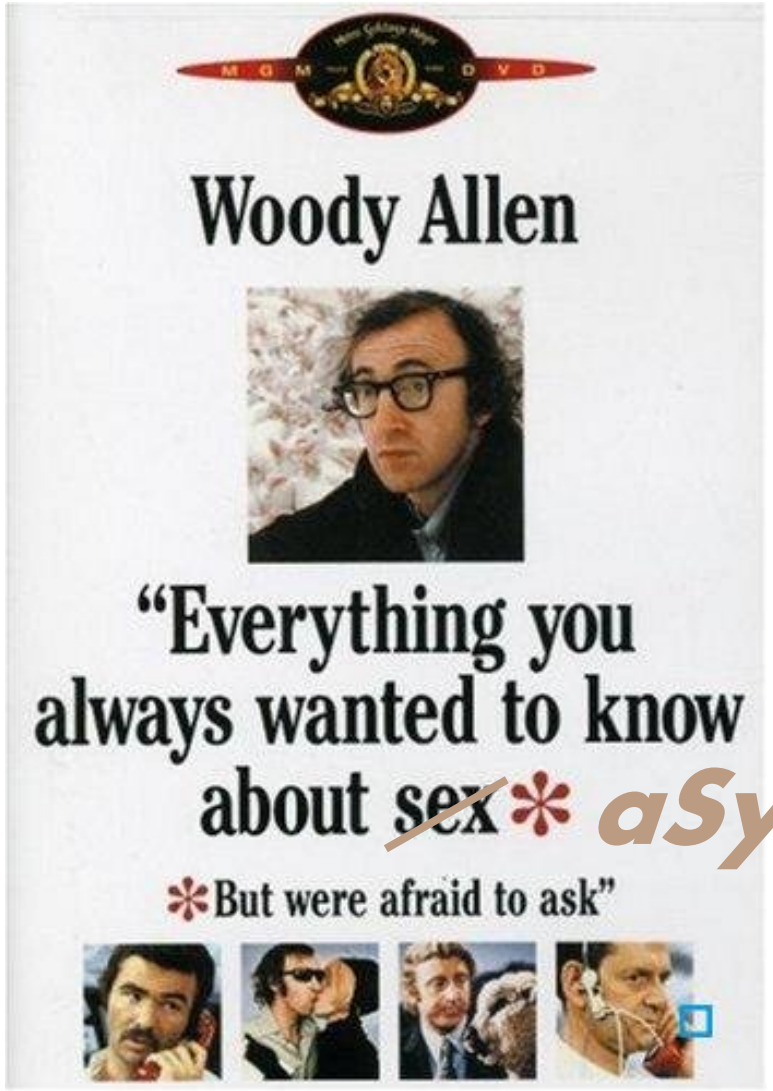
Everything You Always Wanted to Know About Async

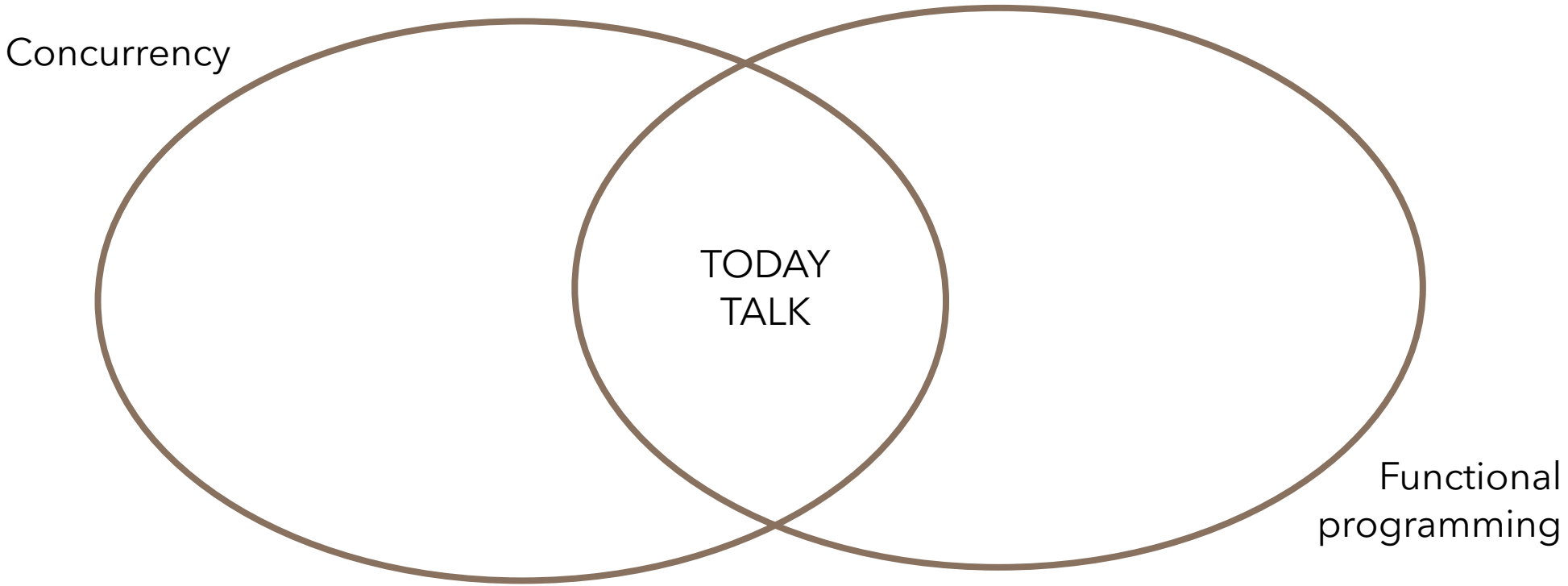
(but were afraid to ask)

Enrico Deusebio

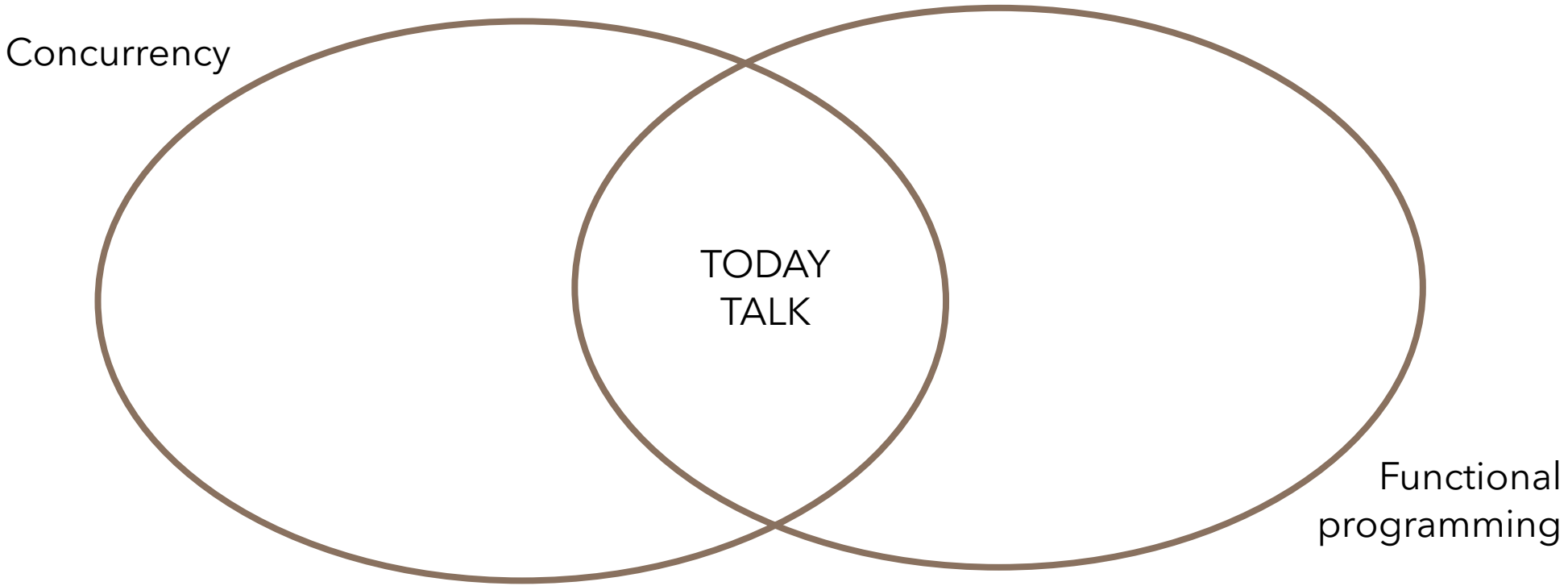
PyCon 2022, Firenze


M G M D V D
Woody Allen
"Everything you always wanted to know about sex*
*But were afraid to ask"

