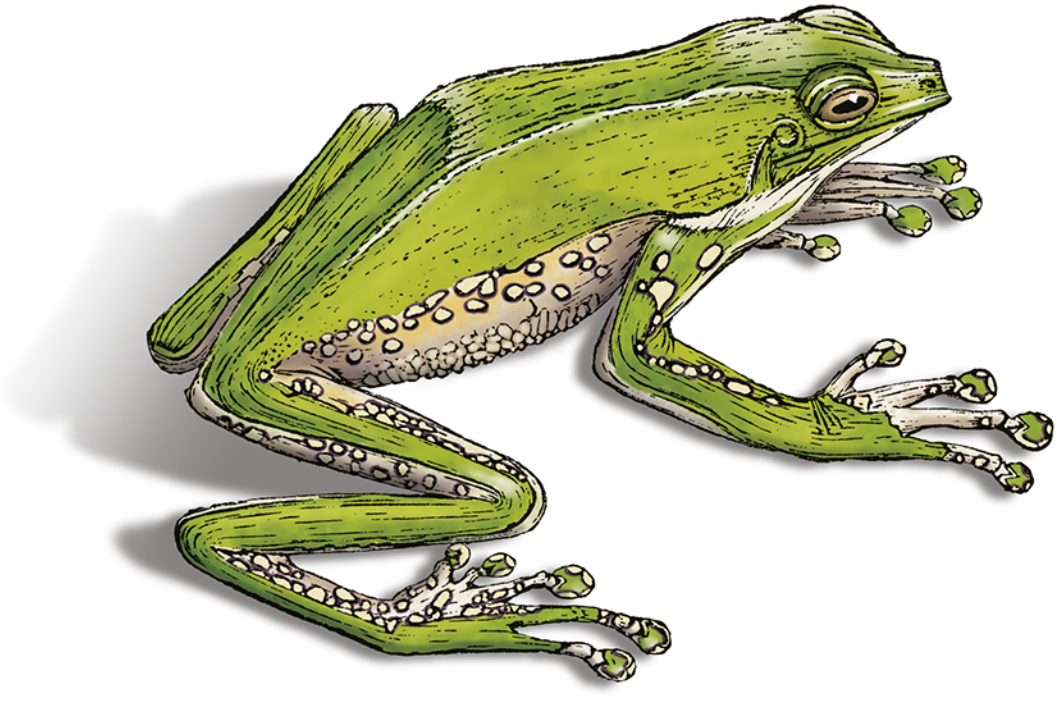
O'REILLY®

Python i Asyncio

Programowanie asynchroniczne



Helion

Caleb Hattingh

Tytuł oryginału: Using Asyncio in Python: Understanding Python's Asynchronous Programming Features

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-7003-6

© 2020 Helion SA

Authorized Polish translation of the English edition of *Using Asyncio in Python*
ISBN 9781492075332 © 2020 Tekmoji Pty Ltd

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/pythas.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pythas
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Kup książkę Poleć książkę

Kup książkę Poleć książkę

ROZDZIAŁ 1.

Prezentacja Asyncio

> Moja historia jest podobna do twojej, tylko ciekawsza, bo są w niej roboty.
>
> — Bender, *Futurama*, odcinek „30% Iron Chef”

Najczęściej zadawanym mi pytaniem odnoście Asyncio w Pythonie 3 jest: „Co to jest i co z tym mogę zrobić?”. Odpowiedź, którą zazwyczaj można usłyszeć na tak postawione pytanie, będzie zapewne mówiła o możliwości wykonywania wielu równoczesnych żądań HTTP w jednym programie. Jednak taka odpowiedź będzie niepełna — i to bardzo. Stosowanie biblioteki Asyncio wymaga zmiany myślenia o strukturze pisanych programów.

Opowieść przedstawiona w dalszej części tego rozdziału zapewnia kontekst, który pozwoli lepiej zrozumieć tę zmianę. Kluczowym zagadnieniem związanym ze stosowaniem biblioteki Asyncio jest to, w jaki sposób można najlepiej wykonywać wiele zadań w tym samym czasie — przy czym nie chodzi o dowolne zadania, a konkretnie o te, które wymagają oczekiwania. Kluczową koncepcją związaną z tym stylem programowania jest to, że podczas oczekiwania na zakończenie *tego* zadania, można zająć się *innymi* zadaniami.

