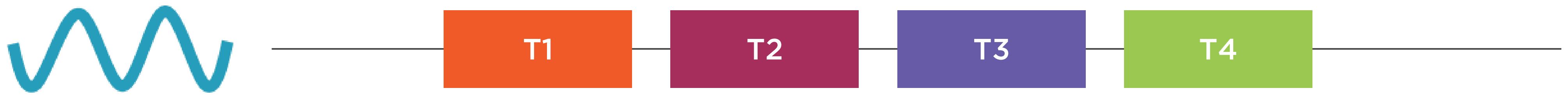
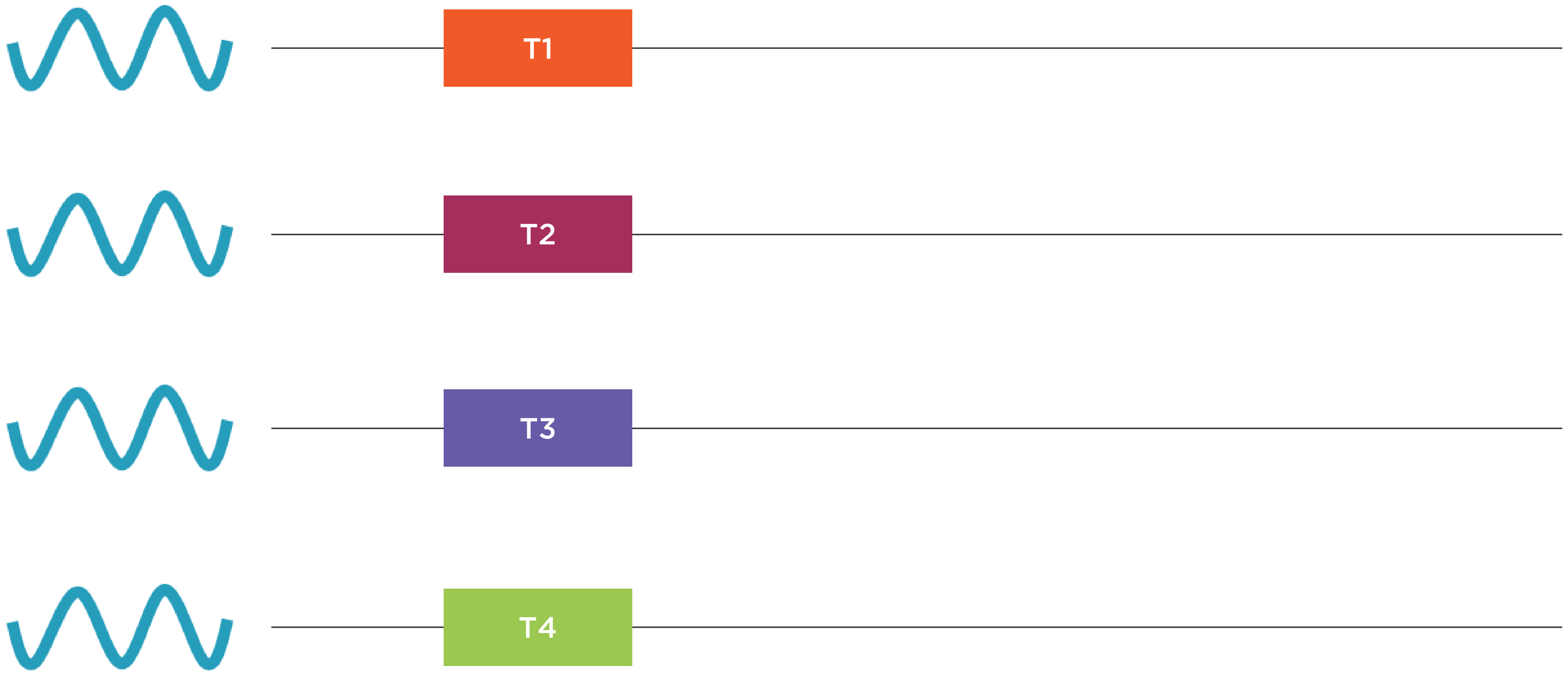
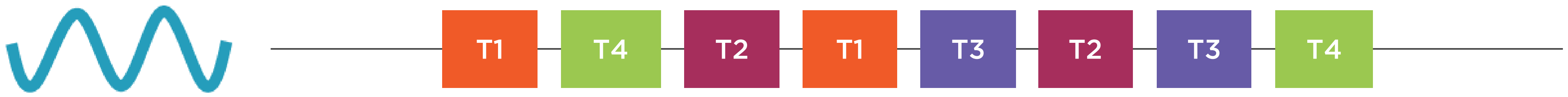
Asynchronous Programming

Tim Ojo
@tim_ojo   www.timojo.com

Single Threaded Asynchrony

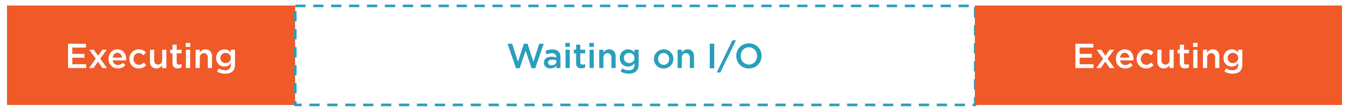

T1
T2
T3
T4


T1
T2
T3
T4


T1
T4
T2
T1
T3
T2
T3
T4

Suitable for IO-Bound Tasks

Suitable for IO-Bound Tasks

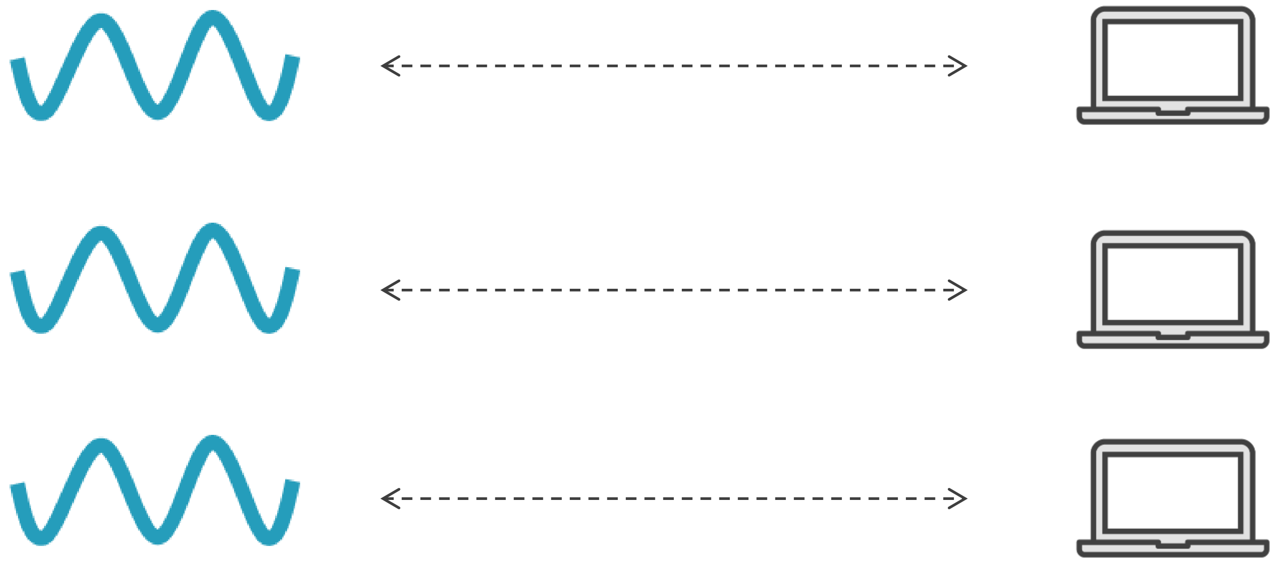


Event Driven Architecture

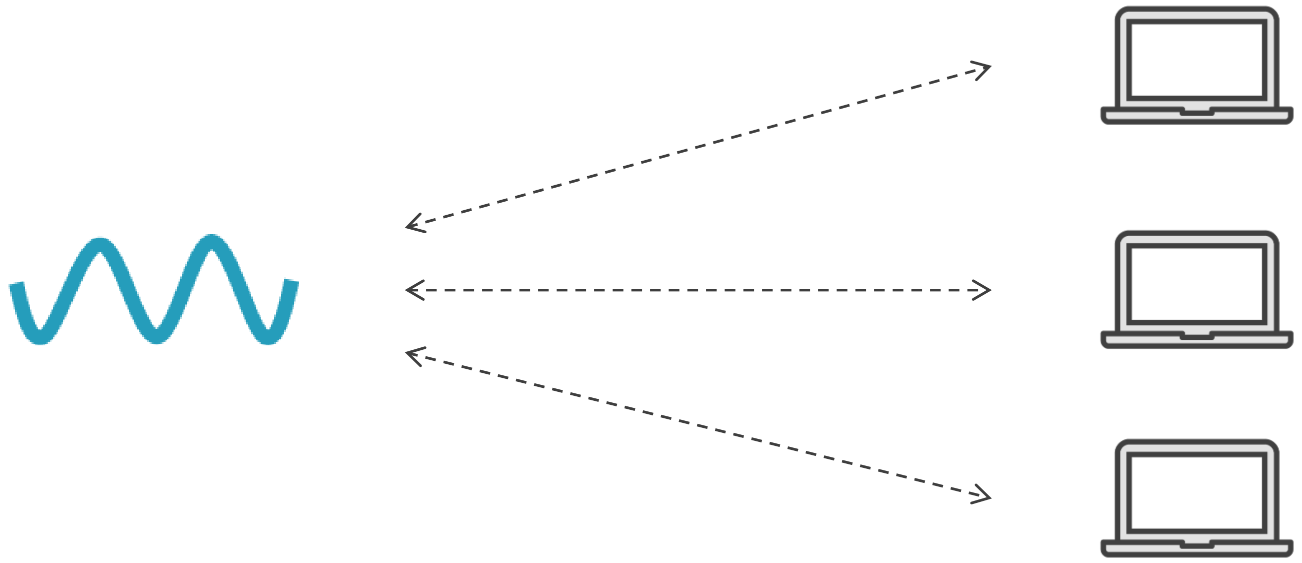
is a software design that orchestrates behavior around the production, detection and consumption of events

