: Any*) → mode.types.signals.SignalT

with_default_sender(*sender: Any = None*) → mode.types.signals.SignalT

class mode.**SyncSignal**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Signal that is synchronous (using regular `def` functions).

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

clone(***kwargs: Any*) → mode.types.signals.SyncSignalT

with_default_sender(*sender: Any = None*) → mode.types.signals.SyncSignalT

class mode.**ForfeitOneForAllSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

If one service in the group crashes, we give up on all of them.

logger = <Logger mode.supervisors (WARNING)>

class mode.**ForfeitOneForOneSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

Supervisor that if a service crashes, we do not restart it.

logger = <Logger mode.supervisors (WARNING)>

class mode.**OneForAllSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

Supervisor that restarts all services when a service crashes.

logger = <Logger mode.supervisors (WARNING)>

class mode.**OneForOneSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

Supervisor simply restarts any crashed service.

logger = <Logger mode.supervisors (WARNING)>

class mode.**SupervisorStrategy**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

Base class for all supervisor strategies.

wakeup() → None

add(**services: mode.types.services.ServiceT*) → None

discard(**services: mode.types.services.ServiceT*) → None

insert(*index: int*, *service: mode.types.services.ServiceT*) → None

service_operational(*service: mode.types.services.ServiceT*) → bool

logger = <Logger mode.supervisors (WARNING)>

class mode.**CrashingSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

Supervisor that crashes the whole program.

logger = <Logger mode.supervisors (WARNING)>

wakeup() → None

class mode.**ServiceT**(**, beacon: mode.utils.types.trees.NodeT = None, loop: asyncio.events.AbstractEventLoop = None*)

Abstract type for an asynchronous service that can be started/stopped.

See also:

mode.Service.

wait_for_shutdown = False

restart_count = 0

supervisor = None

abstract add_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

abstract add_context(*context: ContextManager*) → Any

abstract service_reset() → None

abstract set_shutdown() → None

abstract property started

abstract property crashed

abstract property should_stop

abstract property state

abstract property label

abstract property shortlabel

property beacon

abstract property loop

class mode.**BaseSignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base type for all signals.

abstract clone(***kwargs: Any*) → mode.types.signals.BaseSignalT

abstract with_default_sender(*sender: Any = None*) → mode.types.signals.BaseSignalT

abstract connect(*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]], **kwargs: Any*) → Callable

abstract disconnect(*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]], *, sender: Any = None, weak: bool = True*) → None

class mode.**SignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base class for all async signals (using `async def`).

abstract clone(***kwargs: Any*) → SignalT

abstract with_default_sender(*sender: Any = None*) → SignalT

class mode.**SyncSignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base class for all synchronous signals (using regular `def`).

abstract send(*sender: T_contra*, **args: Any*, ***kwargs: Any*) → None

abstract clone(***kwargs: Any*) → SyncSignalT

abstract with_default_sender(*sender: Any = None*) → SyncSignalT

class mode.**SupervisorStrategyT**(**services: mode.types.supervisors.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = None, replacement: Callable[[mode.types.supervisors.ServiceT, int], Awaitable[mode.types.supervisors.ServiceT]] = None, **kwargs: Any*)

Base type for all supervisor strategies.

abstract wakeup() → None

abstract add(**services: mode.types.supervisors.ServiceT*) → None

abstract discard(**services: mode.types.supervisors.ServiceT*) → None

abstract service_operational(*service: mode.types.supervisors.ServiceT*) → bool

mode.**want_seconds**(*s: float*) → float
Convert `Seconds` to float.

class mode.**flight_recorder**(*logger: Any, *, timeout: Union[datetime.timedelta, float, str], loop: asyncio.events.AbstractEventLoop = None*)
Flight Recorder context for use with `with` statement.

This is a logging utility to log stuff only when something times out.

For example if you have a background thread that is sometimes hanging:

```
class RedisCache(mode.Service):

    @mode.timer(1.0)
    def _background_refresh(self) -> None:
        self._users = await self.redis_client.get(USER_KEY)
        self._posts = await self.redis_client.get(POSTS_KEY)
```

You want to figure out on what line this is hanging, but logging all the time will provide way too much output, and will even change how fast the program runs and that can mask race conditions, so that they never happen.

Use the flight recorder to save the logs and only log when it times out:

```
logger = mode.get_logger(__name__)

class RedisCache(mode.Service):

    @mode.timer(1.0)
    def _background_refresh(self) -> None:
        with mode.flight_recorder(logger, timeout=10.0) as on_timeout:
            on_timeout.info(f'+redis_client.get({USER_KEY!r})')
            await self.redis_client.get(USER_KEY)
            on_timeout.info(f'-redis_client.get({USER_KEY!r})')

            on_timeout.info(f'+redis_client.get({POSTS_KEY!r})')
            await self.redis_client.get(POSTS_KEY)
            on_timeout.info(f'-redis_client.get({POSTS_KEY!r})')
```

If the body of this `with` statement completes before the timeout, the logs are forgotten about and never emitted – if it takes more than ten seconds to complete, we will see these messages in the log:

```
[2018-04-19 09:43:55,877: WARNING]: Warning: Task timed out!
[2018-04-19 09:43:55,878: WARNING]:
    Please make sure it is hanging before restarting.
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1]
    (started at Thu Apr 19 09:43:45 2018) Replaying logs...
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1]
    (Thu Apr 19 09:43:45 2018) +redis_client.get('user')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1]
    (Thu Apr 19 09:43:49 2018) -redis_client.get('user')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1]
```

(continues on next page)

(continued from previous page)

```
    (Thu Apr 19 09:43:46 2018) +redis_client.get('posts')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1] -End of log-
```

Now we know this `redis_client.get` call can take too long to complete, and should consider adding a timeout to it.

wrap_debug(*obj: Any*) → mode.utils.logging.Logwrapped

wrap_info(*obj: Any*) → mode.utils.logging.Logwrapped

wrap_warn(*obj: Any*) → mode.utils.logging.Logwrapped

wrap_error(*obj: Any*) → mode.utils.logging.Logwrapped

wrap(*severity: int*, *obj: Any*) → mode.utils.logging.Logwrapped

activate() → None

cancel() → None

log(*severity: int*, *message: str*, **args: Any*, ***kwargs: Any*) → None

mode.**get_logger**(*name: str*) → logging.Logger
Get logger by name.

mode.**setup_logging**(***, *loglevel: Union[str, int] = None*, *logfile: Union[str, IO] = None*, *loghandlers: List[logging.StreamHandler] = None*, *logging_config: Dict = None*) → int
Configure logging subsystem.

mode.**label**(*s: Any*) → str
Return the name of an object as string.

mode.**shortlabel**(*s: Any*) → str
Return the shortened name of an object as string.

class mode.**Worker**(**services: mode.types.services.ServiceT*, *debug: bool = False*, *quiet: bool = False*, *logging_config: Dict = None*, *loglevel: Union[str, int] = None*, *logfile: Union[str, IO] = None*, *redirect_stdouts: bool = True*, *redirect_stdouts_level: Union[int, str] = None*, *stdout: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *stderr: IO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *console_port: int = 50101*, *loghandlers: List[logging.StreamHandler] = None*, *blocking_timeout: Union[datetime.timedelta, float, str] = 10.0*, *loop: asyncio.events.AbstractEventLoop = None*, *daemon: bool = True*, ***kwargs: Any*)
Start mode service from the command-line.

say(*msg: str*) → None
Write message to standard out.

carp(*msg: str*) → None
Write warning to standard err.

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of service dependencies for this service.

on_setup_root_logger(*logger: logging.Logger*, *level: int*) → None

install_signal_handlers() → None

logger = <Logger mode.worker (WARNING)>

execute_from_commandline() → NoReturn

on_worker_shutdown() → None

stop_and_shutdown() → None

property blocking_detector

mode.debug

Debugging utilities.

exception mode.debug.**Blocking**
Exception raised when event loop is blocked.

class mode.debug.**BlockingDetector**(*timeout: Union[datetime.timedelta, float, str], raises: Type[BaseException] = <class 'mode.debug.Blocking'>, **kwargs: Any*)
Service that detects blocking code using alarm/itimer.

Examples

blockdetect = BlockingDetector(timeout=10.0) await blockdetect.start()

Keyword Arguments

- **timeout** (Seconds) – number of seconds that the event loop can be blocked.
- **raises** (*Type[BaseException]*) – Exception to raise when the blocking timeout is exceeded. Defaults to *Blocking*.

logger = <Logger mode.debug (WARNING)>

mode.exceptions

Custom exceptions.

exception mode.exceptions.**MaxRestartsExceeded**
Supervisor found restarting service too frequently.

mode.locals

Implements thread-local stack using ContextVar (**PEP 567**).

This is a reimplementation of the local stack as used by Flask, Werkzeug, Celery, and other libraries to keep a thread-local stack of objects.

- Supports typing:

```
request_stack: LocalStack[Request] = LocalStack()
```

class mode.locals.**LocalStack**
LocalStack.

Manage state per coroutine (also thread safe).

Most famously used probably in Flask to keep track of the current request object.

push(*obj: T*) → Generator[[None, None], None]
Push a new item to the stack.

push_without_automatic_cleanup(*obj: T*) → None

pop() → Optional[T]
Remove the topmost item from the stack.

Note: Will return the old value or *None* if the stack was already empty.

property stack

property top
Return the topmost item on the stack.

Does not remove it from the stack.

Note: If the stack is empty, `None` is returned.

mode.proxy

Proxy to service.

Works like a service, but delegates to underlying service object.

class mode.proxy.**ServiceProxy**(*, *loop: asyncio.events.AbstractEventLoop = None*)
A service proxy delegates ServiceT methods to a composite service.

Example

```
>>> class MyServiceProxy(ServiceProxy):
...
...     @cached_property
...     def _service(self) -> ServiceT:
...         return ActualService()
```

Notes

This is used by Faust, and probably useful elsewhere! The Faust App is created at module-level, and it uses service proxy to ensure the event loop is not also created just by importing a module.

add_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

add_context(*context: ContextManager*) → Any

service_reset() → None

set_shutdown() → None

property started

property crashed

property should_stop

property state

property label

property shortlabel

abstract = False

property beacon

logger = <Logger mode.proxy (WARNING)>

mode.services

Async I/O services that can be started/stopped/shutdown.

class mode.services.**ServiceBase**(*, *loop: asyncio.events.AbstractEventLoop = None*)
Base class for services.

abstract = True
Set to True if this service class is abstract-only, meaning it will only be used as a base class.

logger = None
Logger used by this service. If not explicitly set this will be based on get_logger(cls.__name__)

property loop

class mode.services.**Diag**(*service: mode.types.services.ServiceT*)
Service diagnostics.

This can be used to track what your service is doing. For example if your service is a Kafka consumer with a background thread that commits the offset every 30 seconds, you may want to see when this happens:

```
DIAG_COMMITTING = 'committing'

class Consumer(Service):

    @Service.task
    async def _background_commit(self) -> None:
        while not self.should_stop:
            await self.sleep(30.0)
            self.diag.set_flag(DIAG_COMITTING)
            try:
                await self._consumer.commit()
            finally:
                self.diag.unset_flag(DIAG_COMMITTING)
```

The above code is setting the flag manually, but you can also use a decorator to accomplish the same thing:

```
@Service.timer(30.0)
async def _background_commit(self) -> None:
    await self.commit()

@Service.transitions_with(DIAG_COMITTING)
async def commit(self) -> None:
    await self._consumer.commit()
```

set_flag(*flag: str*) → None

unset_flag(*flag: str*) → None

class mode.services.**Service**(*, *beacon: mode.utils.types.trees.NodeT = None*, *loop: asyncio.events.AbstractEventLoop = None*)
An asyncio service that can be started/stopped/restarted.

Keyword Arguments

- **beacon** (NodeT) – Beacon used to track services in a graph.
- **loop** (*asyncio.AbstractEventLoop*) – Event loop object.

abstract = False

class Diag(*service: mode.types.services.ServiceT*)
Service diagnostics.

This can be used to track what your service is doing. For example if your service is a Kafka consumer with a background thread that commits the offset every 30 seconds, you may want to see when this happens:

```
DIAG_COMMITTING = 'committing'

class Consumer(Service):

    @Service.task
    async def _background_commit(self) -> None:
        while not self.should_stop:
            await self.sleep(30.0)
            self.diag.set_flag(DIAG_COMITTING)
            try:
                await self._consumer.commit()
            finally:
                self.diag.unset_flag(DIAG_COMMITTING)
```

The above code is setting the flag manually, but you can also use a decorator to accomplish the same thing:

```
@Service.timer(30.0)
async def _background_commit(self) -> None:
    await self.commit()

@Service.transitions_with(DIAG_COMITTING)
async def commit(self) -> None:
    await self._consumer.commit()
```

set_flag(*flag: str*) → None

unset_flag(*flag: str*) → None

wait_for_shutdown = False
Set to True if .stop must wait for the shutdown flag to be set.

shutdown_timeout = 60.0
Time to wait for shutdown flag set before we give up.

restart_count = 0
Current number of times this service instance has been restarted.

mundane_level = 'info'
The log level for mundane info such as *starting*, *stopping*, etc. Set this to `"debug"` for less information.

classmethod from_awaitable(*coro: Awaitable*, ***, *name: str = None*, ***kwargs: Any*) → mode.types.services.ServiceT

classmethod task(*fun: Callable[Any, Awaitable[None]]*) → mode.services.ServiceTask
Decorate function to be used as background task.

Example

```
>>> class S(Service):
...
...     @Service.task
...     async def background_task(self):
...         while not self.should_stop:
...             await self.sleep(1.0)
...             print('Waking up')
```

classmethod timer(*interval: Union[datetime.timedelta, float, str]*) → Callable[Callable[mode.types.services.ServiceT, Awaitable[None]], mode.services.ServiceTask]
Background timer executing every `n` seconds.

Example

```
>>> class S(Service):
...
...     @Service.timer(1.0)
...     async def background_timer(self):
...         print('Waking up')
```

classmethod transitions_to(*flag: str*) → Callable
Decorate function to set and reset diagnostic flag.

add_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT
Add dependency to other service.