Agenda



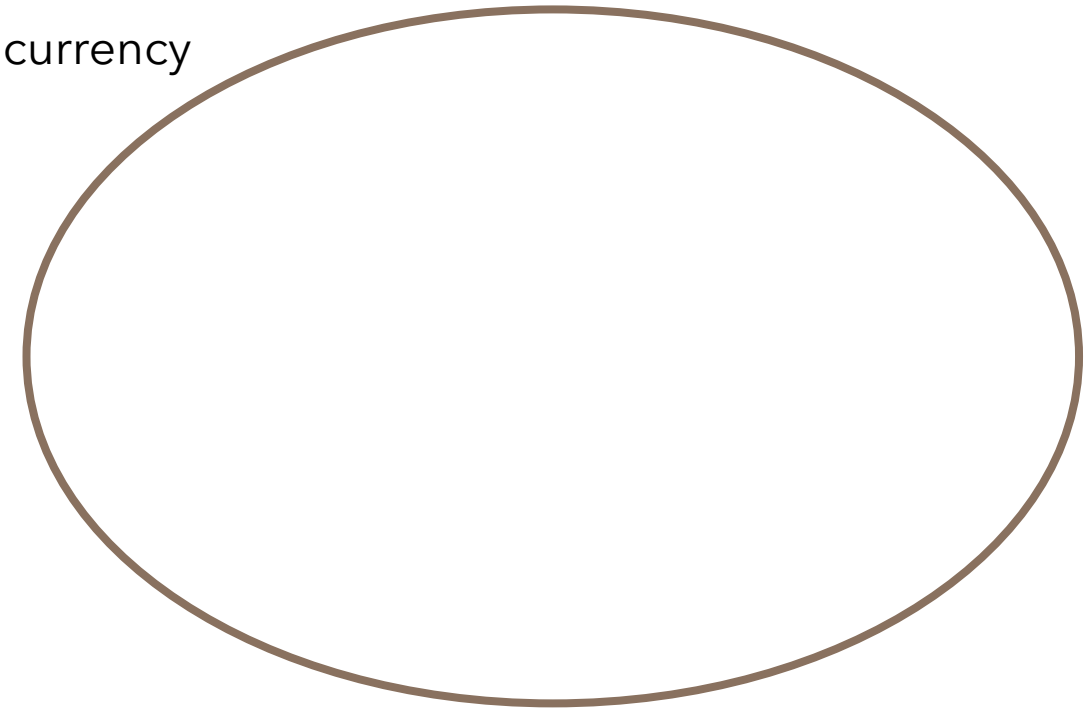
Agenda



Creating **lazy composable operations** to be executed as you please

Agenda

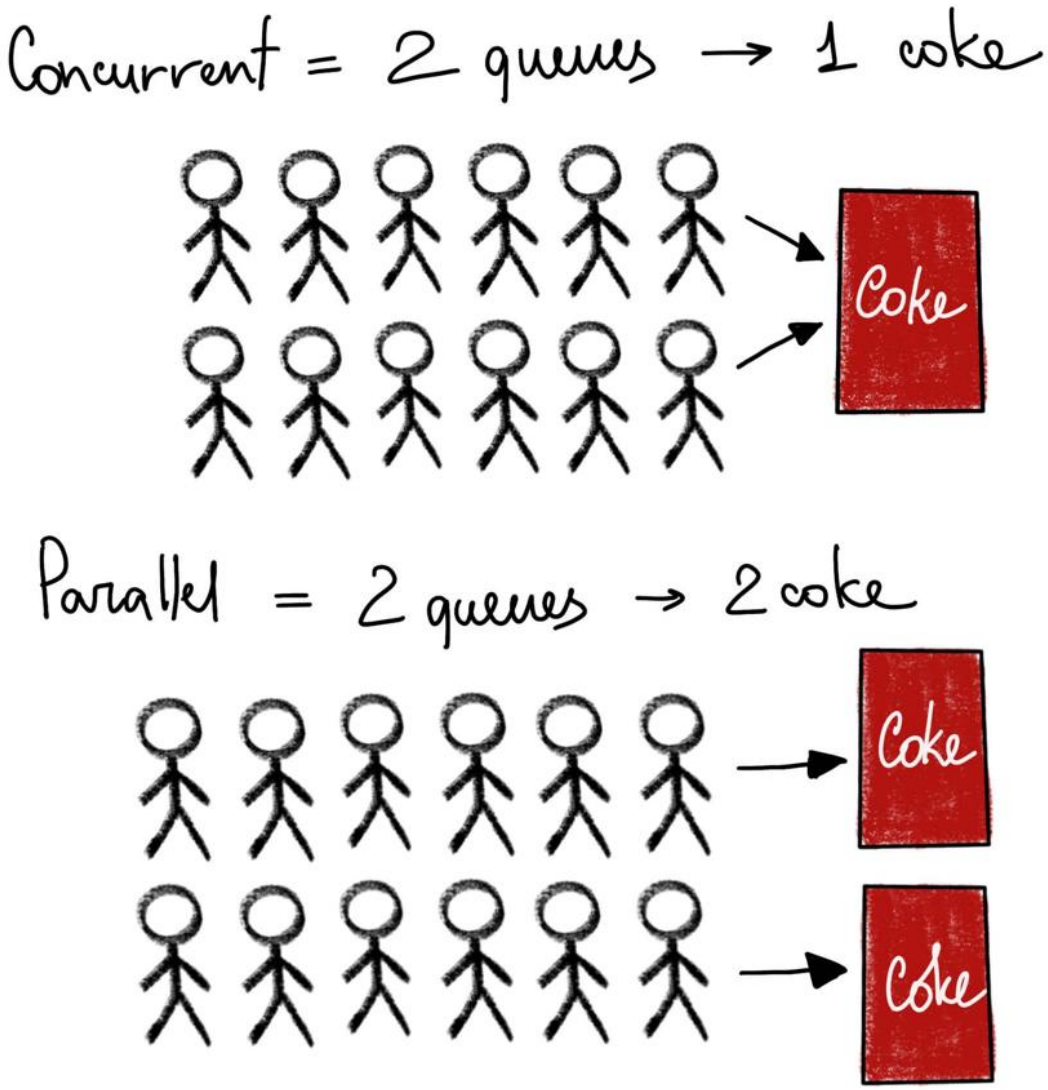
Concurrency



Concurrent

or

Parallel?



ASYNC
THREADING
(IO bound operations)

MULTIPROCESSING
(cpu bound operations)

Where the parallalelization occurs ...