NGINX

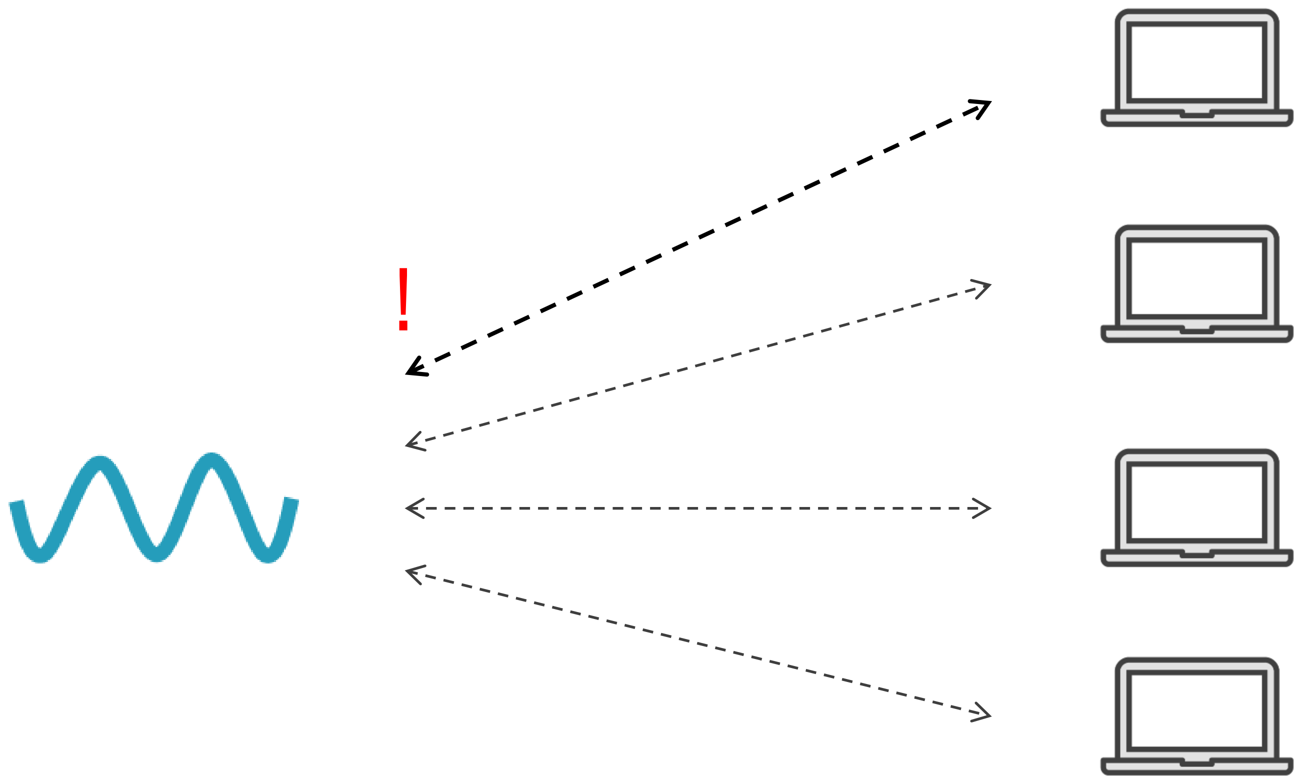
node JS™



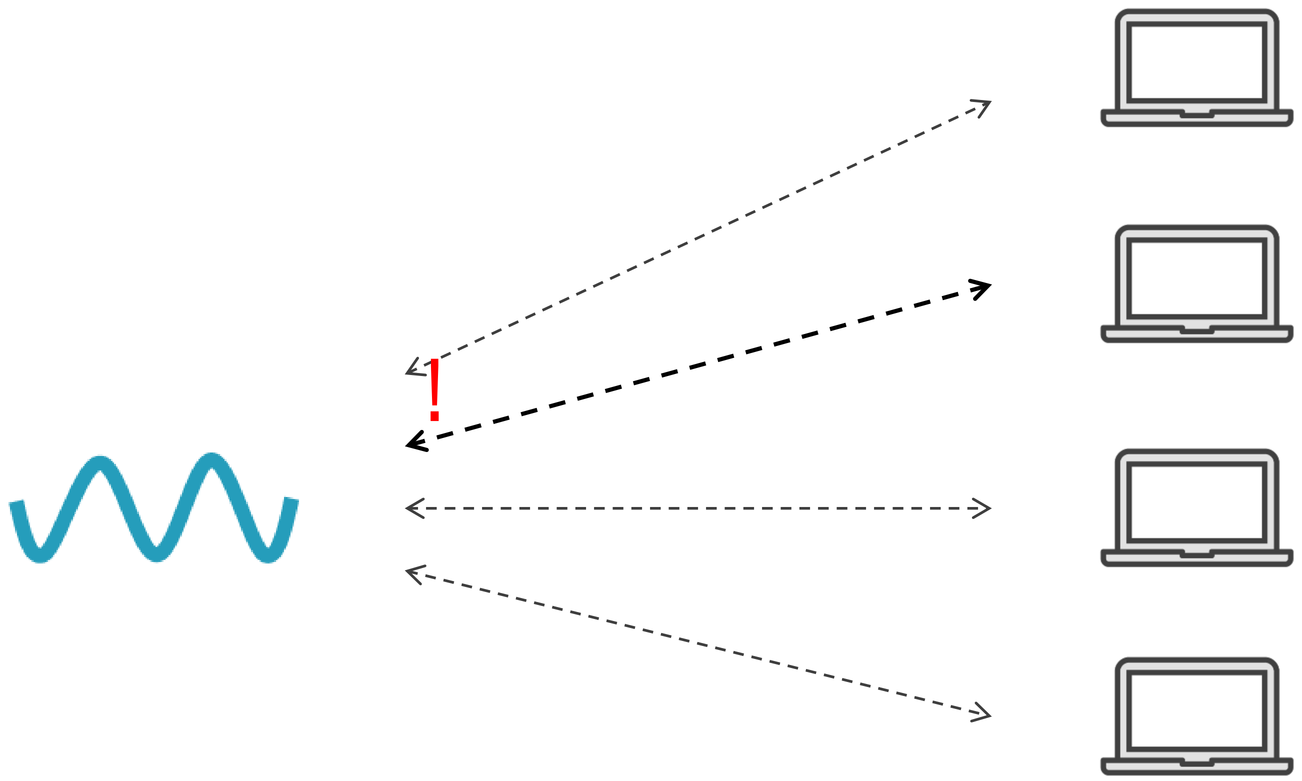
Traditional Model



Event Driven Model



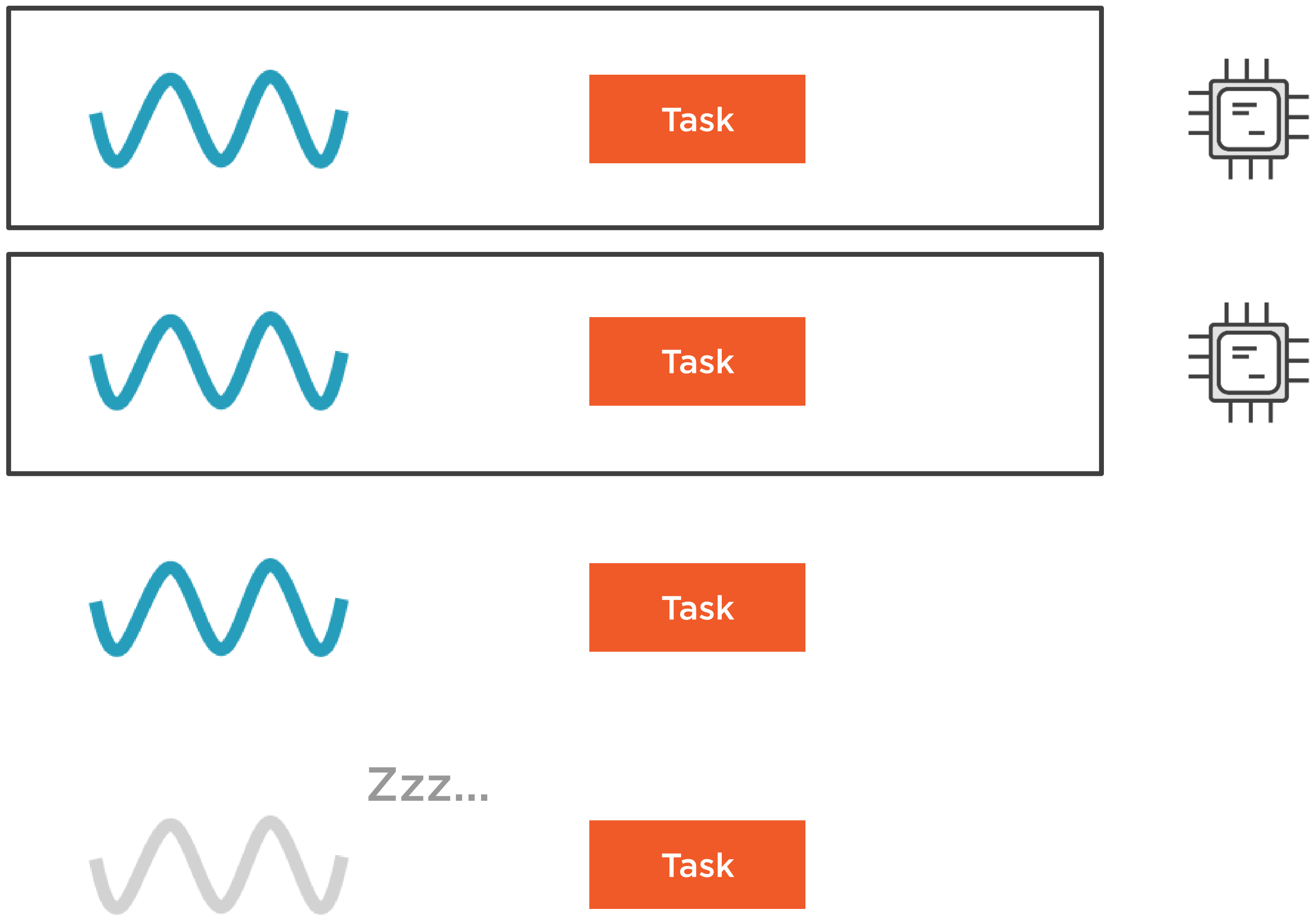
Event Driven Model

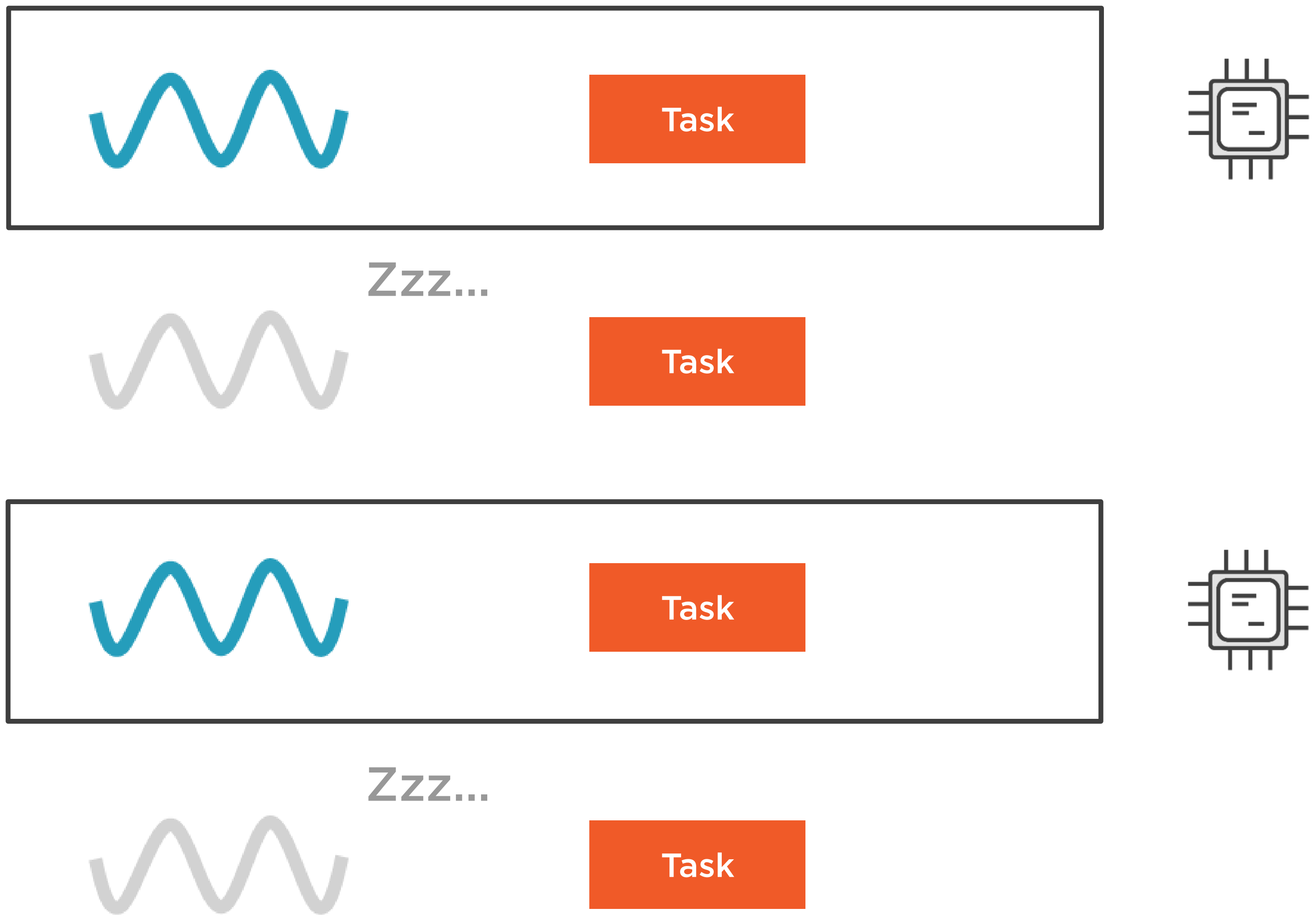


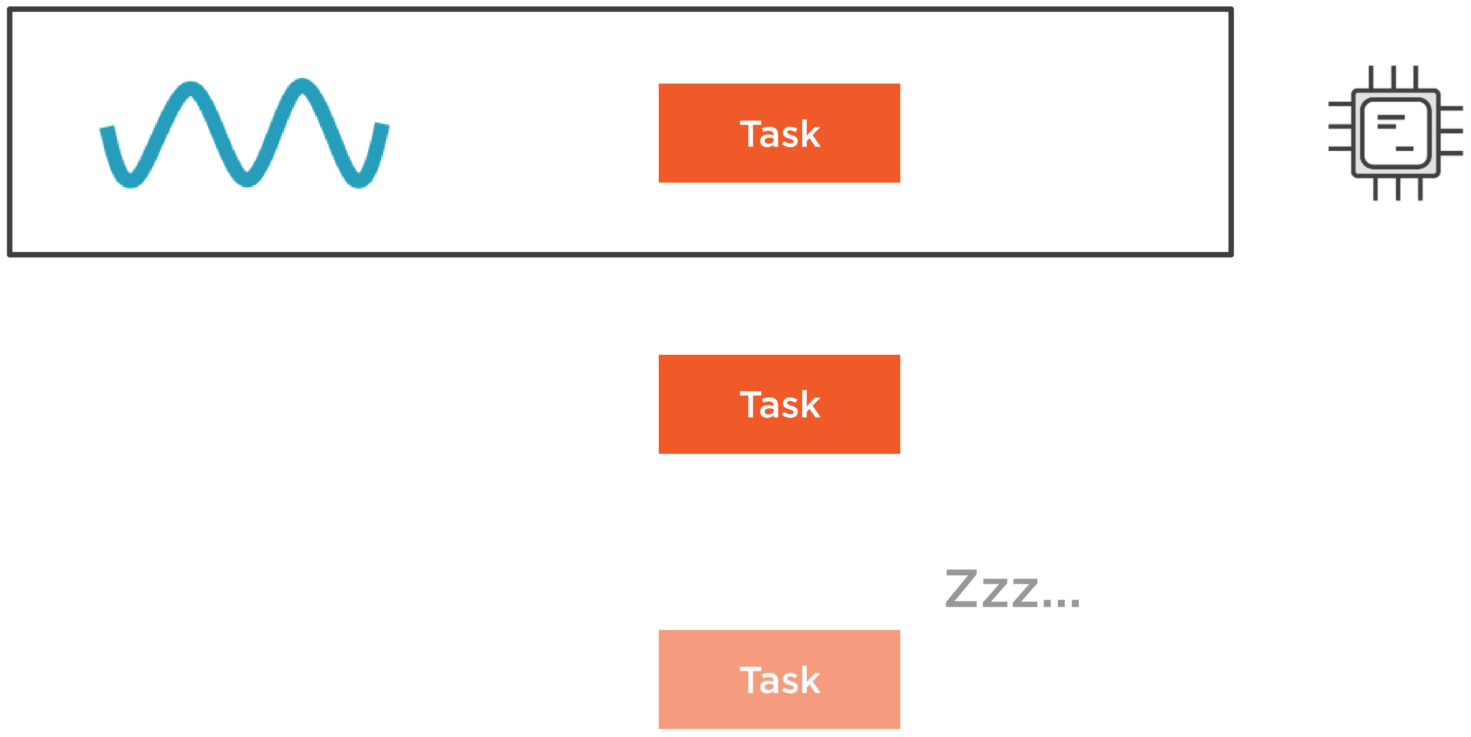
Event Driven Model

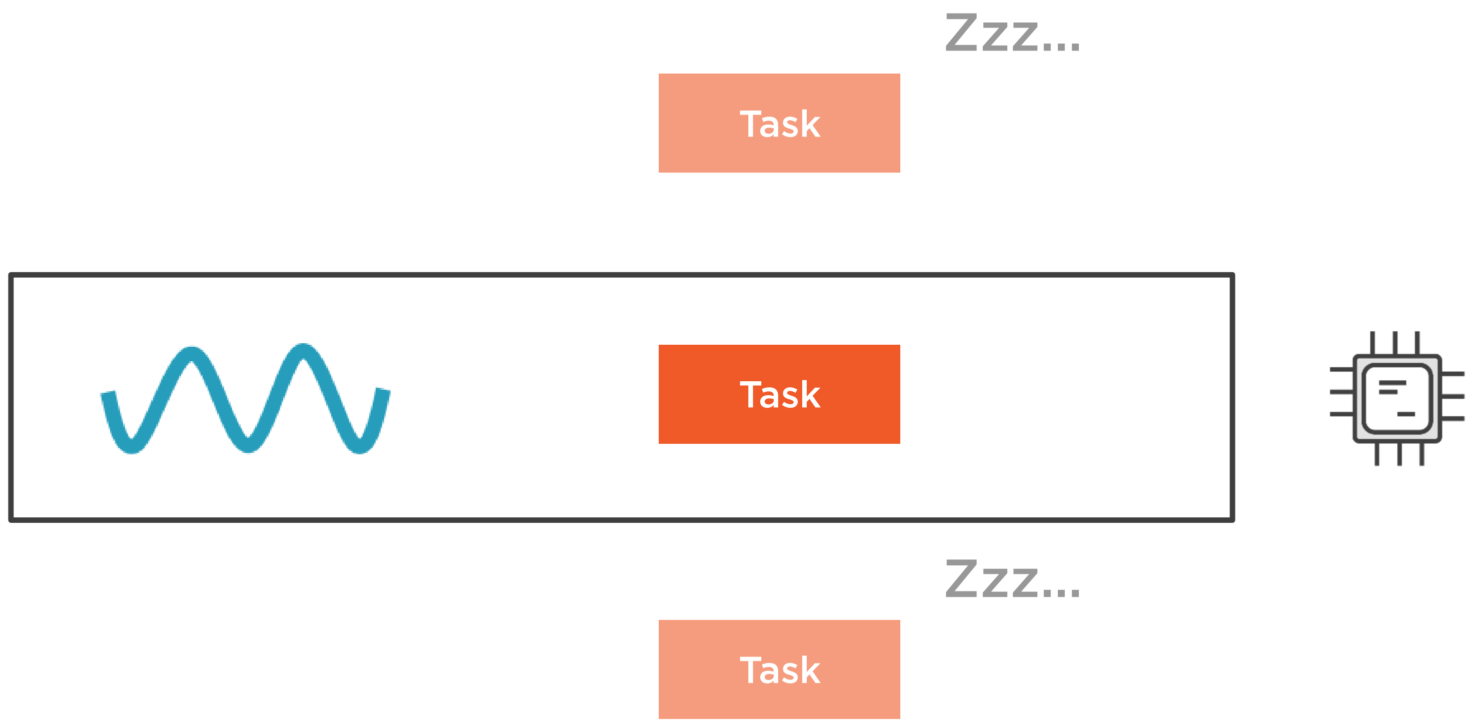
Asynchronous IO

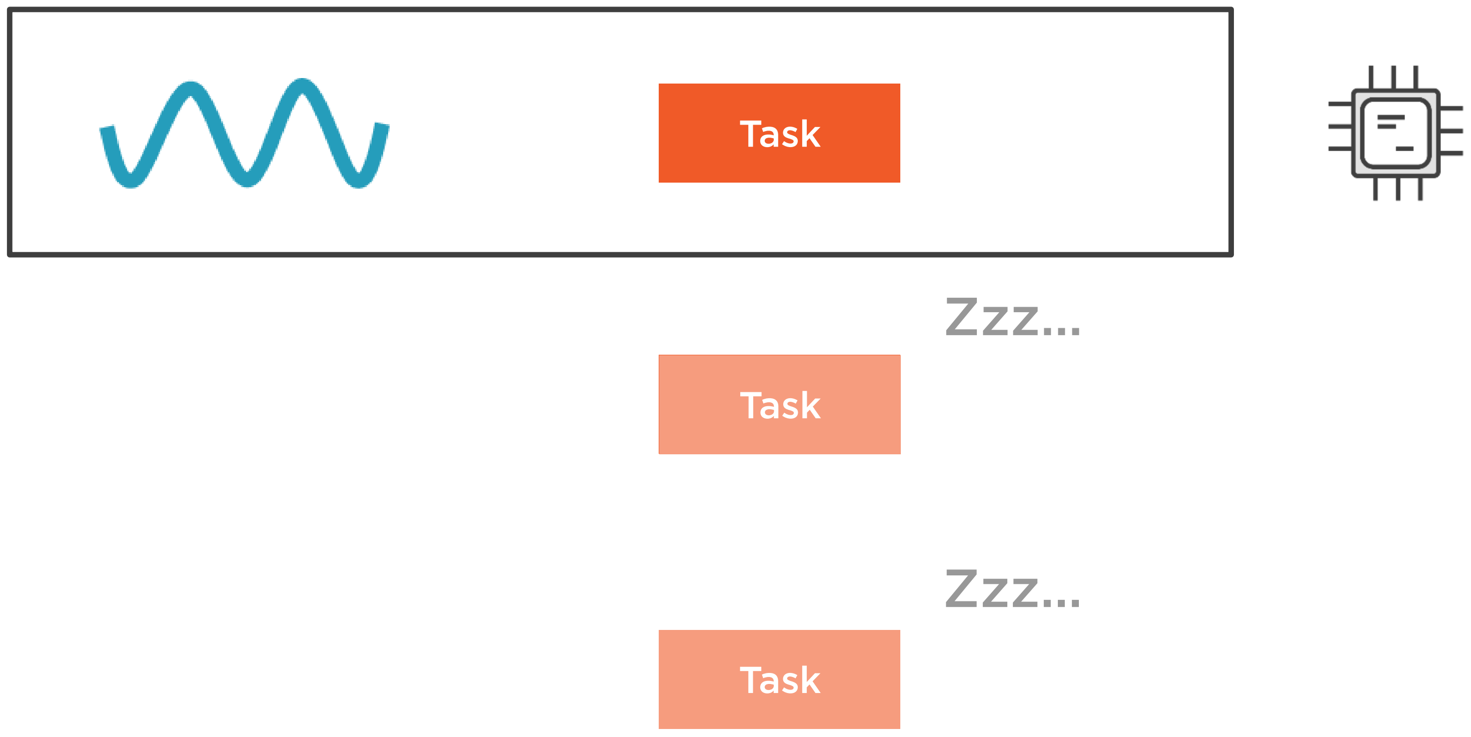
Event Loop


Task
Task
Task
Zzz...
Task


Task
Zzz...
Task
Task
Zzz...
Task


Task
Task
Zzz...
Task


Zzz...
Task
Task
Zzz...
Task


Task
Zzz...
Task
Zzz...
Task

Event Loop

is responsible for getting items from an event queue and handling it

Examples of Events

Change of file state

Timeout occurring

New data at network socket

…

Event Loop

is responsible for getting items from an event queue and handling it

Cooperative Multitasking with Event Loops and Coroutines

Python Event Loop

```
asyncio.get_event_loop()
```

Python Event Loop

```
AbstractEventLoop.run_forever()
```

Python Event Loop

```
