How async and await ended up in Python

EuroPython 2018, Edinburgh

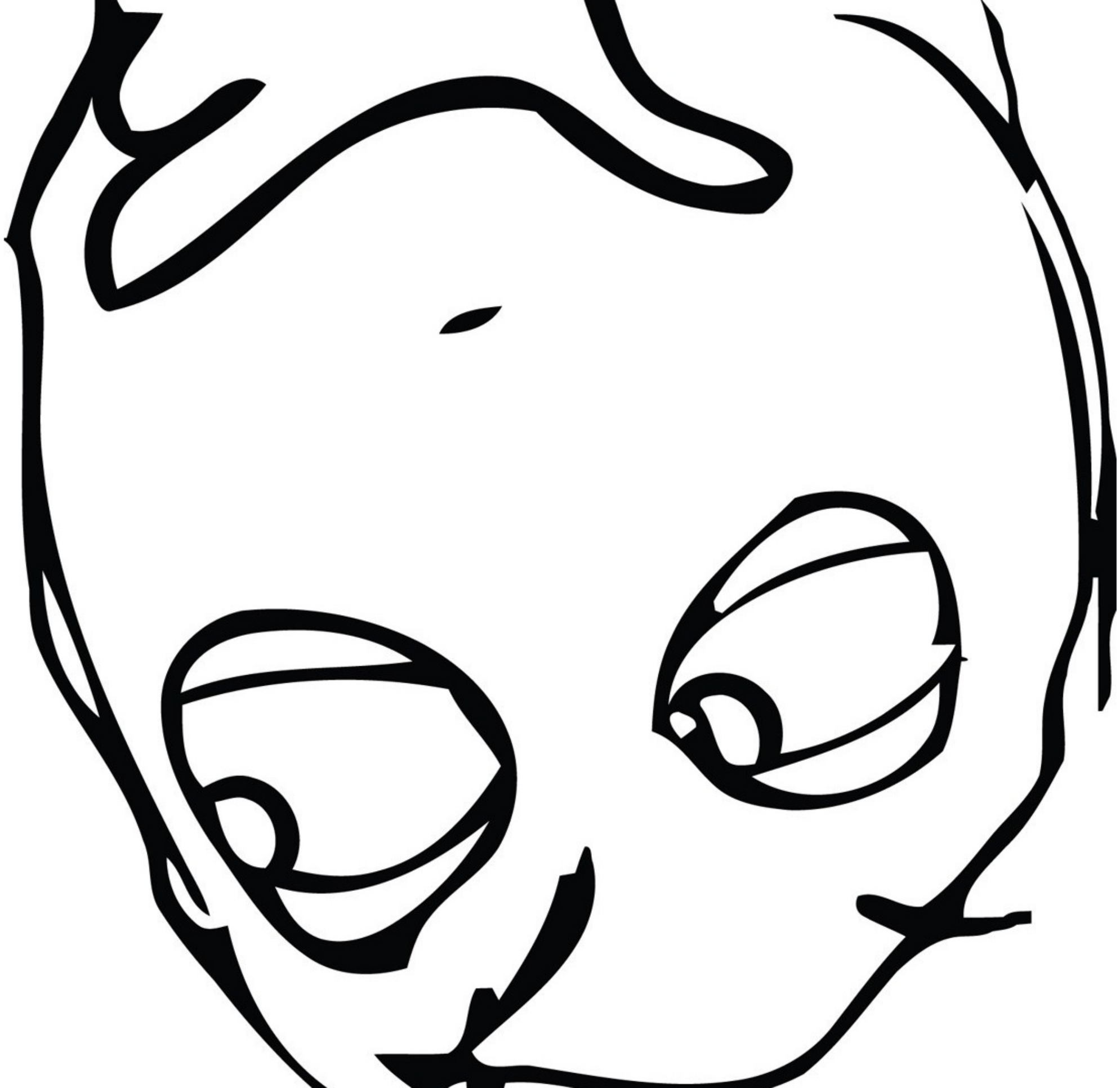
Hello (:

- https://www.hacksoft.io
- Django
- React
- Twitter: @pgergov_

Python is synchronous


Request
Response


Waiting...