ASYNCIO
(asyncio)

APPLICATION LEVEL
We need software/utilities to scheduler, run processes concurrently

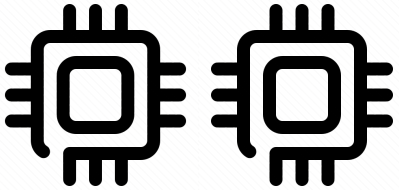
---

THREADING
(threading)

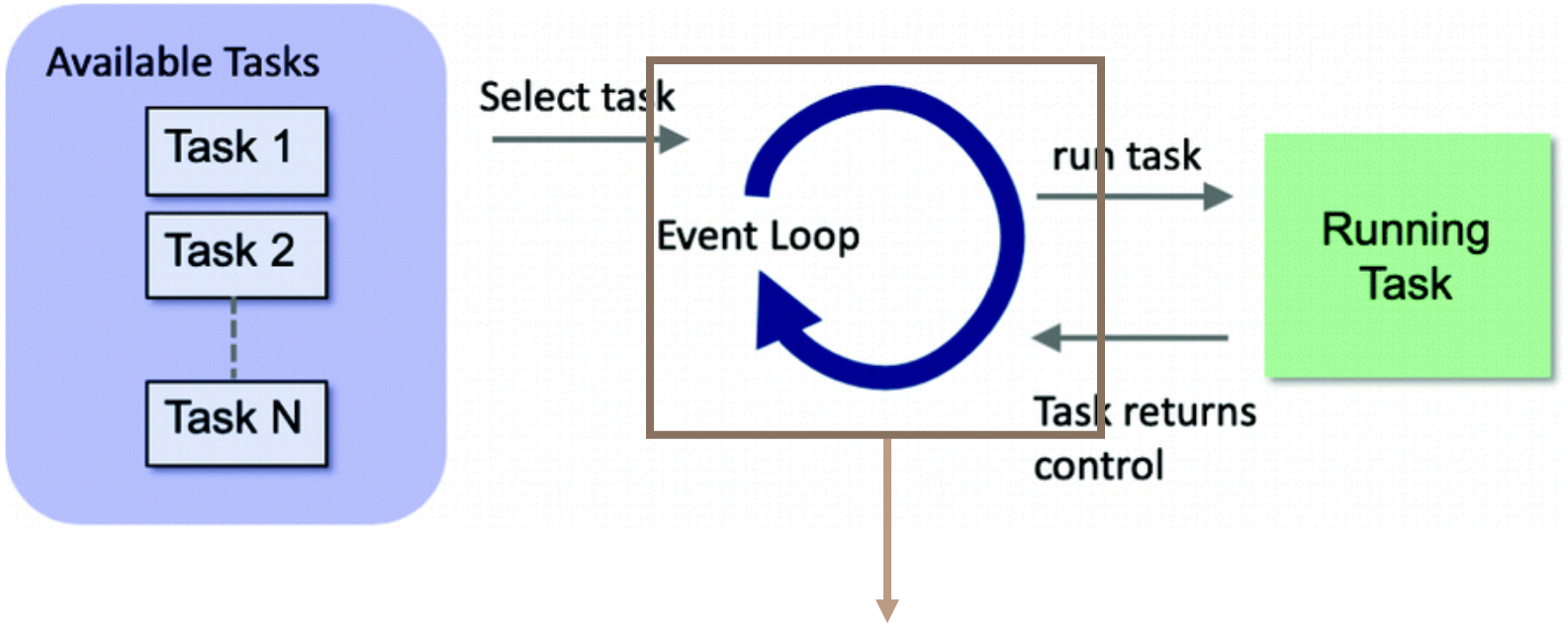
INTERPRETER
Global Interpreter Lock (GIC) does not allow parallel processing even with multi-threading env

---

MULTIPROCESSING
(multiprocessing, joblib, etc)



HARDWARE LEVEL, PROCESSES
Interprocess communication

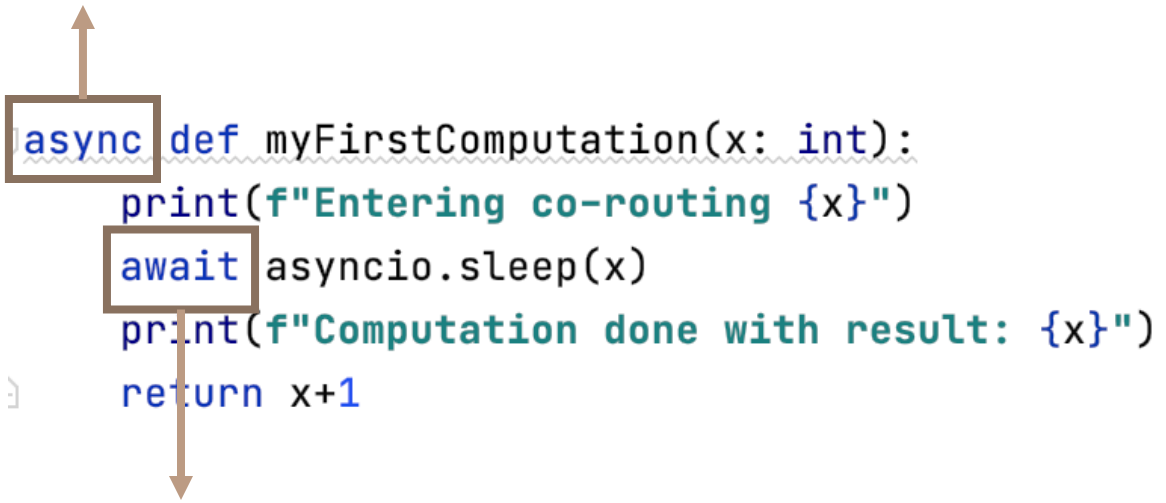


IMPLEMENTATION

The event-loop can be implemented in many ways: this leads to different packages for handling async / concurrent programming event loops, such as ***asyncio*** *and* ***uvloop****.*

CO-ROUTINE

*With async we define a **function** that can be **scheduled on the event-loop**, and therefore that can have within the function some IO-bound operations to be awaited for*



```
async def myFirstComputation(x: int):
    print(f"Entering co-routing {x}")
    await asyncio.sleep(x)
    print(f"Computation done with result: {x}")
    return x+1
```

AWAIT

***Operation that can be waited for**. At this point in the code, the «lock» of this task is released to be taken by a different task.*

Schedule a task on the event loop...

```
# Instantiate the event-loop
loop = asyncio.get_event_loop()

# Create the task on the event-loop
task = loop.create_task(main())

# Run the loop until we complete the task
loop.run_until_complete(task)
```

Co-routine

START EVENT LOOP

We get or create the event-loop where we schedule the task

RUN THE TASK

This instantiate the task to be run in the event loop

WAIT UNTIL COMPLETE

Wait the tasks to be completed

Schedule a task on the event loop...

```
# Instantiate the event-loop
loop = asyncio.get_event_loop()

# Create the task on the event-loop
task = loop.create_task(main())

# Run the loop until we complete the task
loop.run_until_complete(task)
```

From python 3.7+

```
asyncio.run(main())
```

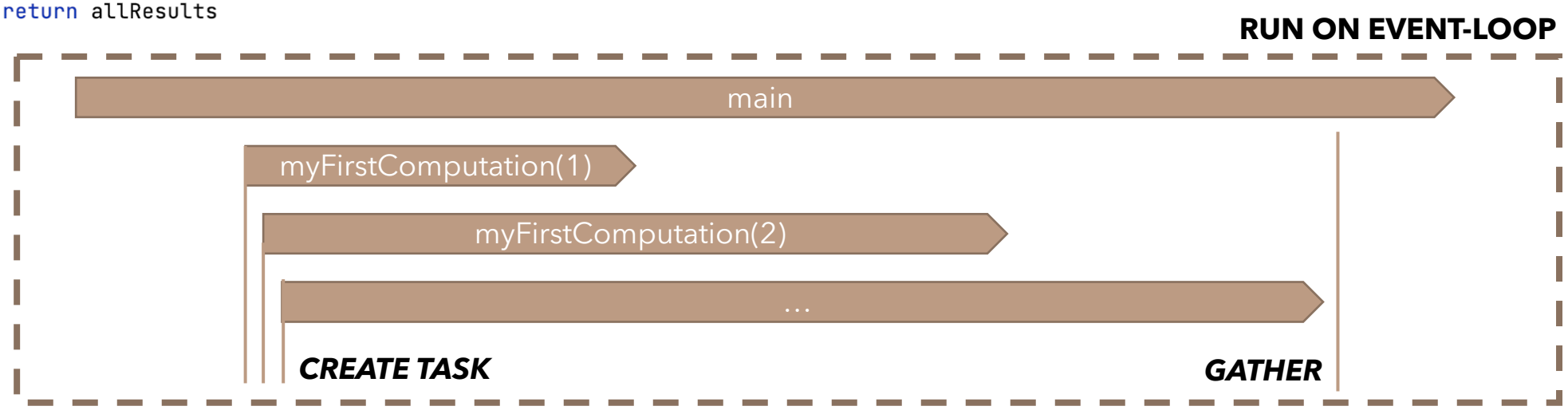
What about nested multiple async operations

```python
async def main():
    print("Starting main async process...")

    computations = [myFirstComputation(2), myFirstComputation(1), ...]

    tasks = [asyncio.create_task(c) for c in computations]

    await asyncio.sleep(1)

    print("Compuration instantiated...")

    allResults = await asyncio.gather(*tasks)

    return allResults
```

