Python Asynchronous Programming
A Hitchhiker's Guide

Motivation

- Python 2 → Python 3, one of the most visible language level upgrade

  (first-class citizen, still ongoing, motivation for upgrade to Python 3)

- A modern programming paradigm (Go, JavaScript, Python, C# ...)

Content

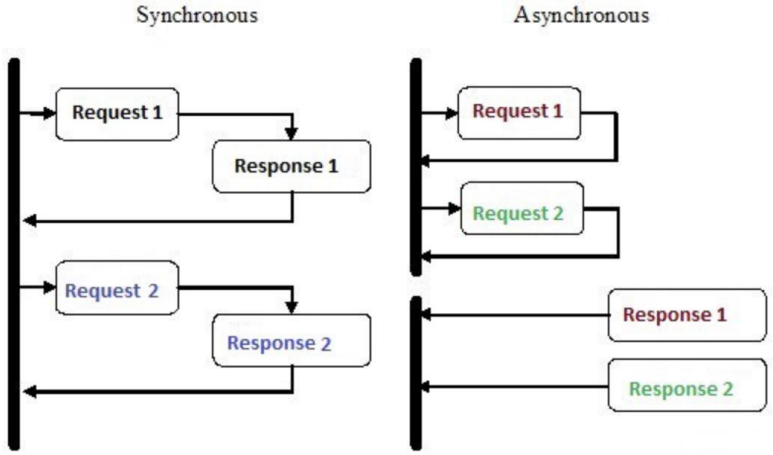
- Python Language Support
- Library Support

Content

- Python Language Support
  - Asynchronous programming
  - Coroutine
  - Python language support and implementation
  - Mixing sync and async code
- Library Support

What asynchronous programming is

a type of parallel programming (language-agnostic), a unit of work is allowed to run separately from the primary application thread, you can handle other tasks while waiting for some resources to respond (cooperative multitasking).



wait for HTTP request

Example (*by Miguel Grinberg's 2017 PyCon talk*):

Playing chess with 24 players simultaneously, 5s for you to think, 55s for other players, 30 rounds totally.

Sync Version: 60s * 30 * 24 = 12h

Async Version: 5s * 24 * 30 = 1h

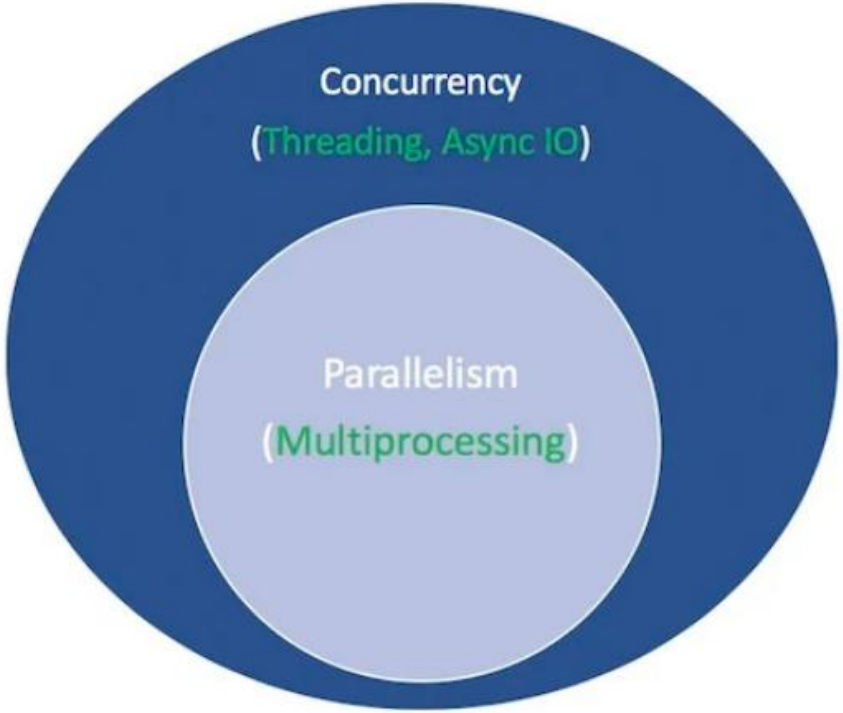
Async IO in Python: A Complete Walkthrough
An Introduction to Asynchronous Programming in Python

