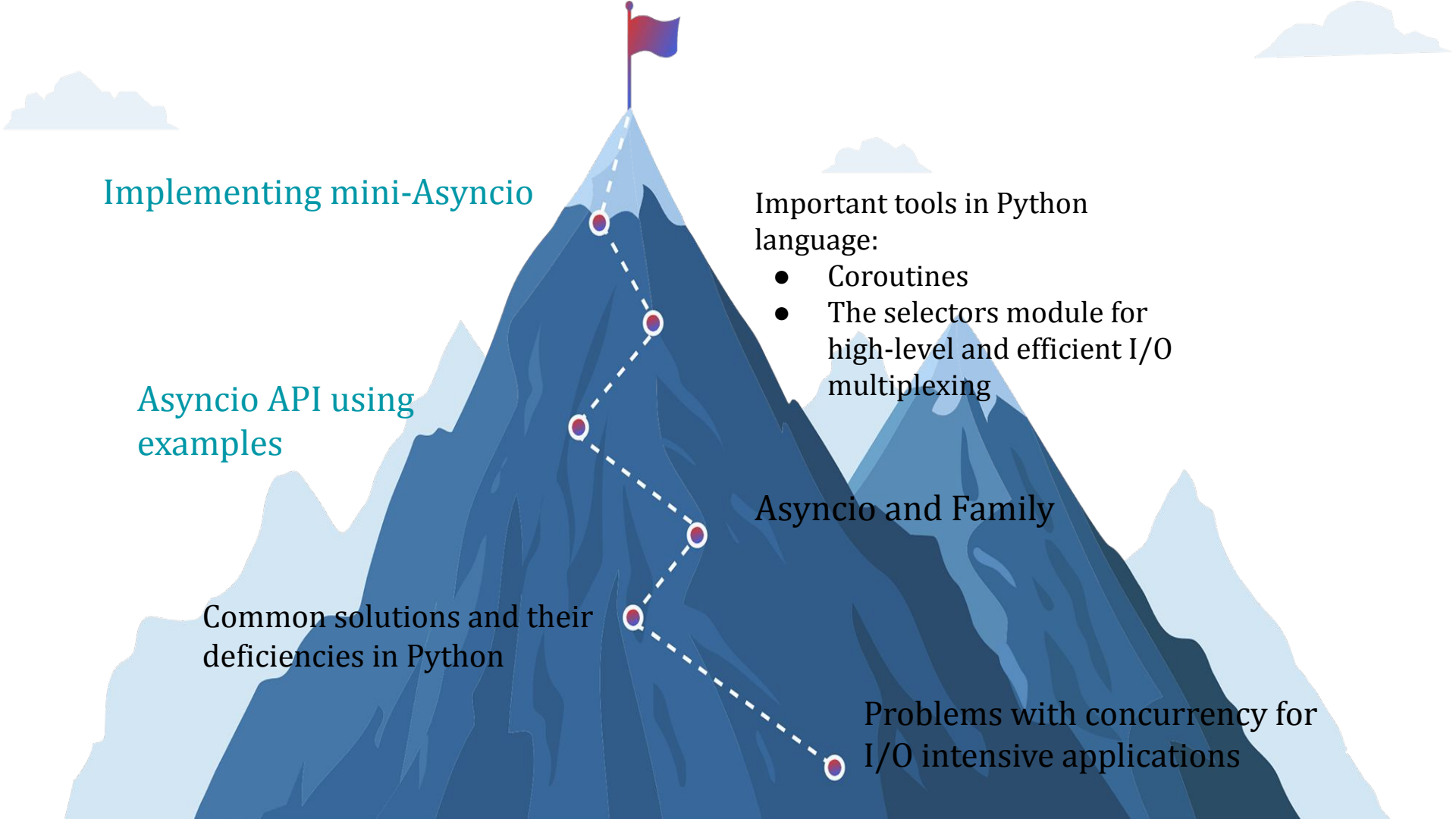
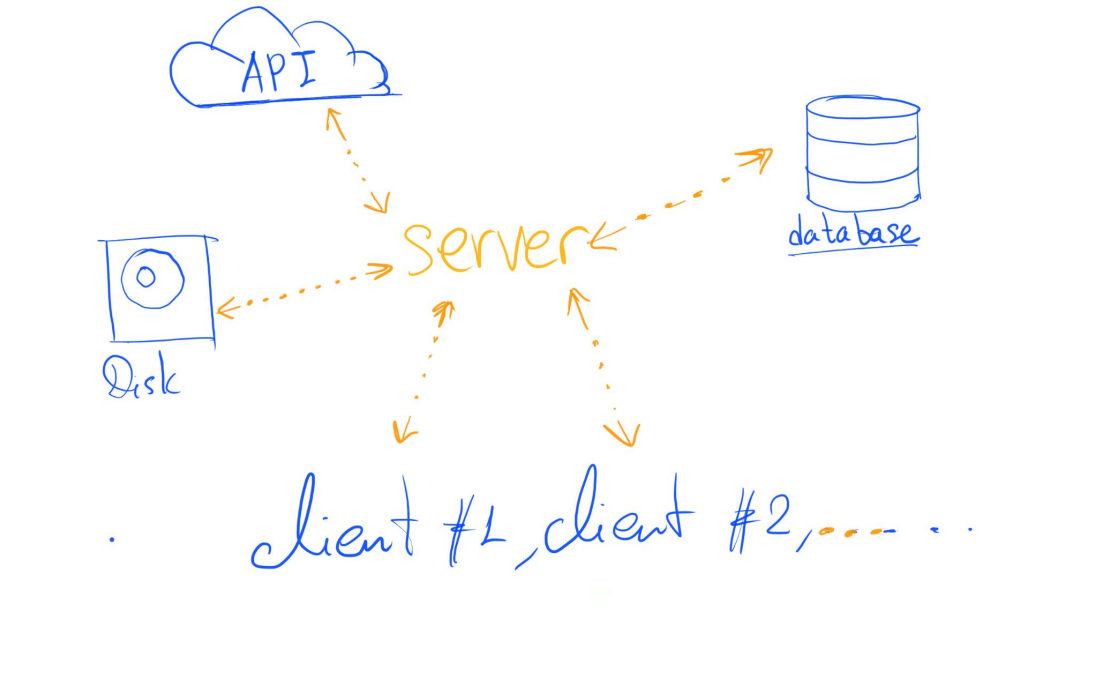
Asynchronous Programming in Python and Asyncio library


Implementing mini-Asyncio
Important tools in Python language:
Coroutines
The selectors module for high-level and efficient I/O multiplexing
Asyncio API using examples
Asyncio and Family
Common solutions and their deficiencies in Python
Problems with concurrency for I/O intensive applications

What is asynchronous programming and why we need it?



the central focus of Asyncio is on how best to best perform multiple tasks at the same time—and not just any tasks, but specifically tasks that involve waiting periods. The key insight required with this style of programming is that while you wait for this task to complete, work on other tasks can be performed.

What is asynchronous programming and why we need it?

Other languages solved those problems differently (mainly two directions)

Coroutine based:

Go - goroutines are lightweight threads managed by the Go runtime. They interact with each other using channels.