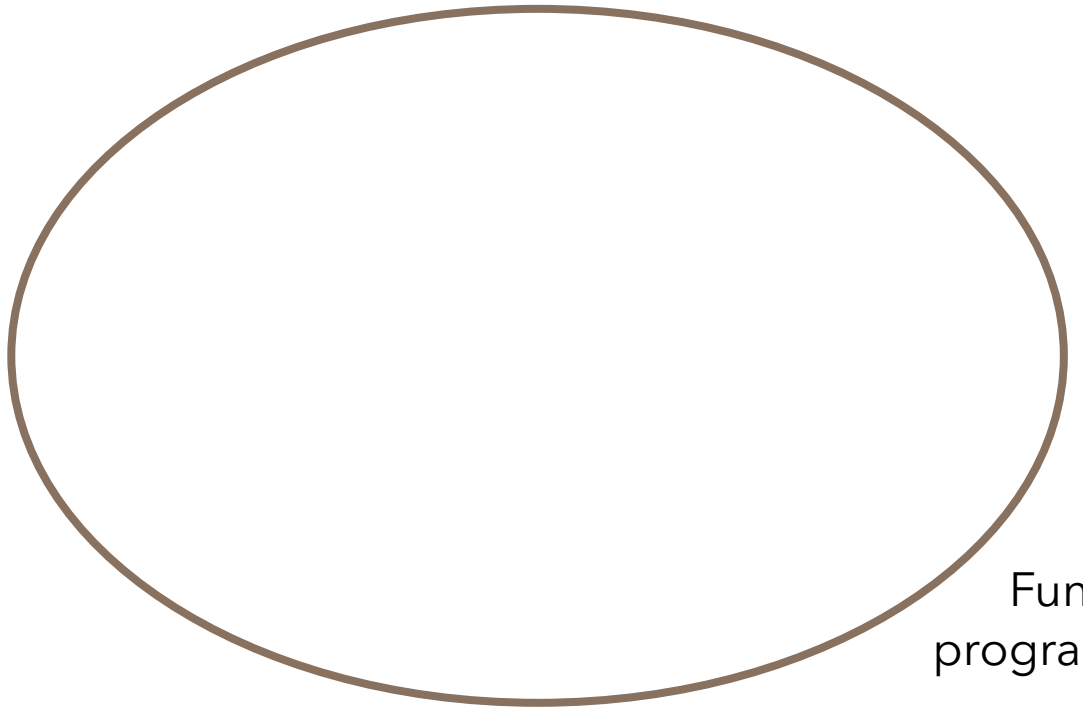
***List** of **co-routines** to be executed asynchrounously*

```python
async def myFirstComputation(x: int) -> int:
    print(f"Entering co-routing {x}")
    await asyncio.sleep(x)
    print(f"Computation done with result: {x}")
    return x+1
```

GATHER: *Returns all results once they have **all been computed***



Agenda



Basic concepts for functional programming

+ *Pure functions*.

  A pure function is a function which:

  - Given the same inputs, always return the same outputs
  - Has no-side effects

+ *Immutability*.

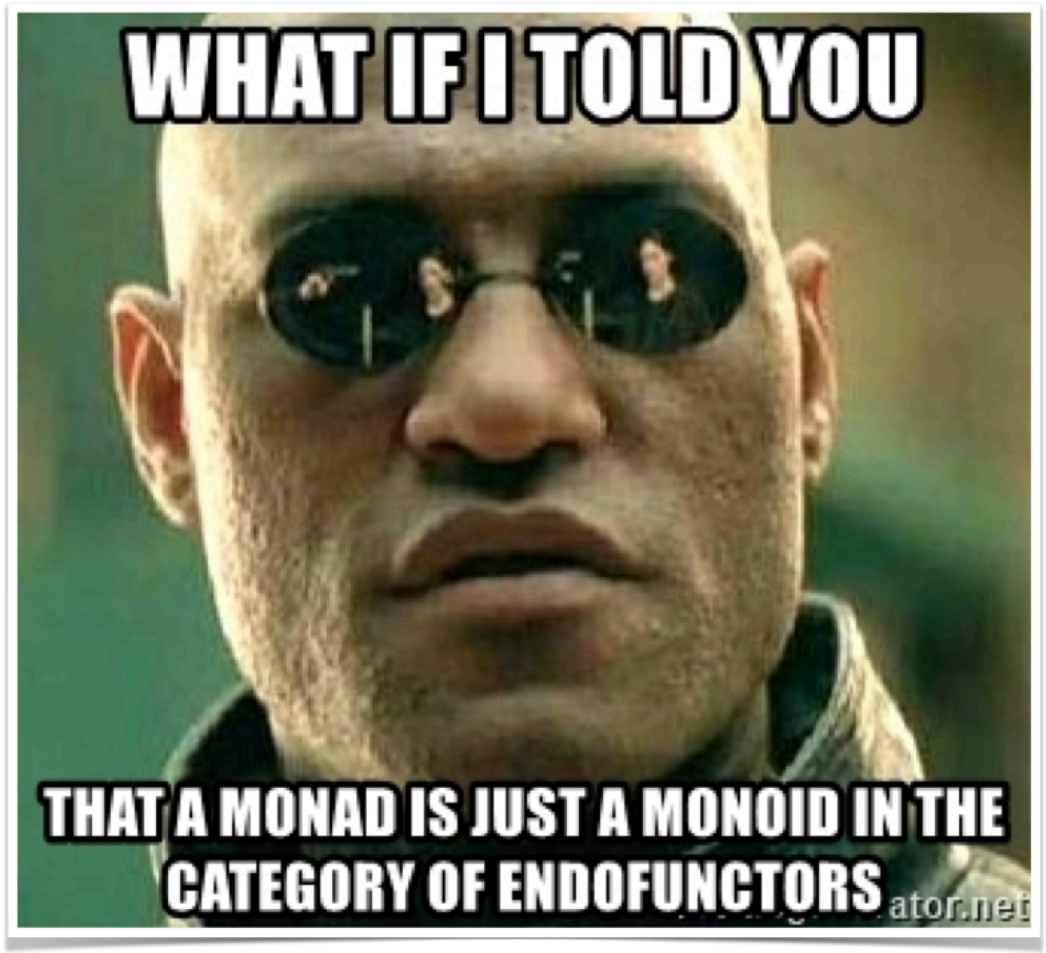
  Objects once instantiated to not change

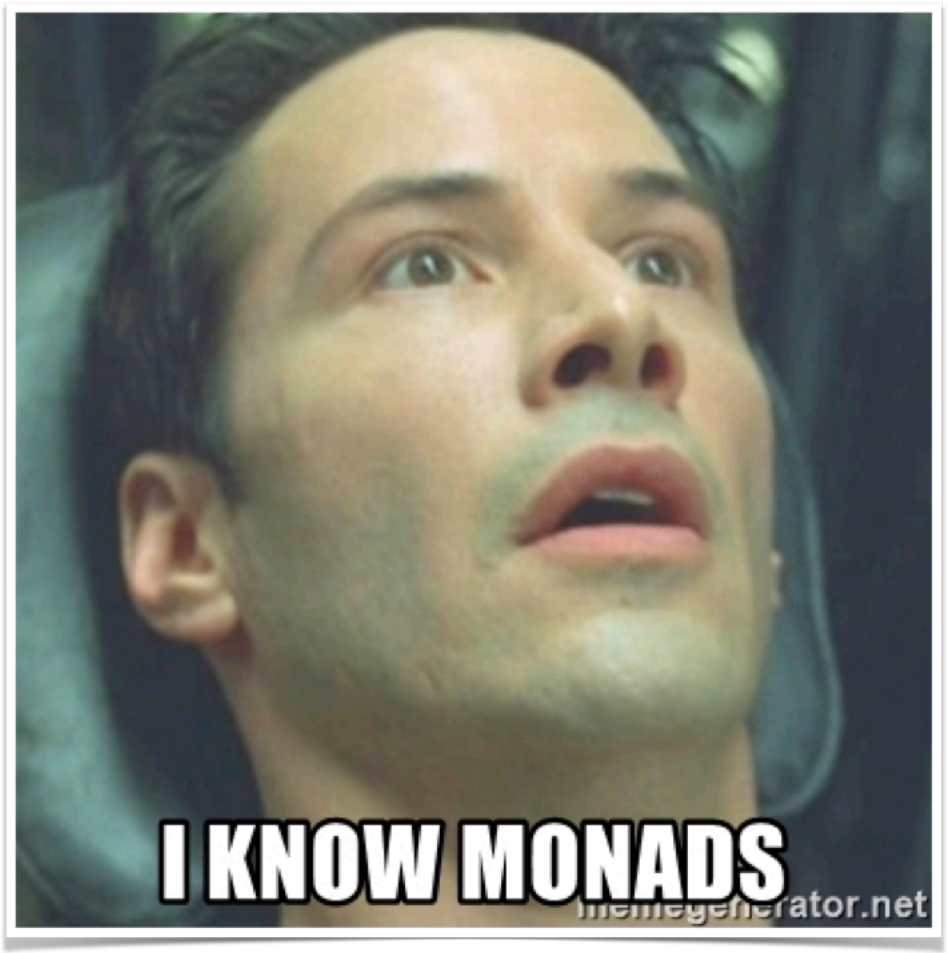
+ *Referential Transparancy*.

