Единственное назначение асинхронности – заполнение задержек между выполнением инструкций.

Реально параллельно код в питоне выполняется только при ресурсозатратном мультипроцессинге (библиотека multiprocessing), запуская отдельные процессы. Асинхронность и многопоточность же лишь заполняют время простоя.

Синхронный код выполняет строки кода друг за другом, даже если выполняется ожидание данных. Примеры:

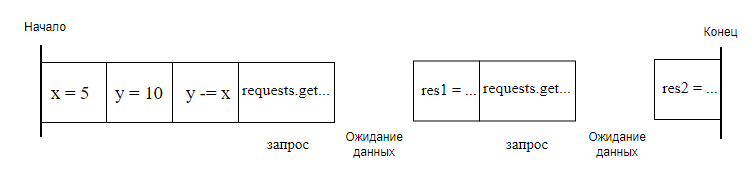
!@#$

x = 5­  
y = 10­­­­

y -= x  
res1 = requests.get("https://www.youtube.com")  
res2 = requests.get("https://vk.com/feed")

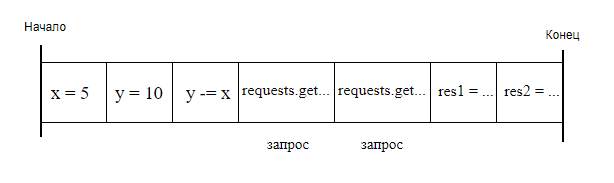
!@#$

Каждая следующая строчка не будет выполняться до момента, пока не выполнена предыдущая. Это значит, что строка **res2 = requests.get("https://vk.com/feed")** не выполнится, пока полностью не отработает строка **res1 = requests.get("https://www.youtube.com")**, в которой выполняется запрос к интернет ресурсу. Это операция ввода. Она требует времени, пока запрос дойдёт до сервера, соберутся нужные данные, и вернётся ответ. Пока все это происходит, наша программа будет просто стоять на месте, чтобы получить результат и записать его в переменную **res1**. Мощности компьютера в этот момент не используются, т.к. не происходит никаких вычислений. Примерно так это выглядит:



А если результат придётся ждать долго? То программа совсем встанет! Чтобы повысить КПД времени простоя, можно заставить код выполняться дальше, пока не придёт ответ.

Асинхронность в питоне реализована в библиотеке asyncio и ключевыми словами async и await. С её помощью во время ожидания программа не будет «висеть», а продолжит выполняться дальше, вернувшись потом в нужную точку за результатом. Выглядеть это будет примерно так:



Код будет выглядеть так:

!@#$

import functools  
import time  
from time import sleep  
  
import aiohttp  
import asyncio  
  
  
async def first\_req():  
 async with aiohttp.ClientSession() as session:  
 async with session.get("https://www.youtube.com") as res1:  
 res1\_text = res1  
 return res1\_text  
  
  
async def second\_req():  
 async with aiohttp.ClientSession() as session:  
 async with session.get("https://vk.com/feed") as res2:  
 res2\_text = res2.status  
 return res2\_text  
  
  
async def main():  
 x = 5  
 y = 10  
 y -= x  
 tasks = []  
  
 tasks.append(asyncio.create\_task(first\_req()))  
 tasks.append(asyncio.create\_task(second\_req()))  
  
 res = await asyncio.gather(\*tasks)  
  
asyncio.run(main())

!@#$

Здесь есть серьезная ошибка – запуск двух сессий, но это пока неважно.

Проверим скорость выполнения кода:

Синхронный:



Асинхронный:



Вот и профит асинхронности! А если запросов будет больше? Попробуем выполнить 6:

!@#$

def main():  
 x = 5  
 y = 10  
 y -= x  
 res1 = requests.get("https://www.youtube.com")  
 res2 = requests.get("https://vk.com/feed")  
 res3 = requests.get("https://vk.com/feed")  
 res4 = requests.get("https://vk.com/feed")  
 res5 = requests.get("https://vk.com/feed")  
 res6 = requests.get("https://vk.com/feed")  
  
  
main()

!@#$

Результат



\*накопировал функции с запросами\*

!@#$

async def main():  
 x = 5  
 y = 10  
 y -= x  
 tasks = []  
  
 tasks.append(asyncio.create\_task(first\_req()))  
 tasks.append(asyncio.create\_task(second\_req()))  
 tasks.append(asyncio.create\_task(third\_req()))  
 tasks.append(asyncio.create\_task(fourth\_req()))  
 tasks.append(asyncio.create\_task(fifth\_req()))  
 tasks.append(asyncio.create\_task(sixth\_req()))  
  
 res = await asyncio.gather(\*tasks)  
  
asyncio.run(main())

!@#$

Результат:



**Основные функции для работы с asyncio**

Чтобы покрыть большую часть задач достаточно выучить лишь фукнций

Asuncio.run() – запускает выполнение асинхронный функций

asyncio.create\_task() – создает обьект, который может быть использован для работы асинхронности (задача), т.к. просто функцию запускать нельзя

asyncio.gather() – позволяет запустить сразу несколько асинхронных задач и возвращает список их результатов. Параметр return\_exceptions означает, что возвращаемый список будет содержать любое исключение, вызванное задачами, вместо того места, где должно было быть значение результата задачи.

async – ключевое слово, обозначающее асинхронную функцию. Используется только при определении функции

await – ключевое слово, обозначающее , что за ним следует асинхронная функция, требующая времени ожидания на своё выполнение. В этом случае код продолжит выполняться дальше и в будущем вернётся в место вызова этой функции за результатом.

В библиотеке asyncio есть встроенные асинхронные аналоги некоторых привычных функций, например, **asyncio.sleep()**. Единственное их отличие в том, что они не блокируют поток, позволяя другим асинхронным функциям выполняться. Вызываются они с ключевым словом await:

!@#$

await asyncio.sleep(1)

!@#$

Подробнее о внутреннем устройстве библиотеки asyncio в этой статье