Restauracja ThreadBotów

Załóżmy, że mamy obecnie rok 2051 i jesteśmy właścicielami restauracji. Podstawą ekonomii jest w tym czasie automatyzacja, którą zapewniają głównie roboty, ale potrzeby ludzi się nie zmieniły — wciąż lubią oni od czasu do czasu gdzieś wyjść i coś zjeść w restauracji. W naszej restauracji wszystkie prace wykonują roboty, oczywiście humanoidalne, jednak nie sposób ich pomylić z człowiekiem. Największym producentem robotów jest firma Threading Inc., a jej roboty przeznaczone do pracy są potocznie nazywane „ThreadBotami”.

Poza tym drobnym „robotycznym szczegółem”, jakim jest obsługa, nasza restauracja wygląda i działa bardzo podobnie do tych staromodnych lokali z roku 2020. Jej goście poszukują tego nieco podstarzałego, klasycznego klimatu. Oczekują świeżego jedzenia przygotowywanego z nieprzetworzonych produktów. Chcą siedzieć przy stolikach. Pragną poczekać trochę, aż ich posiłek zostanie przygotowany — choć z drugiej strony nie chcą czekać zbyt długo. A po zjedzeniu chcą dostać rachunek i zapłacić, a może nawet zostawić napiwek — tak, by poczuć się jak w starych czasach.

Kup książkę Poleć książkę

Ponieważ jesteśmy nowi w biznesie gastronomicznym, robimy to samo, co wszyscy inni: zatrudniamy małą armię robotów: jednego by witał klientów (GreetBot), jednego do zbierania zamówień i obsługi gości (WaitBot), jednego do gotowania (ChefBot) i kolejnego do prowadzenia baru (WineBot).

Głodni klienci wchodzą do restauracji i są witani przez GreetBota, naszego reprezentacyjnego ThreadBota. Następnie są kierowani do stolika, a kiedy już zajmą miejsca, WaitBot zbiera od nich zamówienia. Zapisuje je na karteczkach papieru (zależy nam przecież na zachowaniu staromodnego charakteru restauracji, prawda?), które zanosi do kuchni. ChefBot przegląda te zapisane zamówienia i zaczyna przyrządzać dania. WaitBot od czasu do czasu sprawdza, czy jakieś dania zostały już przygotowane, a kiedy jakieś się pojawią, bezzwłocznie roznosi je do stolików. Kiedy goście już zjedzą i chcą opuścić lokal, udają się z powrotem do GreetBota, który podlicza rachunek, obsługuje proces płatności i kurtuazyjnie życzy gościom miłego wieczoru.

Nasza restauracja cieszy się naprawdę dużą popularnością i w krótkim czasie zyskujemy całkiem spore grono klientów. Zatrudnione roboty robią dokładnie to, co im każemy, a przydzielone zadania realizują doskonale. Wszystko idzie świetnie, a my nie możemy sobie wyobrazić, by mogło być lepiej.

Jednak wraz z upływem czasu zaczynamy zauważać pewien problem. Właściwie to nic poważnego; tylko kilka rzeczy, które wydają się być nie w porządku. Wszyscy inni restauratorzy zatrudniający roboty wydają się borykać z podobnymi błahymi problemami. Niepokojące jest to, że problemy te wydają się nasilać wraz ze wzrostem popularności restauracji.

Zdarzają się, choć sporadycznie, pewne bardzo niepokojące kolizje w działaniach robotów: czasami kiedy taca z daniami jest przygotowywana w kuchni, WaitBot zabiera ją jeszcze *zanim* ChefBot w ogóle położy na niej jakieś naczynia. Zazwyczaj kończy się to porozbijanymi naczyniami i wielkim bałaganem w kuchni. Oczywiście ChefBot wszystko sprząta, niemniej można by sądzić, że tak nowoczesne roboty powinny być nieco lepiej zsynchronizowane. Podobne problemy występują w barze: czasami WineBot kładzie kieliszki na zamówione drinki na barze, a WaitBot zabiera je jeszcze zanim WineBot jest puści, co oczywiście kończy się ich rozbiciem i wylaniem Nederburg Cabernet Suavignon.

Zdarza się także, że GreetBot zaprasza nowych gości do stolika dokładnie w tym samym momencie kiedy WaitBot zaczyna go sprzątać, choć stolik był uznawany za pusty. Goście uważają to za dość dziwne. Próbowaliśmy dodawać pewne opóźnienia do logiki sprzątania stolików WaitBota oraz do funkcji rozlokowywania gości przez GreetBota, jednak nie dały one żadnego widocznego efektu i podobne kolizje wciąż występują. Całe szczęście, że zdarzają się dość rzadko.