  Consequence of dealing with pure functions: if we replace a function call with its return value, nothing should change in the program

+ *Functions as first-class entities and higher order functions*.

  Functions can be in free state, declared anywhere and uses a input and/or outpu values for functions. We can then defined higher-order functions that takes function as arguments and returns functions as outputs. We have some interesing outcome from this as ***Currying*** and ***Composition***





"Once you understand monads, you immediately become incapable of explaining them to anyone else"

(Lady Monadgreen's curse by Gilad Bracha)

A monad is «just» a container...

...Promise!

It represents the **result of a computation**, with the value of the computation to be populated at a later stage, when we **(a)wait**. It really expands the concepts of async opterations by adding two things:

- *Composability*
- *Error Handling*

NOTEBOOK

Abstract Computation

The Promise Container

```
In [1]: from pymonad.promise import Promise, _Promise

In [2]: Promise.insert(1)
Out[2]: <pymonad.promise._Promise at 0x10f034cd0>

In [3]: await Promise.insert(1)
Out[3]: 1
```

Composability (map)

Map Method

In [4]:
```
import asyncio

async def wait_and_then_add_one(x: int) -> int:
    print(f"before waiting {x} sec...")
    await asyncio.sleep(x)
    print(f"after waiting {x} sec...")
    return x+1
```

In [5]:
```
def add_one(x: int) -> int:
    return x + 1
```

In [6]:
```
p = Promise.insert(1).map(wait_and_then_add_one).map(add_one)
```

In [7]:
```
p
```

Out[7]:
```
<pymonad.promise._Promise at 0x10f048c50>
```

In [8]:
```
await Promise.insert(1).map(wait_and_then_add_one).map(wait_and_then_add_one)
```

```
before waiting 1 sec...
after waiting 1 sec...
before waiting 2 sec...
after waiting 2 sec...
```

Out[8]:
```
3
```

Composability (bind and then)

Bind method

In [20]:
```
def generate_new_promise(x: int) -> _Promise[int]:
    print("generating a new promise")
    return Promise.insert(x).map(wait_and_then_add_one)
```

In [21]:
```
p = Promise.insert(1).bind(generate_new_promise)
```

In [22]:
```
await p
```

```
generating a new promise
before waiting 1 sec...
after waiting 1 sec...
```

Out[22]: 2

Then method

In [12]:
```
await Promise.insert(1).then(generate_new_promise).then(wait_and_then_add_one)
```

```
generating a new promise
before waiting 1 sec...
after waiting 1 sec...
before waiting 2 sec...
after waiting 2 sec...
```

Out[12]: 3

Exception Handling (1)

Catching Exception

In [13]:
```
def raise_exception_when_too_large(upper_bound: int):
    def wrapper(value: int):
        if value > upper_bound:
            raise ValueError(f"Too large. Value must be lower than {upper_bound}")
        return value
    return wrapper
```

In [14]:
```
max_limit = 2
```

In [15]:
```
p = Promise.insert(1)\
        .then(generate_new_promise)\
        .then(wait_and_then_add_one)\
        .then(raise_exception_when_too_large(2))
```

In [16]:
```
await p
```

```
generating a new promise
before waiting 1 sec...
after waiting 1 sec...
before waiting 2 sec...
after waiting 2 sec...

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-16-b74e1a80ad7c> in async-def-wrapper()

~/.pyenv/versions/3.7.6/envs/async/lib/python3.7/site-packages/pymonad/promise.py in _then(resolve, _)
```

Exception Handling (2)

In [23]:
```
def resetting_to_upper_bound(upper_bound: int):
    def wrapper(e: Exception):
        print(f"Raised Exception: {e} \n Falling back to {upper_bound}")
        return upper_bound
    return wrapper
```

In [24]:
```
safe_p = p = Promise.insert(1)\
        .then(generate_new_promise)\
        .then(wait_and_then_add_one)\
        .then(raise_exception_when_too_large(2))\
        .catch(resetting_to_upper_bound(max_limit))
```

In [25]:
```
await safe_p
```

```
generating a new promise
before waiting 1 sec...
after waiting 1 sec...
before waiting 2 sec...
after waiting 2 sec...
Raised Exception: Too large. Value must be lower than 2
 Falling back to 2
```

Out[25]:
```
2
```

```
async def main():
    x = Promise.insert(1).map(fakeComputation)
    y = Promise.insert(2).map(fakeComputation)
    z = Promise.insert(3).map(fakeComputation)
    promiseSum = my_func(x, y=1, z=z)
    final_result = promiseSum.map(fakeComputation)
    print("Here we have instantiated the computation, but we have not done anything")
    value = await final_result
    print(value)
```

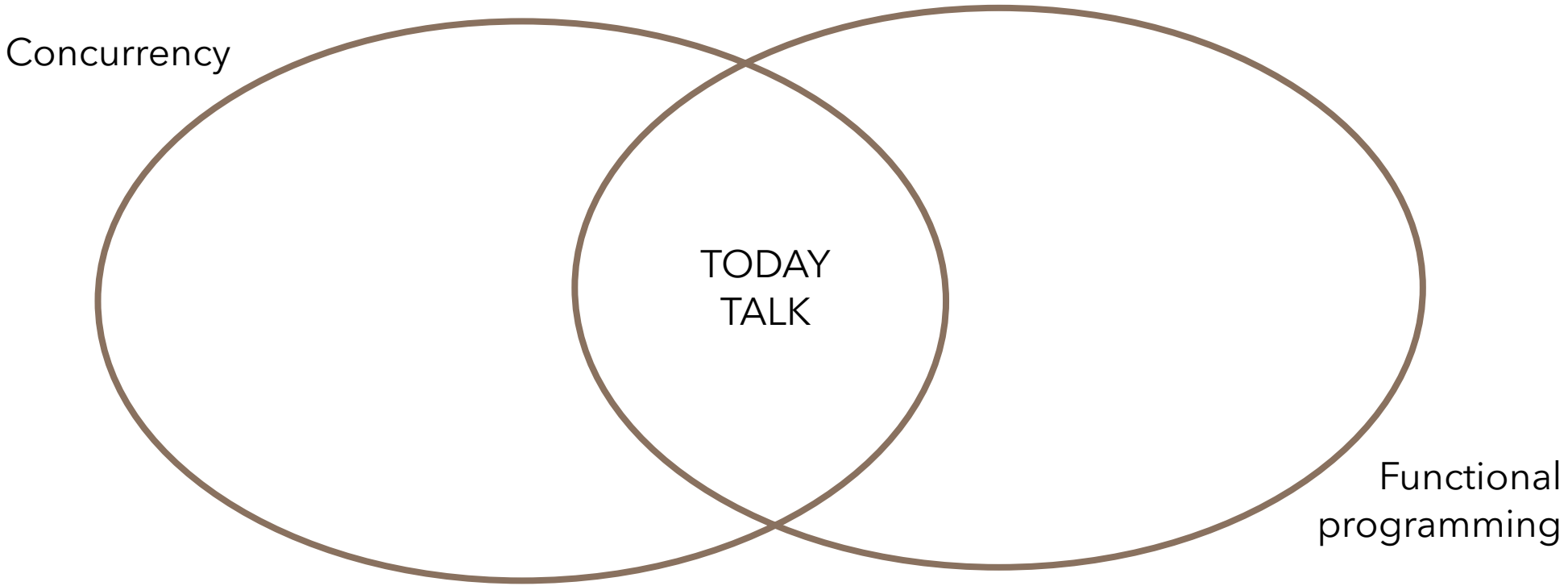
PROMISES DEFINITION

```
async def main():
    x = Promise.insert(1).map(fakeComputation)
    y = Promise.insert(2).map(fakeComputation)
    z = Promise.insert(3).map(fakeComputation)
    promiseSum = my_func(x, y=1, z=z)
    final_result = promiseSum.map(fakeComputation)
    print("Here we have instantiated the computation, but we have not done anything")
    value = await final_result
    print(value)
```