Concurrency VS Parallelism VS Asynchronous

	Concurrency	Parallelism	Asynchronous
Concept	multiple tasks have the ability to run in an overlapping manner *“ doesn't necessarily mean they'll ever both be running at the same instant”*	**performing multiple operations at the same time (simultaneously)** *“ tasks literally run at the same time ”*	single-thread single-process cooperative multitasking *“It’s not about using multiple cores, it’s about using a single core more efficiently”*
Implement	multiprocessing threading asyncio	multiprocessing (entails spreading tasks over a computer’s CPUs)	asyncio coroutine
Application scenarios	CPU-bound IO-bound	**CPU-bound**	IO-bound and high-level structured network code (e.g. network/database connecting process)

Async IO in Python: A Complete Walkthrough
Speed Up Your Python Program With Concurrency
What is the difference between concurrency and parallelism?

Concurrency VS Parallelism VS Asynchronous



relation and implementation

In Python:

threading -- Thread-based parallelism

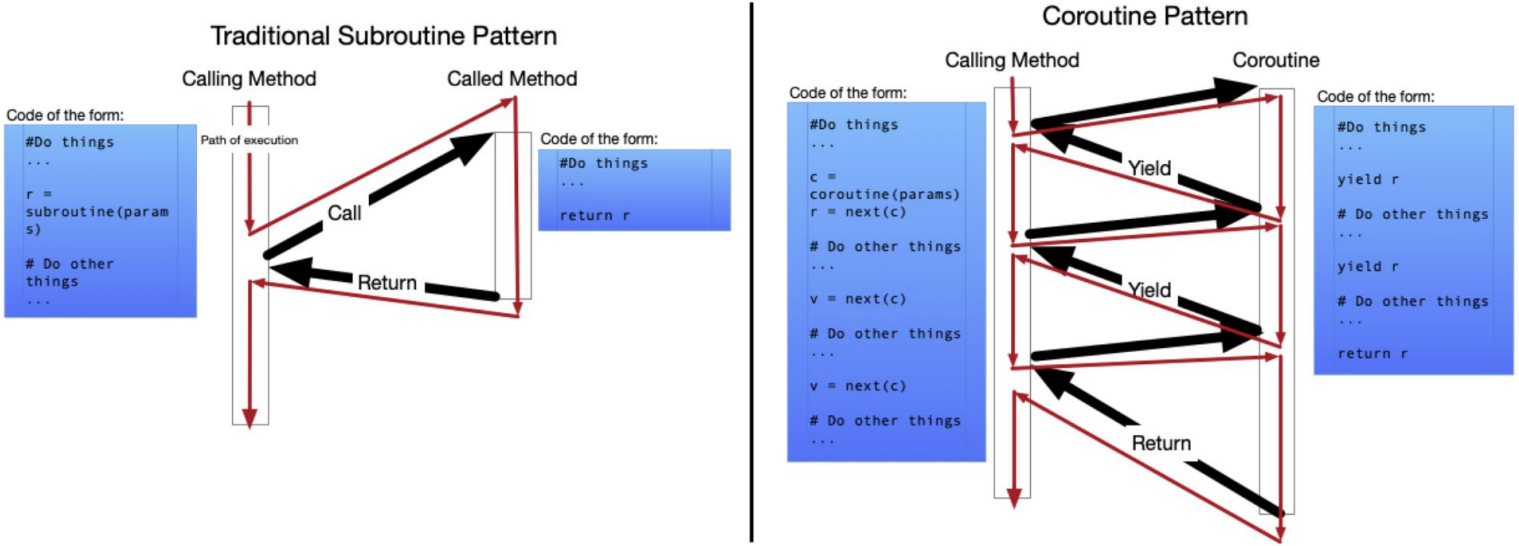
multiprocessing -- Process-based parallelism

asyncio -- asyncio is a library to write concurrent code using the async/await syntax