No cóż, przynajmniej tak było wcześniej. Nasza restauracja stała się tak popularna, że musieliśmy zatrudnić kilka kolejnych ThreadBotów. W piątkowe i sobotnie wieczory, kiedy gości jest wyjątkowo dużo, potrzebny jest drugi GreetBot oraz dwa dodatkowe WaitBoty. Niestety, warunki najmu ThreadBotów są takie, że trzeba je wynajmować na cały tydzień, co oznacza, że przez większą część tygodnia, kiedy jest znacznie spokojniej, musimy ponosić koszty trzech dodatkowych robotów, których praktycznie nie potrzebujemy.

Kolejnym problemem związanym z zasobami, oprócz dodatkowych kosztów, jest to, że dodatkowe roboty oznaczają więcej pracy dla nas. Obsługa czterech robotów i zarządzanie nimi nie stanowiły większego problemu, teraz jednak mamy ich na głowie aż siedem. Nadzorowanie pracy

Kup książkę Poleć książkę

siedmiu ThreadBotów wymaga znacznie więcej wysiłku, a ponieważ nasza restauracja staje się coraz to bardziej znana i popularna, zaczynamy się obawiać, że konieczne będzie wypożyczenie nawet większej ich liczby. Nadzorowanie tych wszystkich robotów powoli staje się pracą na pełen etat. I jeszcze jedno: te dodatkowe ThreadBoty zajmują znacznie więcej miejsca w restauracji. Już z samymi klientami przy stolikach zaczyna się w niej robić ciasno, a co dopiero kiedy kręcą się po niej te wszystkie roboty. Obawiamy się, że kiedy pojawi się konieczność dodania kolejnych maszyn, problem braku miejsca stanie się jeszcze poważniejszy.

Po dodaniu kolejnych trzech robotów pogorszył się także problem kolizji. Teraz zdarza się, że dwa WaitBoty pobierają to samo zamówienie, z tego samego stolika, w tym samym czasie. Wygląda to tak, jakby oba zauważały, że trzeba pobrać zamówienie i robiły to zupełnie nie zwracając uwagi na to, że drugi WaitBot robi dokładnie to samo. Jak można sobie wyobrazić, powoduje to dodatkowe obciążenie kuchni i zwiększa prawdopodobieństwo wystąpienia kolizji podczas pobierania przygotowanych posiłków. Obawiamy się, że kiedy pojawi się konieczność dodania kolejnych robotów, problem ten jeszcze bardziej się nasili.

I znowu mija trochę czasu.

Aż nagle, w pewien wyjątkowo pracowity piątek doznajemy olśnienia: czas zwalnia, nasz umysł się oczyszcza i widzimy fragment restauracji zamrożony w czasie. *Nasze ThreadBoty nic nie robią!* No, może nie do końca nic, ale one jedynie… czekają.

Każdy z trzech WaitBotów stoi przy innym stoliku czekając, aż klienci wybiorą coś z menu i złożą zamówienia. WineBot przygotował 17 drinków (co zabrało mu tylko chwilkę), które teraz czekają na zabranie i rozniesienie, a sam oczekuje na kolejne zamówienia. Jeden z GreetBotów przywitał nową grupę gości i poinformował ich, że muszą minutkę poczekać na odprowadzenie do stolika, i teraz sam czeka na ich odpowiedź. Drugi GreetBot realizuje opłatę kartą kredytową klientów, którzy właśnie zbierają się do wyjścia, i czeka na potwierdzenie wykonania płatności przez terminal. Nawet ChefBot, który przygotowuje 35 dań, nic w tej chwili nie robi, czekając, aż danie, które gotuje, będzie gotowe do ułożenia na talerzu i dostarczenia do stolika przez WaitBota.

I nagle zdajemy sobie sprawę, że choć restauracja jest pełna ThreadBotów, a my nawet myślimy o wynajęciu kolejnych (wraz ze wszystkimi problemami z tym związanymi), to jednak te roboty, którymi dysponujemy, nie są wykorzystywane w pełni.

Ta chwila olśnienia mija, lecz świadomość tego, co zauważyliśmy, pozostaje. Następnego dnia dodajemy do każdego z ThreadBotów moduł gromadzenia danych. Każdy z robotów będzie teraz mierzył, ile czasu spędza na oczekiwaniu, a ile na aktywnym wykonywaniu zadań. Gromadzenie danych zajmuje kolejny tydzień. Po tym czasie, w niedzielny wieczór, siadamy, by przeanalizować dane. Okazuje się, że nawet kiedy restauracja jest w całości zajęta, ThreadBoty przez 98% czasu są bezczynne. Roboty są tak niesłychanie wydajne, że praktycznie każde zadanie mogą wykonać w ułamku sekundy.