Decorator to transform a normal function into a function that can handle Promises

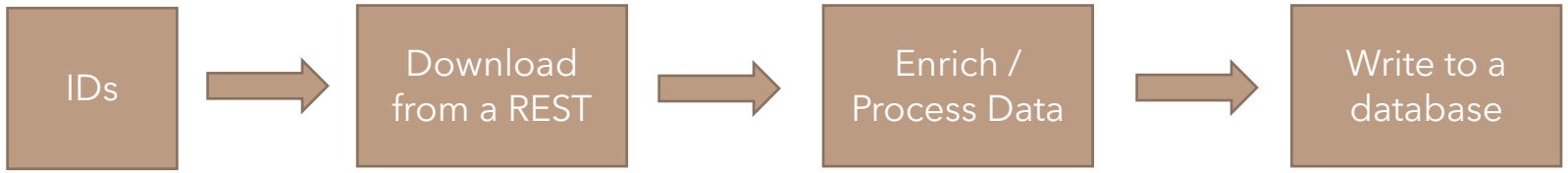
```
@async_func
def my_func(x: int, y: int = 1, z: int = 1):
    return (x + 2*y)/z
```

Agenda



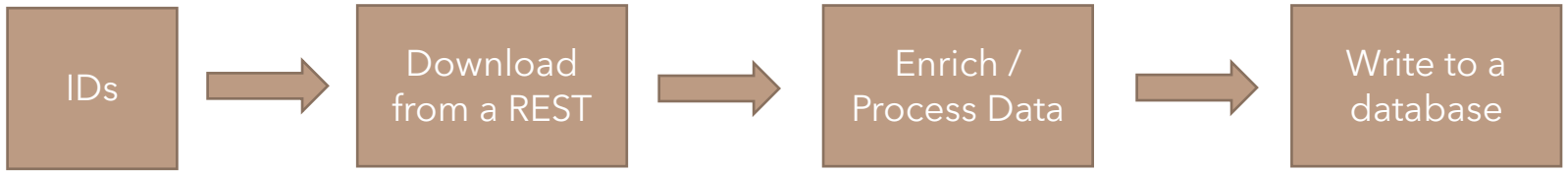
Creating **lazy composable operations** to be executed as you please

USE CASE: Download – Enrich - Store




```
def process(value: T) -> _Promise[S]:
    return Promise.insert(value)\
        .then(getDataFromAPI)\
        .then(myComputation)\
        .then(writeToDb)
```

USE CASE: Download – Enrich - Store




```
def process(value: T) -> _Promise[S]:
    return Promise.insert(value)\
        .then(getDataFromAPI)\
        .then(myComputation)\
        .then(writeToDb)
```

A bit more general and extensible

```
def pipeline(steps: List[Callable[[T], Union[S, _Promise[S]]]]) \
        -> Callable[[T], _Promise[S]]:
    def wrapper(value: T) -> _Promise[S]:
        return reduce(
            lambda promise, step: promise.then(step),
            steps,
            Promise.insert(value)
        )
    return wrapper


process: Callable[[T], _Promise[S]] = pipeline([
    getDataFromAPI,
    myComputation,
    writeToDb
])
```

```python
async def main():
    print("Starting...")

    process: Callable[[T], _Promise[S]] = pipeline([
        getDataFromAPI,
        myComputation,
        writeToDb
    ])

    executions = [Promise.insert(i).then(process) for i in range(10)]



    print("Compuration instantiated...")

    return asyncio.gather(*executions)
```

PIPELINE DEFINITION

We define what we should do and the execution pipeline

ABSTRACT COMPUTATION

We apply the abstract computation to values, but we do not execute them yet

GO!

Now we get the execution of all of them

```python
async def main():
    print("Starting...")

    process: Callable[[T], _Promise[S]] = pipeline([
        getDataFromAPI,
        myComputation,
        writeToDb
    ])

    executions = [Promise.insert(i).then(process) for i in range(10)]



    print("Compuration instantiated...")

    return asyncio.gather(*executions)
```

PIPELINE DEFINITION

We define what we should do and the execution pipeline

ABSTRACT COMPUTATION

We apply the abstract computation to values, but we do not execute them yet

GO!

Now we get the execution of all of them

BUT CAN WE THROTTLE THIS EXECUTION TO LIMIT THE REQUEST MADE TO THE API?

```python
async def main():
    print("Starting...")

    process: Callable[[int], _Promise[Either[Exception, int]]] = pipe
        getDataFromAPI,
        mySafeComputation,
        writeToDb
    ])

    computations = [Promise.insert(i).then(process) for i in range(50)]



    print("Compuration instantiated...")

    rate_limit = AsyncLimiter(max_rate=10, time_period=10)

    return await asyncio.gather(*[throttle(c, rate_limit) for c in computations])
```

```python
from aiolimiter import AsyncLimiter

async def throttle(c, rate_limit):
    async with rate_limit:
        return await c
```

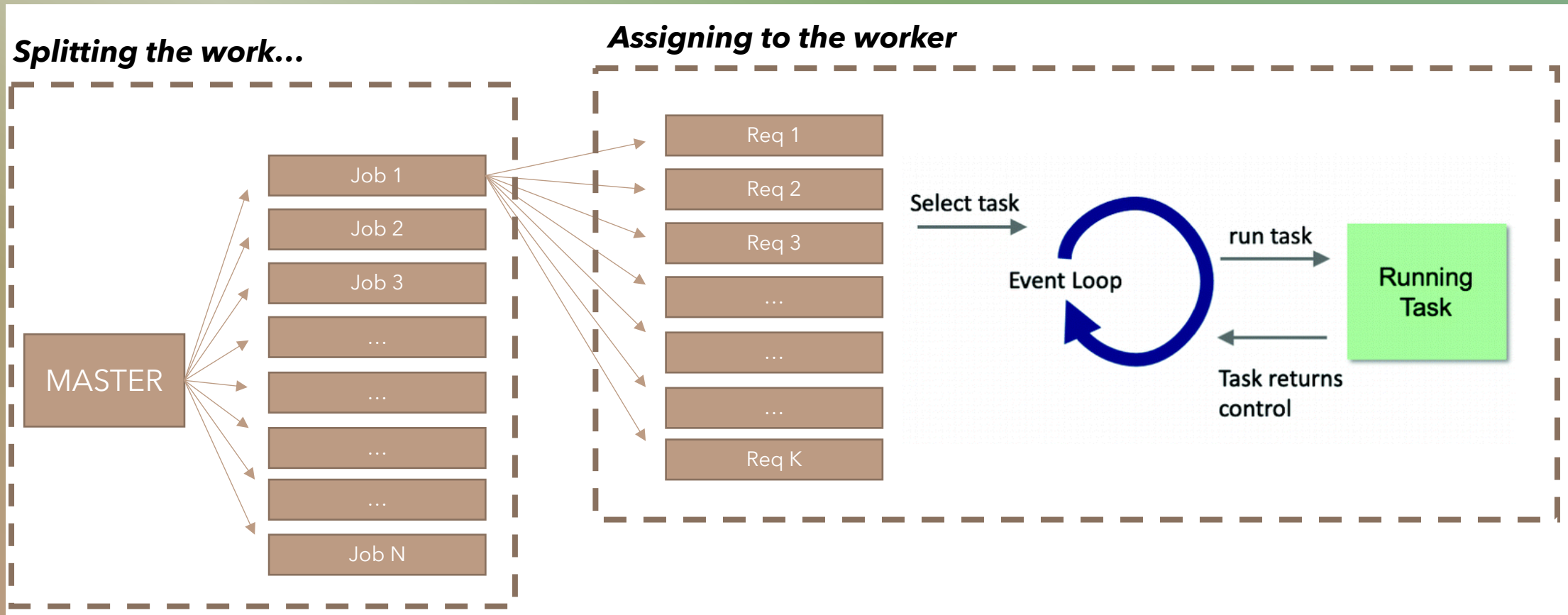
By only changing very few lines and plugging this function we can throttle the execution as we please!

```python
async def main():
    print("Starting...")

    process: Callable[[int], _Promise[Either[Exception, int]]] = pipe
        getDataFromAPI,
        mySafeComputation,
        writeToDb
    ])

    computations = [Promise.insert(i).then(process) for i in range(50)]



    print("Compuration instantiated...")

    rate_limit = AsyncLimiter(max_rate=10, time_period=10)

    return await asyncio.gather(*[throttle(c, rate_limit) for c in computations])
```

```python
from aiolimiter import AsyncLimiter

async def throttle(c, rate_limit):
    async with rate_limit:
        return await c
```