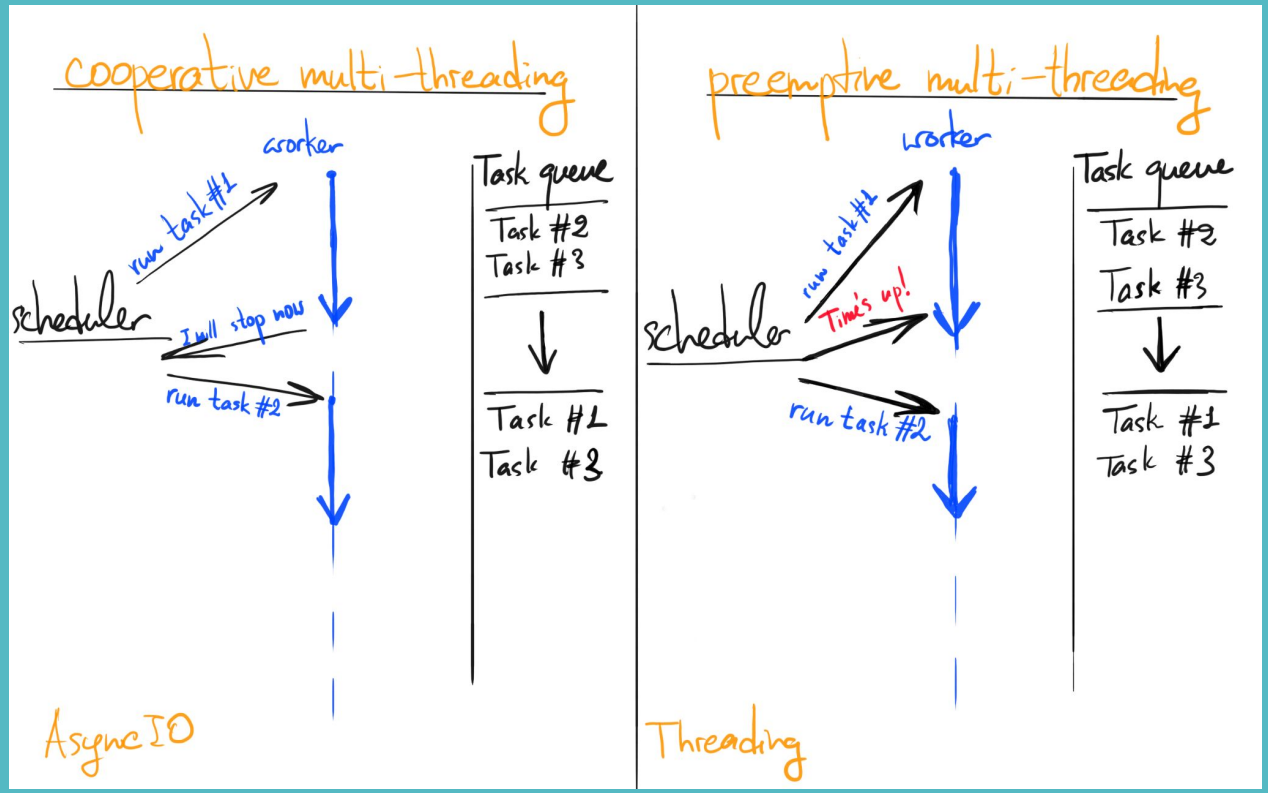
Java - With its latest versions, Java has improved its concurrency model, offering lightweight threads through Project Loom, which aims to make concurrent programming more straightforward

Future based:

Javascript - Futures and Promises. They describe an object that acts as a proxy for a result that is initially unknown, usually because the computation of its value is not yet complete.

Rust - Tokio library, built on Promises, Similar API to Asyncio

Cooperative vs Preemptive threads



Cooperative and Preemptive scheduling (advantages and disadvantages for each)

Example of Windows operating system:

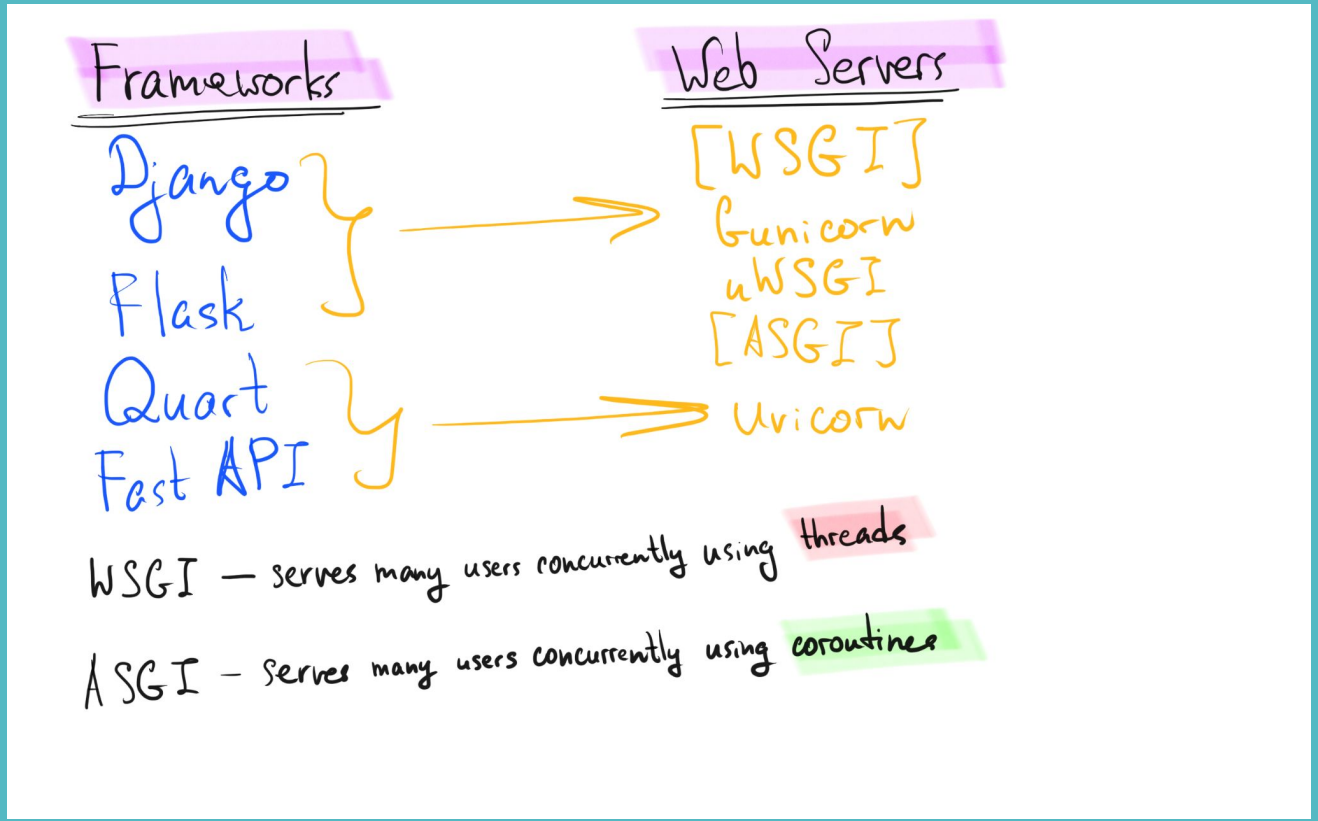
Up until Windows 3.1 :

- Cooperative scheduling, each application was responsible for yielding control back to the operating system (similarly to the relationship between coroutines and Asyncio event loop)
- Advantages:
  - Resource efficiency - required fewer system resources. (As in Asyncio)
- Disadvantages:
  - Stability and care when writing applications: If any application became unresponsive or entered an infinite loop, it could lock up the entire system (As in Asyncio, when writing coroutines care is needed)

With Windows 95 and after:

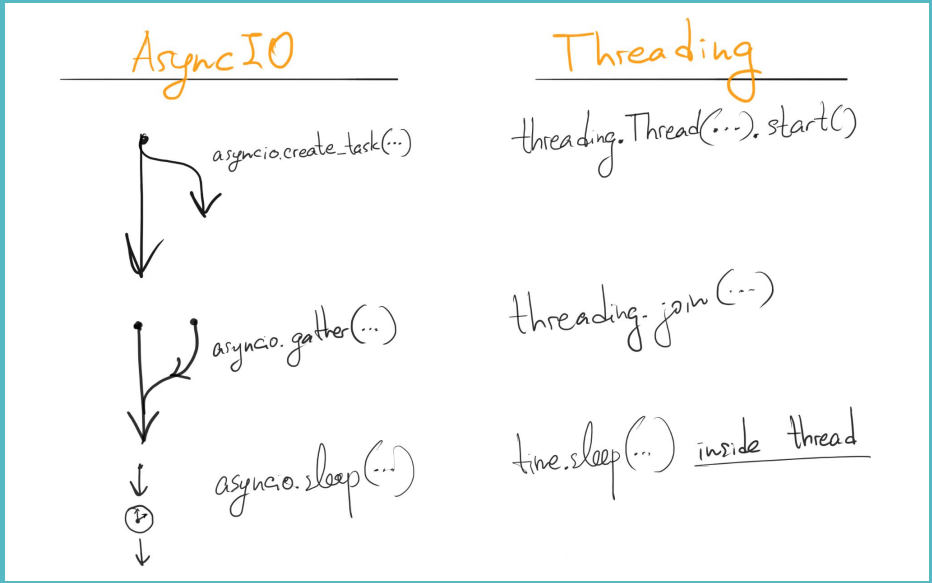
- Preemptive scheduling, the operating system decides when an application should yield control, preventing any single application from monopolizing the CPU (Similar to how Threading library works in Python)
- Advantages:
- Disadvantages:
  - Potential for overhead - more frequent context switching can lead to performance overhead, both memory and time (Similar to Threading library in Python)
  - Common bugs: Race conditions and deadlocks