Ta bezczynność bardzo nas, jako przedsiębiorców, martwi. Zdajemy sobie sprawę, że wszyscy inni właściciele restauracji obsługiwanych przez roboty działają dokładnie tak samo i borykają się z tymi samymi problemami. „Ale przecież musi być jakieś lepsze rozwiązanie!” — stwierdzamy uderzając pięścią w stół.

Kup książkę Poleć książkę

I tak następnego dnia, czyli w leniwy poniedziałek, robimy coś odważnego: programujemy jednego ThreadBota tak, by wykonywał wszystkie czynności. Za każdym razem, kiedy robot zaczyna oczekiwać, przełącza się na kolejne zadanie, jakie trzeba wykonać w restauracji, niezależnie od tego, co nim jest. Takie działanie wydaje się dość niesamowite — jeden ThreadBot wykonujący pracę wszystkich innych robotów; niemniej mamy pewność, że nasze obliczenia są dobre. A poza tym, poniedziałki są spokojne; nawet jeśli coś pójdzie nie tak, to wpływ na działanie restauracji będzie niewielki. Na potrzeby tego projektu nazwaliśmy robota działającego według nowego programu LoopBotem, gdyż w pętli próbuje wykonać wszystkie zadania, jakie są do zrobienia w restauracji.

Napisanie programu dla LoopBota było trudniejsze niż zazwyczaj. Nie chodzi tylko o to, że jeden ThreadBot musi zostać zaprogramowany do wykonywania wszystkich możliwych zadań, lecz także o to, że musimy dodać do niego jakąś logikę określającą, kiedy robot musi przełączyć się na kolejne zadanie. Jednak obecnie mamy już naprawdę duże doświadczenie w programowaniu ThreadBotów, dlatego napisanie takiego programu nie przysparza nam większego problemu.

Bardzo uważnie obserwujemy działania naszego LoopBota. Porusza się po restauracji błyskawicznie, sprawdzając, gdzie jest coś do zrobienia. Niedługo po otworzeniu restauracji wchodzi do niej pierwszy gość. LoopBot zjawia się przy nim niemal natychmiast i pyta, czy klient chciałby stolik przy oknie, czy bliżej baru. Zaraz potem, kiedy LoopBot zaczyna czekać na odpowiedź, jego program nakazuje mu przełączyć się na kolejne zadanie i robot znika. Zamieramy, gdyż wydaje się nam, że to koszmary błąd, lecz w chwili, kiedy klient zaczyna mówić „Przy oknie, proszę", LoopBot ponownie jest już przy nim. Robot słucha odpowiedzi i kieruje gościa do stolika numer 42. I znowu znika, by sprawdzić zamówienia na napoje, dania, wyczyścić stoliki, upewnić się, czy nie ma gości do powitania — i tak w kółko.

Późnym wieczorem tego samego dnia możemy sobie pogratulować znaczącego sukcesu. Sprawdzenie modułu rejestracji danych LoopBota wykazało bowiem, że nawet kiedy ten jeden robot wykonywał wszystkie zadania w restauracji, ilość czasu, kiedy pozostawał w bezczynności, wynosiła 97%. Ten wynik utwierdził nas w przekonaniu, że eksperyment warto kontynuować w kolejne dni tygodnia.

Kiedy nadchodzi pracowity piątek, możemy już przemyśleć wielki sukces naszego eksperymentu. W normalne dni robocze do obsługi restauracji w zupełności wystarczy jeden LoopBot. Co więcej, zauważyliśmy jeszcze jedną rzecz: kolizje zupełnie znikły. Co zresztą ma sens, w końcu restaurację obsługuje tylko jeden robot, więc trudno by było, żeby przeszkadzał samemu sobie. Nie ma już zatem podwójnych zamówień ani konfliktów co do tego, kto zabierze tace z daniami, by dostarczyć ją do stolika.

Rozpoczyna się piątkowy wieczór i — zgodnie z naszymi oczekiwaniami — jeden robot doskonale radzi sobie z obsługą wszystkich gości oraz wszystkich zadań; praca restauracji przebiega nawet lepiej niż wcześniej. Wyobrażamy sobie, że możemy przyjmować nawet więcej gości i nie przejmować się wynajmowaniem kolejnych ThreadBotów. Już podliczamy, ile na tym możemy zaoszczędzić.