AbstractEventLoop.run_forever()
```

```
AbstractEventLoop.run_until_complete(future)
```

Python Event Loop

```
AbstractEventLoop.stop()
```

Python Event Loop

```
AbstractEventLoop.close()
```

Cooperative Multitasking

Cooperative Multitasking

- **Tasks suspend themselves to allow others run**

Cooperative Multitasking

- Tasks suspend themselves to allow others run
- Event loop resumes the task when the IO operation completes

Cooperative Multitasking

- Tasks suspend themselves to allow others run
- Event loop resumes the task when the IO operation completes
- Tasks => Coroutines

Coroutine

Coroutine

- **Coroutine Function**

Coroutine

- **Coroutine Function**
- **Coroutine Object**

```
import asyncio

async def say_hello():
    print("Hello World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(say_hello())
loop.close()
```

```
import asyncio

async def say_hello():
    print("Hello World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(say_hello())
loop.close()
```

```
import asyncio

async def say_hello():
    print("Hello World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(say_hello())
loop.close()
```

```
import asyncio

async def say_hello():
    print("Hello World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(say_hello())
loop.close()
```

```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```

```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```

```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```

Python 3.4

yield from

@asyncio.coroutine

Python 3.5+

await

async

Python 3.4	Python 3.5+
yield from	await
@asyncio.coroutine	async

Python 3.4	Python 3.5+
yield from	await
@asyncio.coroutine	async

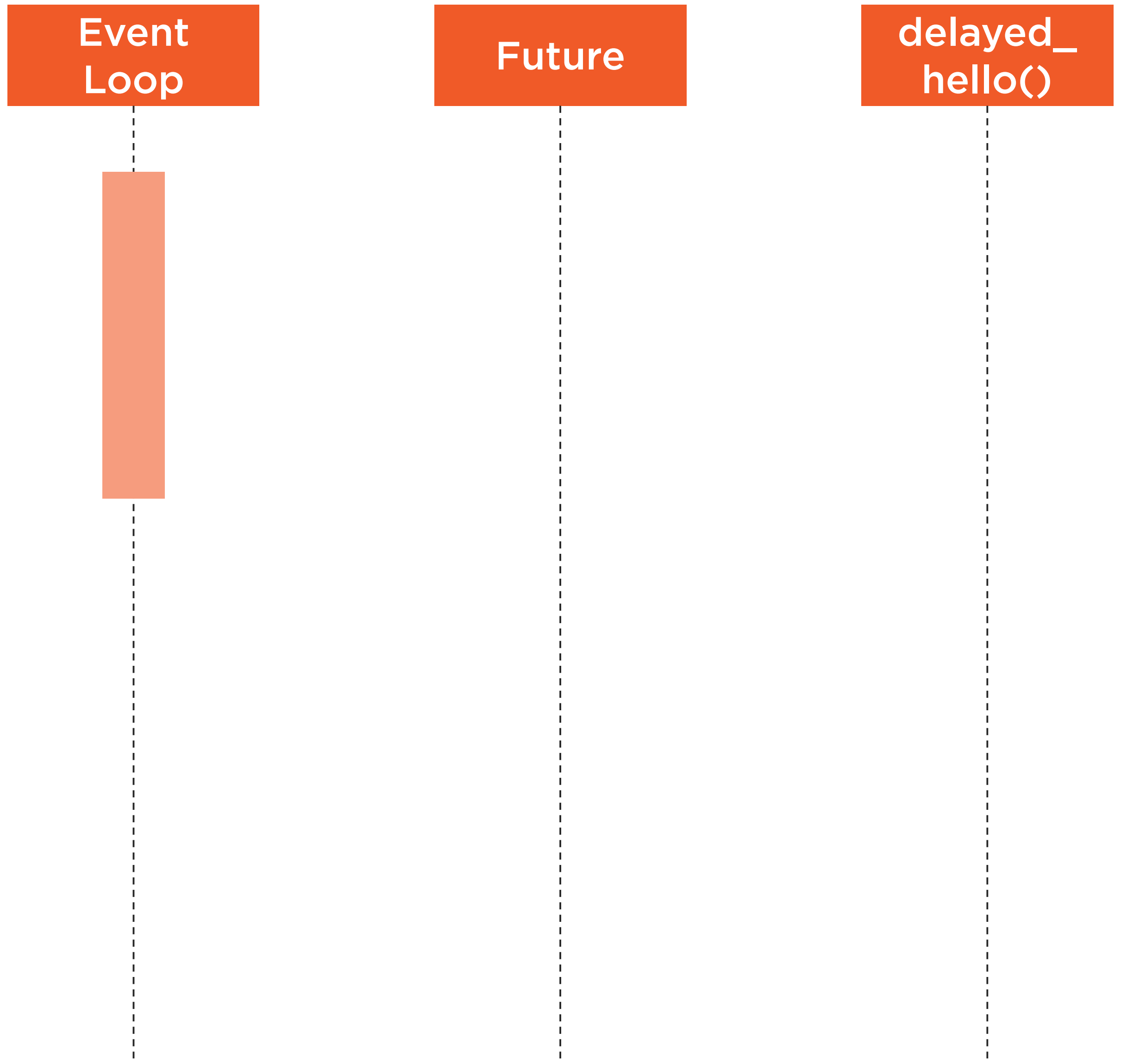
CoroutineObject = CoroutineFunction()

$$\begin{tikzcd} \text{CoroutineObject = CoroutineFunction()} \arrow[d] \\ \text{Future(CoroutineObject)} \end{tikzcd}$$

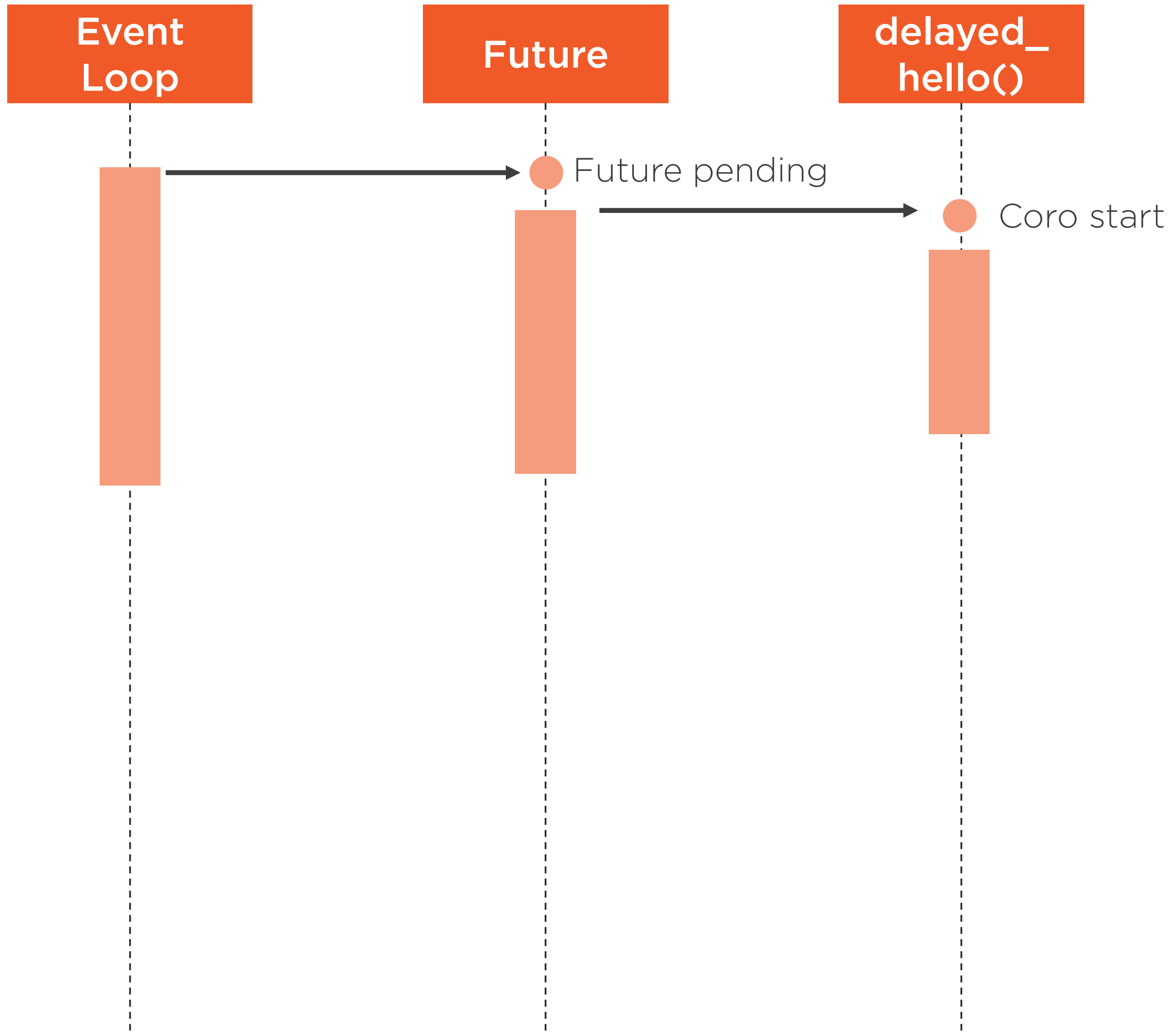
$$\begin{tikzcd}
\text{CoroutineObject = CoroutineFunction()} \arrow[d] \\
\text{Future(CoroutineObject)} \arrow[d] \\
\circlearrowleft
\end{tikzcd}$$