Transitioning to Asynchronous Programming in Python



Transitioning from threading library to Asyncio Library

Asyncio is modelled so that its API corresponds to similar functionality and design patterns in Threading library



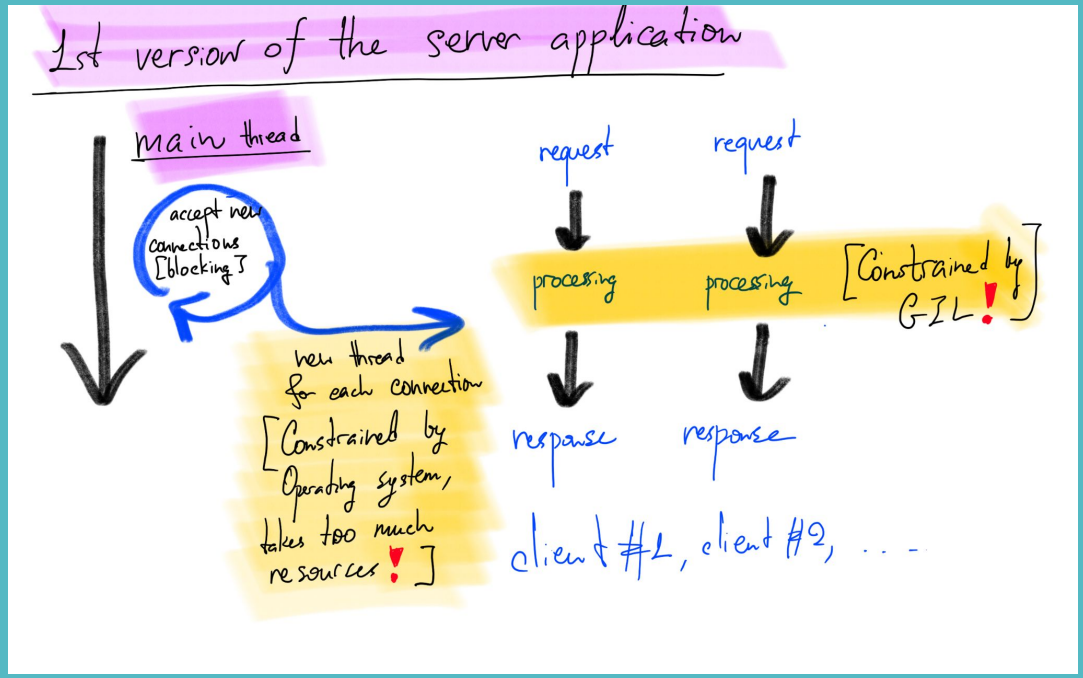
Examples of same functionality implemented in Threading/Asyncio

First Steps

Game plan:
Take a server application that uses Threading library and transform it to the one that uses one thread + our own event queue + socket library + select library

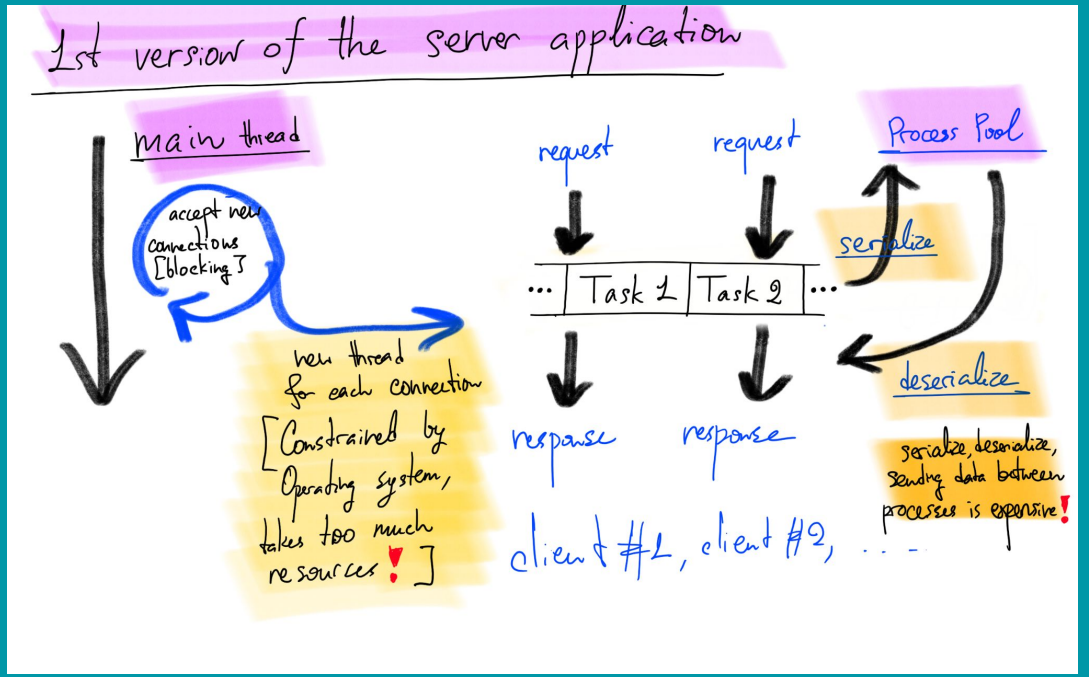
Version #1 of our API server (multithreading)

- The TCP server responds to API requests of the type {n} by sending back {n-th} fibonacci number.
  - This is useful to easily simulate both very simple and complex requests.
- The first version creates a new thread for each of the new client that is connected to the server:
  - Can't serve many users at the same time. The OS bounds the number of threads.
  - We have a problem with GIL [example to show]
  - For more complex example (like chat-server) the synchronization between threads would be complex.



Version #2 of our API server (multiprocessing)

- Deficiencies:
  - The number of clients the server can support in parallel is still limited by OS threshold on number of threads.
  - Transferring the data to other processes and then receiving the response back is time consuming (as processes don't have the same memory space)

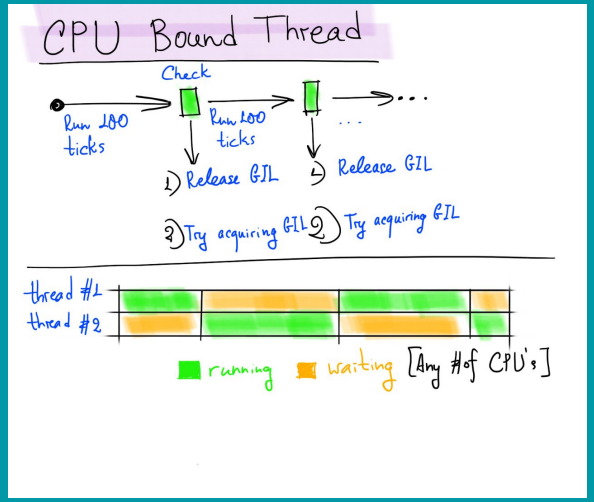


GIL (Global interpreter lock)