Aż nagle, niestety, coś poszło nie tak: jedno z dań (wyszukany suflet) spadło na podłogę. Nigdy wcześniej nie wydarzyło się nic podobnego. Zaczynamy uważnie studiować dane z modułu gromadzenia danych LoopBota. Okazuje się, że przy jednym ze stolików siedzi wyjątkowo gadatliwy klient. Przyszedł on do restauracji sam, a teraz próbuje nawiązać konwersację z LoopBotem, a nawet

Kup książkę Poleć książkę

czasami próbuje go zatrzymać, chwytając za rękę. W takich okolicznościach nasz LoopBot nie był w stanie zajmować się coraz dłuższą listą zadań do wykonania. To właśnie dlatego w kuchni spadł na podłogę pierwszy suflet: LoopBot nie mógł wrócić na czas i zdjąć go z podajnika, bo był zatrzymywany przez gadatliwego klienta.

Piątek się kończy, a my wracamy do domu, by przemyśleć to, czego się dowiedzieliśmy. To prawda, że LoopBot był w stanie wykonywać wszystkie czynności konieczne do obsługi restauracji w piątkowy wieczór, jednak z drugiej strony w kuchni po raz pierwszy przygotowane danie spadło na podłogę, co nie zdarzyło się nigdy wcześniej. Gadatliwi goście cały czas zajmowali WaitBoty, jednak wcześniej nigdy nie miało to żadnego wpływu na prace kuchni.

Wziąwszy po uwagę wszystkie te zagadnienia dochodzimy do wniosku, że najlepiej będzie pozostać przy korzystaniu z jednego LoopBota. W końcu, dzięki niemu udało się nam wyeliminować przykre kolizje i mamy więcej miejsca w restauracji — miejsca, które możemy wykorzystać, by przyjmować więcej klientów. Niemniej zdaliśmy sobie także sprawę z pewnego ważnego zagadnienia dotyczącego LoopBota: może on działać efektywnie wyłącznie w przypadku, gdy wszystkie realizowane przez niego zadania są krótkie, albo przynajmniej gdy można je wykonywać w krótkim czasie. Jeśli jakieś zadanie zatrzyma LoopBota na zbyt długo, to wszystkie inne zadania ucierpią, bo nie będzie się nimi miał kto zająć.

Trudno jest z góry określić, które zadanie może zająć zbyt dużo czasu. Co jeśli klient zamówi drinka wymagającego długotrwałego i skomplikowanego przygotowania, którego zrobienie zajmie więcej czasu niż normalnie? Co jeśli klient będzie chciał poskarżyć się na jakość posiłku, odmówi zapłaty i będzie trzymał LoopBota za ramię, uniemożliwiając mu zajęcie się innymi czynnościami? Dochodzimy do wniosku, że zamiast próbować z góry przewidzieć wszystkie takie sytuacje, lepiej będzie kontynuować obsługę restauracji przez jednego LoopBota i rejestrować możliwie dużo danych, a problemami zajmować się później, w momencie, gdy wystąpią.

I tak znowu mija trochę czasu.

Stopniowo inni restauratorzy zauważają nasz nowy sposób obsługi restauracji i w końcu dochodzą do wniosku, że także oni mogą prowadzić swój biznes używając tylko jednego ThreadBota. Informacje o tym rozchodzą się coraz dalej i dalej. Po jakimś czasie już każda restauracja działa w taki sposób i aż trudno sobie wyobrazić, że kiedyś wszyscy restauratorzy wynajmowali po kilka ThreadBotów.

Epilog

W opisanej historii każdy robotyczny pracownik restauracji jest jednym wątkiem. Najważniejszym wnioskiem z tej historii jest to, że działanie restauracji w przeważającej większości polega na oczekiwaniu, tak jak w przypadku wywołania metody `request.get()`, która większość czasu oczekuje, aż użytkownik coś wpisze.

W przypadku restauracji, kiedy czynności związane z jej obsługą wykonują ludzie, czas, jaki poświęcają na oczekiwanie nie jest zbyt duży, jednak kiedy te same czynności będą wykonywać superefektywne i szybkie roboty, to niemal całe ich działanie będzie się sprowadzać do oczekiwania.

Kup książkę Poleć książkę

W świecie programowania, kiedy mamy do czynienia z programowaniem sieciowym, dzieje się dokładnie tak samo. Procesor wykonuje co ma do zrobienia, a następnie czeka na dane przekazywane przez mechanizmy wejścia-wyjścia. Nowoczesne procesory są bardzo szybkie — setki tysięcy razy szybsze do transmisji sieciowych. Dlatego też procesory wykonujące programy sieciowe większość czasu tracą na oczekiwaniu.