```
import asyncio

async def say_hello():
    print("Hello World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(say_hello())
loop.close()
```

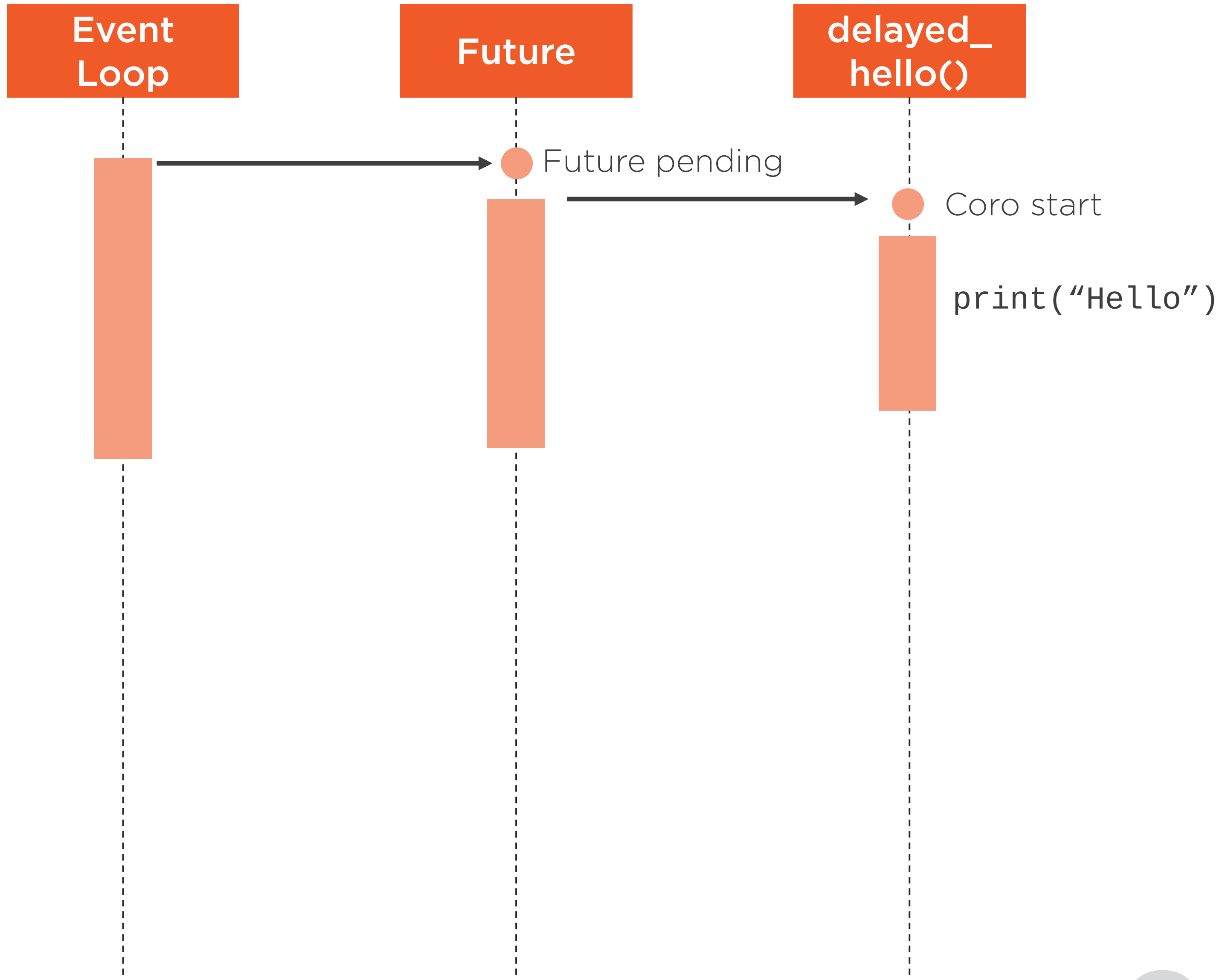
```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```



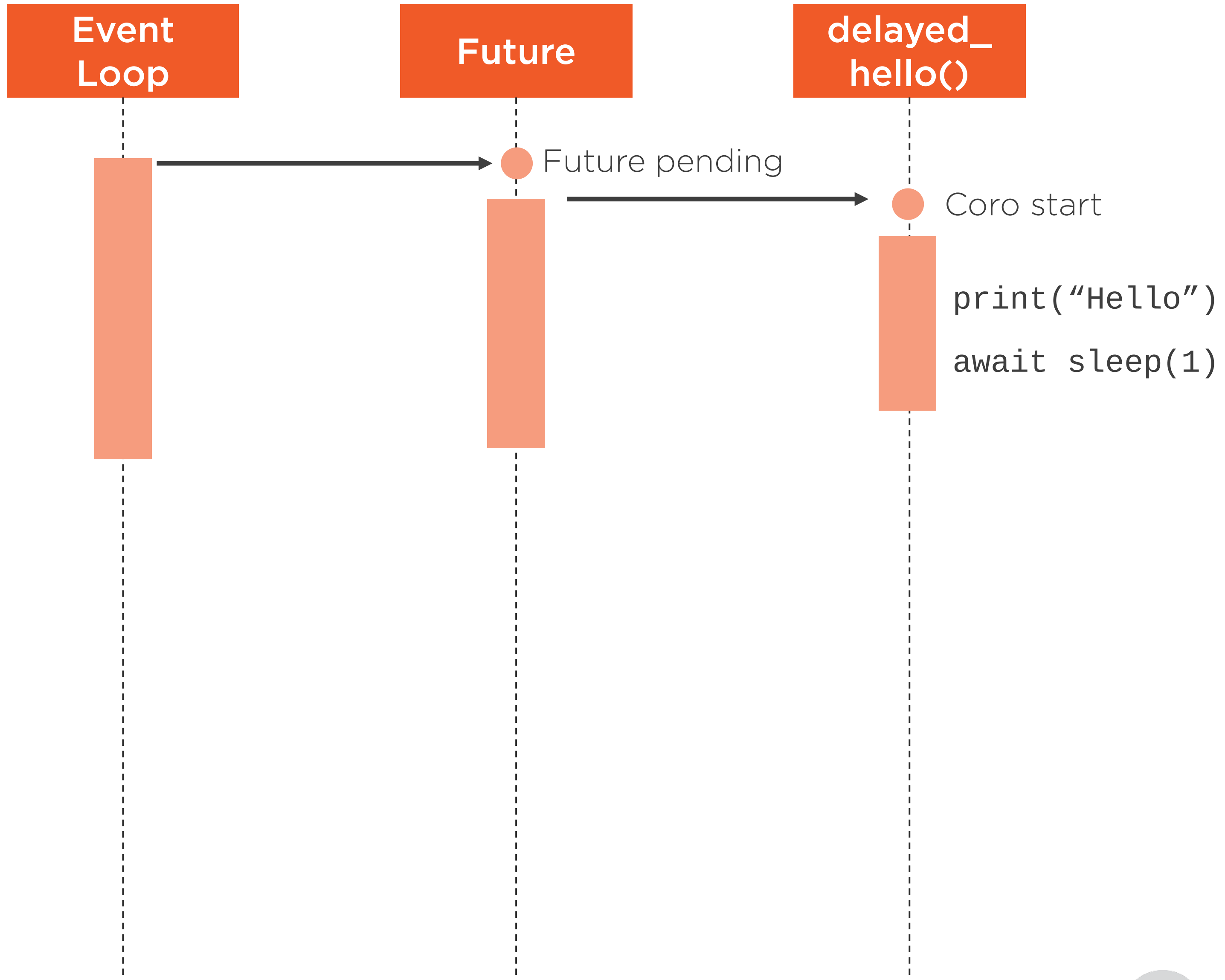
```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```



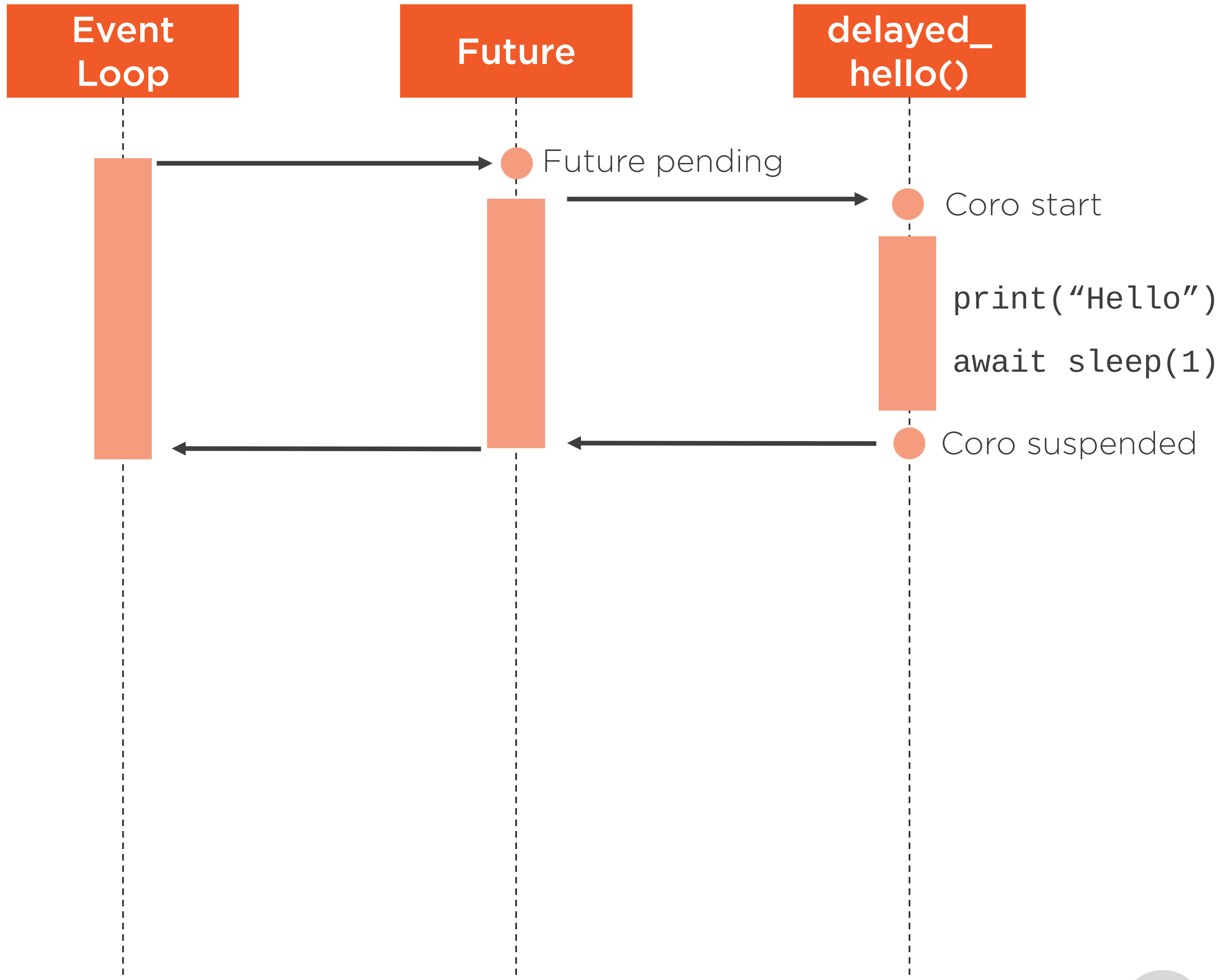
```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```