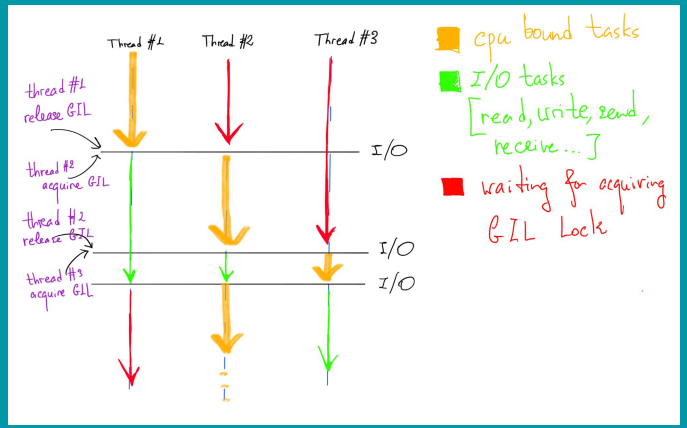
- GIL prevents one Python process from executing more than one Python bytecode instruction at any given time. [Pic. 1] [Examples: synchronous and multithreading calculations of Fibonacci numbers]
- even if we have multiple threads on a machine with multiple cores, a Python process can have only one thread running Python code at a time. [Pic. 2]
- The global interpreter lock is released when I/O operations happen. This lets us employ threads to do concurrent work when it comes to I/O, but not for CPU-bound Python code itself [Pic. 3] [Examples: synchronous and multithreading fetching of data from the Server]

```
>>> def fact(n):
        if n == 1:
            return 0
        else:
            return n * fact(n - 1)

>>> import dis
>>> dis.dis(fact)

  2           0 LOAD_FAST                0 (n)
              2 LOAD_CONST               1 (1)
              4 COMPARE_OP               2 (==)
              6 POP_JUMP_IF_FALSE        6 (to 12)

  3           8 LOAD_CONST               2 (0)
             10 RETURN_VALUE

  5     >>   12 LOAD_FAST                0 (n)
             14 LOAD_GLOBAL              0 (fact)
             16 LOAD_FAST                0 (n)
             18 LOAD_CONST               1 (1)
             20 BINARY_SUBTRACT
             22 CALL_FUNCTION            1
             24 BINARY_MULTIPLY
             26 RETURN_VALUE
```

Pic. 1



Pic. 2



Pic. 3

Version #3 of our API server (coroutines and Select library in Python) Coding Example

Ingredients

For custom event loop

- Modern coroutines in Python (From PEP 342, adding .send(), .throw(), .close() operations):

> In other words, with a few relatively minor enhancements to the language and to the implementation of the generator-iterator type, Python will be able to support performing asynchronous operations without needing to write the entire application as a series of callbacks, and without requiring the use of resource-intensive threads for programs that need hundreds or even thousands of co-operatively multitasking pseudothreads. Thus, these enhancements will give standard Python many of the benefits of the Stackless Python fork, without requiring any significant modification to the CPython core or its APIs. In addition, these enhancements should be readily implementable by any Python implementation (such as Jython) that already supports generators.

  (PEP 380 added yield from, and PEP492 async and await syntax)

- Threads and Multiprocessing
- Socket library in Python
- Select library in Python

GIL (Global interpreter lock)

Necessary for various reasons (Without GIL):

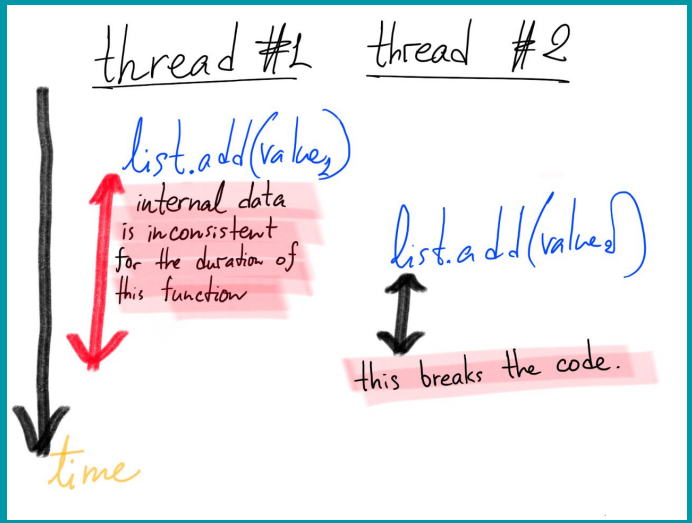
1. Implementation of Garbage Collector (which in Python is a combination of Reference counting + Cycle detection) would not be thread safe otherwise [Pic. 1]
2. Implementation of Python mutable data structures (List, Set, Dict, …) would break [Pic.2]
3. Integration with C libraries would be much more difficult:
   - Numpy
   - Pandas
   - SciPy

   [Example that demonstrates that Numpy indeed releases GIL for efficiency]

```python
class RefCountedObject:
    def __init__(self):
        self.reference_count = 1

    def __del__(self):
        print(f"Object {id(self)} is being garbage collected.")

def allocate_object():
    return RefCountedObject()

def add_reference(obj):
    obj.reference_count += 1 # Problem is here
    # temp = obj.reference_count
    # obj.reference_count = temp + 1

def release_reference(obj):
    obj.reference_count -= 1 # Problem is here
    if obj.reference_count == 0:
        del obj
```

```python
class RefCountedObject:
    def __init__(self):
        self.reference_count = 1

    def __del__(self):
        print(f"Object {id(self)} is being garbage collected.")

def allocate_object():
    return RefCountedObject()

def add_reference(obj):
    ACQUIRE_LOCK()
    obj.reference_count += 1
    RELEISE_LOCK()

def release_reference(obj):
    ACQUIRE_LOCK()
    obj.reference_count -= 1
    if obj.reference_count == 0:
        del obj
    RELEISE_LOCK()
```

Pic. 1



Pic. 2

yield + yield from *vs* async def + await

```
@coroutine
def fibonacci(a, b):
  while True:
    yield a
    a, b = b, a + b

@coroutine
def fibonacci_delayed(a, b, duration):
  asyncio.sleep(duration)
  yield from fibonacci(a, b)
```

yield + yield from

```
class fibonacci:
  def __init__(a, b):
    self.a = a
    self.b = b
  def __await__(self):
    while True:
      yield a
      a, b = b, a + b

async def fibonacci_delayed(a, b, duration):
  asyncio.sleep(duration)
  await fibonacci(a, b)
```

async def + await

Typical Example of Asyncio App (**Single-threaded** *Real-time Chat Application*)

Description:

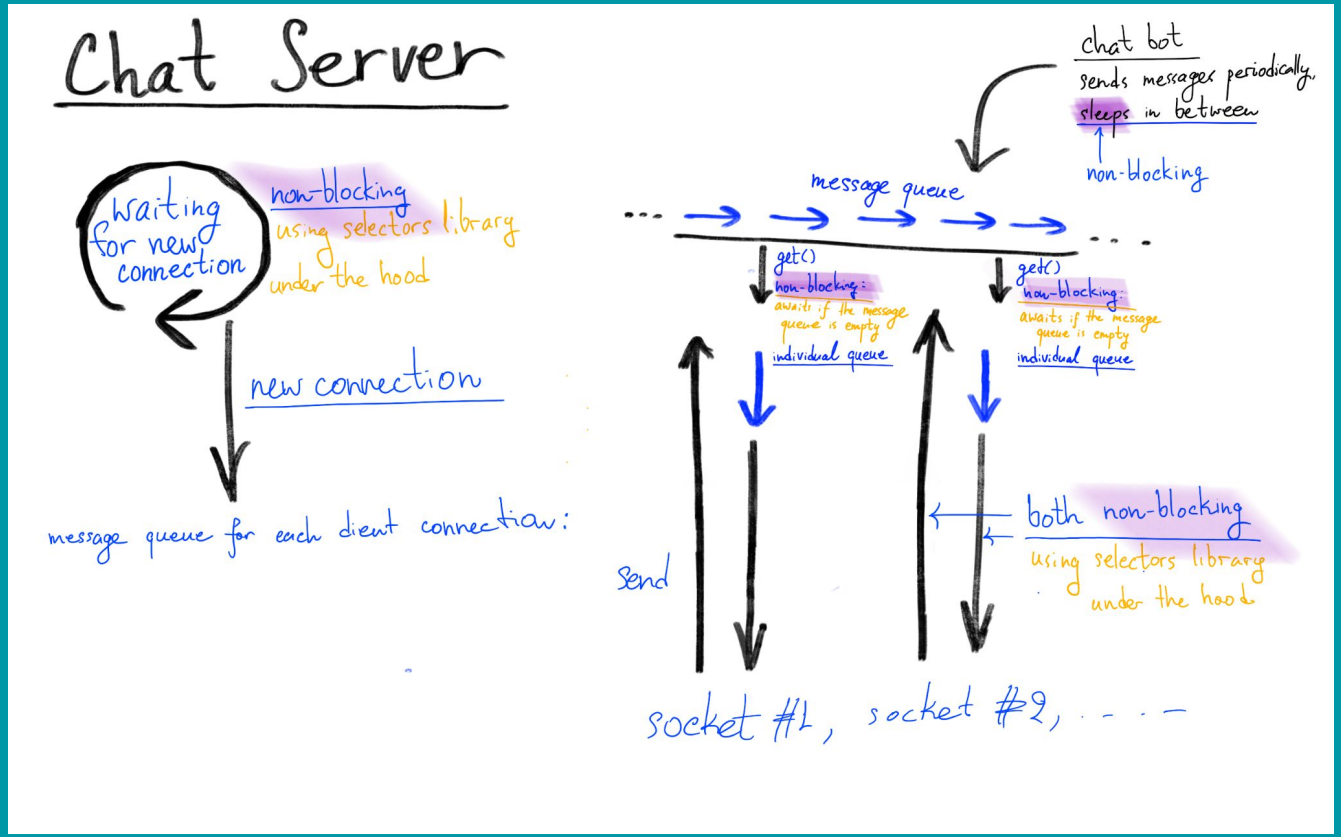
- Users can sign into an application. After choosing the username, they enter into a common server where they can send/receive messages:
- Periodically, a ***chat bot*** broadcasts a message (in our case, inspirational quotes)
- Users get notified when a new user enters/leaves

[Live Demo]

API needed (ones that we will implement, one of the most common functions in Asyncio Python library):

- Asynchronous message **queue:**
  - sending / receiving messages from different clients have different latency. We have to use an asynchronous queue so that the app stays real time for all the users.
  - By asynchronous we mean: if a task wants to take an element from an empty queue it sleeps until some other tasks puts an element in the same queue.
- A task might **spawn** a children tasks and wait for them to **join**:
  - When a new client connects, we create two tasks: for reader and writing to the stream respectively.
- Some tasks might **sleep** for some time before an event loop wakes it up:
  - Chat bot example.
- Tasks might sleep and wait for the **socket to be readable/writable**.

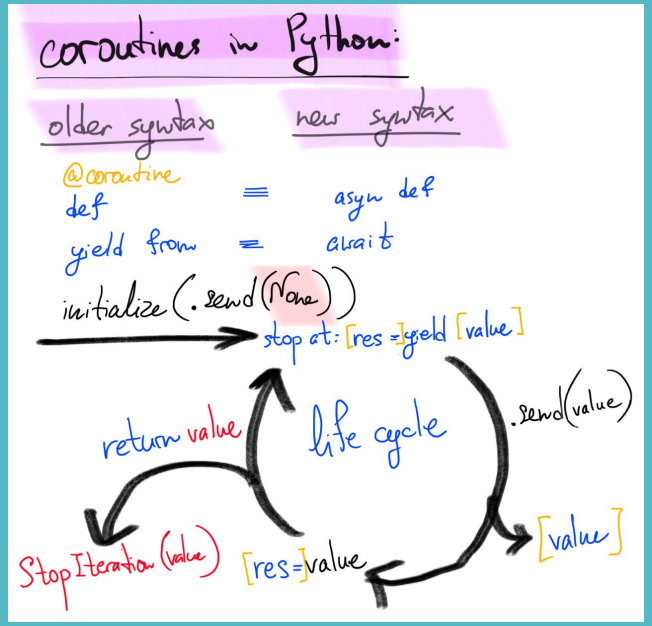
Single-threaded *Real-time Chat Application*

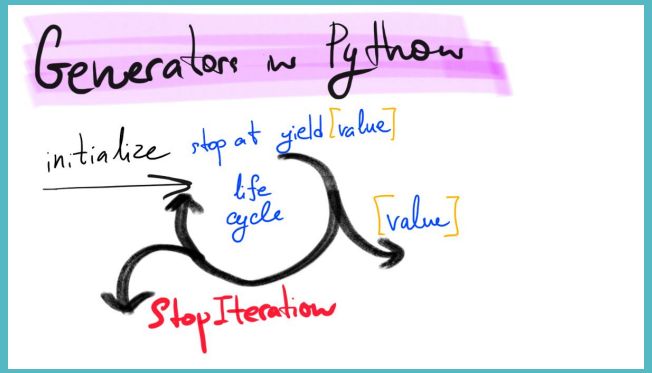


generators *vs* modern coroutines

- Coroutines are a natural way of expressing many algorithms, such as simulations, games, asynchronous I/O, and other forms of event-driven programming or co-operative multitasking.
- Python's generator functions are almost coroutines – but not quite – in that they allow pausing execution to produce a value, but do not provide for values or exceptions to be passed in when execution resumes. They also do not allow execution to be paused within the try portion of try/finally blocks, and therefore make it difficult for an aborted coroutine to clean up after itself.

  [Examples for both]



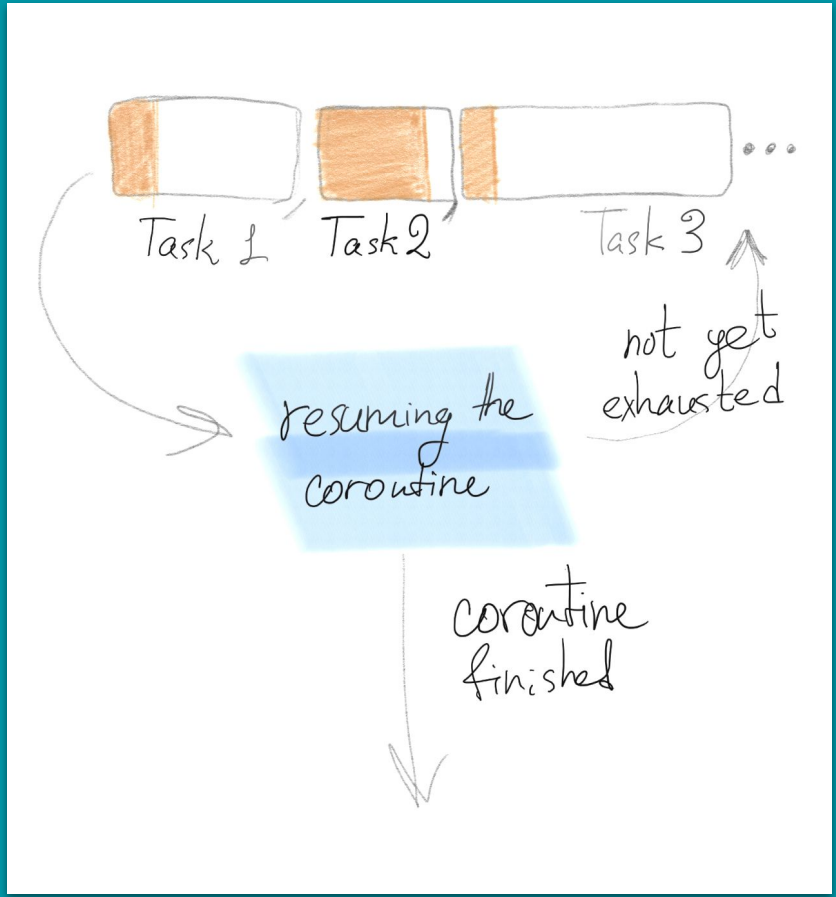


Algorithm #1

The first version supports checking the state of coroutines sequentially (pseudo code follows):

- For each iteration:
  - Consider each active coroutine.
  - Resume it.
  - If it's finished, discard it. Otherwise keep if for the next iteration.