Async IO in Python: A Complete Walkthrough
threading — Thread-based parallelism
asyncio — Asynchronous I/O

Coroutine

a coroutine is a function that can suspend its execution before reaching *return*, and it can indirectly pass control to another coroutine for some time



subroutine, each call is independent

coroutine, continues from left most recently *'yield'* control

[Coroutine Wiki]
[Python Asyncio Part 1 – Basic Concepts and Patterns]

generator-based coroutine

- generator (Python2.2+)
  - *yield* -- indicates where a value is send back to the caller, but don't exit **afterward (the state of the function is remembered)**
  - *yield from* (Python3.3+) -- allowing a generator to delegate part of its operations to another generator

PEP-255: Simple Generators
How to Use Generators and yield in Python

generator-based coroutine(cont)

- asyncio library / with @asyncio.coroutine (Python 3.4)

  "@coroutine" decorator is deprecated since Python 3.8

```
import asyncio

@asyncio.coroutine
def py34_coro():
    """Generator-based coroutine, older syntax"""
    yield from stuff()

async def py35_coro():
    """Native coroutine, modern syntax"""
    await stuff()
```

generator-based and native coroutine

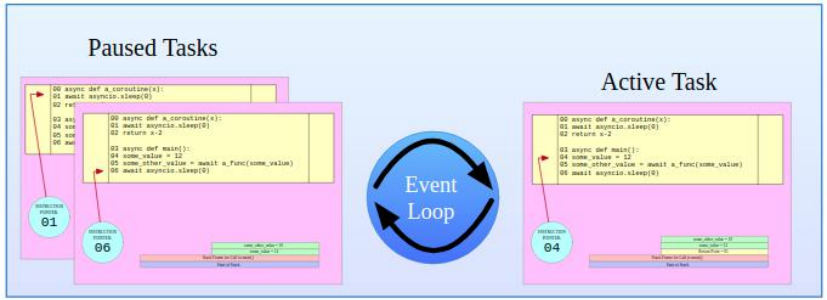
PEP 380 -- Syntax for Delegating to a Subgenerator
PEP 3156 -- Asynchronous IO Support Rebooted: the "asyncio" Module
PEP 492 -- Coroutines with async and await syntax
二、Python异步编程进化史

native coroutine basic

- asyncio is a library to write concurrent code using the async/await syntax
- *async await* keyword (Python 3.5+), *await* can only be used inside asynchronous code blocks
- a function that you introduce with *async def* is a coroutine, and you must *await* it to gets result
- coroutine object should registered to Event Loop and managed by Event Loop