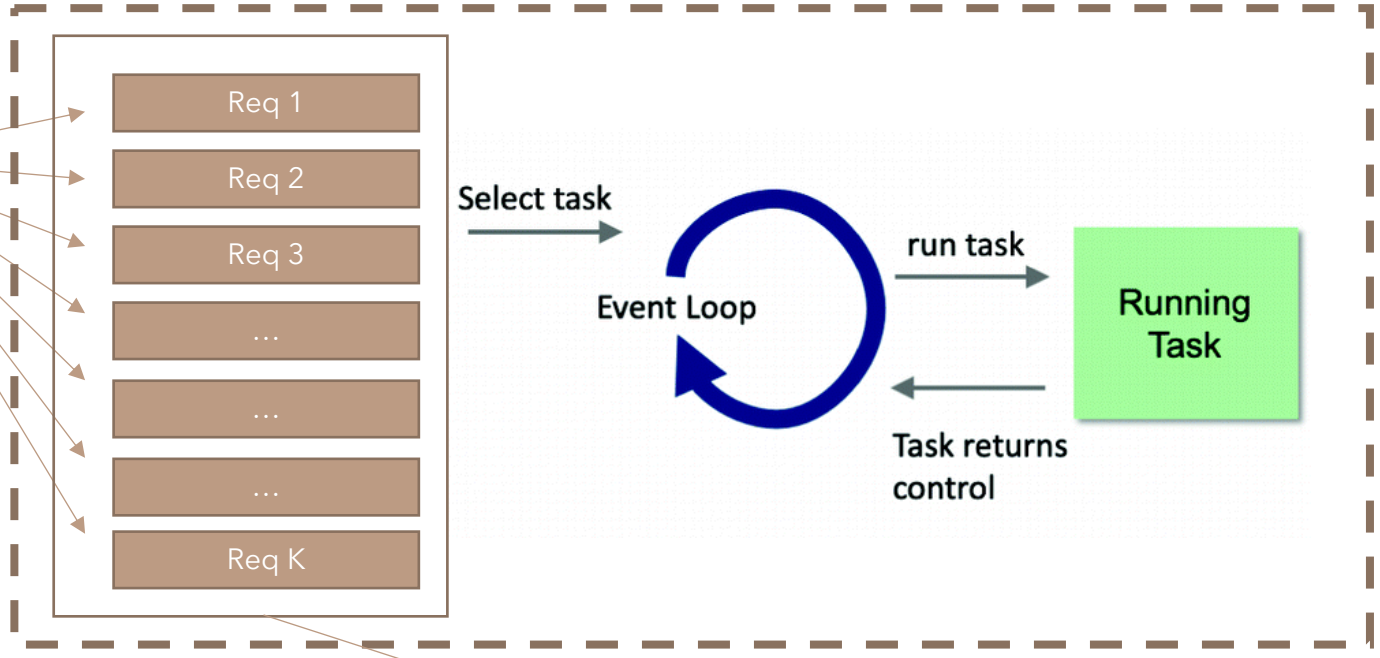
By only changing very few lines and plugging this function we can throttle the execution as we please!

BUT WHAT IF THE OPERATIONS WE HAVE TO PERFORM REQUIRES COMPUTATIONS ?


Splitting the work...
Assigning to the worker
MASTER
Job 1
Job 2
Job 3
...
...
...
...
Job N
Req 1
Req 2
Req 3
...
...
...
Req K
Select task
Event Loop
run task
Running Task
Task returns control

Splitting the work...