[Example for this version of the event Loop]



Implementation (Event Loop):

```
class EventLoop:
  def __init__(self):
    self.queue = []
    self.active_coroutines = []
  def append_tasks(tasks_list):
    self.active_coroutines.extend(task_list)
  def run():
    while self.active_coroutines:
      current_coroutine = sefl.active_coroutines.pop(0)
      try:
        current_coroutine.send(None)
        self.active_coroutines.append(current_coroutine)
      except StopIteration:
        pass
```

```
class EventLoop:
  def __init__(self):
    self.queue = []
    self.active_coroutines = []
  def append_tasks(tasks_list):
    self.active_coroutines.extend(task_list)
  def run():
    while self.active_coroutines:
      current_iteration, next_iteration = self.active_coroutines, []
      for current_coroutine in current_iteration:
        try:
          current_coroutine.send(None)
          next_iteration.append(current_iteration)
        except:
          pass
      self.active_coroutines = next_iteration
```

Implementation (Tasks):

Without event_loop API

```
@coroutine
def sort_words_by_frequency(file_path):
    word_count = {}
    # Read file and count word frequency
    with open(file_path, 'r') as file:

        line_count = 0
        lines_at_a_time = 10000

        for line in file:
            words = line.split()
            for word in words:
                word = word.lower()  # Optional: makes it case-insensitive
                word_count[word] = word_count.get(word, 0) + 1

            line_count = line_count + 1
            if line_count % lines_at_a_time == 0:
              line_count = 0

              # yield to the **event_loop**
              yield

    # Sort words by frequency in decreasing order
    sorted_words = sorted(word_count.items(), key=lambda x: x[1], reverse=True)

    first_n = 200
    print(file_path)
    for id in range(min(first_n, len(sorted_words))):
      print("{} - {} times".format(sorted_words[id][0], sorted_words[id][1]))
```

With event_loop API

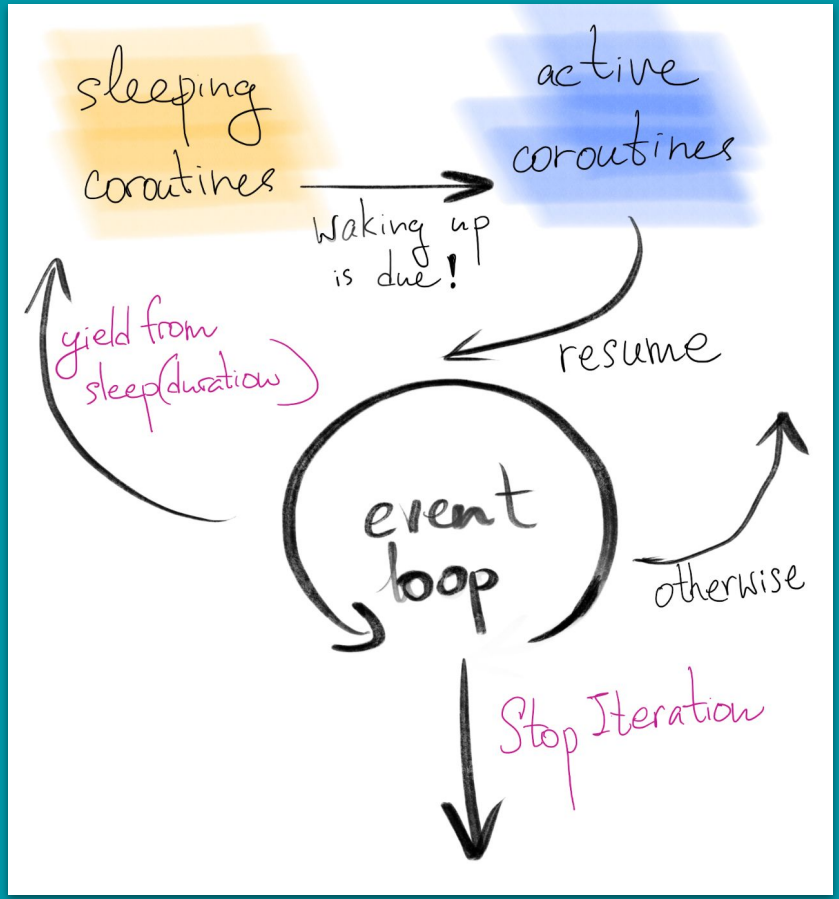
```
@coroutine
def yield_to_event_loop():
  yield

@coroutine
def sort_words_by_frequency(file_path):
    word_count = {}
    # Read file and count word frequency
    with open(file_path, 'r') as file:

        line_count = 0
        lines_at_a_time = 10000

        for line in file:
            words = line.split()
            for word in words:
                word = word.lower()  # Optional: makes it case-insensitive
                word_count[word] = word_count.get(word, 0) + 1

            line_count = line_count + 1
            if line_count % lines_at_a_time == 0:
              line_count = 0

              # yield to the **event_loop**
              yield from yield_to_event_loop()

    # Sort words by frequency in decreasing order
    sorted_words = sorted(word_count.items(), key=lambda x: x[1], reverse=True)

    first_n = 200
    print(file_path)
    for id in range(min(first_n, len(sorted_words))):
      print("{} - {} times".format(sorted_words[id][0], sorted_words[id][1]))
```

Algorithm #2

The second version introduces the possibility for a task to periodically resume execution (pseudo-code follows):

- For each iteration:
  - Consider each active coroutine.
    - Resume them.
    - Depending on the resume value put them into sleeping list or put them back into active coroutines for the next iteration
  - If there are no active coroutines for the next iteration:
    - Find the coroutine that should resume the soonest and sleep the thread until that time using a system call.
    - Append all tasks that are due to continue to the active coroutine list.

[Show the example for this version of the event loop]



Implementation #2 (Event Loop):

```
@coroutine
def take_nap(duration):
  yield ("take nap", duration)

class Scheduler:
  def __init__(self):
    self.task_queue = deque()
    self.sleeping = []

    # implementation detail, needed for the case when more then one coroutines are scheduled for exactly
the same time
    self.priority_index = 1

  def create_task(self, coroutine):
    self.task_queue.append(coroutine)

  def run(self):
    while any([self.task_queue, self.task_queue]):
      current_tasks, new_tasks = self.task_queue, []
      for coroutine in current_tasks:
        try:
          yield_data = coroutine.send(None)
          if yield_data and yield_data[0] == "take nap":
            duration = yield_data[1]
            self.sleeping.append((time.time() + duration, self.priority_index, coroutine))
            self.sleeping.sort()
            self.priority_index = self.priority_index + 1
          else:
            new_tasks.append(coroutine)
        except StopIteration:
          pass
      if not new_tasks and self.sleeping:
        delta_time = max(self.sleeping[0][0] - time.time(), 0)
        time.sleep(delta_time)
        while self.sleeping and self.sleeping[0][0] < time.time():
          new_tasks.append(self.sleeping[0][2])
          self.sleeping = self.sleeping[1:]
      self.task_queue = new_tasks
```