Wniosek z tej historii jest taki, że programy można celowo pisać w taki sposób, by umożliwiały procesorowi zmianę wykonywanego zadania, jeśli pojawi się taka konieczność. Choć zapewnia to korzyści ekonomiczne (użycie mniejszej liczby procesorów do wykonywania tej samej pracy), to jednak prawdziwą zaletą takiego rozwiązania w porównaniu z pracą wielowątkową (wykorzystującą wiele procesorów) jest eliminacja zjawiska wyścigu (ang. *race condition*).

Takie rozwiązanie nie jest jednak idealne: jak opisano w historii, także i ono, jak większość rozwiązań technologicznych, ma swoje zalety i wady. Wprowadzenie LoopBota rozwiązało pewną klasę problemów, lecz jednocześnie spowodowało pojawienie się innych, nowych — a jednym spośród nich, i to wcale nie najmniejszym, było to, że właściciel restauracji musiał nauczyć się nieco innego sposobu programowania.

Jakie problemy stara się rozwiązywać Asyncio?

Jeśli chodzi o operacje wejścia-wyjścia, istnieją dokładnie (tylko!) dwa powody przemawiające za wykorzystaniem pracy równoległej bazującej na asynchroniczności w porównaniu z pracą równoległą bazującą na użyciu wielu wątków:

- Biblioteka Asyncio stanowi bezpieczniejszą alternatywę niż wielozadaniowość z wywłaszczaniem (na przykład stosowanie wielu wątków), pozwala unikać błędów, wyścigów oraz wielu innych niedeterministycznych zagrożeń, które często występują w nietrywialnych aplikacjach wielowątkowych.
- Biblioteka Asyncio zapewnia proste możliwości wsparcia dla korzystania z wielu tysięcy *jednoczesnych* połączeń korzystających z gniazd, w tym także możliwość obsługi wielu długotrwałych połączeń realizowanych przy użyciu nowych technologii, takich jak WebSockets czy MQTT, wykorzystywanych przez aplikacje Internetu Rzeczy (IoT).

I tyle.

Wątki, jako model programistyczny, najlepiej nadają się do rozwiązywania pewnego rodzaju zadań obliczeniowych, które najlepiej jest wykonywać korzystając z wielu procesorów i współdzielonej pamięci, używanej do efektywnej komunikacji pomiędzy poszczególnymi wątkami. W takich zastosowaniach, użycie przetwarzania wielordzeniowego z wykorzystaniem wspólnej pamięci jest złem koniecznym, gdyż wymaga tego dziedzina problemu.

Jednak programowanie sieciowe *nie* należy do tej dziedziny. Wnioski płynące z przedstawionej historii pokazują, że programowanie sieciowe w głównej mierze sprowadza się do „oczekiwania aż coś się stanie" i z tego względu w programach sieciowych nie musimy korzystać z systemu operacyjnego i jego możliwości efektywnej realizacji zadań na wielu procesorach. Co więcej, nie potrzebujemy wcale ryzyka, z jakim wiąże się wielozadaniowość z wywłaszczaniem, takiego jak wyścigi, które mogą występować podczas korzystania ze współdzielonej pamięci.

Kup książkę Poleć książkę

Jednak istnieje wiele nieporozumień i błędnych informacji dotyczących innych potencjalnych korzyści zapewnianych przez modele programowania bazujące na zdarzeniach. Poniżej przedstawiłem kilka przykładów takich często spotykanych błędnych opinii:

Asyncio sprawi, że nasz kod będzie piekielnie szybki.

Niestety — to nieprawda. W rzeczywistości większość pomiarów pokazuje, że rozwiązania korzystające z wątków są nieco szybsze od porównywalnych rozwiązań korzystających z Asyncio. Gdyby za metrykę wydajności uznać sam stopień współbieżności, to zastosowanie Asyncio *faktycznie* w pewnym stopniu ułatwiłoby tworzenie bardzo wielu równoczesnych połączeń sieciowych. Systemy operacyjne często narzucają pewien górny limit liczby tworzonych wątków, a liczba ta jest znacząco mniejsza od liczby możliwych do utworzenia połączeń sieciowych. Te ograniczenia narzucane przez system operacyjny można zmienić, niemniej łatwiej jest skorzystać z biblioteki Asyncio. I choć oczekujemy, że utworzenie wielu tysięcy wątków powinno powodować dodatkowe koszty związane z ich *przełączaniem*, których można uniknąć dzięki zastosowaniu koprocedur (ang. *coroutines*), to jednak w praktyce okazuje się, że efekty są trudnie do zmierzenia[1]. Nie, bez wątpienia szybkość nie jest zaletą stosowania Asyncio w Pythonie, a jeśli to właśnie jej ktoś poszukuje, po powinien zainteresować się *Cythonem*.