The service will be started/stopped with this service.

add_context(*context: ContextManager*) → Any

add_future(*coro: Awaitable*) → _asyncio.Future
Add relationship to asyncio.Future.

The future will be joined when this service is stopped.

on_init() → None

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of service dependencies for this service.

service_reset() → None

set_shutdown() → None
Set the shutdown signal.

Notes

If *wait_for_shutdown* is set, stopping the service will wait for this flag to be set.

property started
Return `True` if the service was started.

property crashed

property should_stop
Return `True` if the service must stop.

property state
Service state - as a human readable string.

property label
Label used for graphs.

property shortlabel
Label used for logging.

property beacon
Beacon used to track services in a dependency graph.

logger = <Logger mode.services (WARNING)>

mode.services.**task**(*fun: Callable[Any, Awaitable[None]]*) → mode.services.ServiceTask
Decorate function to be used as background task.

Example

```
>>> class S(Service):
...
...     @Service.task
...     async def background_task(self):
...         while not self.should_stop:
...             await self.sleep(1.0)
...             print('Waking up')
```

mode.services.**timer**(*interval: Union[datetime.timedelta, float, str]*) → Callable[Callable[mode.types.services.ServiceT, Awaitable[None]], mode.services.ServiceTask]
Background timer executing every n seconds.

Example

```
>>> class S(Service):
...
...     @Service.timer(1.0)
...     async def background_timer(self):
...         print('Waking up')
```

mode.signals

Signals - implementation of the Observer pattern.

class mode.signals.**BaseSignal**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)
Base class for signal/observer pattern.

asdict() → Mapping[str, Any]

clone(***kwargs: Any*) → mode.types.signals.BaseSignalT

with_default_sender(*sender: Any = None*) → mode.types.signals.BaseSignalT

unpack_sender_from_args(**args: Any*) → Tuple[T, Tuple[Any, ...]]

connect (*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]] = None, **kwargs: Any*) → Callable

disconnect (*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]], *, weak: bool = False, sender: Any = None*) → None

iter_receivers (*sender: T_contra*) → Iterable[Union[Callable[[T, Any, mode.types.signals.BaseSignalT, Any], None], Callable[[T, Any, mode.types.signals.BaseSignalT, Any], Awaitable[None]]]]

property ident

property label

class mode.signals.**Signal**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Asynchronous signal (using `async def` functions).

clone (***kwargs: Any*) → mode.types.signals.SignalT

with_default_sender (*sender: Any = None*) → mode.types.signals.SignalT

class mode.signals.**SyncSignal**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Signal that is synchronous (using regular `def` functions).

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

clone (***kwargs: Any*) → mode.types.signals.SyncSignalT

with_default_sender (*sender: Any = None*) → mode.types.signals.SyncSignalT

mode.supervisors

Supervisors.

Naming here is taken from Erlang ;-)

Don't know supervisors? Read about them them here: http://learnyousomeerlang.com/supervisors

class mode.supervisors.**SupervisorStrategy**(**services: mode.types.services.ServiceT*, *max_restarts: Union[datetime.timedelta, float, str] = 100.0*, *over: Union[datetime.timedelta, float, str] = 1.0*, *raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>*, *replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None*, ***kwargs: Any*)

Base class for all supervisor strategies.

wakeup () → None

add (**services: mode.types.services.ServiceT*) → None

discard (**services: mode.types.services.ServiceT*) → None

insert (*index: int*, *service: mode.types.services.ServiceT*) → None

service_operational (*service: mode.types.services.ServiceT*) → bool

logger = <Logger mode.supervisors (WARNING)>

class mode.supervisors.**OneForOneSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

Supervisor simply restarts any crashed service.

logger = <Logger mode.supervisors (WARNING)>

class mode.supervisors.**OneForAllSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

Supervisor that restarts all services when a service crashes.

logger = <Logger mode.supervisors (WARNING)>

class mode.supervisors.**ForfeitOneForOneSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

Supervisor that if a service crashes, we do not restart it.

logger = <Logger mode.supervisors (WARNING)>

class mode.supervisors.**ForfeitOneForAllSupervisor**(**services: mode.types.services.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = <class 'mode.exceptions.MaxRestartsExceeded'>, replacement: Callable[[mode.types.services.ServiceT, int], Awaitable[mode.types.services.ServiceT]] = None, **kwargs: Any*)

If one service in the group crashes, we give up on all of them.

logger = <Logger mode.supervisors (WARNING)>

mode.threads

ServiceThread - Service that starts in a separate thread.

Will use the default thread pool executor (`loop.set_default_executor()`), unless you specify a specific executor instance.

Note: To stop something using the thread's loop, you have to use the `on_thread_stop` callback instead of the on_stop callback.

class mode.threads.**QueuedMethod**

Describe a method to be called by thread.

property promise

Alias for field number 0

property method

Alias for field number 1

property args

Alias for field number 2

property kwargs

Alias for field number 3

class mode.threads.**WorkerThread**(*service: mode.threads.ServiceThread*, ***kwargs: Any*)

Thread class used for services running in a dedicated thread.

run() → None

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

stop() → None

class mode.threads.**ServiceThread**(**, executor: Any = None, loop: asyncio.events.AbstractEventLoop = None, thread_loop: asyncio.events.AbstractEventLoop = None, Worker: Type[mode.threads.WorkerThread] = None, **kwargs: Any*)

Service subclass running within a dedicated thread.

abstract = False

wait_for_shutdown = True

wait_for_thread = True
Set this to False if s.start() should not wait for the underlying thread to be fully started.

Worker
alias of *WorkerThread*

on_crash(*msg: str*, **fmt: Any*, ***kwargs: Any*) → None

logger = <Logger mode.threads (WARNING)>

class mode.threads.**QueueServiceThread**(*, *executor: Any = None*, *loop: asyncio.events.AbstractEventLoop = None*, *thread_loop: asyncio.events.AbstractEventLoop = None*, *Worker: Type[mode.threads.WorkerThread] = None*, ***kwargs: Any*)

Service running in separate thread.

Uses a queue to run functions inside the thread, so you can delegate calls.

logger = <Logger mode.threads (WARNING)>

abstract = False

property method_queue

mode.timers

AsyncIO Timers.

mode.timers.**timer_intervals**(*interval: Union[datetime.timedelta, float, str], max_drift_correction: float = 0.1, name: str = '', clock: Callable[float] = <built-in function perf_counter>*) → Iterator[float]

Generate timer sleep times.

Example

```
>>> async def my_timer(interval=1.0):
...     # wait interval before running first time.
...     await asyncio.sleep(interval)
...     for sleep_time in timer_intervals(1.0, name='my_timer'):
...         # do something that takes a while.
...         await asyncio.sleep(sleep_time)
```

mode.worker

Worker - Starts services from the command-line.

Workers add signal handling, logging, and other things required to start and manage services in a process environment.

class mode.worker.**Worker**(**services: mode.types.services.ServiceT, debug: bool = False, quiet: bool = False, logging_config: Dict = None, loglevel: Union[str, int] = None, logfile: Union[str, IO] = None, redirect_stdouts: bool = True, redirect_stdouts_level: Union[int, str] = None, stdout: IO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>, stderr: IO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>, console_port: int = 50101, loghandlers: List[logging.StreamHandler] = None, blocking_timeout: Union[datetime.timedelta, float, str] = 10.0, loop: asyncio.events.AbstractEventLoop = None, daemon: bool = True, **kwargs: Any*)

Start mode service from the command-line.

say(*msg: str*) → None
Write message to standard out.

carp(*msg: str*) → None
Write warning to standard err.

on_init_dependencies() → Iterable[mode.types.services.ServiceT]
Return list of service dependencies for this service.

on_setup_root_logger(*logger: logging.Logger*, *level: int*) → None

install_signal_handlers() → None

logger = <Logger mode.worker (WARNING)>

execute_from_commandline() → NoReturn

on_worker_shutdown() → None

stop_and_shutdown() → None

property blocking_detector

1.5.2 Typehints

mode.types

class mode.types.**DiagT**(*service: mode.types.services.ServiceT*)
Diag keeps track of a services diagnostic flags.

abstract set_flag(*flag: str*) → None

abstract unset_flag(*flag: str*) → None

class mode.types.**ServiceT**(**, beacon: mode.utils.types.trees.NodeT = None, loop: asyncio.events.AbstractEventLoop = None*)
Abstract type for an asynchronous service that can be started/stopped.

See also:

mode.Service.

wait_for_shutdown = False

restart_count = 0

supervisor = None

abstract add_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

abstract add_context(*context: ContextManager*) → Any

abstract service_reset() → None

abstract set_shutdown() → None

abstract property started

abstract property crashed

abstract property should_stop

abstract property state

abstract property label

abstract property shortlabel

property beacon

abstract property loop

class mode.types.**BaseSignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base type for all signals.

abstract clone(***kwargs: Any*) → mode.types.signals.BaseSignalT

abstract with_default_sender(*sender: Any = None*) → mode.types.signals.BaseSignalT

abstract connect(*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]], **kwargs: Any*) → Callable

abstract disconnect(*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]], *, sender: Any = None, weak: bool = True*) → None

class mode.types.**SignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base class for all async signals (using `async def`).

abstract clone(***kwargs: Any*) → SignalT

abstract with_default_sender(*sender: Any = None*) → SignalT

class mode.types.**SyncSignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base class for all synchronous signals (using regular `def`).

abstract send(*sender: T_contra*, **args: Any*, ***kwargs: Any*) → None

abstract clone(***kwargs: Any*) → SyncSignalT

abstract with_default_sender(*sender: Any = None*) → SyncSignalT

class mode.types.**SupervisorStrategyT**(**services: mode.types.supervisors.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = None, replacement: Callable[[mode.types.supervisors.ServiceT, int], Awaitable[mode.types.supervisors.ServiceT]] = None, **kwargs: Any*)

Base type for all supervisor strategies.

abstract wakeup() → None

abstract add(**services: mode.types.supervisors.ServiceT*) → None

abstract discard(**services: mode.types.supervisors.ServiceT*) → None

abstract service_operational(*service: mode.types.supervisors.ServiceT*) → bool

mode.types.services

Type classes for *mode.services*.

class mode.types.services.**DiagT**(*service: mode.types.services.ServiceT*)

Diag keeps track of a services diagnostic flags.

abstract set_flag(*flag: str*) → None

abstract unset_flag(*flag: str*) → None

class mode.types.services.**ServiceT**(**, beacon: mode.utils.types.trees.NodeT = None, loop: asyncio.events.AbstractEventLoop = None*)

Abstract type for an asynchronous service that can be started/stopped.

See also:

mode.Service.

wait_for_shutdown = False

restart_count = 0

supervisor = None

abstract add_dependency(*service: mode.types.services.ServiceT*) → mode.types.services.ServiceT

abstract add_context(*context: ContextManager*) → Any

abstract service_reset() → None

abstract set_shutdown() → None

abstract property started

abstract property crashed

abstract property should_stop

abstract property state

abstract property label

abstract property shortlabel

property beacon

abstract property loop

mode.types.signals

Type classes for *mode.signals*.

mode.types.signals.**FilterReceiverMapping**
alias of typing.MutableMapping

class mode.types.signals.**BaseSignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base type for all signals.

abstract clone(***kwargs: Any*) → mode.types.signals.BaseSignalT

abstract with_default_sender(*sender: Any = None*) → mode.types.signals.BaseSignalT

abstract connect(*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]], **kwargs: Any*) → Callable

abstract disconnect(*fun: Union[Callable[[T, Any, BaseSignalT, Any], None], Callable[[T, Any, BaseSignalT, Any], Awaitable[None]]], *, sender: Any = None, weak: bool = True*) → None

class mode.types.signals.**SignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base class for all async signals (using `async def`).

abstract clone(***kwargs: Any*) → SignalT

abstract with_default_sender(*sender: Any = None*) → SignalT

class mode.types.signals.**SyncSignalT**(*, *name: str = None*, *owner: Type = None*, *loop: asyncio.events.AbstractEventLoop = None*, *default_sender: Any = None*, *receivers: MutableSet[Any] = None*, *filter_receivers: MutableMapping[Any, MutableSet[Any]] = None*)

Base class for all synchronous signals (using regular `def`).

abstract send(*sender: T_contra*, **args: Any*, ***kwargs: Any*) → None

abstract clone(***kwargs: Any*) → SyncSignalT

abstract with_default_sender(*sender: Any = None*) → SyncSignalT

mode.types.supervisors

Type classes for *mode.supervisors*.

class mode.types.supervisors.**SupervisorStrategyT**(**services: mode.types.supervisors.ServiceT, max_restarts: Union[datetime.timedelta, float, str] = 100.0, over: Union[datetime.timedelta, float, str] = 1.0, raises: Type[BaseException] = None, replacement: Callable[[mode.types.supervisors.ServiceT, int], Awaitable[mode.types.supervisors.ServiceT]] = None, **kwargs: Any*)

Base type for all supervisor strategies.

abstract wakeup() → None

abstract add(**services: mode.types.supervisors.ServiceT*) → None

abstract discard(**services: mode.types.supervisors.ServiceT*) → None

abstract service_operational(*service: mode.types.supervisors.ServiceT*) → bool

1.5.3 Event Loops

mode.loop

AsyncIO event loop implementations.

This contains a registry of different AsyncIO loop implementations to be used with Mode.

The choices available are:

aio default Normal `asyncio` event loop policy.

eventlet Use eventlet as the event loop.

This uses aioeventlet and will apply the eventlet monkey-patches.

To enable execute the following as the first thing that happens when your program starts (e.g. add it as the top import of your entrypoint module):

```
>>> import mode.loop
>>> mode.loop.use('eventlet')
```

gevent Use gevent as the event loop.

This uses aiogevent (+modifications) and will apply the gevent monkey-patches.

This choice enables you to run blocking Python code as if they have invisible *async/await* syntax around it (NOTE: C extensions are not usually gevent compatible).