Simple example

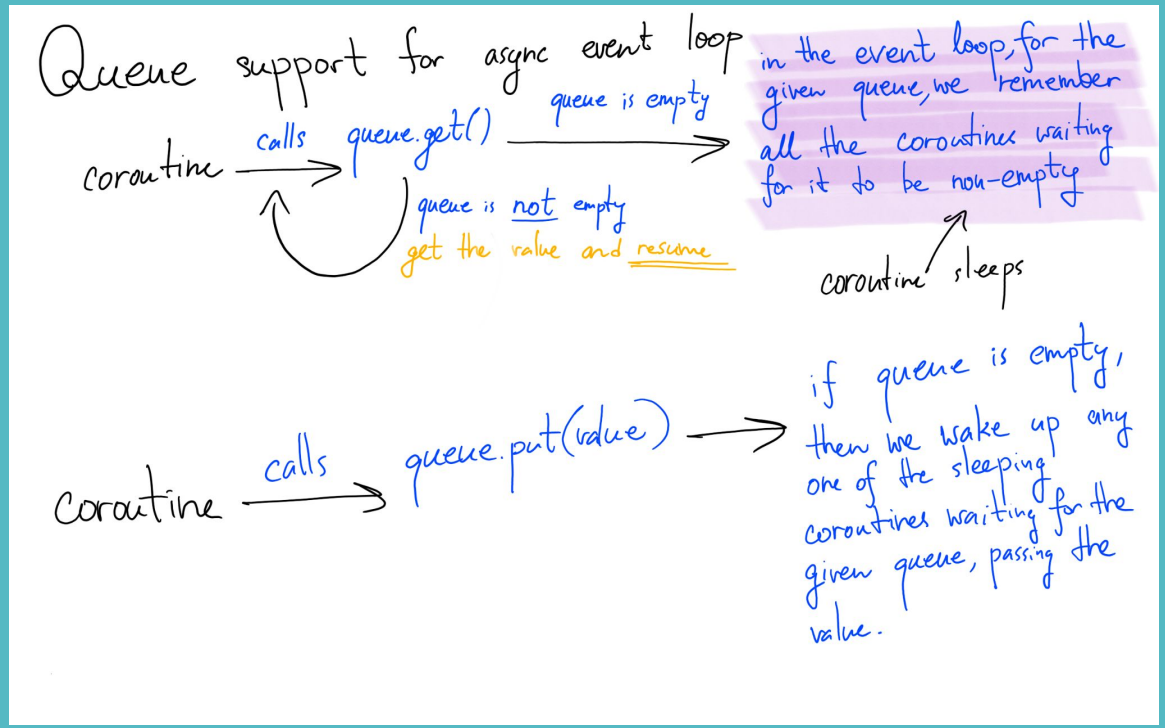
```
def countdown(n):
  while n > 0:
    print("Count down", n)
    yield from take_nap(2) # sleep for 2 minutes
    n -= 1

def countup(n):
  x = 1
  while x <= n:
    print("Count up", x)
    yield from take_nap(4) # sleep for 4 minutes
    x += 1

scheduler = Scheduler()

scheduler.create_task(countdown(20))
scheduler.create_task(countup(10))

scheduler.run()
```

Algorithm #3

The Third version supports asynchronous Queue for Our event loop

Algorithm:

- For consumers:
  - If the queue is empty then sleep and wait
  - Otherwise pop the value
- For producers:
  - Push the value
  - If the queue was empty, then wake up any of the consumers waiting for the queue



Implementation #4 (Event Loop):

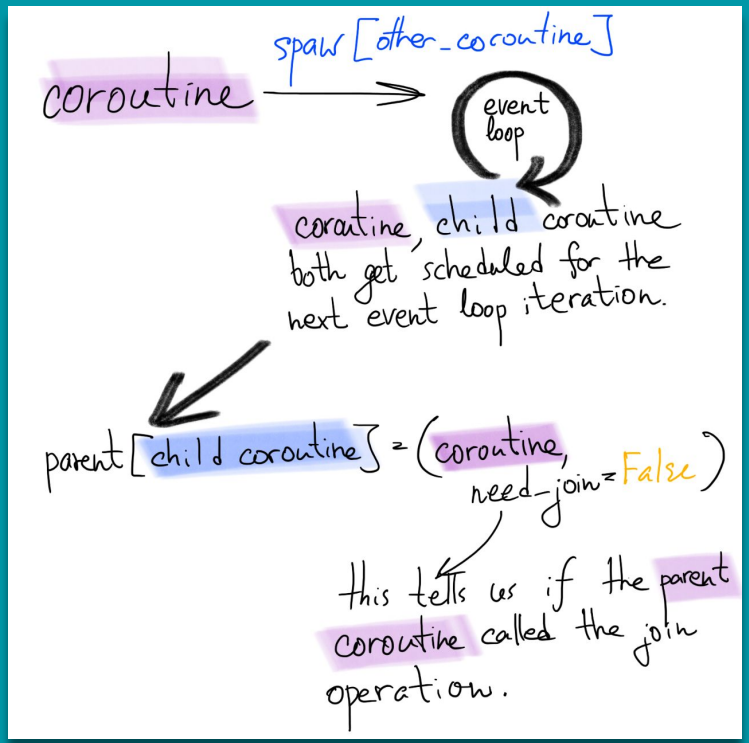
```
class AsyncEventLoop:
    def __init__(self):
        self.task_queue = []
        self.waiting_queue_dict = {}

    def append_task(self, coroutine):
        self.task_queue.append((coroutine, None))

    def run(self):
        while self.task_queue:
            coroutine, what_to_send = self.task_queue.pop(0)
            try:
                data = coroutine.send(what_to_send)
                if data:
                    if data[0] == "waiting on the given queue":
                        # append the coroutine to the waiting list for the given queue
                        if data[1] in self.waiting_queue_dict:
                            self.waiting_queue_dict[data[1]].append(coroutine)
                        else:
                            self.waiting_queue_dict[data[1]] = [coroutine]

                    elif data[0] == "wake up anyone for the given queue":
                        self.task_queue.append((coroutine, None))

                        # is a coroutine found for the given queue, wake it up
                        sleeping_coroutine = "nothing found"
                        if self.waiting_queue_dict[data[1]]:
                            sleeping_coroutine = self.waiting_queue_dict[data[1]].pop(0)

                        if sleeping_coroutine != "nothing found":
                            self.task_queue.append((sleeping_coroutine, None))
            except StopIteration:
                pass
```

```
class Queue:
    def __init__(self):
        self.queue = []

    @coroutine
    def get(self):
        while True:
            if self.queue:
                return self.queue.pop(0)
            else:
                # ask the event loop to notify this coroutine if the given loop is non-empty as some point
                yield ("waiting on the given queue", self)

    @coroutine
    def put(self, value):
        self.queue.append(value)
        # ask the event loop to notify anyone coroutine that is waiting for the given queue
        yield ("wake up anyone for the given queue", self)
```

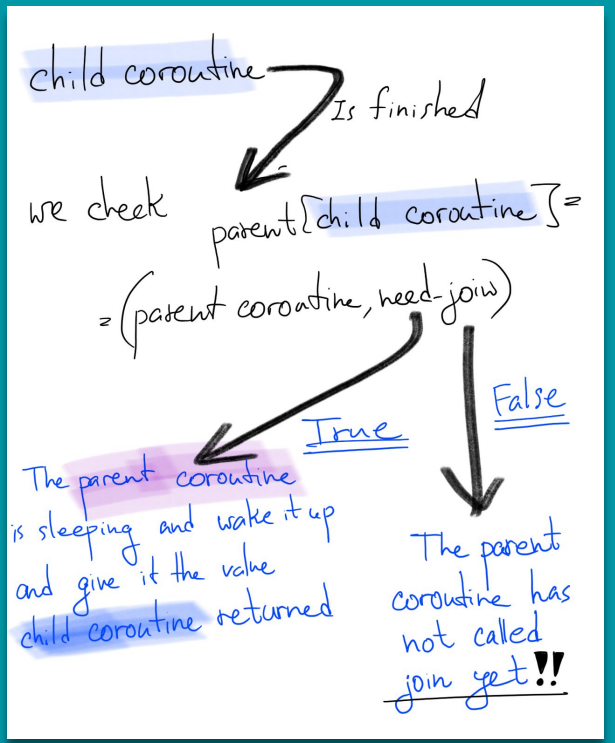
Algorithm #4

The fourth version introduces the possibility for a task to spawn new tasks and then wait for them to finish.