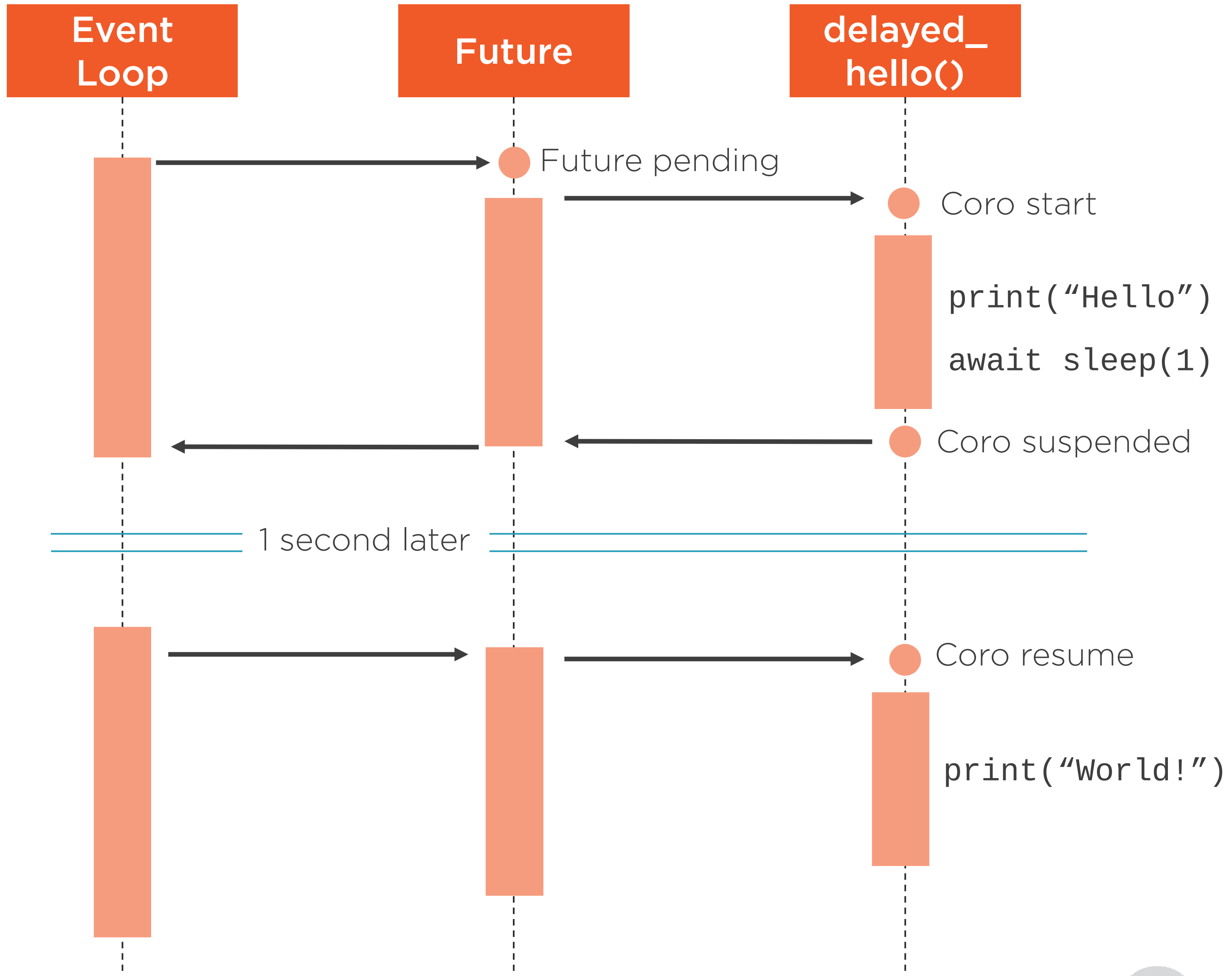
```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```



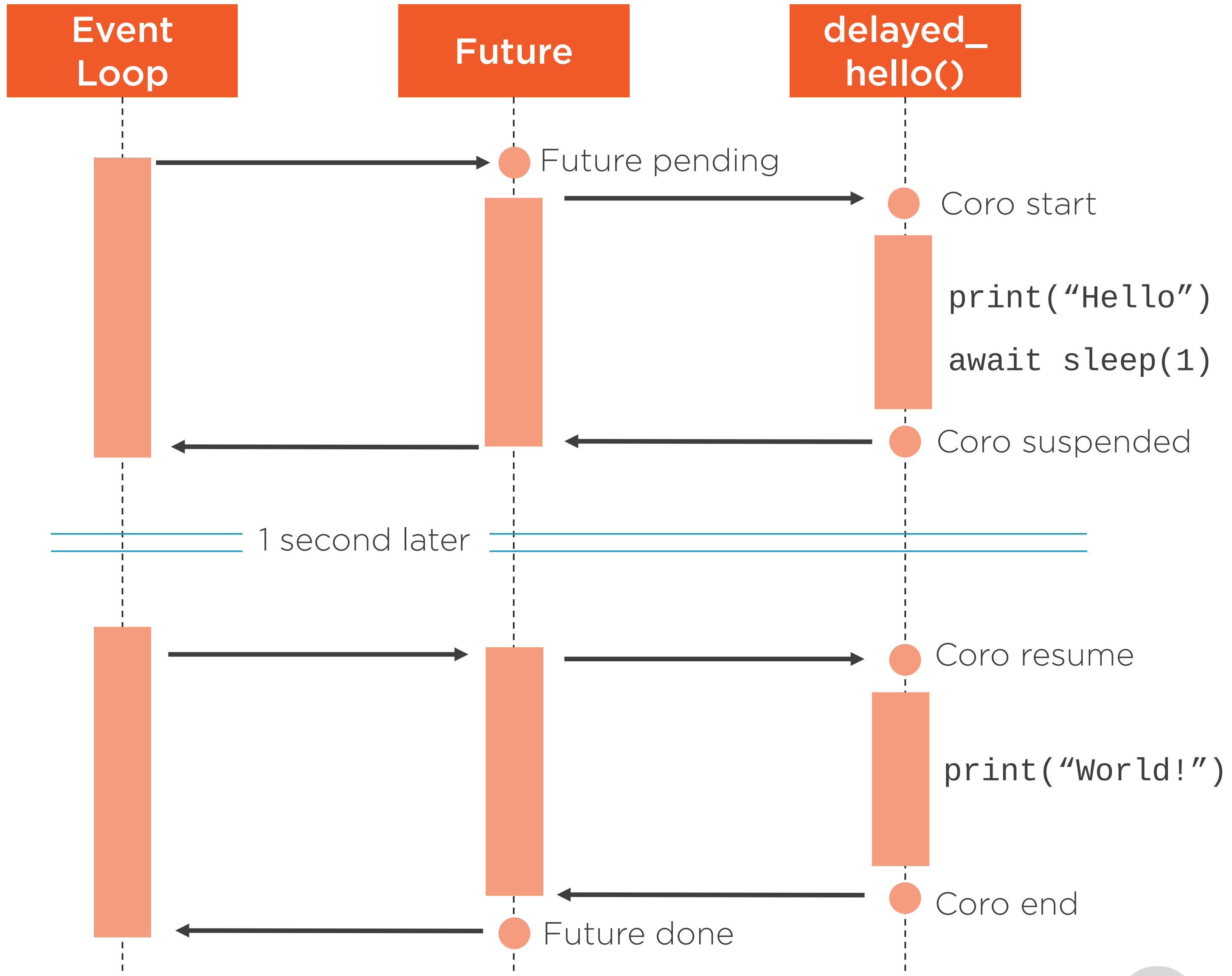
```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```



```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```



```
import asyncio

async def delayed_hello():
    print("Hello ")
    await asyncio.sleep(1)
    print("World!")

loop = asyncio.get_event_loop()
loop.run_until_complete(delayed_hello())
loop.close()
```



More Asyncio Concepts

Future

manages the execution and represents the eventual result of a computation

Future Methods

```
cancel() # cancels the future
```

Future Methods

```
done() # returns true if completed or canceled
```

Future Methods

```
result() # returns the result
```

Future Methods

```
exception() # returns any exception raised during execution
```

Future Methods

```
add_done_callback(fn) # adds callback to be run when done
```

Non-Blocking	Blocking
asyncio.Future	*concurrent.future.Future*
.result()	.result(timeout)
.exception()	.exception(timeout)

Non-Blocking	Blocking
asyncio.Future	*concurrent.future.Future*
.result()	.**result**(timeout)
.exception()	.exception(timeout)

Non-Blocking

asyncio.Future

.result()

.exception()

Blocking

concurrent.future.Future

.result(timeout)

.exception(timeout)

Non-Blocking

asyncio.Future

.result()

.exception()

Blocking

concurrent.future.Future

.result(timeout)

.exception(timeout)

Waiting for a Future to Complete

```
await future # pause execution until future is done
```

Waiting for a Future to Complete

```
loop.run_until_complete(future) # loop stops after future is complete
```

Task

a subclass of Future that is used to wrap and manage the execution of a coroutine in an event loop

Creating a Task

```
asyncio.ensure_future(coro_or_future, *, loop=None)
```

Creating a Task

```
asyncio.ensure_future(coro_or_future, *, loop=None)
```

```
AbstractEventLoop.create_task(coro)
```

Coroutine Chaining

A coroutine awaiting another coroutine