To enable execute the following as the first thing that happens when your program starts (e.g. add it as the top import of your entrypoint module):

```
>>> import mode.loop
>>> mode.loop.use('gevent')
```

uvloop Event loop using uvloop.

To enable execute the following as the first thing that happens when your program starts (e.g. add it as the top import of your entrypoint module):

```
>>> import mode.loop
>>> mode.loop.use('uvloop')
```

mode.loop.**use**(*loop: str*) → None
Specify the event loop to use as a string.

Loop must be one of: aio, eventlet, gevent, uvloop.

mode.loop.eventlet

Warning: Importing this module directly will set the global event loop. See `faust.loop` for more information.

mode.loop.gevent

Warning: Importing this module directly will set the global event loop. See `faust.loop` for more information.

mode.loop.uvloop

Warning: Importing this module directly will set the global event loop. See `faust.loop` for more information.

1.5.4 Utils

mode.utils.aiter

Async iterator lost and found missing methods: aiter, anext, etc.

mode.utils.aiter.**aenumerate**(*it: AsyncIterable[Any], start: int = 0*) → AsyncIterator[Tuple[int, Any]]
`async for` version of `enumerate`.

mode.utils.aiter.**aiter**(*it: Any*) → AsyncIterator
Create iterator from iterable.

Notes

If the object is already an iterator, the iterator should return self when `__aiter__` is called.

class mode.utils.aiter.**arange**(**slice_args: int, **slice_kwargs: Any*)
Async generator that counts like `range`.

count(*n: int*) → int

index(*n: int*) → int

mode.utils.aiter.**aslice**(*ait, *slice_args*)
Extract slice from async generator.

mode.utils.aiter.**chunks**(*it: AsyncIterable, n: int*) → AsyncIterable[List]
Split an async iterator into chunks with *n* elements each.

Example

n == 2 >>> x = chunks(arange(10), 2) >>> [item async for item in x] [[0, 1], [2, 3], [4, 5], [6, 7], [8, 9], [10]]

n == 3 >>> x = chunks(arange(10)), 3) >>> [item async for item in x] [[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10]]

mode.utils.collections

Custom data structures.

class mode.utils.collections.**FastUserDict**
Proxy to dict.

Like `collection.UserDict` but reimplements some methods for better performance when the underlying dictionary is a real dict.

classmethod fromkeys(*iterable: Iterable[KT], value: VT = None*) → mode.utils.collections.FastUserSet

copy() → dict

update(*[E]*, ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

clear() → None. Remove all items from D.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

class mode.utils.collections.**FastUserSet**
Proxy to set.

copy() → Set[T]

difference(*other: Union[AbstractSet[T], Iterable[T]]*) → Set[T]

intersection(*other: Union[AbstractSet[T], Iterable[T]]*) → Set[T]

isdisjoint(*other: AbstractSet[T]*) → bool
Return True if two sets have a null intersection.

issubset(*other: AbstractSet[T]*) → bool

issuperset(*other: AbstractSet[T]*) → bool

symmetric_difference(*other: Union[AbstractSet[T], Iterable[T]]*) → Set[T]

union(*other: Union[AbstractSet[T], Iterable[T]]*) → Set[T]

add(*element: T*) → None
Add an element.

clear() → None
This is slow (creates N new iterators!) but effective.

difference_update(*other: Union[AbstractSet[T], Iterable[T]]*) → None

discard(*element: T*) → None
Remove an element. Do not raise an exception if absent.

intersection_update(*other: Union[AbstractSet[T], Iterable[T]]*) → None

pop() → T
Return the popped value. Raise KeyError if empty.

remove(*element: T*) → None
Remove an element. If not a member, raise a KeyError.

symmetric_difference_update(*other: Union[AbstractSet[T], Iterable[T]]*) → None

update(*other: Union[AbstractSet[T], Iterable[T]]*) → None

class mode.utils.collections.**FastUserList**(*initlist=None*)
Proxy to list.

class mode.utils.collections.**LRUCache**(*limit: int = None*, *, *thread_safety: bool = False*)
LRU Cache implementation using a doubly linked list to track access.

Parameters

- **limit** (*int*) – The maximum number of keys to keep in the cache. When a new key is inserted and the limit has been exceeded, the *Least Recently Used* key will be discarded from the cache.
- **thread_safety** (*bool*) – Enable if multiple OS threads are going to access/mutate the cache.

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

items() → a set-like object providing a view on D's items

incr(*key: KT*, *delta: int = 1*) → int

class mode.utils.collections.**ManagedUserSet**
A MutableSet that adds callbacks for when keys are get/set/deleted.

on_add(*value: T*) → None

on_discard(*value: T*) → None

on_clear() → None

on_change(*added: Set[T], removed: Set[T]*) → None

add(*element: T*) → None
Add an element.

clear() → None
This is slow (creates N new iterators!) but effective.

discard(*element: T*) → None
Remove an element. Do not raise an exception if absent.

pop() → T
Return the popped value. Raise KeyError if empty.

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

difference_update(*other: Union[AbstractSet[T], Iterable[T]]*) → None

intersection_update(*other: Union[AbstractSet[T], Iterable[T]]*) → None

symmetric_difference_update(*other: Union[AbstractSet[T], Iterable[T]]*) → None

update(*other: Union[AbstractSet[T], Iterable[T]]*) → None

class mode.utils.collections.**ManagedUserDict**
A UserDict that adds callbacks for when keys are get/set/deleted.

on_key_get(*key: KT*) → None
Handle that key is being retrieved.

on_key_set(*key: KT*, *value: VT*) → None
Handle that value for a key is being set.

on_key_del(*key: KT*) → None
Handle that a key is deleted.

on_clear() → None
Handle that the mapping is being cleared.

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

clear() → None. Remove all items from D.

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

class mode.utils.collections.**AttributeDictMixin**
Mixin for Mapping interface that adds attribute access.

I.e., *d.key -> d[key]*).

class mode.utils.collections.**AttributeDict**
Dict subclass with attribute access.

class mode.utils.collections.**DictAttribute**(*obj: Any*)
Dict interface to attributes.

obj[k] -> obj.k obj[k] = val -> obj.k = val

obj = None

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

mode.utils.collections.**force_mapping**(*m: Any*) → Mapping
Wrap object into supporting the mapping interface if necessary.

mode.utils.compat

Compatibility utilities.

mode.utils.compat.**current_task**()
Return the currently running task in an event loop or None.

By default the current task for the current event loop is returned.

None is returned when called not in the context of a Task.

class mode.utils.compat.**AsyncContextManager**

class mode.utils.compat.**ChainMap**(**maps*)

class mode.utils.compat.**Counter**(***kwds*)

class mode.utils.compat.**Deque**

mode.utils.compat.**OrderedDict**
alias of builtins.dict

mode.utils.compat.**want_bytes**(*s: AnyStr*) → bytes
Convert string to bytes.

mode.utils.compat.**want_str**(*s: AnyStr*) → str
Convert bytes to string.

mode.utils.compat.**isatty**(*fh: IO*) → bool
Return True if fh has a controlling terminal.

Notes

Use with e.g. sys.stdin.

class mode.utils.compat.**DummyContext**(*enter_result=None*)
Context for with-statement doing nothing.

mode.utils.contexts

Context manager utilities.

class mode.utils.contexts.**AbstractAsyncContextManager**
An abstract base class for asynchronous context managers.

class mode.utils.contexts.**AsyncExitStack**
Async *ExitStack*.

Context manager for dynamic management of a stack of exit callbacks.

Examples

```
async with AsyncExitStack() as stack:
    connections = [
        await stack.enter_async_context(get_connection())
        for i in range(5)
    ]
    # All opened connections will automatically be released at the
    # end of the async with statement, even if attempts to open a
    # connection later in the list raise an exception.
```

push_async_exit(*exit: Union[AsyncContextManager, Callable[..., Awaitable]]*) → Union[AsyncContextManager, Callable[..., Awaitable]]
Register coroutine with the standard __aexit__ method signature.

Can suppress exceptions the same way __aexit__ method can. Also accepts any object with an __aexit__ method (registering a call to the method instead of the object itself).

push_async_callback(*callback: Callable[..., Awaitable], *args: Any, **kwds: Any*) → Callable[..., Awaitable]
Register an arbitrary coroutine function and arguments.

Cannot suppress exceptions.

class mode.utils.contexts.**ExitStack**
Context manager for dynamic management of a stack of exit callbacks.

For example:

with ExitStack() as stack: files = [stack.enter_context(open(fname)) for fname in filenames] # All opened files will automatically be closed at the end of # the with statement, even if attempts to open files later # in the list raise an exception.

close() → None
Immediately unwind the context stack.

mode.utils.contexts.**asynccontextmanager**(*func*)
@asynccontextmanager decorator.

Typical usage:

```
@asynccontextmanager
async def some_async_generator(<arguments>):
    <setup>
    try:
        yield <value>
    finally:
        <cleanup>
```

This makes this:

```
async with some_async_generator(<arguments>) as <variable>:
    <body>
```

equivalent to this:

```
<setup>
try:
    <variable> = <value>
    <body>
finally:
    <cleanup>
```

class mode.utils.contexts.**nullcontext**(*enter_result=None*)
Context manager that does no additional processing.

Used as a stand-in for a normal context manager, when a particular block of code is only sometimes used with a normal context manager:

cm = optional_cm if condition else nullcontext() with cm:

Perform operation, using optional_cm if condition is True

class mode.utils.contexts.**asyncnullcontext**(*enter_result: Any = None*)
Context for async-with statement doing nothing.

mode.utils.futures

Async I/O Future utilities.

mode.utils.futures.**all_tasks**(*loop: asyncio.events.AbstractEventLoop*) → Set[_asyncio.Task]

mode.utils.futures.**current_task**()
Return the currently running task in an event loop or None.

By default the current task for the current event loop is returned.

None is returned when called not in the context of a Task.

class mode.utils.futures.**stampede**(*fget: Callable*, *, *doc: str = None*)
Descriptor for cached async operations providing stampede protection.

See also thundering herd problem.

Adding the decorator to an async callable method:

Examples

Here's an example coroutine method connecting a network client:

```
class Client:

    @stampede
    async def maybe_connect(self):
        await self._connect()

    async def _connect(self):
        return Connection()
```

In the above example, if multiple coroutines call `maybe_connect` at the same time, then only one of them will actually perform the operation. The rest of the coroutines will wait for the result, and return it once the first caller returns.

mode.utils.futures.**done_future**(*result: Any = None*, *, *loop: asyncio.events.AbstractEventLoop = None*) → _asyncio.Future
Return `asyncio.Future` that is already evaluated.

mode.utils.futures.**maybe_cancel**(*fut: _asyncio.Future*) → bool
Cancel future if it is cancellable.

mode.utils.futures.**notify**(*fut: Optional[_asyncio.Future]*, *result: Any = None*) → None
Set `asyncio.Future` result if future exists and is not done.

mode.utils.graphs

class mode.utils.graphs.**GraphFormatter**(*root: Any = None*, *type: str = None*, *id: str = None*, *indent: int = 0*, *inw: str = ' '*, ***scheme: Any*)
Format dependency graphs.

edge_scheme = {'arrowcolor': 'black', 'arrowsize': 0.7, 'color': 'darkseagreen4'}

node_scheme = {'color': 'palegreen4', 'fillcolor': 'palegreen3'}

term_scheme = {'color': 'palegreen2', 'fillcolor': 'palegreen1'}

scheme = {'arrowhead': 'vee', 'fontname': 'HelveticaNeue', 'shape': 'box', 'style':

graph_scheme = {'bgcolor': 'mintcream'}

attr(*name: str*, *value: Any*) → str

attrs(*d: Mapping = None*, *scheme: Mapping = None*) → str

head(***attrs: Any*) → str

tail() → str

label(*obj: _T*) → str

node(*obj: _T*, ***attrs: Any*) → str

terminal_node(*obj: _T*, ***attrs: Any*) → str

edge(*a: _T*, *b: _T*, ***attrs: Any*) → str

FMT(*fmt: str*, **args: Any*, ***kwargs: Any*) → str

draw_edge(*a: _T*, *b: _T*, *scheme: Mapping = None*, *attrs: Mapping = None*) → str

draw_node(*obj: _T*, *scheme: Mapping = None*, *attrs: Mapping = None*) → str

class mode.utils.graphs.**DependencyGraph**(*it: Iterable = None*, *formatter: mode.utils.types.graphs.GraphFormatterT[_T] = None*)

A directed acyclic graph of objects and their dependencies.

Supports a robust topological sort to detect the order in which they must be handled.

Takes an optional iterator of `(obj, dependencies)` tuples to build the graph from.

> **Warning:** Does not support cycle detection.

add_arc(*obj: _T*) → None
Add an object to the graph.

add_edge(*A: _T*, *B: _T*) → None
Add an edge from object `A` to object `B`.

I.e. `A` depends on `B`.

connect(*graph: mode.utils.types.graphs.DependencyGraphT[_T]*) → None
Add nodes from another graph.

topsort() → Sequence
Sort the graph topologically.

Returns of objects in the order in which they must be handled.

Return type List

valency_of(*obj: _T*) → int
Return the valency (degree) of a vertex in the graph.

update(*it: Iterable*) → None
Update graph with data from a list of `(obj, deps)` tuples.

edges() → Iterable
Return generator that yields for all edges in the graph.

to_dot(*fh: IO*, *, *formatter: mode.utils.types.graphs.GraphFormatterT[_T] = None*) → None
Convert the graph to DOT format.

Parameters

- **fh** (*IO*) – A file, or a file-like object to write the graph to.
- **formatter** (*celery.utils.graph.GraphFormatter*) – Custom graph formatter to use.

items() → a set-like object providing a view on D's items

mode.utils.imports

Importing utilities.

class mode.utils.imports.**FactoryMapping**(**args: Mapping, **kwargs: str*)

Class plugin system.

This is an utility to maintain a mapping from name to fully qualified Python attribute path, and also supporting the use of these in URLs.

