SuperFastPython.com Cheat Sheet for Python asyncio

Why asyncio?

Asyncio provides coroutine-based concurrency suited to non-blocking socket I/O applications.

Coroutine

Import
```
import asyncio
```

Define a coroutine
```
async def custom_coroutine():
    # ...
```

Create coroutine object
```
coro = custom_coroutine()
```

Run coroutine as entry point
```
asyncio.run(main())
```

Suspend and run coroutine from a coroutine
```
await custom_coroutine()
```

Sleep a coroutine
```
await asyncio.sleep(1)
```

Async Comprehensions and Loops

Asynchronous comprehension
```
res = [r async for r in async_gen()]
```

Await comprehension
```
res = [r await a in awaitables]
```

Asynchronous for-loop
```
async for item in async_gen():
    print(item)
```

Task

A task schedules a coroutine to run independently.

Create and schedule task (high-level)
```
task = asyncio.create_task(coro)
```

Create and schedule task (low-level)
```
task = asyncio.ensure_future(coro)
```

Suspend and wait for a task to finish
```
await task
```

Get the current task
```
task = asyncio.current_task()
```

Get all running tasks
```
tasks = asyncio.all_tasks()
```

Get task result
```
value = task.result()
```

Get task unhandled exception
```
ex = task.exception()
```

Cancel a task result
```
was_canceled = task.cancel()
```

Check if the task is done (not running)
```
if task.done():
    # ...
```

Check if the task was canceled
```
if task.cancelled():
    # ...
```

Add done callback function
```
task.add_done_callback(handler)
```

Remove done callback function
```
task.remove_done_callback(handler)
```

Set and get task name
```
task.set_name('MyTask')
name = task.get_name()
```

Tasks

Operations on an awaitable, task, or tasks.

Wait for awaitable with a timeout
```
try:
    await asyncio.wait_for(tk,
timeout=1)
except asyncio.TimeoutError:
    # ...
```

Shield a task from cancelation
```
shielded = asyncio.shield(task)
```

Run blocking function in new thread
```
coro = asyncio.to_thread(myfunc)
```

Run coroutine in asyncio event loop
```
fut = run_coroutine_threadsafe(coro,
loop)
```

Run many awaitables as a group
```
await asyncio.gather(c1(), c2())
```

Wait for all tasks in a collection
```
done,pen = await asyncio.wait(tasks)
```

Wait for all tasks with a timeout in seconds
```
Try:
    done,pen = await
asyncio.wait(tasks, timeout=5)
except asyncio.TimeoutError:
    # ...
```

Wait for the first task in a collection
```
done,pen = await asyncio.wait(tasks,
return_when=FIRST_COMPLETED)
```

Wait for the first task to fail
```
done,pen = await asyncio.wait(tasks,
return_when=FIRST_EXCEPTION)
```

Get results in task completion order
```
for c in
asyncio.as_completed(tasks):
    result = await c
```

Non-blocking IO Subprocesses

Run command as subprocess
```
p = await
create_subprocess_exec('ls')
```

Run shell command as subprocess
```
p = await
create_subprocess_shell('ls')
```

Wait for subprocess to finish
```
await process.wait()
```

Read from subprocess
```
data = await process.communicate()
```

Read from subprocess
```
await
process.communicate(input=data)
```

Terminate a subprocess
```
process.terminate()
```

Non-blocking IO Streams

Open a client tcp connection
```
reader, writer = await
open_connection(google,com', 80)
```

Start a tcp server
```
server = await start_server(handle,
'127.0.0.1', 9876)
```

Read from socket
```
data = await reader.readline()
```

Write to socket
```
writer.write(data)
```

Drain socket until ready
```
await writer.drain()
```

Close socket connection
```
writer.close()
await writer.wait_closed()
```

Semaphores and Events, and Conditions

Semaphore, set num positions
```
semaphore = asyncio.Semaphore(10)
await semaphore.acquire()
# ...
semaphore.release()
```

Semaphore, context manager
```
async with semaphore:
    # ...
```

Create event, then set event
```
event = asyncio.Event()
event.set()
```

Check if event is set
```
if event.is_set():
    # ...
```

Wait for event to be set (blocking)
```
await event.wait()
```

Condition variable
```
condition = asyncio.Condition()
await condition.acquire()
# ...
condition.release()
```

Wait on condition to be notified (blocking)
```
async with condition:
    await condition.wait()
```

Wait on condition for expression (blocking)
```
async with condition:
    await condition.wait_for(check)
```

Notify any single thread waiting on condition
```
async with condition:
    condition.notify(n=1)
```

Notify all threads waiting on condition
```
async with condition:
    condition.notify_all()
```

Async Locks

Mutex lock
```
lock = asyncio.Lock()
await lock.acquire()
# ...
lock.release()
```

Mutex lock, context manager
```
async with lock:
    # ...
```

Queues

Via `Queue`, `LifoQueue`, `PriorityQueue`

Create queue
```
queue = asyncio.Queue()
```

Create queue with limited capacity
```
queue = asyncio.Queue(100)
```

Add item to queue (blocking, if limited)
```
await queue.put(item)
```

Retrieve item from queue (blocking)
```
item = await queue.get()
```

Check if queue is empty
```
if queue.empty():
    # ...
```

Check if queue is full
```
if queue.full():
    # ...
```

Get current capacity of queue
```
capacity = queue.qsize()
```

Mark unit of work complete
```
queue.task_done()
```

Wait for all units to be complete
```
await queue.join()
```

Async Generators and Iterators

Define asynchronous generator

```
async def async_generator():
    for i in range(10):
        await asyncio.sleep(1)
        yield i
```

Define asynchronous iterator

```
class AsyncIterator():
    def __init__(self):
        self.counter = 0
    def __aiter__(self):
        return self
    async def __anext__(self):
        if self.counter >= 10:
            raise StopAsyncIteration
        await asyncio.sleep(1)
        self.counter += 1
        return self.counter
```

Use asynchronous iterator

```
async for value in AsyncIterator():
    # ...
```

Async Context Managers

Define asynchronous context manager

```
class AsyncContextManager():
    async def __aenter__(self):
        await asyncio.sleep(1)
    def __aexit__(self, et, exc,
tb):
        await asyncio.sleep(1)
```

Use asynchronous context manager

```
async with CustomClass() as mgmr:
    # ...
```