Spawning a task



Joining a task



Implementation #3 (Event Loop):

Spawning a task

```python
@coroutine
def spawn(task):
    # Ask the event loop to put task to active coroutine's list
    child_handler = yield ("spawn", task)
    # we need this handler for the join operation later on
    return child_handler
```

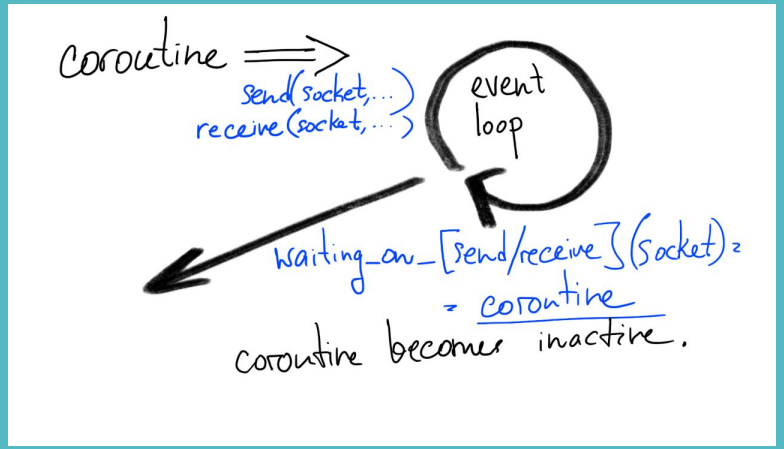
joining a task

```python
@coroutine
def join(task):
    # the event handler is responsible to attach value
    # to the send operation
    calculation_result = yield ("join", task)
    return calculation_result
```

```python
def run_until_complete(task):
  # list of active coroutines, along with values that we pass for the next .send()
  active_tasks = [(task, None)]

  # each coroutines has a parent, along with joined = [True, False] depending
  # whether the parent called join or not
  parent_dict = defaultdict(list)

  # for coroutines that finished but their parents have not calle join yet, also saves
  # the value child coroutine returned
  parent_dict_with_children_finished = defaultdict(list)

  while active_tasks:
    current_queue, active_tasks_next_iteration = active_tasks, []
    for coroutine, value_to_pass in current_queue:
      try:
        received_data = coroutine.send(value_to_pass)
      except StopIteration as result_wrapper:
        # the coroutine finished
        parent_list = parent_dict[coroutine]
        if parent_list:  # if is has a parent
          parent_joined = parent_list[0][1]
          if parent_joined:
            # if the parent already wait for the child to join, we wake up the parent
            # passing the value child coroutine returned
            active_tasks_next_iteration.append((parent_list[0][0], result_wrapper.value))
          else:
            # otherwise, we remember the return value for when the parent calls join
            parent_dict_with_children_finished[coroutine].append(result_wrapper.value)
          parent_dict.pop(coroutine)
      else:
        if received_data and received_data[0] == "spawn":
          # child coroutine is active for the next iteration
          active_tasks_next_iteration.append((received_data[1], None))
          # parent coroutine need a child_handler to call join afterwards
          active_tasks_next_iteration.append((coroutine, received_data[1]))
          # child coroutine now has a parent
          parent_dict[received_data[1]].append((coroutine, False))
        elif received_data and received_data[0] == "join":
          children_not_finished = list(parent_dict.keys())
          children_finished = list(parent_dict_with_children_finished.keys())
          child_handler = received_data[1]
          if child_handler in children_not_finished:
            parent_dict[child_handler][0] = (parent_dict[child_handler][0][0], True)
          elif child_handler in children_finished:
            active_tasks_next_iteration.append(
              (coroutine, parent_dict_with_children_finished.pop(child_handler)[0])
            )
          else:
            active_tasks_next_iteration.append((coroutine, None))
    active_tasks = active_tasks_next_iteration
```

Algorithm #5

The fifth version supports non-blocking socket I/O



polling on sockets:

next iteration of the event loop

- active coroutine list is non-empty: Non-blocking Polling
- otherwise: sleeping coroutines:
  - empty ⇒ Poll(duration = INF)
  - ≠ [(coro$_1$, t_1), ..., (coro$_n$, t_n)] with $t_1 \leq \ldots \leq t_n$

    ⇓

    Poll(t_1 - current_time)

Implementation #5 (Event Loop):

- We check if at least one of the sockets are either readable or writable using select function. For such events we notify the corresponding coroutines.
- Select function can be blocking, so we choose the maximal blocking timeout that will not interfere with other coroutines:
  - If some are active:
    - Timeout = 0
  - If non are active, some are sleeping:
    - Timeout = minimal time before at least one sleeping coroutine is scheduled to wake up

```
class Scheduler:
  def __init__(self):
    self.ready = deque()
    self.sleeping = []
    self.sequence = 0
    self._read_waiting = {}
    self._write_waiting = {}

  def call_soon(self, func):
    self.ready.append(func)

  def call_later(self, delay, func):
    self.sequence += 1
    deadline = time.time() + delay
    heapq.heappush(self.sleeping, (deadline, self.sequence, func))

  def read_wait(self, fileno, func):
    self._read_waiting[fileno] = func

  def write_wait(self, fileno, func):
    self._write_waiting[fileno] = func

  def run(self):
    while (self.ready or self.sleeping or self._read_waiting or self._write_waiting):
      if not self.ready:
        if self.sleeping:
          deadline, _, func = self.sleeping[0]
          timeout = deadline - time.time()
          if timeout < 0:
            timeout = 0
        else:
          timeout = None

        can_read, can_write, _ = select(self._read_waiting, self._write_waiting, [], timeout)
        for fd in can_read:
          self.ready.append(self._read_waiting.pop(fd))
        for fd in can_write:
          self.ready.append(self._write_waiting.pop(fd))

        now = time.time()
        while self.sleeping:
          if now > self.sleeping[0][0]:
            self.ready.append(heapq.heappop(self.sleeping))
          else:
            break
      while self.ready:
        func = self.ready.popleft()
        print(func)
        func()
```

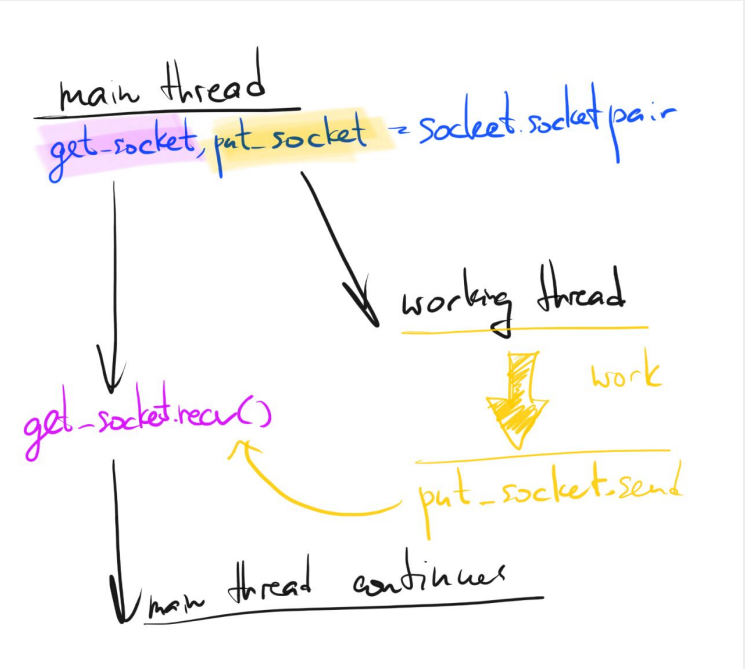
How to incorporate threads in the event loop that generally uses only coroutines

Some CPU intensive functions can not be refactored to become coroutines and periodically yield to the event loop

In that case it becomes necessary to still have threads but they be transparent for the library:

Solution: We use socket.socketpair() function to connect two threads to each other (The standard solution to periodically check whether the working thread finished would be wasteful)

[Examples]



Implementation (worker threads):

```
@coroutine
def run_in_thread(function, *args_list):
  get_socket, put_socket = socket.socketpair()

  def thread_worker():
    answer = function(*args_list)
    put_socket.send(str(answer).encode())
    put_socket.close()

  threading.Thread(target = thread_worker).start()
  answer = yield("wait socket to read", get_socket)
  get_socket.close()
  return answer
```

The event loop creates a socket pair to create a two directional channel between threads. The calling coroutine sleeps until the channel is readable, which means that the worker thread finished.

The end

Thanks!