二、Python异步编程进化史
Event Loop

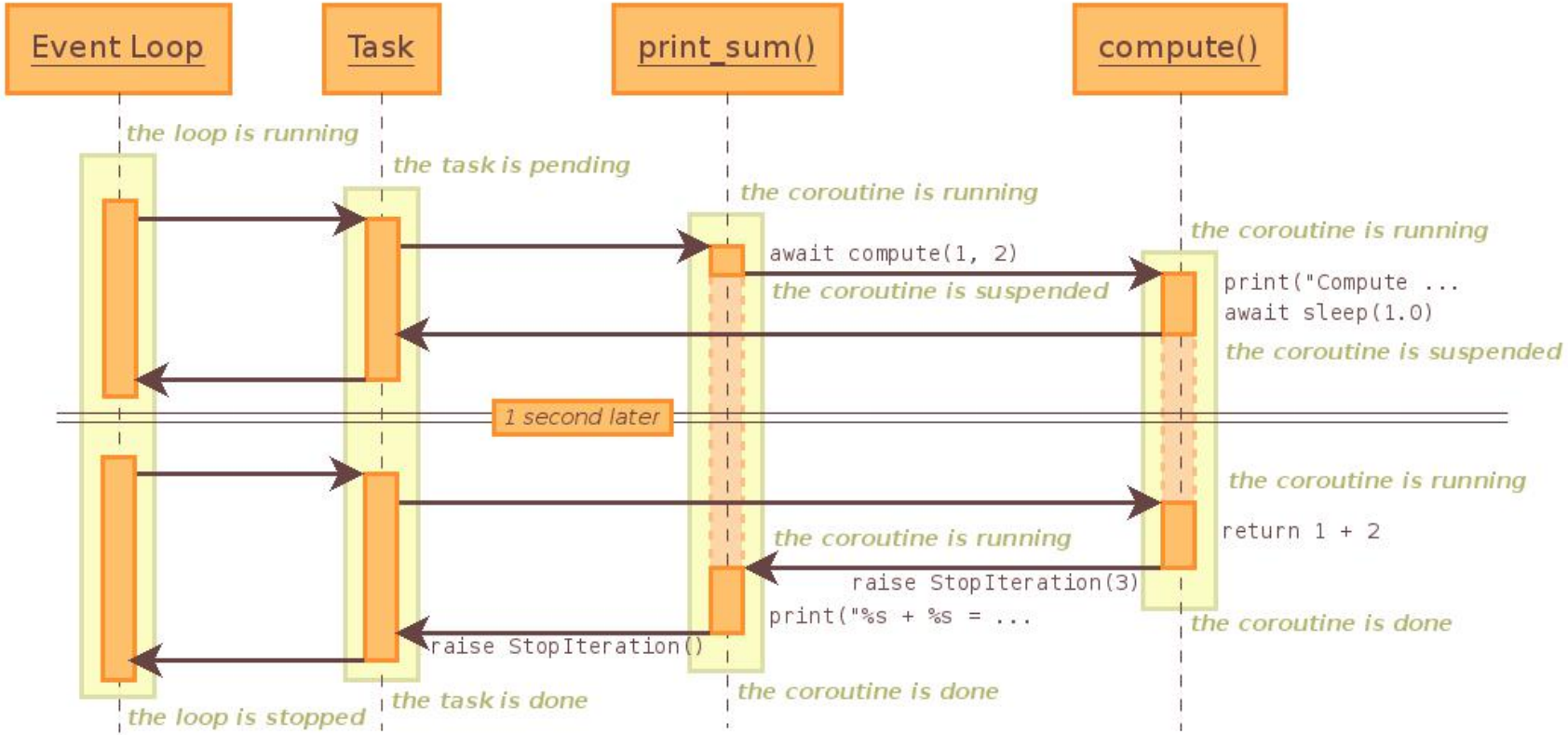
Event Loop

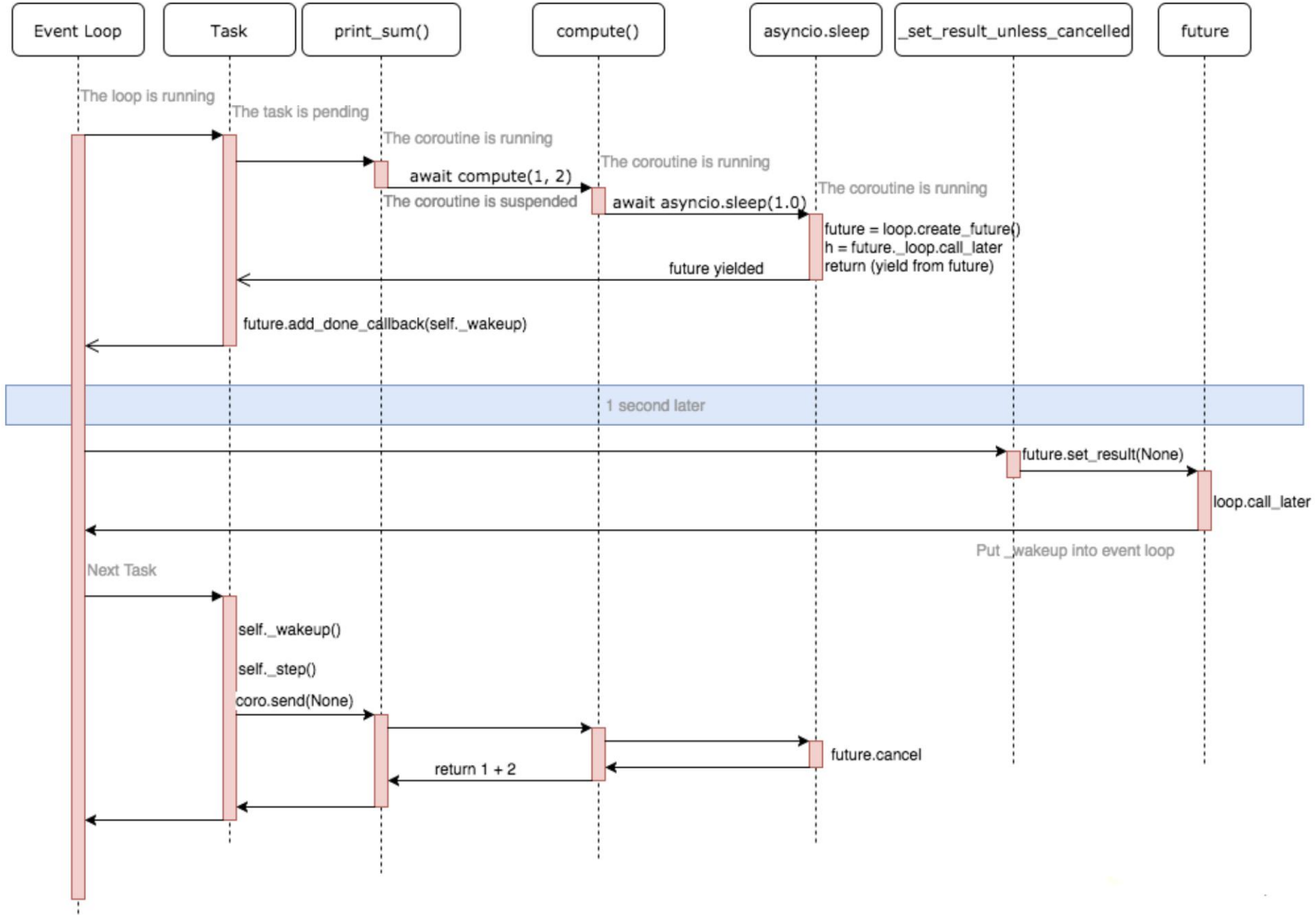


- *“The event loop is the core of every asyncio application. Event loops run asynchronous tasks and callbacks, perform network IO operations, and run subprocesses.”*
- event loop contains within it a list of objects called Tasks. Each Task maintains a single stack, and its own execution pointer as well
- at any one time the event loop can only have one Task actually executing
- pluggable, e.g: uvloop

Event Loop
The Event Loop and asyncio.run()
Explaining event loop in 100 lines of code

Demo


Event Loop
Task
print_sum()
compute()
the loop is running
the task is pending
the coroutine is running
the coroutine is running
await compute(1, 2)
the coroutine is suspended
print("Compute ...
await sleep(1.0)
the coroutine is suspended
1 second later
the coroutine is running
the coroutine is running
return 1 + 2
raise StopIteration(3)
print("%s + %s = ...
the coroutine is done
raise StopIteration()
the coroutine is done
the loop is stopped
the task is done


Event Loop
Task
print_sum()
compute()
asyncio.sleep
_set_result_unless_cancelled
future
The loop is running
The task is pending
The coroutine is running
The coroutine is running
The coroutine is running
await compute(1, 2)
The coroutine is suspended
await asyncio.sleep(1.0)
future = loop.create_future()
h = future._loop.call_later
return (yield from future)
future yielded
future.add_done_callback(self._wakeup)
1 second later
future.set_result(None)
loop.call_later
Put _wakeup into event loop
Next Task
self._wakeup()
self._step()
coro.send(None)
future.cancel
return 1 + 2