Example

```
>>> # Specifying the type enables mypy to know that
>>> # this factory returns Driver subclasses.
>>> drivers: FactoryMapping[Type[Driver]]
>>> drivers = FactoryMapping({
...     'rabbitmq': 'my.drivers.rabbitmq:Driver',
...     'kafka': 'my.drivers.kafka:Driver',
...     'redis': 'my.drivers.redis:Driver',
... })
```

```
>>> drivers.by_url('rabbitmq://localhost:9090')
<class 'my.drivers.rabbitmq.Driver'>
```

```
>>> drivers.by_name('redis')
<class 'my.drivers.redis.Driver'>
```

iterate() → Iterator[_T]

by_url(*url: Union[str, mode.utils.imports.URL]*) → _T

Get class associated with URL (scheme is used as alias key).

by_name(*name: Union[_T, str]*) → _T

get_alias(*name: str*) → str

include_setuptools_namespace(*namespace: str*) → None

data

mode.utils.imports.**symbol_by_name**(*name: Union[_T, str], aliases: Mapping[str, str] = None, imp: Any = None, package: str = None, sep: str = '.', default: Any = None, **kwargs: Any*) → Any

Get symbol by qualified name.

The name should be the full dot-separated path to the class:

```
modulename.ClassName
```

Example:

```
mazecache.backends.redis.RedisBackend
                       ^- class name
```

or using ':' to separate module and symbol:

```
mazecache.backends.redis:RedisBackend
```

If *aliases* is provided, a dict containing short name/long name mappings, the name is looked up in the aliases first.

Examples

```
>>> symbol_by_name('mazecache.backends.redis:RedisBackend')
<class 'mazecache.backends.redis.RedisBackend'>
```

```
>>> symbol_by_name('default', {
...     'default': 'mazecache.backends.redis:RedisBackend'})
<class 'mazecache.backends.redis.RedisBackend'>
```

Does not try to look up non-string names. >>> from mazecache.backends.redis import RedisBackend >>> symbol_by_name(RedisBackend) is RedisBackend True

mode.utils.imports.**load_extension_classes**(*namespace: str*) → Iterable[mode.utils.imports.EntrypointExtension]

Yield extension classes for setuptools entrypoint namespace.

If an entrypoint is defined in `setup.py`:

```
entry_points={
    'faust.codecs': [
        'msgpack = faust_msgpack:msgpack',
    ],
```

Iterating over the 'faust.codecs' namespace will yield the actual attributes specified in the path (`faust_msgpack:msgpack`):

```
>>> from faust_msgpack import msgpack
>>> attrs = list(load_extension_classes('faust.codecs'))
assert msgpack in attrs
```

mode.utils.imports.**load_extension_class_names**(*namespace: str*) → Iterable[mode.utils.imports.RawEntrypointExtension]

Get setuptools entrypoint extension class names.

If the entrypoint is defined in `setup.py` as:

```
entry_points={
    'faust.codecs': [
        'msgpack = faust_msgpack:msgpack',
    ],
```

Iterating over the 'faust.codecs' namespace will yield the name:

```
>>> list(load_extension_class_names('faust.codecs'))
[('msgpack', 'faust_msgpack:msgpack')]
```

mode.utils.imports.**cwd_in_path**() → Generator

Context adding the current working directory to sys.path.

mode.utils.imports.**import_from_cwd**(*module: str*, *, *imp: Callable = None*, *package: str = None*) → module

Import module, temporarily including modules in the current directory.

Modules located in the current directory has precedence over modules located in *sys.path*.

mode.utils.imports.**smart_import**(*path: str*, *imp: Any = None*) → Any
Import module if module, otherwise same as *symbol_by_name()*.

mode.utils.logging

Logging utilities.

mode.utils.logging.**FormatterHandler**
alias of typing.Callable

mode.utils.logging.**get_logger**(*name: str*) → logging.Logger
Get logger by name.

class mode.utils.logging.**LogSeverityMixin**
Mixin class that delegates standard logging methods to logger.

The class that mixes in this class must define the `log` method.

Example

```
>>> class Foo(LogSeverityMixin):
...
...     logger = get_logger('foo')
...
...     def log(self,
...             severity: int,
...             msg: str,
...             *args: Any, **kwargs: Any) -> None:
...         return self.logger.log(severity, msg, *args, **kwargs)
```

dev(*msg: str*, **args: Any*, ***kwargs: Any*) → None

debug(*msg: str*, **args: Any*, ***kwargs: Any*) → None

info(*msg: str*, **args: Any*, ***kwargs: Any*) → None

warn(*msg: str*, **args: Any*, ***kwargs: Any*) → None

warning(*msg: str*, **args: Any*, ***kwargs: Any*) → None

error(*msg: str*, **args: Any*, ***kwargs: Any*) → None

crit(*msg: str*, **args: Any*, ***kwargs: Any*) → None

critical(*msg: str*, **args: Any*, ***kwargs: Any*) → None

exception(*msg: str*, **args: Any*, ***kwargs: Any*) → None

class mode.utils.logging.**CompositeLogger**(*logger: logging.Logger*, *formatter: Callable[..., str] = None*)
Composite logger for classes.

The class can be used as both mixin and composite, and may also define a `.formatter` attribute which will reformat any log messages sent.

Service uses this to add logging methods:

```
class Service(ServiceT):

    log: CompositeLogger
```

(continues on next page)

(continued from previous page)

```
    def __init__(self):
        self.log = CompositeLogger(
            logger=self.logger,
            formatter=self._format_log,
        )

    def _format_log(self, severity: int, msg: str,
                    *args: Any, **kwargs: Any) -> str:
        return (f'[^{"-" * (self.beacon.depth - 1)}'
                f'{self.shortlabel}]: {msg}')
```

This means those defining a service may also use it to log:

```
>>> service.log.info('Something happened')
```

and when logging additional information about the service is automatically included.

log(*severity: int*, *msg: str*, **args: Any*, ***kwargs: Any*) → None

format(*severity: int*, *msg: str*, **args: Any*, ***kwargs: Any*) → str

mode.utils.logging.**formatter**(*fun: Callable[Any, Any]*) → Callable[Any, Any]
Register formatter for logging positional args.

class mode.utils.logging.**ExtensionFormatter**(*stream: IO = None*, ***kwargs: Any*)
Formatter that can register callbacks to format args.

Extends colorlog.

format(*record: logging.LogRecord*) → str
Format a message from a record object.

mode.utils.logging.**level_name**(*loglevel: int*) → str
Convert log level to number.

mode.utils.logging.**level_number**(*loglevel: int*) → int
Convert log level number to name.

mode.utils.logging.**setup_logging**(***, *loglevel: Union[str, int] = None*, *logfile: Union[str, IO] = None*, *loghandlers: List[logging.StreamHandler] = None*, *logging_config: Dict = None*) → int
Configure logging subsystem.

class mode.utils.logging.**Logwrapped**(*obj: Any*, *logger: Any = None*, *severity: Union[int, str] = None*, *ident: str = ''*)
Wrap all object methods, to log on call.

mode.utils.logging.**cry**(*file: IO*, ***, *sep1: str = '='*, *sep2: str = '-'*, *sep3: str = '~'*, *seplen: int = 49*) → None
Return stack-trace of all active threads.

See also:

Taken from https://gist.github.com/737056.

class mode.utils.logging.**flight_recorder**(*logger: Any*, ***, *timeout: Union[datetime.timedelta, float, str]*, *loop: asyncio.events.AbstractEventLoop = None*)
Flight Recorder context for use with `with` statement.

This is a logging utility to log stuff only when something times out.

For example if you have a background thread that is sometimes hanging:

```
class RedisCache(mode.Service):

    @mode.timer(1.0)
    def _background_refresh(self) -> None:
        self._users = await self.redis_client.get(USER_KEY)
        self._posts = await self.redis_client.get(POSTS_KEY)
```

You want to figure out on what line this is hanging, but logging all the time will provide way too much output, and will even change how fast the program runs and that can mask race conditions, so that they never happen.

Use the flight recorder to save the logs and only log when it times out:

```
logger = mode.get_logger(__name__)

class RedisCache(mode.Service):

    @mode.timer(1.0)
    def _background_refresh(self) -> None:
        with mode.flight_recorder(logger, timeout=10.0) as on_timeout:
            on_timeout.info(f'+redis_client.get({USER_KEY!r})')
            await self.redis_client.get(USER_KEY)
            on_timeout.info(f'-redis_client.get({USER_KEY!r})')

            on_timeout.info(f'+redis_client.get({POSTS_KEY!r})')
            await self.redis_client.get(POSTS_KEY)
            on_timeout.info(f'-redis_client.get({POSTS_KEY!r})')
```

If the body of this `with` statement completes before the timeout, the logs are forgotten about and never emitted – if it takes more than ten seconds to complete, we will see these messages in the log:

```
[2018-04-19 09:43:55,877: WARNING]: Warning: Task timed out!
[2018-04-19 09:43:55,878: WARNING]:
    Please make sure it is hanging before restarting.
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1]
    (started at Thu Apr 19 09:43:45 2018) Replaying logs...
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1]
    (Thu Apr 19 09:43:45 2018) +redis_client.get('user')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1]
    (Thu Apr 19 09:43:49 2018) -redis_client.get('user')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1]
    (Thu Apr 19 09:43:46 2018) +redis_client.get('posts')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1] -End of log-
```

Now we know this `redis_client.get` call can take too long to complete, and should consider adding a timeout to it.

wrap_debug(*obj: Any*) → mode.utils.logging.Logwrapped

wrap_info(*obj: Any*) → mode.utils.logging.Logwrapped

wrap_warn(*obj: Any*) → mode.utils.logging.Logwrapped

wrap_error(*obj: Any*) → mode.utils.logging.Logwrapped

wrap(*severity: int*, *obj: Any*) → mode.utils.logging.Logwrapped

activate() → None

cancel() → None

log(*severity: int*, *message: str*, **args: Any*, ***kwargs: Any*) → None

class mode.utils.logging.**FileLogProxy**(*logger: logging.Logger*, ***, *severity: Union[int, str] = None*)

File-like object that forwards data to logger.

mode = 'w'

name = None

closed = False

severity = 30

write(*data: Any*) → None

writelines(*lines: Sequence[str]*) → None

flush() → None

close() → None

isatty() → bool

mode.utils.logging.**redirect_stdouts**(*logger: logging.Logger = <Logger mode.redirect (WARNING)>*, ***, *severity: Union[int, str] = None*, *stdout: bool = True*, *stderr: bool = True*) → ContextManager[mode.utils.logging.FileLogProxy]

Redirect sys.stdout and sys.stdout to logger.

mode.utils.loops

Event loop utilities.

mode.utils.loops.**clone_loop**(*loop: asyncio.events.AbstractEventLoop*) → asyncio.events.AbstractEventLoop

Clone loop retaining signal handlers.

mode.utils.loops.**call_asap**(*callback: Callable*, **args: Any*, *context: Any = None*, *loop: asyncio.events.AbstractEventLoop = None*) → asyncio.events.Handle

Call function asap by pushing at the front of the line.

mode.utils.mocks

Mocking and testing utilities.

class mode.utils.mocks.**Mock**(*spec=None*, *side_effect=None*, *return_value=sentinel.DEFAULT*, *wraps=None*, *name=None*, *spec_set=None*, *parent=None*, *_spec_state=None*, *_new_name=''*, *_new_parent=None*, ***kwargs*)

Mock object.

global_call_count = None

call_counts = None

reset_mock(**args*, ***kwargs*)

Restore the mock object to its initial state.

class mode.utils.mocks.**AsyncMock**(**args: Any*, *name: str = None*, ***kwargs: Any*)

Mock for async def function/method or anything awaitable.

class mode.utils.mocks.**AsyncMagicMock**(**args: Any*, *name: str = None*, ***kwargs: Any*)

A magic mock type for async def functions/methods.

class mode.utils.mocks.**AsyncContextManagerMock**(**args: Any*, *aenter_return: Any = None*, *aexit_return: Any = None*, ***kwargs: Any*)

Mock for typing.AsyncContextManager.

You can use this to mock asynchronous context managers, when an object with a fully defined __aenter__ and __aexit__ is required.

Here's an example mocking an aiohttp client:

```
import http
from aiohttp.client import ClientSession
from aiohttp.web import Response
from mode.utils.mocks import AsyncContextManagerMock, AsyncMock, Mock

@pytest.fixture()
def session(monkeypatch):
    session = Mock(
        name='http_client',
        autospec=ClientSession,
        request=Mock(
            return_value=AsyncContextManagerMock(
                return_value=Mock(
                    autospec=Response,
                    status=http.HTTPStatus.OK,
                    json=AsyncMock(
                        return_value={'hello': 'json'},
                    ),
                ),
            ),
        ),
    )
    monkeypatch.setattr('where.is.ClientSession', session)
    return session

@pytest.mark.asyncio
async def test_session(session):
    from where.is import ClientSession
    session = ClientSession()
    async with session.get('http://example.com') as response:
        assert response.status == http.HTTPStatus.OK
        assert await response.json() == {'hello': 'json'}
```

class mode.utils.mocks.**FutureMock**(**args*, ***kwargs*)

Mock a asyncio.Future.

awaited = False

assert_awaited()

assert_not_awaited()

class mode.utils.mocks.**MagicMock**(**args*, ***kw*)

MagicMock is a subclass of Mock with default implementations of most of the magic methods. You can use MagicMock without having to configure the magic methods yourself.

If you use the *spec* or *spec_set* arguments then *only* magic methods that exist in the spec will be created.

Attributes and the return value of a *MagicMock* will also be *MagicMocks*.

mock_add_spec(*spec*, *spec_set=False*)

Add a spec to a mock. *spec* can either be an object or a list of strings. Only attributes on the *spec* can be fetched as attributes from the mock.

If *spec_set* is True then only attributes on the spec can be set.

mode.utils.mocks.**call**

A tuple for holding the results of a call to a mock, either in the form *(args, kwargs)* or *(name, args, kwargs)*.