```
async def perform_task():
    print('performing task')
    print('waiting for result1')
    result1 = await subtask1()
    print('waiting for result2')
    result2 = await subtask2(result1)
    return (result1, result2)

async def subtask1():
    print('perform subtask 1')
    return 'result1'

async def subtask2(arg):
    print('perform subtask 2')
    return 'result2 relies on {}'.format(arg)

loop = asyncio.get_event_loop()
result = loop.run_until_complete(perform_task())
event_loop.close()
```

```
async def perform_task():
    print('performing task')
    print('waiting for result1')
    result1 = await subtask1()
    print('waiting for result2')
    result2 = await subtask2(result1)
    return (result1, result2)

async def subtask1():
    print('perform subtask 1')
    return 'result1'

async def subtask2(arg):
    print('perform subtask 2')
    return 'result2 relies on {}'.format(arg)

loop = asyncio.get_event_loop()
result = loop.run_until_complete(perform_task())
event_loop.close()
```

```
async def perform_task():
    print('performing task')
    print('waiting for result1')
    result1 = await subtask1()
    print('waiting for result2')
    result2 = await subtask2(result1)
    return (result1, result2)

async def subtask1():
    print('perform subtask 1')
    return 'result1'

async def subtask2(arg):
    print('perform subtask 2')
    return 'result2 relies on {}'.format(arg)

loop = asyncio.get_event_loop()
result = loop.run_until_complete(perform_task())
event_loop.close()
```

```
async def perform_task():
    print('performing task')
    print('waiting for result1')
    result1 = await subtask1()
    print('waiting for result2')
    result2 = await subtask2(result1)
    return (result1, result2)

async def subtask1():
    print('perform subtask 1')
    return 'result1'

async def subtask2(arg):
    print('perform subtask 2')
    return 'result2 relies on {}'.format(arg)

loop = asyncio.get_event_loop()
result = loop.run_until_complete(perform_task())
event_loop.close()
```

```
async def perform_task():
    print('performing task')
    print('waiting for result1')
    result1 = await subtask1()
    print('waiting for result2')
    result2 = await subtask2(result1)
    return (result1, result2)

async def subtask1():
    print('perform subtask 1')
    return 'result1'

async def subtask2(arg):
    print('perform subtask 2')
    return 'result2 relies on {}'.format(arg)

loop = asyncio.get_event_loop()
result = loop.run_until_complete(perform_task())
event_loop.close()
```

```
async def perform_task():
    print('performing task')
    print('waiting for result1')
    result1 = await subtask1()
    print('waiting for result2')
    result2 = await subtask2(result1)
    return (result1, result2)

async def subtask1():
    print('perform subtask 1')
    return 'result1'

async def subtask2(arg):
    print('perform subtask 2')
    return 'result2 relies on {}'.format(arg)

loop = asyncio.get_event_loop()
result = loop.run_until_complete(perform_task())
event_loop.close()
```

Parallel Execution of Tasks

```
coroutine asyncio.wait(futures, *,
                       loop=None,
                       timeout=None,
                       return_when=ALL_COMPLETED)
```

```
coroutine asyncio.wait(futures, *,
                       loop=None,
                       timeout=None,
                       return_when=ALL_COMPLETED)
```

```
coroutine asyncio.wait(futures, *,
                       loop=None,
                       timeout=None,
                       return_when=ALL_COMPLETED)

- Returns (DONE_FUTURES, PENDING_FUTURES)
```

```
coroutine asyncio.wait(futures, *,
                       loop=None,
                       timeout=None,
                       return_when=ALL_COMPLETED)

- Returns (DONE_FUTURES, PENDING_FUTURES)
```

```
import asyncio

async def get_item(i):
    await asyncio.sleep(i)
    return 'item ' + str(i)

async def get_items(num_items):
    print('getting items')
    item_coros = [
        get_item(i)
        for i in range(num_items)
    ]
    print('waiting for tasks to complete')
    completed, pending = await asyncio.wait(item_coros)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(get_items(4))
finally:
    loop.close()
```

```
import asyncio

async def get_item(i):
    await asyncio.sleep(i)
    return 'item ' + str(i)

async def get_items(num_items):
    print('getting items')
    item_coros = [
        get_item(i)
        for i in range(num_items)
    ]
    print('waiting for tasks to complete')
    completed, pending = await asyncio.wait(item_coros)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(get_items(4))
finally:
    loop.close()
```

```
import asyncio

async def get_item(i):
    await asyncio.sleep(i)
    return 'item ' + str(i)

async def get_items(num_items):
    print('getting items')
    item_coros = [
        get_item(i)
        for i in range(num_items)
    ]
    print('waiting for tasks to complete')
    completed, pending = await asyncio.wait(item_coros)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(get_items(4))
finally:
    loop.close()
```

```
import asyncio

async def get_item(i):
    await asyncio.sleep(i)
    return 'item ' + str(i)

async def get_items(num_items):
    print('getting items')
    item_coros = [
        get_item(i)
        for i in range(num_items)
    ]
    print('waiting for tasks to complete')
    completed, pending = await asyncio.wait(item_coros)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(get_items(4))
finally:
    loop.close()
```

```
...
async def get_items(num_items):
    print('getting items')
    item_coros = [
        get_item(i)
        for i in range(num_items)
    ]
    print('waiting 2 seconds for tasks to complete')
    completed, pending = await asyncio.wait(item_coros, timeout=2)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))

    if pending:
        print('canceling remaining tasks')
        for t in pending:
            t.cancel()

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(get_items(4))
finally:
    loop.close()
```

```
...
async def get_items(num_items):
    print('getting items')
    item_coros = [
        get_item(i)
        for i in range(num_items)
    ]
    print('waiting 2 seconds for tasks to complete')
    completed, pending = await asyncio.wait(item_coros, timeout=2)
    results = [t.result() for t in completed]
    print('results: {!r}'.format(results))

    if pending:
        print('canceling remaining tasks')
        for t in pending:
            t.cancel()

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(get_items(4))
finally:
    loop.close()
```

```
coroutine asyncio.wait_for(future, timeout, *, loop=None)
```

```
try:
    result = await asyncio.wait_for(task, 30.0)
except asyncio.TimeoutError:
    print('task did not complete in 30 seconds so it was canceled')
```

```
asyncio.as_completed(fs, *, loop=None, timeout=None)
```

```
for task in asyncio.as_completed(tasks):
    result = await task
```

```
asyncio.gather(*coros_or_futures, loop=None, return_exceptions=False)
asyncio.gather(*coros_or_futures, loop=None, return_exceptions=False)
asyncio.gather(*coros_or_futures, loop=None, return_exceptions=False)
```

```
import asyncio

async def get_item(i):
    await asyncio.sleep(i)
    return 'item ' + str(i)

async def get_items(num_items):
    print('getting items')
    item_coros = [
        get_item(i)
        for i in range(num_items)
    ]
    print('waiting for tasks to complete')
    results = await asyncio.gather(*item_coros)
    print('results: {!r}'.format(results))

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(get_items(4))
finally:
    loop.close()
```

```
import asyncio

async def get_item(i):
    await asyncio.sleep(i)
    return 'item ' + str(i)

async def get_items(num_items):
    print('getting items')
    item_coros = [
        get_item(i)
        for i in range(num_items)
    ]
    print('waiting for tasks to complete')
    results = await asyncio.gather(*item_coros)
    print('results: {!r}'.format(results))

loop = asyncio.get_event_loop()
try:
    loop.run_until_complete(get_items(4))
finally:
    loop.close()
```

Asyncio Libraries

aiohttp

```
pip install aiohttp
```

aiohttp server

```
import aiohttp.web

async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    return web.Response(text=text)

app = web.Application()
app.router.add_get('/', handle)
app.router.add_get('/{name}', handle)

web.run_app(app)
```

aiohttp server

```
import aiohttp.web

async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    return web.Response(text=text)

app = web.Application()
app.router.add_get('/', handle)
app.router.add_get('/{name}', handle)

web.run_app(app)
```

aiohttp server

```
import aiohttp.web

async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    return web.Response(text=text)

app = web.Application()
app.router.add_get('/', handle)
app.router.add_get('/{name}', handle)

web.run_app(app)
```

aiohttp server

```
import aiohttp.web

async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    return web.Response(text=text)

app = web.Application()
app.router.add_get('/', handle)
app.router.add_get('/{name}', handle)

web.run_app(app)
```

aiohttp server

```
import aiohttp.web

async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    return web.Response(text=text)

app = web.Application()
app.router.add_get('/', handle)
app.router.add_get('/{name}', handle)

web.run_app(app)
```

```
import aiohttp
import asyncio
import async_timeout

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

```
import aiohttp
import asyncio
import async_timeout

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

```
import aiohttp
import asyncio
import async_timeout

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

```
import aiohttp
import asyncio
import async_timeout

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

```python
import aiohttp
import asyncio
import async_timeout

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

aiofiles

aiofiles

- asyncio enabled alternative to standard file API

aiofiles

- asyncio enabled alternative to standard file API
- Similar API

aiofiles

- asyncio enabled alternative to standard file API
- Similar API
- Supports async and await

Standard File API

```
with open('filename', mode='r') as f:
    contents = f.read()
print(contents)
```

aiofiles API

```
async with aiofiles.open('filename', 
mode='r') as f:
    contents = await f.read()
print(contents)
```

Standard File API

```
with open('filename', mode='w') as f:
    f.write('data')
```

aiofiles API

```
async with aiofiles.open('filename',
mode='w') as f:
    await f.write('data')
```

aiofiles

```
pip install aiofiles
```

asyncio Libraries

asyncio Libraries

✓ aiohttp – Asynchronous web requests

asyncio Libraries

- ✓ aiohttp – Asynchronous web requests
- ✓ aiofiles – Asynchronous file I/O

asyncio Libraries

- ✓ aiohttp – Asynchronous web requests
- ✓ aiofiles – Asynchronous file I/O
- ? Other functions?