Mixing sync and async code

- You cannot run async code unless you have an event loop to run it in
- You can call non-async code from async-code, but might “block”
- Submit long-running blocking work with a thread pool to manage it, use *run_in_executor*

Python Asyncio Part 5 – Mixing Synchronous and Asynchronous Code

Advanced Topic

- Integrate asyncio with other concurrency technologies
- Async IO design patterns (Chaining coroutines, Queues)
- Implement a event loop
- Performance analysis
- … ...

Content

- Python Language Support
- Library Support
  - async web components
  - async in Tornado
  - asynchronous requests with aiohttp

async web components

	Tornado Tornado	aiohttp	fastapi FastAPI
Latest	v1.0.0 (2010) Current: 6.10 (Oct, 2020)	v1.0.0 (2016) Current: 3.7.3 (Nov, 2020)	v0.1.11 (Dec, 2018) Current: 0.68.0(1 month ago)
Feature	asynchronous servers and clients *(tornado.httpserver, tornado.httpclient)* networking modules *(tornado.ioloop, tornado.iostream)* coroutines library *(tornado.gen, tornado.locks, tornado.queues)*	HTTP implementation Web-Sockets out-of-the-box and avoids Callback Hell middlewares and plugable routing	Fast *" One of the fastest Python frameworks available"* Automatic API doc generation
Python Version	Python 2.7, 3.5+, since 5.0 with *asyncio (2 not supported)*	Python 3.7+	Python 3.6+
Stars	20.1k	11.5k	35k

awesome-asyncio
Top 5 Asynchronous Web Frameworks for Python
aio-libs (libs support async/await natively)

async web components(Cont)

	sanic	vibora	starlette	Pydantic
Latest	v18.12 (Dec, 2018) Current: 21.6.2 (2 month ago)	“being completely re-written”	Current: 0.16.0 (1 month ago)	v1.0 (Oct, 2019) Current: v1.8.2(2021.05.11)
Feature	both framework and web server supports the simple and universal async/await syntax out of the box microframework + async	close cousin of Sanic *“2 fast than sanic”* *“Just like Flask”* 	a lightweight ASGI framework/toolkit	Data validation and settings management using python type annotations pydantic enforces type hints at runtime, and provides user friendly errors when data is invalid
Python Version	Python 3.7+	Python 3.6+	Python 3.6+	Python 3.6+
Stars	15.3k	5.7k	5.9k	7.2k

async in Tornado

	Before 4.3	v4.3	v5.X (wishpost v5.1.1)	v6.0+
feature	coroutine based on *yield*	begin support async/await keywords @gen.coroutine → async def	integrated with *asyncio*	use native coroutines internally

implementation	by framework tornado.ioloop
	
	“On Python 2, it uses ``epoll`` (Linux) or ``kqueue`` (BSD and Mac OS X) if they are available, or else we fall back on select().”
			IOLoop.current().start()
			get asyncio's ioloop
			`try:`
			`  import asyncio`
			`except ImportError:`
			`  asyncio = None`
				Compatible with asyncio
				
				“As of Tornado 6.0, `IOLoop` is a wrapper around the `asyncio` event loop.”

Python Version			2.7.9+ / 3.5+	3.5+

Tornado Web Server
python3中tornado框架和asyncio这两个都是异步IO，有什么本质区别吗？
当tornado 集成python 的asynico后 并发现状是如何？

Asynchronous requests with aiohttp

- Read a sequence of URLs from a local file.
- Send GET requests for the URLs and decode the resulting content. If this fails, stop there for a URL.
- Search for the URLs within href tags in the HTML of the responses.
- Write the results to results.txt.
- Do all of the above as asynchronously and concurrently as possible. (Use *aiohttp* for the requests, and *aiofiles* for the file-appends.

aiohttp
Making 1 million requests with python-aiohttp
A Full Program: Asynchronous Requests