If args or kwargs are empty then a call tuple will compare equal to a tuple without those values. This makes comparisons less verbose:

```
_Call(('name', (), {})) == ('name',)
_Call(('name', (1,), {})) == ('name', (1,))
_Call(((), {'a': 'b'})) == ({'a': 'b'},)
```

The *_Call* object provides a useful shortcut for comparing with call:

```
_Call(((1, 2), {'a': 3})) == call(1, 2, a=3)
_Call(('foo', (1, 2), {'a': 3})) == call.foo(1, 2, a=3)
```

If the _Call has no name then it will match any name.

mode.utils.mocks.**patch**(*target*, *new=sentinel.DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with a *MagicMock*. If *patch* is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If *patch* is used as a context manager the created mock is returned by the context manager.

target should be a string in the form *'package.module.ClassName'*. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling *patch* from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the *MagicMock* if patch is creating one for you.

In addition you can pass *spec=True* or *spec_set=True*, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *MagicMock* is used.

A more powerful form of *spec* is *autospec*. If you set *autospec=True* then the mock will be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a *TypeError* if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class.

Instead of *autospec=True* you can pass *autospec=some_object* to use an arbitrary object as the spec instead of the one being replaced.

By default *patch* will fail to replace attributes that don't exist. If you pass in *create=True*, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Patch can be used as a *TestCase* class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. *patch* finds tests by looking

for method names that start with *patch.TEST_PREFIX*. By default this is *test*, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting *patch.TEST_PREFIX*.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if *patch* is creating a mock object for you.

patch takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction.

patch.dict(...), *patch.multiple(...)* and *patch.object(...)* are available for alternate use-cases.

mode.utils.objects

Object utilities.

mode.utils.objects.**FieldMapping**
alias of typing.Mapping

mode.utils.objects.**DefaultsMapping**
alias of typing.Mapping

exception mode.utils.objects.**InvalidAnnotation**
Raised by *annotations()* when encountering an invalid type.

class mode.utils.objects.**Unordered**(*value: _T*)
Shield object from being ordered in heapq/__le__/etc.

class mode.utils.objects.**KeywordReduce**
Mixin class for objects that can be "pickled".

"Pickled" means the object can be serialiazed using the Python binary serializer – the pickle module.

Python objects are made pickleable through defining the __reduce__ method, that returns a tuple of: (restore_function, function_starargs):

```
class X:

    def __init__(self, arg1, kw1=None):
        self.arg1 = arg1
        self.kw1 = kw1

    def __reduce__(self) -> Tuple[Callable, Tuple[Any, ...]]:
        return type(self), (self.arg1, self.kw1)
```

This is *tedious* since this means you cannot accept **kwargs in the constructur, so what we do is define a __reduce_keywords__ argument that returns a dict instead:

```
class X:

    def __init__(self, arg1, kw1=None):
        self.arg1 = arg1
        self.kw1 = kw1

    def __reduce_keywords__(self) -> Mapping[str, Any]:
        return {
            'arg1': self.arg1,
            'kw1': self.kw1,
        }
```

mode.utils.objects.**qualname**(*obj: Any*) → str
Get object qualified name.

mode.utils.objects.**shortname**(*obj: Any*) → str
Get object name (non-qualified).

mode.utils.objects.**canoname**(*obj: Any*, *, *main_name: str = None*) → str
Get qualname of obj, trying to resolve the real name of __main__.

mode.utils.objects.**canonshortname**(*obj: Any*, *, *main_name: str = None*) → str
Get non-qualified name of obj, resolve real name of __main__.

mode.utils.objects.**annotations**(*cls: Type*, *, *stop: Type = <class 'object'>*, *invalid_types: Set = None*, *alias_types: Mapping = None*, *skip_classvar: bool = False*, *globalns: Dict[str, Any] = None*, *localns: Dict[str, Any] = None*) → Tuple[Mapping[str, Type], Mapping[str, Any]]
Get class field definition in MRO order.

Parameters

- **cls** – Class to get field information from.
- **stop** – Base class to stop at (default is object).
- **invalid_types** – Set of types that if encountered should raise *InvalidAnnotation* (does not test for subclasses).
- **alias_types** – Mapping of original type to replacement type.
- **globalns** – Global namespace to use when evaluating forward references (see typing.ForwardRef).
- **localns** – Local namespace to use when evaluating forward references (see typing.ForwardRef).

Returns

Tuple with two dictionaries, the first containing a map of field names to their types, the second containing a map of field names to their default value. If a field is not in the second map, it means the field is required.

Return type Tuple[FieldMapping, DefaultsMapping]

Raises ***InvalidAnnotation*** – if a list of invalid types are provided and an invalid type is encountered.

Examples

```
>>> class Point:
...     x: float
...     y: float

>>> class 3DPoint(Point):
...      z: float = 0.0

>>> fields, defaults = annotations(3DPoint)
>>> fields
{'x': float, 'y': float, 'z': 'float'}
>>> defaults
{'z': 0.0}
```

mode.utils.objects.**eval_type**(*typ: Any*, *globalns: Dict[str, Any] = None*, *localns: Dict[str, Any] = None*, *invalid_types: Set = None*, *alias_types: Mapping = None*) → Type

Convert (possible) string annotation to actual type.

Examples

```
>>> eval_type('List[int]') == typing.List[int]
```

mode.utils.objects.**iter_mro_reversed**(*cls: Type*, *stop: Type*) → Iterable[Type]

Iterate over superclasses, in reverse Method Resolution Order.

The stop argument specifies a base class that when seen will stop iterating (well actually start, since this is in reverse, see Example for demonstration).

Parameters

- **cls** (*Type*) – Target class.
- **stop** (*Type*) – A base class in which we stop iteration.

Notes

The last item produced will be the class itself (*cls*).

Examples

```
>>> class A: ...
>>> class B(A): ...
>>> class C(B): ...
```

```
>>> list(iter_mro_reverse(C, object))
[A, B, C]
```

```
>>> list(iter_mro_reverse(C, A))
[B, C]
```

Yields *Iterable[Type]* – every class.

mode.utils.objects.**guess_polymorphic_type**(*typ: Type*, ***, *set_types: Tuple[Type, ...] = (typing.AbstractSet, typing.FrozenSet, typing.MutableSet, typing.Set)*, *list_types: Tuple[Type, ...] = (typing.List, typing.Sequence, typing.MutableSequence)*, *tuple_types: Tuple[Type, ...] = (typing.Tuple,)*, *dict_types: Tuple[Type, ...] = (typing.Dict, typing.Mapping, typing.MutableMapping)*) → Tuple[Type, Type]

Try to find the polymorphic and concrete type of an abstract type.

Returns tuple of `(polymorphic_type, concrete_type)`.

Examples

```
>>> guess_polymorphic_type(List[int])
(list, int)
```

```
>>> guess_polymorphic_type(Optional[List[int]])
(list, int)
```

```
>>> guess_polymorphic_type(MutableMapping[int, str])
(dict, str)
```

mode.utils.objects.**label**(*s: Any*) → str
Return the name of an object as string.

mode.utils.objects.**shortlabel**(*s: Any*) → str
Return the shortened name of an object as string.

class mode.utils.objects.**cached_property**(*fget: Callable[Any, RT], fset: Callable[[Any, RT], None] = None, fdel: Callable[[Any, RT], None] = None, doc: str = None, class_attribute: str = None*)
Cached property.

A property descriptor that caches the return value of the get function.

Examples

```
@cached_property
def connection(self):
    return Connection()

@connection.setter  # Prepares stored value
def connection(self, value):
    if value is None:
        raise TypeError('Connection must be a connection')
    return value

@connection.deleter
def connection(self, value):
    # Additional action to do at del(self.attr)
    if value is not None:
        print(f'Connection {value!r} deleted')
```

is_set(*obj: Any*) → bool

setter(*fset: Callable[[Any, RT], None]*) → mode.utils.objects.cached_property

deleter(*fdel: Callable[[Any, RT], None]*) → mode.utils.objects.cached_property

mode.utils.queues

Queue utilities - variations of asyncio.Queue.

class mode.utils.queues.**FlowControlEvent**(**, initially_suspended: bool = True, loop: asyncio.events.AbstractEventLoop = None*)
Manage flow control *FlowControlQueue* instances.

The FlowControlEvent manages flow in one or many queue instances at the same time.

To flow control queues, first create the shared event:

```
>>> flow_control = FlowControlEvent()
```

Then pass that shared event to the queues that should be managed by it:

```
>>> q1 = FlowControlQueue(maxsize=1, flow_control=flow_control)
>>> q2 = FlowControlQueue(flow_control=flow_control)
```

If you want the contents of the queue to be cleared when flow is resumed, then specify that by using the `clear_on_resume` flag:

```
>>> q3 = FlowControlQueue(clear_on_resume=True,
...                       flow_control=flow_control)
```

To suspend production into queues, use `flow_control.suspend`:

```
>>> flow_control.suspend()
```

While the queues are suspend, any producer attempting to send something to the queue will hang until flow is resumed.

To resume production into queues, use `flow_control.resume`:

```
>>> flow_control.resume()
```

Notes

In Faust queues are managed by the `app.flow_control` event.

manage_queue(*queue: mode.utils.queues.FlowControlQueue*) → None
Add *FlowControlQueue* to be cleared on resume.

suspend() → None
Suspend production into queues managed by this event.

resume() → None
Resume production into queues managed by this event.

is_active() → bool

clear() → None

class mode.utils.queues.**FlowControlQueue**(*maxsize: int = 0, *, flow_control: mode.utils.queues.FlowControlEvent = None, clear_on_resume: bool = False, **kwargs: Any*)
`asyncio.Queue` managed by *FlowControlEvent*.

See also:

FlowControlEvent.

clear() → None

class mode.utils.queues.**ThrowableQueue**(**args: Any, **kwargs: Any*)
Queue that can be notified of errors.

empty() → bool
Return True if the queue is empty, False otherwise.

clear() → None

get_nowait() → _T
Remove and return an item from the queue.

Return an item if one is immediately available, else raise QueueEmpty.

mode.utils.text

Text and string manipulation utilities.

class mode.utils.text.**FuzzyMatch**
Fuzzy match resut.

property ratio
Alias for field number 0

property value
Alias for field number 1

mode.utils.text.**title**(*s: str*) → str
Capitalize sentence.

```
"foo bar" -> "Foo Bar"
```

```
"foo-bar" -> "Foo Bar"
```

mode.utils.text.**didyoumean**(*haystack: Iterable[str], needle: str, *, fmt_many: str = 'Did you mean one of {alt}?', fmt_one: str = 'Did you mean {alt}?', fmt_none: str = '', min_ratio: float = 0.6*) → str
Generate message with helpful list of alternatives.

Examples

```
>>> raise Exception(f'Unknown mode: {mode}! {didyoumean(modes, mode)}')
```

```
>>> didyoumean(['foo', 'bar', 'baz'], 'boo')
'Did you mean foo?'
```

```
>>> didyoumean(['foo', 'moo', 'bar'], 'boo')
'Did you mean one of foo, moo?'
```

```
>>> didyoumean(['foo', 'moo', 'bar'], 'xxx')
''
```

Parameters

- **haystack** – List of all available choices.
- **needle** – What the user provided.
- **fmt_many** – String format returned when there are more than one alternative. Default is: `"Did you mean one of {alt}?"`.
- **fmt_one** – String format returned when there's a single fuzzy match. Default is: `"Did you mean {alt}?"`.
- **fmt_none** – String format returned when there are no fuzzy matches. Default is: `""` (empty string, error message is usually printed before the alternatives so user has context).

- **min_ratio** – Minimum fuzzy ratio before word is considered a match. Default is 0.6.

mode.utils.text.**fuzzymatch_choices**(*haystack: Iterable[str], needle: str, *, fmt_many: str = 'one of {alt}', fmt_one: str = '{alt}', fmt_none: str = '', min_ratio: float = 0.6*) → str
Fuzzy match reducing to error message suggesting an alternative.

mode.utils.text.**fuzzymatch_iter**(*haystack: Iterable[str], needle: str, *, min_ratio: float = 0.6*) → Iterator[mode.utils.text.FuzzyMatch]
Fuzzy Match: Including actual ratio.

Yields *FuzzyMatch* – tuples of `(ratio, value)`.

mode.utils.text.**fuzzymatch_best**(*haystack: Iterable[str], needle: str, *, min_ratio: float = 0.6*) → Optional[str]
Fuzzy Match - Return best match only (single scalar value).

mode.utils.text.**abbr**(*s: str*, *max: int*, *suffix: str = '...'*, *words: bool = False*) → str
Abbreviate word.

mode.utils.text.**abbr_fqdn**(*origin: str*, *name: str*, ***, *prefix: str = ''*) → str
Abbreviate fully-qualified Python name, by removing origin.

`app.origin` is the package where the app is defined, so if this is `examples.simple`:

```
>>> app.origin
'examples.simple'
>>> abbr_fqdn(app.origin, 'examples.simple.Withdrawal', prefix='[...]')
'[...]Withdrawal'

>>> abbr_fqdn(app.origin, 'examples.other.Foo', prefix='[...]')
'examples.other.foo'
```

shorten_fqdn() is similar, but will always shorten a too long name, abbr_fqdn will only remove the origin portion of the name.

mode.utils.text.**shorten_fqdn**(*s: str*, *max: int = 32*) → str
Shorten fully-qualified Python name (like "os.path.isdir").

mode.utils.text.**pluralize**(*n: int*, *text: str*, *suffix: str = 's'*) → str
Pluralize term when n is greater than one.

mode.utils.text.**maybecat**(*s: Optional[AnyStr]*, *suffix: str = ''*, ***, *prefix: str = ''*) → AnyStr
Concatenate string only if existing string s' is defined.

Keyword Arguments

- **suffix** – add suffix if string s' is defined.
- **prefix** – add prefix is string s' is defined.

mode.utils.times

Time, date and timezone related utilities.

mode.utils.times.**Seconds = typing.Union[datetime.timedelta, float, str]**
Seconds can be expressed as float or `timedelta`,

class mode.utils.times.**Bucket**(*rate: Union[datetime.timedelta, float, str], over: Union[datetime.timedelta, float, str] = 1.0, *, fill_rate: Union[datetime.timedelta, float, str] = None, capacity: Union[datetime.timedelta, float, str] = None, raises: Type[BaseException] = None, loop: asyncio.events.AbstractEventLoop = None*)

Rate limiting state.