More asyncio Libraries

More asyncio Libraries

- aiomysql

More asyncio Libraries

- aiomysql
- aiopg

More asyncio Libraries

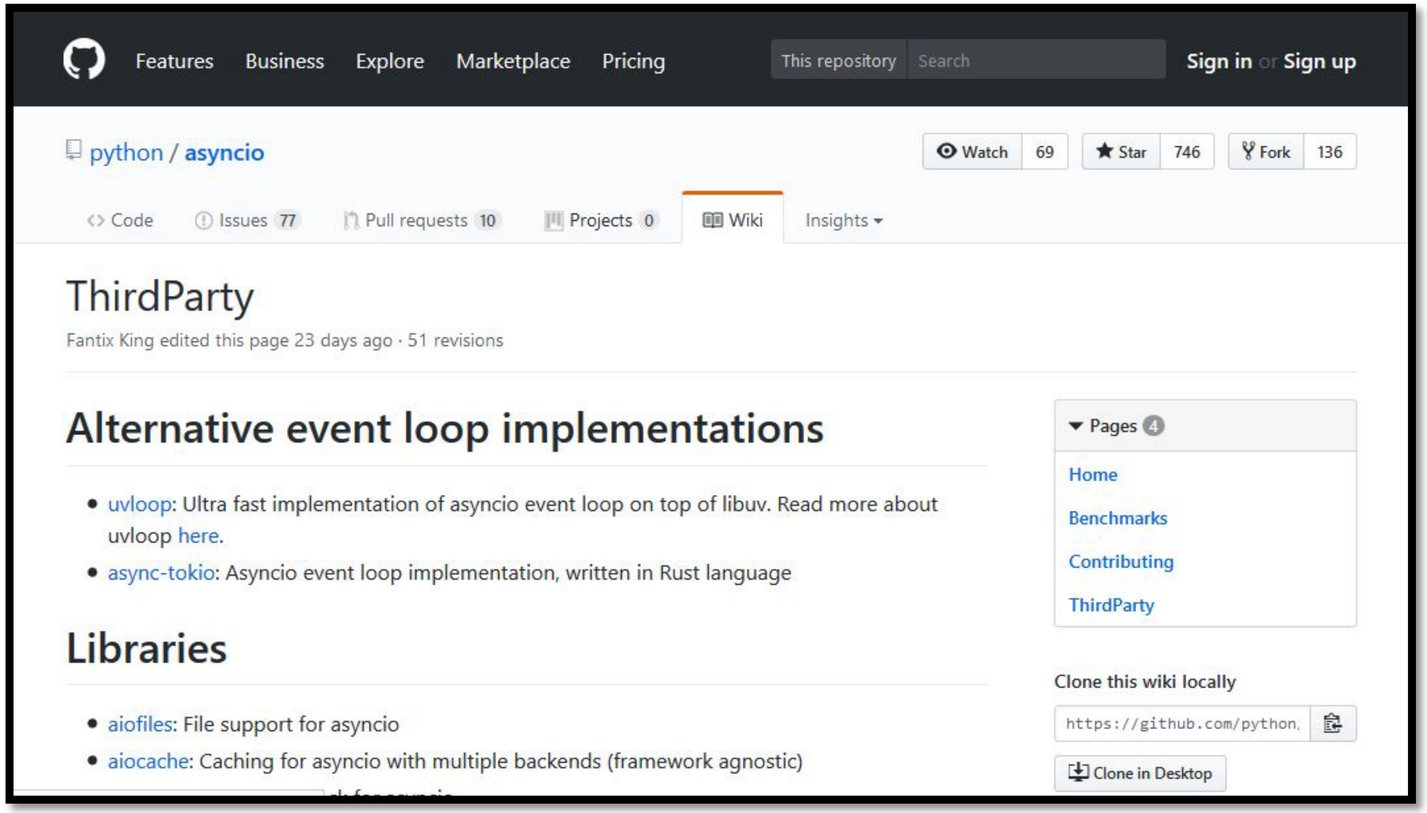
- aiomysql
- aiopg
- aiocouchdb

More asyncio Libraries

- aiomysql
- aiopg
- aiocouchdb
- aiocassandra

More asyncio Libraries

https://github.com/python/asyncio/wiki/ThirdParty



Features Business Explore Marketplace Pricing This repository Search Sign in or Sign up

python / asyncio Watch 69 Star 746 Fork 136

Code Issues 77 Pull requests 10 Projects 0 Wiki Insights

ThirdParty

Fantix King edited this page 23 days ago · 51 revisions

Alternative event loop implementations

- uvloop: Ultra fast implementation of asyncio event loop on top of libuv. Read more about uvloop here.
- async-tokio: Asyncio event loop implementation, written in Rust language

Libraries

- aiofiles: File support for asyncio
- aiocache: Caching for asyncio with multiple backends (framework agnostic)

Pages 4

Home
Benchmarks
Contributing
ThirdParty

Clone this wiki locally

https://github.com/python,

Clone in Desktop

Combining Coroutines with Threads and Processes

```
coroutine AbstractEventLoop.run_in_executor(executor, func, *args)
```

coroutine AbstractEventLoop.run_in_executor(executor, **func**, *args) *coroutine* AbstractEventLoop.run_in_executor(**executor**, func, *args) *coroutine* AbstractEventLoop.run_in_executor(executor, func, ***args**)

```
import concurrent.futures

def blocking_func(n):
    time.sleep(0.5)
    return n ** 2

async def main(loop, executor):
    print('creating executor tasks')
    blocking_tasks = [
            loop.run_in_executor(executor, blocking_func, i)
            for i in range(6)
    ]
    print('waiting for tasks to complete')
    results = await asyncio.gather(*blocking_tasks)
    print('results: {!r}'.format(results))

if __name__ == '__main__':
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=3)
    loop = asyncio.get_event_loop()
    try:
            loop.run_until_complete(main(loop, executor))
    finally:
            loop.close()
```

```
import concurrent.futures

def blocking_func(n):
    time.sleep(0.5)
    return n ** 2

async def main(loop, executor):
    print('creating executor tasks')
    blocking_tasks = [
        loop.run_in_executor(executor, blocking_func, i)
        for i in range(6)
    ]
    print('waiting for tasks to complete')
    results = await asyncio.gather(*blocking_tasks)
    print('results: {!r}'.format(results))

if __name__ == '__main__':
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=3)
    loop = asyncio.get_event_loop()
    try:
        loop.run_until_complete(main(loop, executor))
    finally:
        loop.close()
```

```
import concurrent.futures

def blocking_func(n):
    time.sleep(0.5)
    return n ** 2

async def main(loop, executor):
    print('creating executor tasks')
    blocking_tasks = [
            loop.run_in_executor(executor, blocking_func, i)
            for i in range(6)
    ]
    print('waiting for tasks to complete')
    results = await asyncio.gather(*blocking_tasks)
    print('results: {!r}'.format(results))

if __name__ == '__main__':
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=3)
    loop = asyncio.get_event_loop()
    try:
            loop.run_until_complete(main(loop, executor))
    finally:
            loop.close()
```

```
import concurrent.futures

def blocking_func(n):
    time.sleep(0.5)
    return n ** 2

async def main(loop, executor):
    print('creating executor tasks')
    blocking_tasks = [
            loop.run_in_executor(executor, blocking_func, i)
            for i in range(6)
    ]
    print('waiting for tasks to complete')
    results = await asyncio.gather(*blocking_tasks)
    print('results: {!r}'.format(results))

if __name__ == '__main__':
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=3)
    loop = asyncio.get_event_loop()
    try:
            loop.run_until_complete(main(loop, executor))
    finally:
            loop.close()
```

```
import concurrent.futures

def blocking_func(n):
    time.sleep(0.5)
    return n ** 2

async def main(loop, executor):
    print('creating executor tasks')
    blocking_tasks = [
            loop.run_in_executor(executor, blocking_func, i)
            for i in range(6)
    ]
    print('waiting for tasks to complete')
    results = await asyncio.gather(*blocking_tasks)
    print('results: {!r}'.format(results))

if __name__ == '__main__':
    executor = concurrent.futures.ThreadPoolExecutor(max_workers=3)
    loop = asyncio.get_event_loop()
    try:
            loop.run_until_complete(main(loop, executor))
    finally:
            loop.close()
```

```
import concurrent.futures

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

async def main(loop, executor, n):
    n_factorial = await loop.run_in_executor(executor, factorial, n)
    print('The factorial of {} is {}'.format(n, n_factorial))

if __name__ == '__main__':
    executor = concurrent.futures.ProcessPoolExecutor(max_workers=1)
    loop = asyncio.get_event_loop()
    n = 25
    try:
            loop.run_until_complete(main(loop, executor, n))
    finally:
            loop.close()
```

```
import concurrent.futures

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

async def main(loop, executor, n):
    n_factorial = await loop.run_in_executor(executor, factorial, n)
    print('The factorial of {} is {}'.format(n, n_factorial))

if __name__ == '__main__':
    executor = concurrent.futures.ProcessPoolExecutor(max_workers=1)
    loop = asyncio.get_event_loop()
    n = 25
    try:
            loop.run_until_complete(main(loop, executor, n))
    finally:
            loop.close()
```

```
import concurrent.futures

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

async def main(loop, executor, n):
    n_factorial = await loop.run_in_executor(executor, factorial, n)
    print('The factorial of {} is {}'.format(n, n_factorial))

if __name__ == '__main__':
    executor = concurrent.futures.ProcessPoolExecutor(max_workers=1)
    loop = asyncio.get_event_loop()
    n = 25
    try:
            loop.run_until_complete(main(loop, executor, n))
    finally:
            loop.close()
```

```
import concurrent.futures

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

async def main(loop, executor, n):
    n_factorial = await loop.run_in_executor(executor, factorial, n)
    print('The factorial of {} is {}'.format(n, n_factorial))

if __name__ == '__main__':
    executor = concurrent.futures.ProcessPoolExecutor(max_workers=1)
    loop = asyncio.get_event_loop()
    n = 25
    try:
            loop.run_until_complete(main(loop, executor, n))
    finally:
            loop.close()
```

```
import concurrent.futures

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

async def main(loop, executor, n):
    n_factorial = await loop.run_in_executor(executor, factorial, n)
    print('The factorial of {} is {}'.format(n, n_factorial))

if __name__ == '__main__':
    executor = concurrent.futures.ProcessPoolExecutor(max_workers=1)
    loop = asyncio.get_event_loop()
    n = 25
    try:
            loop.run_until_complete(main(loop, executor, n))
    finally:
            loop.close()
```

Concurrency in Python

Summary

Single Threaded Asynchrony

Cooperative Multitasking with Coroutines and Event Loops

More Asyncio Concepts

Parallel Execution of Tasks

Asyncio Libraries

Combining Coroutines with Threads and Processes

Threading

Create and manage native OS Threads

Provides synchronization and communication mechanisms

Is hampered by the GIL for CPU-bound tasks

Multiprocessing

Similar API to the Threading API

Provides synchronization and communication mechanisms

Shares state via shared memory or manager process

concurrent.futures

Provides an abstraction over threads and processes

Introduces Futures

asyncio

Brings single threaded asynchronous programming to Python

Introduces coroutines, await, async context managers, etc…

More Learning Resources

More Learning Resources

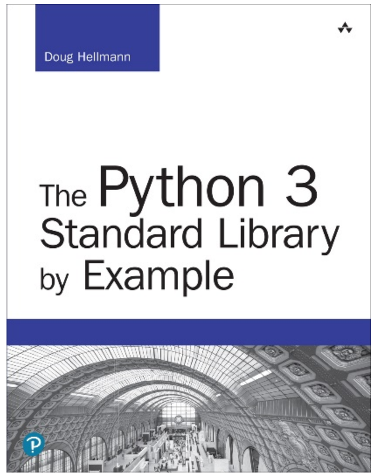
- Python documentation

More Learning Resources

- Python documentation
- Python Module of the Week blog (www.pymotw.com)

More Learning Resources

- Python documentation
- Python Module of the Week blog (www.pymotw.com)
  - The Python 3 Standard Library by Example



Python Concurrency Getting Started

Tim Ojo

@tim_ojo www.timojo.com