Asyncio sprawia, że wątki stają się niepotrzebne.

Bez dwóch zdań — nie! Prawdziwą zaletą wątków jest możliwość pisania programów działających na wielu procesorach, w których różne zadania obliczeniowe mogą używać tej samej pamięci. Z możliwości tych korzysta na przykład biblioteka obliczeniowa `numpy`, która przyspiesza niektóre obliczenia macierzowe wykonując je na wielu procesorach i używając do tego współdzielonej pamięci. Jeśli chodzi o samą efektywność, to ten model programowania zagadnień obliczeniowych nie ma sobie równych.

Asyncio eliminuje problem z GIL.

Także to stwierdzenie nie jest prawdziwe. Faktem jest, że w przypadku korzystania z Asyncio nie trzeba się przejmować problemem GIL[2], jednak wynika to tylko z tego, że problem ten występuje wyłącznie w programach wielowątkowych. „Problem GIL", o którym tu mowa, jest związany z brakiem możliwości równoczesnego wykonywania kodu na wielu rdzeniach podczas korzystania z wielu wątków. Biblioteka Asyncio (niemal z definicji) korzysta z jednego wątku, dlatego też GIL nie ma żadnego wpływu na jej działanie, jednak nawet ona nie daje nam możliwości korzystania z wielu rdzeni procesora[3]. Warto także zaznaczyć, że na procesorach

[1] Trudno jest znaleźć badania w tym zakresie, jednak wyniki pokazują, że przełączenie kontekstu wątku w systemie Linux na nowoczesnym sprzęcie zajmuje około 50 mikrosekund. Wyobraźmy to sobie w (bardzo) ogólny i niedokładny sposób: zastosowanie 1000 wątków powodowałoby stratę 50 milisekund na samo przełączanie kontekstu. Te narzuty czasowe się sumują, jednak bez wątpienia nie sprawią one, że aplikacja nie będzie się nadawać do użycia.

[2] Globalna blokada interpretera (ang. *global interpreter lock*, w skrócie *GIL*) sprawia, że kod interpretera Pythona (przy czym nie chodzi tu o kod, który sami piszemy!) jest bezpieczny pod względem wielowątkowym, gdyż ma miejsce blokowanie przetwarzania kodów operacji (ang. *opcodes*). Niestety ma to ten niefortunny skutek, że działanie interpretera jest ograniczone do jednego procesora, przez co tracona jest możliwość równoległej pracy na wielu rdzeniach.

[3] Z podobnego względu problem GIL w ogóle nie występuje w języku JavaScript — programy pisane w tym języku są wykonywane tylko w jednym wątku.

Kup książkę Poleć książkę

wielordzeniowych GIL może powodować dodatkowe problemy oprócz tych, które już zostały wymienione: na konferencji PyCon 2010 Dave Beazley wygłosił na ten temat wykład zatytułowany *Understanding the Python GIL*, a wiele z poruszanych w nim zagadnień wciąż jest aktualnych.

Asyncio zapobiega występowaniu wyścigów.
Nieprawda. W programowaniu wielowątkowym niebezpieczeństwo wystąpienia wyścigu istnieje zawsze, niezależnie od tego, czy używane będą wątki, czy model programowania bazujący na zdarzeniach. To prawda, że stosowanie biblioteki Asyncio pozwala niemal w całości wyeliminować pewną *klasę* wyścigów, występujących powszechnie w programach wielowątkowych, takich jak rywalizacja o dostęp do współdzielonej pamięci używanej przez wiele procesów. Z drugiej strony, Asyncio nie eliminuje możliwości występowania innych rodzajów wyścigów, takich jak występujące w rozproszonych architekturach mikrousług wyścigi pomiędzy różnymi procesami korzystającymi z tych samych zasobów. Wciąż trzeba zwracać uwagę na sposób korzystania ze współdzielonych zasobów. Podstawową zaletą Asyncio w porównaniu z kodem korzystającym z wielu wątków jest to, że punkty przekazywania sterowania pomiędzy koprocedurami są *widoczne* (dzięki zastosowaniu słowa kluczowego `await`), dzięki czemu znacznie łatwiej można zorientować się jak i kiedy wykonywane są odwołania do współdzielonych zasobów.