A bucket "pours" tokens at a rate of `rate` per second (or over').

Calling *bucket.pour()*, pours one token by default, and returns `True` if that amount can be poured now, or `False` if the caller has to wait.

If this returns `False`, it's prudent to either sleep or raise an exception:

```
if not bucket.pour():
    await asyncio.sleep(bucket.expected_time())
```

If you want to consume multiple tokens in one go then specify the number:

```
if not bucket.pour(10):
    await asyncio.sleep(bucket.expected_time(10))
```

This class can also be used as an async. context manager, but in that case can only consume one tokens at a time:

```
async with bucket:
    # do something
```

By default the async. context manager will suspend the current coroutine and sleep until as soon as the time that a token can be consumed.

If you wish you can also raise an exception, instead of sleeping, by providing the `raises` keyword argument:

```
# hundred tokens in one second, and async with: raises TimeoutError

class MyError(Exception):
    pass

bucket = Bucket(100, over=1.0, raises=MyError)

async with bucket:
    # do something
```

abstract pour(*tokens: int = 1*) → bool

abstract expected_time(*tokens: int = 1*) → float

abstract property tokens

property fill_rate

class mode.utils.times.**TokenBucket**(*rate: Union[datetime.timedelta, float, str], over: Union[datetime.timedelta, float, str] = 1.0, *, fill_rate: Union[datetime.timedelta, float, str] = None, capacity: Union[datetime.timedelta, float, str] = None, raises: Type[BaseException] = None, loop: asyncio.events.AbstractEventLoop = None*)

Rate limiting using the token bucket algorithm.

pour(*tokens: int = 1*) → bool

expected_time(*tokens: int = 1*) → float

property tokens

mode.utils.times.**rate**(*r: float*) → float
Convert rate string (*"100/m"*, *"2/h"* or *"0.5/s"*) to seconds.

mode.utils.times.**rate_limit**(*rate: float*, *over: Union[datetime.timedelta, float, str] = 1.0*, ***, *bucket_type: Type[mode.utils.times.Bucket] = mode.utils.times.TokenBucket*, *raises: Type[BaseException] = None*, *loop: asyncio.events.AbstractEventLoop = None*) → mode.utils.times.Bucket
Create rate limiting manager.

mode.utils.times.**want_seconds**(*s: float*) → float
Convert *Seconds* to float.

mode.utils.tracebacks

Traceback utilities.

mode.utils.tracebacks.**print_task_stack**(*task: _asyncio.Task*, ***, *file: IO = <_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>*, *limit: int = 125*, *capture_locals: bool = False*) → None
Print the stack trace for an asyncio.Task.

mode.utils.tracebacks.**format_task_stack**(*task: _asyncio.Task*, ***, *limit: int = 125*) → None
Format asyncio.Task stack trace as a string.

class mode.utils.tracebacks.**Traceback**(*frame: frame*, *lineno: int = None*, *lasti: int = None*)
Traceback object with truncated frames.

classmethod from_task(*task: _asyncio.Task*, ***, *limit: int = 125*) → mode.utils.tracebacks.Traceback

classmethod from_coroutine(*coro: Union[Coroutine, Generator]*, ***, *depth: int = 0*, *limit: int = 125*) → mode.utils.tracebacks.Traceback

mode.utils.trees

Data structure: Trees.

class mode.utils.trees.**Node**(*data: _T*, ***, *root: mode.utils.types.trees.NodeT = None*, *parent: mode.utils.types.trees.NodeT = None*, *children: List[mode.utils.types.trees.NodeT[_T]] = None*)
Tree node.

Notes

Nodes have a link to

- the .root node (or None if this is the top-most node)
- the .parent node (if this is a child node).
- a list of children

A Node may have arbitrary .data associated with it, and arbitrary data may also be stored in .children.

Parameters **data** (*Any*) – Data to associate with node.

Keyword Arguments

- **root** (NodeT) – Root node.
- **parent** (NodeT) – Parent node.
- **children** (*List[NodeT]*) – List of child nodes.

new(*data: _T*) → mode.utils.types.trees.NodeT
Create new node from this node.

reattach(*parent: mode.utils.types.trees.NodeT[_T]*) → mode.utils.types.trees.NodeT[_T]
Attach this node to *parent* node.

add(*data: _T*) → None
Add node as a child node.

discard(*data: _T*) → None
Remove node so it's no longer a child of this node.

traverse() → Iterator[mode.utils.types.trees.NodeT[_T]]
Iterate over the tree in BFS order.

walk() → Iterator[mode.utils.types.trees.NodeT[_T]]
Iterate over hierarchy backwards.

This will yield parent nodes all the way up to the root.

as_graph() → mode.utils.types.graphs.DependencyGraphT
Convert to *DependencyGraph*.

property depth

property path

property parent

property root

mode.utils.types.graphs

Type classes for *mode.utils.graphs*.

class mode.utils.types.graphs.**GraphFormatterT**(*root: Any = None*, *type: str = None*, *id: str = None*, *indent: int = 0*, *inw: str = ' '*, ***scheme: Any*)

Type class for graph formatters.

abstract attr(*name: str*, *value: Any*) → str

abstract attrs(*d: Mapping = None*, *scheme: Mapping = None*) → str

abstract head(***attrs: Any*) → str

abstract tail() → str

abstract label(*obj: _T*) → str

abstract node(*obj: _T*, ***attrs: Any*) → str

abstract terminal_node(*obj: _T*, ***attrs: Any*) → str

abstract edge(*a: _T*, *b: _T*, ***attrs: Any*) → str

abstract FMT(*fmt: str*, **args: Any*, ***kwargs: Any*) → str

abstract draw_edge(*a: _T*, *b: _T*, *scheme: Mapping = None*, *attrs: Mapping = None*) → str

abstract draw_node(*obj: _T*, *scheme: Mapping = None*, *attrs: Mapping = None*) → str

class mode.utils.types.graphs.**DependencyGraphT**(*it: Iterable[_T] = None*, *formatter: mode.utils.types.graphs.GraphFormatterT[_T] = None*)

Type class for dependency graphs.

abstract add_arc(*obj: _T*) → None

abstract add_edge(*A: _T*, *B: _T*) → None

abstract connect(*graph: mode.utils.types.graphs.DependencyGraphT*) → None

abstract topsort() → Sequence

abstract valency_of(*obj: _T*) → int

abstract update(*it: Iterable*) → None

abstract edges() → Iterable

abstract to_dot(*fh: IO*, ***, *formatter: mode.utils.types.graphs.GraphFormatterT[_T] = None*) → None

mode.utils.types.trees

Type classes for *mode.utils.trees*.

class mode.utils.types.trees.**NodeT**

Node in a tree data structure.

children = None

data = None

abstract new(*data: _T*) → mode.utils.types.trees.NodeT

abstract add(*data: _T*) → None

abstract discard(*data: _T*) → None

abstract reattach(*parent: mode.utils.types.trees.NodeT*) → mode.utils.types.trees.NodeT

abstract traverse() → Iterator[mode.utils.types.trees.NodeT]

abstract walk() → Iterator[mode.utils.types.trees.NodeT]

abstract as_graph() → mode.utils.types.graphs.DependencyGraphT

abstract property parent

abstract property root

abstract property depth

abstract property path

mode.utils.typing

Backport of `typing` additions in Python 3.7.

class mode.utils.typing.**AsyncContextManager**

class mode.utils.typing.**ChainMap**(**maps*)

class mode.utils.typing.**Counter**(***kwds*)

class mode.utils.typing.**Deque**

1.6 Change history

1.6.1 3.1.3

release-date 2019-04-04 08:37 P.M PST

release-by Ask Solem (@ask)

- `mode.utils.worker.exiting` now takes option to print exceptions.
- Threads: Method queue "starting…" logs now logged with debug severity.
- Worker: execute_from_commandline no longer swallow errors if loop closed.
- Adds *mode.locals.LocalStack*.

1.6.2 3.1.2

release-date 2019-04-04 08:37 P.M PST

release-by Ask Solem (@ask)

- **Revoked release**: Version without changelog entry was uploaded to PyPI. Please upgrade to 3.1.3.

1.6.3 3.1.1

release-date 2019-03-27 10:02 A.M PST

release-by Ask Solem (@ask)

- Service: property `should_stop` is now true if service crashed.
- Timers: Avoid drift + introduce a tiny bit of drift to timers.

  Thanks to Bob Haddleton (@bobh66) for discovering this issue.

1.6.4 3.1.0

release-date 2019-03-21 03:26 P.M PST

release-by Ask Solem (@ask)

- Adds *nullcontext* and *asyncnullcontext*.

  Backported from Python 3.7 you can import these from *mode.utils.contexts*.
- Mode project changes:
  - Added bandit to CI lint build.
  - Added pydocstyle to CI lint build.

1.6.5 3.0.13

release-date 2019-03-20 04:58 P.M PST

release-by Ask Solem (@ask)

- Adds `CompositeLogger.warning` alias to `warn`.

  flake8-logging-format has a rule that says you are only allowed to use `.warning`, so going with that.

1.6.6 3.0.12

release-date 2019-03-20 03:23 P.M PST

release-by Ask Solem (@ask)

- Adds *`all_tasks()`* as a backward compatible `asyncio.all_tasks()`.
- Signal: Fixes `.connect()` decorator to work with parens and without

  Signal decorator now works with parens:

```
@signal.connect()
def my_handler(sender, **kwargs):
    ...
```

  and without parens:

```
@signal.connect
def my_handler(sender, **kwargs):
...
```

- Signal: Do not use weakref by default.

  Using weakref by default meant it was too easy to connect a signal handler to only have it disappear because there were no more references to the object.

1.6.7 3.0.11

release-date 2019-03-19 08:50 A.M PST

release-by Ask Solem (@ask)

- Adds ThrowableQueue._throw() for non-async version of .throw().

1.6.8 3.0.10

release-date 2019-03-14 03:55 P.M PST

release-by Ask Solem (@ask)

- Worker: was giving successful exit code when an exception is raised.

  The `.execute_from_commandline` method now always exits (its return type is `typing.NoReturn`).

- Adds `NoReturn` to *`mode.utils.compat`*.

  Import `typing.NoReturn` from here to support Python versions before 3.6.3.

1.6.9 3.0.9

release-date 2019-03-08 01:20 P.M PDT

release-by Ask Solem (@ask)

- Threads: Add multiple workers for thread method queue.

  Default number of workers is now 2, to allow for two recursive calls.

- Signal: Use *Signal.label* instead of `.indent` to be more consistent.
- Signal: Properly set `.name` when signal is member of class.
- Adds ability to log the FULL traceback of `asyncio.Task`.
- Service: Stop faster if stopped immediately after start
- Service: Correctly track dependencies for services added using `Service.on_init_dependencies` (Issue #40).

  Contributed by Nimi Wariboko Jr (@nemosupremo).

1.6.10 3.0.8

release-date 2019-01-25 03:54 P.M PDT

release-by Ask Solem (@ask)

- Fixes `DeprecationWarning` importing from `collections`.
- stampede: Fixed edge case where stampede wrapped function called multiple times.

  Calling the same stampede wrapped function multiple times within the same event loop iteration would previously call the function multiple times.

  For example using `asyncio.gather()`:

  Previously this would call the function four times, but with the fix it's only called once and provides the expected result.

- **Mocks: Adds `mask_module()` and** `patch_module()`.
- CI: Added Windows build.
- CI: Enabled random order for tests.

1.6.11 3.0.7

release-date 2019-01-18 01:12 P.M PDT

release-by Ask Solem (@ask)

- **ServiceThread** `.stop()` would wait for thread shutdown even if thread was never started.
- **CI**: Adds CPython 3.7.2 and 3.6.8 to build matrix

1.6.12 3.0.6

release-date 2019-01-07 12:10 P.M PDT

release-by Ask Solem (@ask)

- Adds `%(extra)s` as log format option.

  To add additional context to your logging statements use for example:

```
logger.error('Foo', extra={'data': {'database': 'db1'}})
```

1.6.13 3.0.5

release-date 2018-12-19 04:40 P.M PDT

release-by Ask Solem (@ask)

- Fixes compatibility with colorlog 4.0.x.

  Contributed by Ryan Whitten (@rwhitten577).

1.6.14 3.0.4

release-date 2018-12-07 04:40 P.M PDT

release-by Ask Solem (@ask)

- Now depends on mypy_extensions.

1.6.15 3.0.3

release-date 2018-12-07 3:22 P.M PDT

release-by Ask Solem (@ask)

- Threads: Fixed delay in shutdown if `on_thread_stop` callback raises exception.
- Service: Stopping of children no longer propagates exceptions, to ensure other services are still stopped.
- Worker: Fixed race condition if worker stopped before being fully started.

  This would lead the worker to shutdown early before fully stopping all dependent services.
- Tests: Adds *`AsyncMagicMock`*

1.6.16 3.0.2

release-date 2018-12-07 1:14 P.M PDT

release-by Ask Solem (@ask)

- Worker: Fixes crash on Windows where signal handlers cannot be registered.
- Utils: Adds *`shortname()`* to get non-qualified object path.
- Utils: Adds *`canonshortname()`* to get non-qualified object path that attempts to resolve the real name of `__main__`.

1.6.17 3.0.1

release-date 2018-12-06 10:20 A.M PDT

release-by Ask Solem (@ask)

- Worker: Added new callback `on_worker_shutdown`.
- Worker: Do not stop twice, instead wait for original stop to complete.

  Signals would start multiple stopping coroutines, leading to the worker shutting down too fast.
- Threads: All `ServiceThread` services needs a keepalive coroutine to be scheduled.
- Supervisor: Fixed issue with `CrashingSupervisor` where service would not crash.

1.6.18 3.0.0

release-date 2018-11-30 4:48 P.M PDT

release-by Ask Solem (@ask)

- `ServiceThread` no longer uses `run_in_executor`.