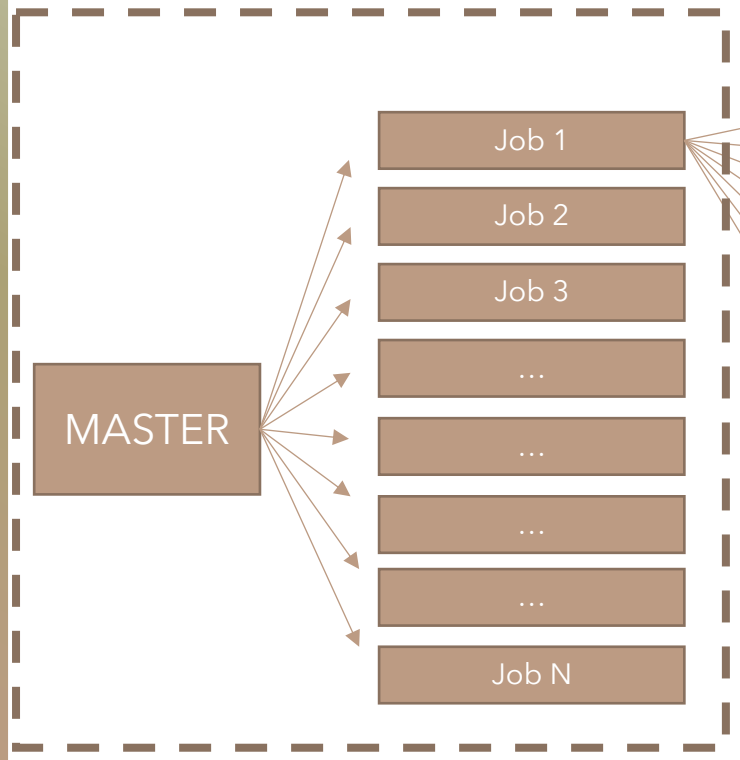
Assigning to the worker



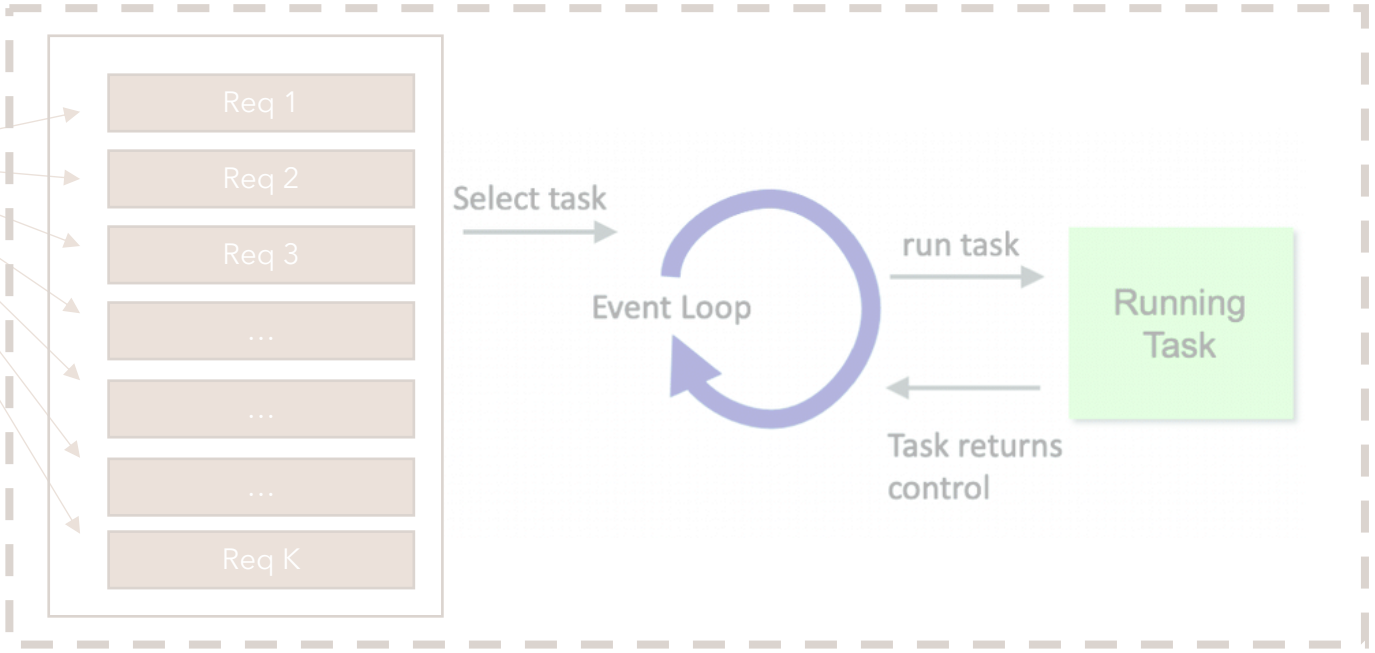

```
async def loopOverRange(input_range, process):

    computations = [Promise.insert(i).then(process) for i in input_range]

    rate_limit = AsyncLimiter(max_rate=10, time_period=10)

    await asyncio.sleep(1)

    print("Compuration instantiated...")

    return await asyncio.gather(*[throttle(c, rate_limit) for c in computations])
```

```
def getProcess() -> Callable[[int], _Promise[Either[Exception, int]]]:
    return pipeline([
        getDataFromAPI,
        mySafeComputation,
        writeToDb
    ])
```

Splitting the work...



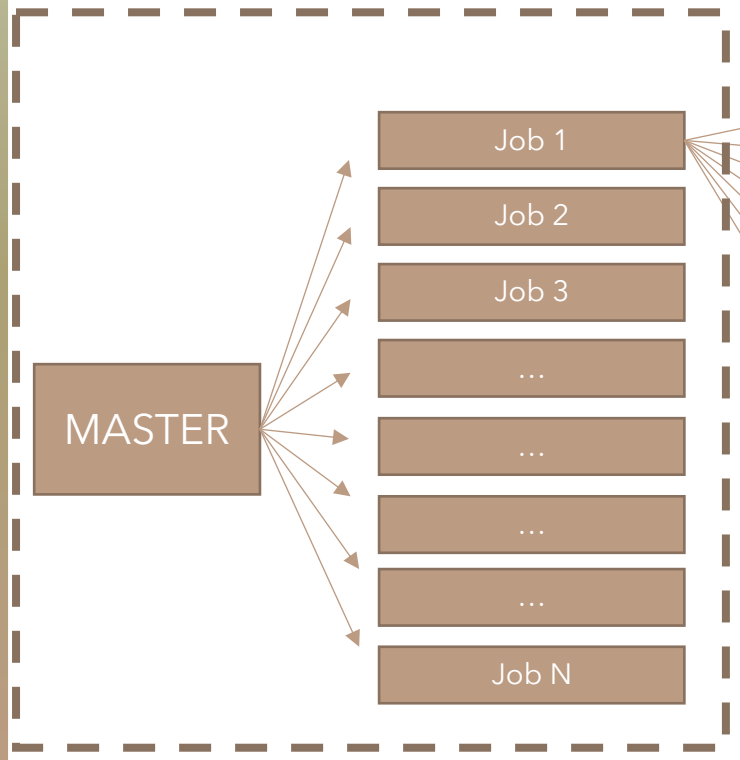
Assigning to the worker



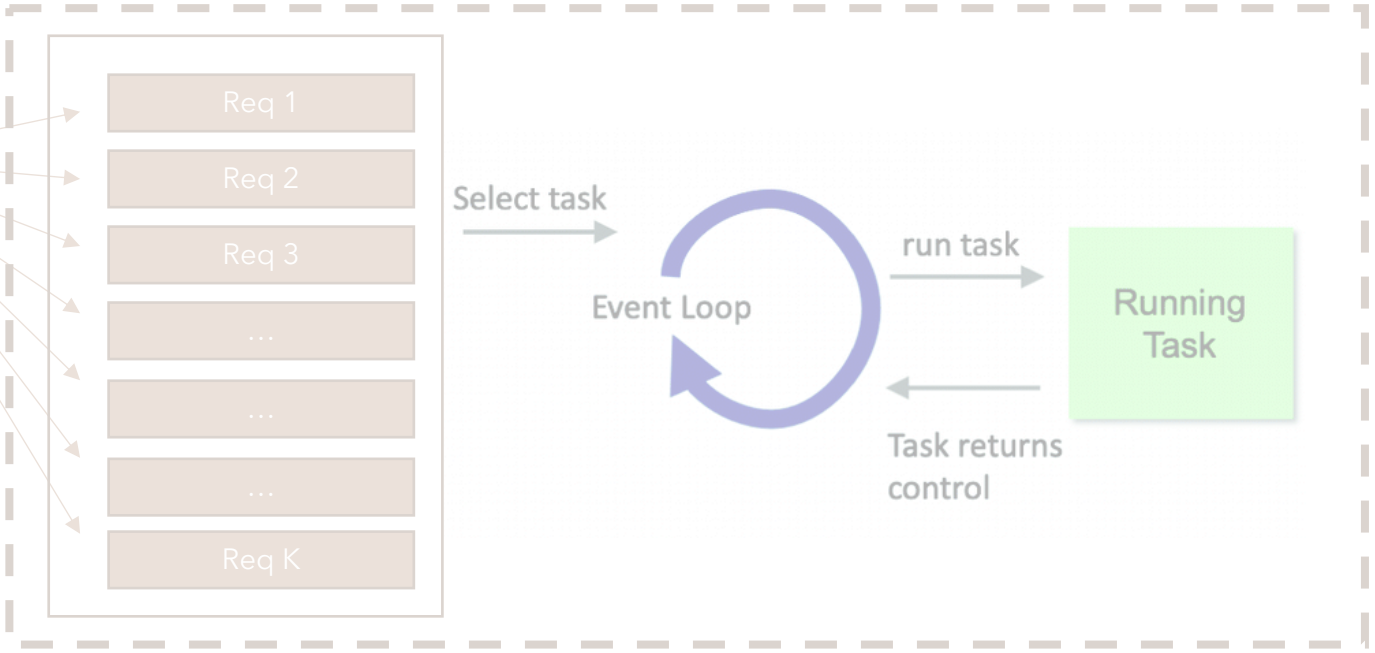

```python
from cgnal.utils.dict import groupIterable
from concurrent.futures import ProcessPoolExecutor
import concurrent

batch_size = 10
total_range = list(range(100))

futures = []

jobs = groupIterable(total_range, batch_size)

with ProcessPoolExecutor(max_workers=2) as executor:
    for ibatch, batch in enumerate(jobs):
        futures.append(executor.submit(basicMain, ibatch, batch))
```

Job splitting

Splitting the work...



Assigning to the worker



async-job definition

```
def basicMain(ibatch, input_range):
    print(f"Running batch {ibatch}")
    return asyncio.run(
        loopOverRange(
            input_range, getProcess()
        )
    )
```

```
from cgnal.utils.dict import groupIterable
from concurrent.futures import ProcessPoolExecutor
import concurrent

batch_size = 10
total_range = list(range(100))

futures = []

jobs = groupIterable(total_range, batch_size)
```

Worker creation and scheduling

```
with ProcessPoolExecutor(max_workers=2) as executor:
    for ibatch, batch in enumerate(jobs):
        futures.append(executor.submit(basicMain, ibatch, batch))
```

References

- Python libraries

  https://docs.python.org/3/library/asyncio.html

  https://github.com/jasondelaat/pymonad

  https://github.com/dry-python/returns

- Examples

  https://github.com/CGnal/async-examples


Thank you!