Asyncio sprawia, że programowanie współbieżne staje się łatwe.
Hm… jak w ogóle mam się do tego ustosunkować?

Ten ostatni z przedstawionych mitów jest najbardziej niebezpieczny. Programowanie współbieżne *zawsze* jest złożone i trudne, niezależnie od tego, czy używamy wielowątkowości, czy Asyncio. Kiedy eksperci stwierdzają, że „biblioteka Asyncio sprawia, że programowanie współbieżne jest łatwiejsze", tak na prawdę mają na myśli to, że stosowanie tej biblioteki nieco ułatwia unikanie pewnych rodzajów prawdziwie koszmarnych błędów związanych ze zjawiskiem wyścigów — błędów, które przyprawiają o bezsenność i o których historie są opowiadane innym programistom cichym szeptem przy ognisku, kiedy wilki wyją w oddali…

Jednak nawet kiedy stosujemy Asyncio, wciąż musimy sobie radzić z ogromną złożonością. W jaki sposób nasza aplikacja będzie przechodzić testy bezpieczeństwa? W jaki sposób będziemy komunikować się z bazą danych, która zezwala tylko na kilka równoczesnych połączeń — znacznie mniej niż na pięć tysięcy połączeń sieciowych do klientów? W jaki sposób nasz program łagodnie zakończy wszystkie połączenia, kiedy odbierze sygnał do zamknięcia? Jak będziemy obsługiwać (blokujące!) odwołania do dysku oraz operacje rejestracji? To przykłady tylko kilku spośród bardzo wielu trudnych decyzji projektowych, które trzeba będzie podjąć.

Zaprojektowanie aplikacji wciąż będzie złożonym problemem, jednak można mieć nadzieję, że nieco łatwiej będzie określić logikę jej działania, kiedy aplikacja będzie używać tylko jednego wątku.

Kup książkę Poleć książkę


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Nowa wizja bezpieczeństwa kodu Pythona!

Programowanie współbieżne jest ważną techniką w tworzeniu nowoczesnych rozwiązań sieciowych. Programiści Pythona często w tym celu korzystają z wątków i mechanizmu wywłaszczania. Z tym że nie jest to optymalne rozwiązanie — z uwagi na ryzyko naruszenia bezpieczeństwa. Istnieje też możliwość programowania asynchronicznego z wykorzystaniem biblioteki Asyncio, która została dodana w Pythonie 3.4. Złożoność API Asyncio budzi jednak obawy programistów Pythona, również biegle posługujących się tym językiem. Mimo to wysiłek włożony w zrozumienie działania Asyncio jest opłacalny, gdyż biblioteka ta pozwala na skuteczne rozwiązywanie problemów ze współbieżnym programowaniem sieciowym.

Lektura tej książki ułatwi Ci pozbycie się obaw przed biblioteką Asyncio. Zrozumiesz jej podstawowe elementy, co pozwoli Ci na rozpoczęcie programowania sterowanego zdarzeniami i prostą obsługę tysięcy jednoczesnych połączeń sieciowych. Dowiesz się, dlaczego Asyncio stanowi bezpieczniejszą alternatywę dla wielozadaniowości z wywłaszczaniem wątków, i dogłębnie zrozumiesz koncepcję programowania asynchronicznego. Następnie przeanalizujesz wprowadzone w Pythonie zmiany, dzięki którym możliwe jest programowanie asynchroniczne. Dowiesz się także, w jakich konkretnie sytuacjach biblioteka Asyncio jest wyjątkowo użyteczna i których narzędzi należy wtedy używać. W książce pokazano najlepsze sposoby wykorzystania nowych możliwości Asyncio.

W książce między innymi:

- porównanie programowania współbieżnego z wykorzystaniem Asyncio i wątków
- podstawy programowania bazującego na zdarzeniach
- możliwości Asyncio ważne dla programistów końcowych oraz twórców frameworków
- składnia async/await, w tym API koprocedur i klasy Future
- szczegółowe przypadki użycia kilku bibliotek zgodnych z Asyncio

Caleb Hattingh programuje w Pythonie od mniej więcej dwudziestu lat. Używał go do modelowania reakcji chemicznych, tworzenia systemów rezerwacji miejsc w hotelach, budowy systemów CRM, witryn WWW czy też do tworzenia oprogramowania wykorzystującego system GPS. Często występuje jako prelegent podczas konferencji PyCon AU. Chętnie angażuje się w pomoc młodym programistom, na przykład jako mentor CoderDojo. Bierze też udział w Software Carpentry, a nawet w Govhacker.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-7003-6

9 788328 370036

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 39,90 zł