  Since services are long running, it is not a good idea for them to block pool worker threads. Instead we run one thread for every ServiceThread.
- Adds `QueuedServiceThread`

  This subclass of *`ServiceThread`* enables the use of a queue to send work to the service thread.

  This is useful for services that wrap blocking network clients for example.

  If you have a blocking Redis client you could run it in a separate thread like this:

```
class Redis(QueuedServiceThread):
    _client: StrictRedis = None

    async def on_start(self) -> None:
        self._client = StrictRedis()

    async def get(self, key):
        return await self.call_thread(self._client.get, key)

    async def set(self, key, value):
        await self.call_thread(self._client.set, key, value)
```

  The actual redis client will be running in a separate thread (with a separate event loop). The `get` and `set` methods will delegate to the thread, and return only when the thread is finished handling them and is ready with a result:

```
async def use_redis():
    # We use async-with-statement here, but
    # can also do `await redis.start()` then `await redis.stop()`
    async with Redis() as redis:
        await redis.set(key='foo', value='bar')
        assert await redis.get(key='foo') == 'bar'
```

- Collections: `FastUserSet` and `ManagedUserSet` now implements all `set` operations.
- Collections are now generic types.

You can now subclass collections with typing information:

- `class X(FastUserDict[str, int]): ...`
- `class X(ManagedUserDict[str, int]): ...`
- `class X(FastUserSet[str]): ...`
- `class X(ManagedUserSet[str]): ...`

- `maybe_async()` utility now also works with `@asyncio.coroutine` decorated coroutines.
- Worker: SIGUSR1 cry handler: Fixed crash when coroutine does not have `__name__` attribute.

1.6.19 2.0.4

release-date 2018-11-19 1:07 P.M PST

release-by Ask Solem (@ask)

- `FlowControlQueue.clear` now cancels all waiting for `Queue.put`.

1.6.20 2.0.3

release-date 2018-11-05 5:20 P.M PDT

release-by Ask Solem (@ask)

- Adds *Service.wait_first(*coros)*

  Wait for the first coroutine to return, where coroutines can also be `asyncio.Event`.

  Returns `mode.services.WaitResults` with fields:

  - `.done` - List of arguments that are now done.
  - `.results` - List of return values in order of `.done`.
  - `.stopped` - Set to True if the service was stopped.

1.6.21 2.0.2

release-date 2018-11-03 9:07 A.M PST

release-by Ask Solem (@ask)

- Now depends on aiocontextvars 0.2

  This release uses **PEP 508** syntax for conditional requirements, as *2.0.1* did not work when installing wheel.

1.6.22 2.0.1

release-date 2018-11-02 7:38 P.M PST

release-by Ask Solem (@ask)

- Now depends on aiocontextvars 0.2

1.6.23 2.0.0

release-date 2018-11-02 9:12 A.M PST

release-by Ask Solem (@ask)

- Services now create the event loop on demand.

  This means the event loop is no longer created in *Service.__init__* so that services can be defined at module scope without initializing the loop.

  This makes the `ServiceProxy` pattern redundant for most use cases.

- Adds `.utils.compat.current_task` as alias for `asyncio.current_task`.
- Adds support for contextvars in Python 3.6 using aiocontextvars.

  In mode services you can now use `contextvars` module even on Python 3.6, thanks to the work of @fantix.

1.6.24 1.18.2

release-date 2018-11-30 6:23 P.M PDT

release-by Ask Solem (@ask)

- Worker: SIGUSR1 cry handler: Fixed crash when coroutine does not have `__name__` attribute.

1.6.25 1.18.1

release-date 2018-10-03 2:49 P.M PDT

release-by Ask Solem (@ask)

- **Service**: `Service.from_awaitable(coro)` improvements.

  The resulting `service.start` will now:

  - Convert awaitable to `asyncio.Task`.
  - Wait for task to complete.

  then `service.stop` will:

  - Cancel the task.

  This ensures an `asyncio.sleep(10.0)` within can be cancelled. If you need some operation to absolutely finish you must use *asyncio.shield*.

- **Utils**: `cached_property` adds new `.is_set(o)` method on descriptor

  This can be used to test for the attribute having been cached/used.

  If you have a class with a `cached_property`:

```
from mode.utils.objects import cached_property

class X:

    @cached_property
    def foo(self):
        return 42
```

(continues on next page)

(continued from previous page)

```
x = X()
print(x.foo)
```

From an instance you can now check if the property was accessed:

```
if type(x).foo.is_set(x):
    print(f'Someone accessed x.foo and it was cached as: {x.foo}')
```

1.6.26 1.18.0

release-date 2018-10-02 3:32 P.M PDT

release-by Ask Solem (@ask)

- **Worker**: Fixed error when starting aioconsole on `--debug`

  The worker would crash with:

  ```
  TypeError: Use `self.add_context(ctx)` for non-async context
  ```

  when started with the `--debug` flag.

- **Worker**: New `daemon` argument controls shutdown of worker.

  When the flag is enabled, the default, the worker will not shut down until the worker instance is either explicitly stopped, or it receives a terminating process signal (`SIGINT`/`SIGTERM`/etc.)

  When disabled, the worker for the given service will shut down as soon as `await service.start()` returns.

  You can think of it as a flag for daemons, but one that doesn't actually do any of the UNIX daemonization stuff (detaching, etc.). It merely means the worker continues to run in the background until stopped by signal.

- **Service**: Added class method: `Service.from_awaitable`.

  This can be used to create a service out of any coroutine or `Awaitable`:

  ```
  from mode import Service, Worker

  async def me(interval=1.0):
      print('ME STARTING')
      await asyncio.sleep(interval)
      print('ME STOPPING')

  def run_worker(interval=1.0):
      coro = me(interval=1.0)
      Worker(Service.from_awaitable(coro)).execute_from_commandline()

  if __name__ == '__main__':
      run_worker()
  ```

  Note: Using a service with `await self.sleep(1.0)` is often not what you want, as stopping the service will have to wait for the sleep to finish.

  `Service.from_awaitable` is as such a last resort for cases where you're provided a coroutine you cannot implement as a service.

`Service.sleep()` is useful as it will stop sleeping immediately if the service is stopped:

class Me(Service):

async def on_start(self) -> None: await self.sleep(1.0)

- **Service: New method `_repr_name` can be used to override the service** class name used in `repr(service)`.

1.6.27 1.17.3

release-date 2018-09-18 4:00 P.M PDT

release-by Ask Solem (@ask)

- Service: New attribute `mundane_level` decides the logging level of mundane logging events such as "[X] Starting…", for starting/stopping and tasks being cancelled.

  The value for this must be a logger level name, and is `"info"` by default.

  If logging for a service is noisy at info-level, you can move it to debug level by setting this attribute to `"debug"`:

```
class X(Service):
    mundane_level = 'debug'
```

1.6.28 1.17.2

release-date 2018-09-17 3:00 P.M PDT

release-by Ask Solem (@ask)

- Removed and fixed import from `collections` that will be moved to `collections.abc` in Python 3.8.

  This also silences a `DeprecationWarning` that was being emitted on Python 3.7.

- Type annotations now passing checks on mypy 0.630.

1.6.29 1.17.1

release-date 2018-09-13 6:27 P.M PDT

release-by Ask Solem (@ask)

- Fixes several bugs related to unwrapping `Optional[List[..]]` in *mode.utils.objects.annotations()*.

  This functionality is not really related to mode at all, so should be moved out of this library. Faust uses it for models.

1.6.30 1.17.0

release-date 2018-09-12 5:39 P.M PDT

release-by Ask Solem (@ask)

- New async iterator utility: *arange*

Like `range` but returns an async iterator:

```
async for n in arange(0, 10, 2):
    ...
```

- New async iterator utility: *`aslice()`*

  Like `itertools.islice` but works on asynchronous iterators.

- New async iterator utility: *`chunks()`*

  `chunks` takes an async iterable and divides it up into chunks of size n:

```
# Split range of 100 numbers into chunks of 10 each.
async for chunk in chunks(arange(100), 10):
    yield chunk
```

  This gives chunks like this:

```
[
    [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
    ...,
]
```

1.6.31 1.16.0

release-date 2018-09-11 1:37 P.M PDT

release-by Ask Solem

- **Requirements**
  - Now depends on Mode 1.15.1.

    Contributed by Michael Seifert

- **Distribution: Installing mode no longer installs the `t` directory**

  containing tests as a Python package.

  Contributed by Michael Seifert

- **Testing**: New *`AsyncContextManagerMock`*

  You can use this to mock asynchronous context managers.

  Please see *`AsyncContextManagerMock`* for an example.

- **CI**: Python 3.7.0 and 3.6.0 was added to the build matrix.

1.6.32 1.15.1

release-date 2018-08-15 11:17 A.M PDT

release-by Ask Solem

- Tests now passing on CPython 3.7.0
- **Utils**: Adds `remove_optional` function in *`mode.utils.objects`*

  This can be used to extract the concrete type from `Optional[Foo]`.

- **Utils**: Adds `humanize_seconds` function to *`mode.utils.times`*

1.6.33 1.15.0

release-date 2018-06-27 1:39 P.M PDT

release-by Ask Solem

- Worker: Logging can now be set up using dictionary config, by passing the `logging_config` argument to *`mode.Worker`*.

  Contributed by Allison Wang.

- Worker: No longer supports the `logformat` argument.

  To set up custom log format you must now pass in dict configuration via the `logging_config` argument.

- Service: `start()` accidentally silenced `asyncio.CancelledError`.
- Service: Invalid assert caused *`CrashingSupervisor`* to crash with strange error

1.6.34 1.14.1

release-date 2018-06-06 1:26 P.M PDT

release-by Ask Solem

- Service: Fixed "coroutine x was never awaited" for background tasks (`@Service.task` decorator) when service is started and stopped in quick succession.

1.6.35 1.14.0

release-date 2018-06-05 12:13 P.M PDT

release-by Ask Solem

- Adds method `Service.wait_many(futures, *, timeout=None)`

1.6.36 1.13.0

release-date 2018-05-16 1:26 P.M PDT

release-by Ask Solem

- Mode now registers as a library having static type annotations.

  This conforms to **PEP 561** – a new specification that defines how Python libraries register type stubs to make them available for use with static analyzers like mypy and pyre-check.

- The code base now passes `--strict-optional` type checks.

1.6.37 1.12.5

release-date 2018-05-14 4:48 P.M PDT

release-by Ask Solem

- Supervisor: Fixed wrong index calculation in management of index-based service restart.

1.6.38 1.12.4

release-date 2018-05-07 3:20 P.M PDT

release-by Ask Solem

- Adds new mock class for async functions: *mode.utils.mocks.AsyncMock()*

  This can be used to mock an async callable:

```
from mode.utils.mocks import AsyncMock

class App(Service):

    async def on_start(self):
        self.ret = await self.some_async_method('arg')

    async def some_async_method(self, arg):
        await asyncio.sleep(1)

@pytest.fixture
def app():
    return App()

@pytest.mark.asyncio
async def test_something(*, app):
    app.some_async_method = AsyncMock()
    async with app: # starts and stops the service, calling on_start
        app.some_async_method.assert_called_once_with('arg')
        assert app.ret is app.some_async_method.coro.return_value
```

- Added 100% test coverage for modules:
  - *mode.proxy*
  - *mode.threads*
  - *mode.utils.aiter*

1.6.39 1.12.3

release-date 2018-05-07 3:33 P.M PDT

release-by Ask Solem

Important Notes

- Moved to https://github.com/ask/mode

Changes

- Signal: Improved repr when signal has a default sender.
- DictAttribute: Now supports `len` and `del(d[key])`.
- Worker: If overriding `on_first_start` you can now call `default_on_first_start` instead of super.

  Example:

```
class MyWorker(Worker):

    async def on_first_start(self) -> None:
        print('FIRST START')
        await self.default_on_first_start()
```

1.6.40 1.12.2

release-date 2018-04-26 11:47 P.M PDT

release-by Ask Solem

- Fixed shutdown error in *`ServiceThread`*.

1.6.41 1.12.1

release-date 2018-04-24 11:28 P.M PDT

release-by Ask Solem

- Now works with CPython 3.6.1 and 3.6.0.

1.6.42 1.12.0

release-date 2018-04-23 1:28 P.M PDT

release-by Ask Solem

Backward Incompatible Changes

- Changed `Service.add_context`
  - To add an async context manager (`AsyncContextManager`), use `add_async_context()`:

```
class S(Service):

    async def on_start(self) -> None:
        self.context = await self.add_async_context(MyAsyncContext())
```

  - To add a regular context manager (`ContextManager`), use *`add_context()`*:

```
class S(Service):

    async def on_start(self) -> None:
        self.context = self.add_context(MyContext())
```

This change was made so that contexts can be added from non-async functions. To add an *async context* you still need to be within an async function definition.

News

- **Worker: Now redirects `sys.stdout` and `sys.stderr` to the**

  logging subsystem by default.

  - To disable this pass `Worker(redirect_stdouts=False)`.
  - The default severity level for `print` statements are `logging.WARN`, but you can change this using `Worker(redirect_stdouts_level='INFO')`.

- `Seconds`/*`want_seconds()`* can now be expressed as strings and rate strings:
  - float as string: `want_seconds('1.203') == 1.203`
  - *10 in one second*: `want_seconds('10/s') == 10.0`
  - *10.33 in one hour*: `want_seconds('10.3/h') == 0.0028611111111111116`
  - *100 in one hour*: `want_seconds('100/h') == 0.02777777777777778`
  - *100 in one day*: `want_seconds('100/d') == 0.0011574074074074076`

  This is especially useful for the rate argument to the *`rate_limit`* helper.

- Added new context manager: *`mode.utils.logging.redirect_stdouts()`*.
- Module *`mode.types`* now organized by category:
  - Service types: *`mode.types.services`*
  - Signal types: *`mode.types.signals`*
  - Supervisor types: *`mode.types.supervisors`*
- *`mode.flight_recorder`* can now wrap objects so that every method call on that object will result in the call and arguments to that call being logged.