Threads

Asynchronous

asyncio

hello_asyncio.py

```
import asyncio


async def hello_world_coroutine(delay):
    print('Hello')
    await asyncio.sleep(delay)
    print(f'World, with delay: {delay}')


loop = asyncio.get_event_loop()

loop.create_task(hello_world_coroutine(1))
loop.create_task(hello_world_coroutine(2))

loop.run_forever()
```

Python 3.6.3

```
HeLL
Hello
World, with delay 1
World, with delay 2
```

hello_asyncio.py

```python
import asyncio


async def hello_world_coroutine(delay):
    print('Hello')
    await asyncio.sleep(delay)
    print(f'World, with delay: {delay}')


loop = asyncio.get_event_loop()

loop.create_task(hello_world_coroutine(1))
loop.create_task(hello_world_coroutine(2))

loop.run_forever()
```

Coroutines are computer-program components that generalize subroutines for non-preemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations

Wikipedia

Dude...

Wait, what?

How async and await ended up in python

- Order of execution - raise and return
- Iterable and Iterator
- Generator functions - yield and .send
- Python 3.3 yield from
- Definition for coroutine in Python
- Python 3.4, @asyncio.coroutine
- Python 3.5, @types.coroutine
- async and await

Order of execution

throw_exception.py

```
def throw_exception():
    print('Will raise an Exception')
    raise Exception('Raised inside `throw_exception`')
print('This message won\'t be printed')
```

hello_world.py

```
def hello_world():
    print('Hello world!')
    return 42
    print('This message will never be printed')
```

yield

Iterable

__iter__

for x in iterable:

Iterator

__next__

__iter__

iterator.py

```
class MyIterator:
    def __init__(self):
        self.counter = 1

    def __iter__(self):
        return self

    def __next__(self):
        counter = self.counter

        if counter > 3:
            raise StopIteration

        self.counter += 1

        return counter
```

```
iterator = MyIterator()
next(iterator) # returns 1
next(iterator) # returns 2
next(iterator) # returns 3
next(iterator) # raises StopIteration
```

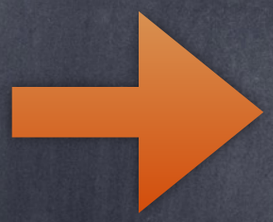
```
iterator = MyIterator()

for numb in iterator:
    print(numb)
```

```
1
2
3
```

Generator function

generator_function.py

```
def generator_function():
    print('Going to yield first value')
    yield 1
    print('Yielding second value')
    yield 2
```



```
gen = generator_function()
next(gen)
'Going to yield first value'
1
next(gen)
'Yielding second value'
2
next(gen) # raises StopIteration
```

.send

generator_send.py

```
def generator_send():
    print('Going to yield a value')
    received = yield 42
    print(f'Received {received}')
```

```
gen = generator_function()
gen.send(None)
'Going to yield value'
42
gen.send('Hello generator')
'Received Hello generator'
StopIteration is raised
```

Python 3.3

yield from

```
for x in iterator:
    yield x
```

```
yield from iterator
```

yield_from.py

```
def first_generator():
    yield 1
    print('In the middle of first generator')
    yield 2
```

```
def second_generator():
    gen = first_generator()

    yield from gen
    print('In the middle of second generator')
    yield 3
```

```
gen = second_generator()
next(gen)
1
next(gen)
In the middle of first generator
2
next(gen)
In the middle of second generator
3
next(gen) # raises StopIteration
```

Coroutines are computer-program components that generalize subroutines for non-preemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations

Wikipedia

Python 3.3

definition of coroutine in Python

Python 3.4

@asyncio.coroutine

Python 3.6.3

```
import asyncio


async def hello_world_coroutine(delay):
    print('Hello')
    await asyncio.sleep(delay)
    print(f'World, with delay: {delay}')


loop = asyncio.get_event_loop()

loop.create_task(hello_world_coroutine(1))
loop.create_task(hello_world_coroutine(2))

loop.run_forever()
```

Python 3.4

```
import asyncio


@asyncio.coroutine
def hello_world_coroutine(delay):
    print('Hello')
    yield from asyncio.sleep(delay)
    print(f'World, with delay: {delay}')


loop = asyncio.get_event_loop()

loop.create_task(hello_world_coroutine(1))
loop.create_task(hello_world_coroutine(2))

loop.run_forever()
```

Python 3.5

@types.coroutine

Python 3.5

async

async def

Python 3.5

await

__await__

Conclusion

What's next?

The superpowers of async and await

Thank you

Here's a kiss for you!

https://github.com/pgergov/europython-2018