  Example logging statements with `INFO` severity:

```
with flight_recorder(logger, timeout=10.0) as on_timeout:
    redis = on_timeout.wrap_info(self.redis)
    await redis.get(key)
```

  There's also `wrap_debug(o)`, `wrap_warn(o)`, `wrap_error(o)`, and for any severity: `wrap(logging.CRIT, o)`.

Fixes

- Fixed bug in `Service.wait` on Python 3.7.

1.6.43 1.11.5

release-date 2018-04-19 3:12 P.M PST

release-by Ask Solem

- FlowControlQueue now available in `mode.utils.queues`.

This is a backward compatible change.

- Tests for FlowControlQueue

1.6.44 1.11.4

release-date 2018-04-19 9:36 A.M PST

release-by Ask Solem

- Adds *mode.flight_recorder*

  This is a logging utility to log stuff only when something times out.

  For example if you have a background thread that is sometimes hanging:

```
class RedisCache(mode.Service):

    @mode.timer(1.0)
    def _background_refresh(self) -> None:
        self._users = await self.redis_client.get(USER_KEY)
        self._posts = await self.redis_client.get(POSTS_KEY)
```

  You want to figure out on what line this is hanging, but logging all the time will provide way too much output, and will even change how fast the program runs and that can mask race conditions, so that they never happen.

  Use the flight recorder to save the logs and only log when it times out:

```
logger = mode.get_logger(__name__)

class RedisCache(mode.Service):

    @mode.timer(1.0)
    def _background_refresh(self) -> None:
        with mode.flight_recorder(logger, timeout=10.0) as on_timeout:
            on_timeout.info(f'+redis_client.get({USER_KEY!r})')
            await self.redis_client.get(USER_KEY)
            on_timeout.info(f'-redis_client.get({USER_KEY!r})')

            on_timeout.info(f'+redis_client.get({POSTS_KEY!r})')
            await self.redis_client.get(POSTS_KEY)
            on_timeout.info(f'-redis_client.get({POSTS_KEY!r})')
```

  If the body of this `with` statement completes before the timeout, the logs are forgotten about and never emitted – if it takes more than ten seconds to complete, we will see these messages in the log:

```
[2018-04-19 09:43:55,877: WARNING]: Warning: Task timed out!
[2018-04-19 09:43:55,878: WARNING]: Please make sure it is hanging before␣
↪restarting.
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1] (started at Thu Apr␣
↪19 09:43:45 2018) Replaying logs...
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1] (Thu Apr 19 09:43:45␣
↪2018) +redis_client.get('user')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1] (Thu Apr 19 09:43:49␣
↪2018) -redis_client.get('user')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1] (Thu Apr 19 09:43:46␣
↪2018) +redis_client.get('posts')
[2018-04-19 09:43:55,878: INFO]: [Flight Recorder-1] -End of log-
```

Now we know this `redis_client.get` call can take too long to complete, and should consider adding a timeout to it.

1.6.45 1.11.3

release-date 2018-04-18 5:22 P.M PST

relese-by Ask Solem

- Cry handler (*kill -USR1*): Truncate huge data in stack frames.
- ServiceProxy: Now supports `_crash` method.

1.6.46 1.11.2

release-date 2018-04-18 5:02 P.M PST

release-by Ask Solem

- Service: `add_future()` now maintains futures in a set and futures are automatically removed from it when done.
- Cry handler (*kill -USR1*) now shows name of Service.task background tasks.
- Stampede: Now propagates cancellation.

1.6.47 1.11.1

release-date 2018-04-18 11:08 P.M PST

release-by Ask Solem

- Service.add_context: Now works with AsyncContextManager.
- CI now runs functional tests.
- Added supervisor and service tests.

1.6.48 1.11.0

release-date 2018-04-17 1:23 P.M PST

release-by Ask Solem

- Supervisor: Fixes bug with max restart triggering too early.
- Supervisor: Also restart child services.
- Service: Now supports `__post_init__` like Python 3.7 dataclasses.
- Service: Crash is logged even if crashed multiple times.

1.6.49 1.10.4

release-date 2018-04-13 3:53 P.M PST

release-by Ask Solem

- Supervisor: Log full traceback when restarting service.

1.6.50 1.10.3

release-date 2018-04-11 10:58 P.M PST

release-by Ask Solem

- setup_logging: now ensure logging is setup by clearing root logger handlers.

1.6.51 1.10.2

release-date 2018-04-03 4:50 P.M PST

release-by Ask Solem

- Fixed wrong version number in Changelog.

1.6.52 1.10.1

release-date 2018-04-03 4:43 P.M PST

release-by Ask Solem

- **Service.wait: If the future we are waiting for is cancelled we must** propagate `CancelledError`.

1.6.53 1.10.0

release-date 2018-03-30 12:36 P.M PST

release-by Ask Solem

- New supervisor: *`ForfeitOneForOneSupervisor`*.

  If a service in the group crashes we give up on that service and don't start it again.

- New supervisor: `ForfeitOneForAllSupervisor`.

  If a service in the group crashes we give up on it, but also stop all services in the group and give up on them also.

- Service Logging: Renamed `self.log.crit` to `self.log.critical`.

  The old name is still available and is not deprecated at this time.

1.6.54 1.9.2

release-date 2018-03-20 10:17 P.M PST

release-by Ask Solem

- Adds `FlowControlEvent.clear()` to clear all contents of flow controlled queues.
- `FlowControlEvent` now starts in a suspended state.

  To disable this pass `FlowControlEvent(initially_suspended=False))`.

- Adds `Service.service_reset` method to reset service start/stopped/crashed/etc., flags

1.6.55 1.9.1

release-date 2018-03-05 11:51 P.M PST

release-by Ask Solem

- No longer depends on terminaltables.

1.6.56 1.9.0

release-date 2018-03-05 11:33 P.M PST

release-by Ask Solem

1.6.57 Backward Incompatible Changes

- Module `mode.utils.debug` renamed to *mode.debug*.

  This is unlikely to affect users as this module is only used by mode internally.

  This module had to move because it imports `mode.Service`, and the `mode.utils` package is not allowed to import from the *mode* package at all.

1.6.58 News

- Added function `mode.utils.import.smart_import()`.
- Added non-async version of *mode.Signal*: *mode.SyncSignal*.

  The signal works exactly the same as the asynchronous version, except `Signal.send` must not be `await`-ed:

```
on_configured = SyncSignal()
on_configured.send(sender=obj)
```

- Added method `iterate` to *mode.utils.imports.FactoryMapping*.

  This enables you to iterate over the extensions added to a `setuptools` entrypoint.

1.6.59 Fixes

- `StampedeWrapper` now correctly clears flag when original call done.

1.6.60 1.8.0

release-date 2018-02-20 04:01 P.M PST

release-by Ask Solem

Backward Incompatible Changes

- API Change to fix memory leak in `Service.wait`.

  The `Service.wait(*futures)` method was added to be able to wait for this list of futures but also stop waiting if the service is stopped or crashed:

```
import asyncio
from mode import Service

class X(Service):
    on_thing_ready: asyncio.Event

    def __post_init__(self):
        self.on_thing_ready = asyncio.Event(loop=loop)

    @Service.task
    async def _my_background_task(self):
        while not self.should_stop:
            # wait for flag to be set (or service stopped/crashed)
            await self.wait(self.on_thing_ready.wait())
            print('FLAG SET')
```

  The problem with this was

  1) The wait flag would return None and not raise an exception if the service is stopped/crashed.
  2) Futures would be scheduled on the event loop but not properly cleaned up, creating a very slow memory leak.
  3) No return value was returned for succesful feature.

  So to properly implement this we had to change the API of the `wait` method to return a tuple instead, and to only allow a single coroutine to be passed to wait:

```
@Service.task
async def _my_background_task(self):
    while not self.should_stop:
        # wait for flag to be set (or service stopped/crashed)
        result, stopped = await self.wait(self.on_thing_ready)
        if not stopped:
            print('FLAG SET')
```

  This way the user can provide an alternate path when the service is stopped/crashed while waiting for this event.

  A new shortcut method `wait_for_stopped(fut)` was also added:

```
# wait for flag to be set (or service stopped/crashed)
if not await self.wait_for_stopped(self.on_thing_ready):
    print('FLAG SET')
```

  Moreover, you can now pass `asyncio.Event` objects directly to `wait()`.

News

- Added *mode.utils.collections.DictAttribute*.
- Added *mode.utils.collections.AttributeDict*.

Bugs

- Signals can create clone of signal with default sender already set

```
signal: Signal[int] = Signal()
signal = signal.with_default_sender(obj)
```

1.6.61 1.7.0

release-date 2018-02-05 12:28 P.M PST

release-by Ask Solem

- Adds *mode.utils.aiter* for missing `aiter` and `anext` functions.
- Adds *mode.utils.futures* for `asyncio.Task` related tools.
- Adds *mode.utils.collections* for custom mapping/set and list data structures.
- Adds *mode.utils.imports* for importing modules at runtime, as well as utilities for typed `setuptools` entry-points.
- Adds *mode.utils.text* for fuzzy matching user input.

1.6.62 1.6.0

release-date 2018-02-05 11:10 P.M PST

release-by Ask Solem

- Fixed bug where `@Service.task` background tasks were not started in subclasses.
- Service: Now has two exit stacks: `.exit_stack` & `.async_exit_stack`.

  This is a backward incompatible change, but probably nobody was accessing `.exit_stack` directly.

  Use `await Service.enter_context(ctx)` with both regular and asynchronous context managers:

```
class X(Service):

    async def on_start(self) -> None:
        # works with both context manager types.
        await self.enter_context(async_context)
        await self.enter_context(context)
```

- Adds `asynccontextmanager`()` decorator from CPython 3.7b1.

  This decorator works exactly the same as `contextlib.contextmanager()`, but for `async with`.

  Import it from *mode.utils.contexts*:

```
from mode.utils.contexts import asynccontextmanager

@asynccontextmanager
async def connection_or_default(conn: Connection = None) -> Connection:
    if connection is None:
```

(continues on next page)

(continued from previous page)

```
        async with connection_pool.acquire():
            yield
    else:
        yield connection

async def main():
    async with connection_or_default() as connection:
        ...
```

- Adds *AsyncExitStack* from CPython 3.7b1

  This works like `contextlib.ExitStack`, but for asynchronous context managers used with `async with`.

- Logging: Worker debug log messages are now colored blue when colors are enabled.

1.6.63 1.5.0

release-date 2018-01-04 03:43 P.M PST

release-by Ask Solem

- Service: Adds new `await self.add_context(context)`

  This adds a new context manager to be entered when the service starts, and exited once the service exits.

  The context manager can be either a `typing.AsyncContextManager` (`async with`) or a regular `typing.ContextManager` (`with`).

- Service: Added `await self.add_runtime_dependency()` which unlike `add_dependency` starts the dependent service if the self is already started.

- Worker: Now supports a new `console_port` argument to specify a port for the aiomonitor console, different than the default (50101).

  Note: The aiomonitor console is only started when `Worker(debug=True, ...)` is set.

1.6.64 1.4.0

release-date 2017-12-21 09:50 A.M PST

release-by Ask Solem

- Worker: Add support for parameterized logging handlers.

  Contributed by Prithvi Narasimhan.

1.6.65 1.3.0

release-date 2017-12-04 01:17 P.M PST

release-by Ask Solem

- Now supports color output in logs when logging to a terminal.

- Now depends on colorlog
- Added *mode.Signal*: async. implementation of the observer pattern (think Django signals).
- DependencyGraph is now a generic type: `DependencyGraph[int]`
- Node is now a generic type: `Node[Service]`.

1.6.66 1.2.1

release-date 2017-11-06 04:50 P.M PST

release-by Ask Solem

- Service: Subclasses can now override a Service.task method.

  Previously it would unexpectedly start two tasks: the task defined in the superclass and the task defined in the subclass.

1.6.67 1.2.0

release-date 2017-11-02 03:17 P.M PDT

release-by Ask Solem

- Renames PoisonpillSupervisor to CrashingSupervisor.
- Child services now stopped even if not fully started.

  Previously `child_service.stop()` would not be called if *child_service.start()* never completed, but as a service might be in the process of starting other child services, we need to call stop even if not fully started.

1.6.68 1.1.1

release-date 2017-10-25 04:34 P.M PDT

release-by Ask Solem

- Added alternative event loop implementations: eventlet, gevent, uvloop.

  E.g. to use gevent as the event loop, install mode using:

  ```
  $ pip install mode[gevent]
  ```

  and add this line to the top of your worker entrypoint module:

  ```
  import mode.loop
  mode.loop.use('gevent')
  ```

- Service: More fixes for the weird *__init_subclass__* behavior only seen in Python 3.6.3.
- ServiceThread: Now propagates errors raised in the thread to the main thread.

1.6.69 1.1.0

release-date 2017-10-19 01:35 P.M PDT

release-by Ask Solem

- ServiceThread: Now inherits from Service, and uses `loop.run_in_executor()` to start the service as a thread.
- setup_logging: filename argument is now respected.

1.6.70 1.0.2

release-date 2017-10-10 01:51 P.M PDT

release-by Ask Solem

- Adds support for Python 3.6.0
- Adds backports of typing improvements in CPython 3.6.1 to `mode.utils.compat`: `AsyncContextManager`, `ChainMap`, `Counter`, and `Deque`.
- `Supervisor.add` and `.discard` now takes an arbitrary number of services to add/discard as star arguments.
- Fixed typo in example: `Service.task` -> `mode.Service.task`.

  Contributed by Xu Jing.

1.6.71 1.0.1

release-date 2017-10-05 02:53 P.M PDT

release-by Ask Solem

- Fixes compatibility with Python 3.6.3.

  Python 3.6.3 badly broke `__init_subclass__`, in such a way that any class attribute set is set for all subclasses.

1.6.72 1.0.0

release-date 2017-10-04 01:29 P.M PDT

release-by Ask Solem

- Initial release

1.7 Glossary

thread safe A function or process that is thread safe means that multiple POSIX threads can execute it in parallel without race conditions or deadlock situations.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W