Efektywny Python

90 sposobów na lepszy kod

WYDANIE II

Helion

Brett Slatkin

Tytuł oryginału: Effective Python: 90 Specific Ways to Write Better Python (2nd Edition)

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-6732-6

Authorized translation from the English language edition, entitled: EFFECTIVE PYTHON: 90 SPECIFIC WAYS TO WRITE BETTER PYTHON, 2nd Edition by SLATKIN, BRETT, published by Pearson Education, Inc, publishing as Addison-Wesley Professional.

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)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/efpyt2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Wprowadzenie **11**

Podziękowania **15**

O autorze **17**

Rozdział 1. Programowanie zgodne z duchem Pythona **19**

Sposób 1. Ustalenie używanej wersji Pythona 19
Sposób 2. Stosuj styl PEP 8 21
Sposób 3. Różnice między typami bytes i str 23
Sposób 4. Wybieraj interpolowane ciągi tekstowe f zamiast ciągów tekstowych formatowania w stylu C i funkcji str.format() 28
Sposób 5. Decyduj się na funkcje pomocnicze zamiast na skomplikowane wyrażenia 38
Sposób 6. Zamiast indeksowania wybieraj rozpakowanie wielu operacji przypisania 41
Sposób 7. Preferuj użycie funkcji enumerate() zamiast range() 44
Sposób 8. Używaj funkcji zip() do równoczesnego przetwarzania iteratorów 46
Sposób 9. Unikaj bloków else po pętlach for i while 48
Sposób 10. Unikaj powtórzeń w wyrażeniach przypisania 50

Rozdział 2. Lista i słownik **57**

Sposób 11. Umiejętnie podziel sekwencje 57
Sposób 12. Unikaj użycia indeksów początek, koniec i wartości kroku w pojedynczej operacji podziału 60
Sposób 13. Wybieraj rozpakowanie typu catch-all zamiast tworzenia wycinków 62
Sposób 14. Używaj parametru key podczas sortowania według skomplikowanych kryteriów 66
Sposób 15. Zachowaj ostrożność, gdy polegasz na kolejności wstawiania elementów do obiektu typu dict 71

Sposób 16. Podczas obsługi brakujących kluczy słownika wybieraj funkcję get() zamiast operatora in i wyjątku KeyError 78
Sposób 17. Podczas obsługi brakujących elementów w wewnętrznym stanie wybieraj typ defaultdict zamiast metody setdefault() 82
Sposób 18. Wykorzystaj metodę __missing__() do tworzenia wartości domyślnych w zależności od klucza 84

Rozdział 3. Funkcje 89

Sposób 19. Gdy funkcja zwraca wiele wartości, nie rozpakowuj więcej niż trzech zmiennych 89
Sposób 20. Preferuj wyjątki zamiast zwrotu wartości None 92
Sposób 21. Zobacz, jak domknięcia współdziałają z zakresem zmiennej 95
Sposób 22. Zmniejszenie wizualnego zagmatwania za pomocą zmiennej liczby argumentów pozycyjnych 98
Sposób 23. Zdefiniowanie zachowania opcjonalnego za pomocą argumentów w postaci słów kluczowych 101
Sposób 24. Użycie None i docstring w celu dynamicznego określenia argumentów domyślnych 105
Sposób 25. Wymuszaj czytelność kodu, stosując jedynie argumenty w postaci słów kluczowych 108
Sposób 26. Dekoratory funkcji definiuj za pomocą functools.wraps 112

Rozdział 4. Konstrukcje składane i generatory 117

Sposób 27. Używaj list składanych zamiast funkcji map() i filter() 117
Sposób 28. Unikaj więcej niż dwóch wyrażeń na liście składanej 119
Sposób 29. Stosuj wyrażenia przypisania, aby unikać powielania zadań w konstrukcjach składanych 121
Sposób 30. Rozważ użycie generatorów, zamiast zwracać listy 124
Sposób 31. Podczas iteracji przez argumenty zachowuj postawę defensywną 126
Sposób 32. Rozważ użycie generatora wyrażeń dla dużych list składanych 131
Sposób 33. Twórz wiele generatorów za pomocą wyrażenia yield from 132
Sposób 34. Unikaj wstrzykiwania danych do generatorów za pomocą metody send() 135
Sposób 35. Unikaj w generatorach przejścia między stanami za pomocą metody throw() 140
Sposób 36. Rozważ stosowanie modułu itertools w pracy z iteratorami i generatorami 144

Rozdział 5. Klasy i interfejsy 151

Sposób 37. Twórz klasy, zamiast zagnieżdżać wiele poziomów typów wbudowanych 151
Sposób 38. Dla prostych interfejsów akceptuj funkcje zamiast klas 157
Sposób 39. Użycie polimorfizmu @classmethod w celu ogólnego tworzenia obiektów 160
Sposób 40. Inicjalizacja klasy nadrzędnej za pomocą wywołania super() 165

Sposób 41. Rozważ łączenie funkcjonalności za pomocą klas domieszek 169
Sposób 42. Preferuj atrybuty publiczne zamiast prywatnych 173
Sposób 43. Stosuj dziedziczenie po collections.abc w kontenerach typów niestandardowych 178

Rozdział 6. Metaklasy i atrybuty 183

Sposób 44. Używaj zwykłych atrybutów zamiast metod typu getter i setter 183
Sposób 45. Rozważ użycie @property zamiast refaktoryzacji atrybutów 187
Sposób 46. Stosuj deskryptory, aby wielokrotnie wykorzystywać metody udekorowane przez @property 191
Sposób 47. Używaj metod __getattr__(), __getattribute__() i __setattr__() dla opóźnionych atrybutów 196
Sposób 48. Sprawdzaj podklasy za pomocą __init_subclass__ 201
Sposób 49. Rejestruj istniejące klasy za pomocą __init_subclass__() 208
Sposób 50. Adnotacje atrybutów klas dodawaj za pomocą metody __set_name__() 212
Sposób 51. Dla złożonych rozszerzeń klas wybieraj dekoratory klas zamiast metaklas 216

Rozdział 7. Współbieżność i równoległość 223

Sposób 52. Używaj modułu subprocess do zarządzania procesami potomnymi 224
Sposób 53. Użycie wątków dla operacji blokujących wejście- yjście, unikanie równoległości 228
Sposób 54. Używaj klasy Lock, aby unikać stanu wyścigu w wątkach 232
Sposób 55. Używaj klasy Queue do koordynacji pracy między wątkami 236
Sposób 56. Naucz się rozpoznawać, kiedy współbieżność jest niezbędna 244
Sposób 57. Unikaj tworzenia nowych egzemplarzy Thread na żądanie fan-out 248
Sposób 58. Pamiętaj, że stosowanie Queue do obsługi współbieżności wymaga refaktoringu 252
Sposób 59. Rozważ użycie klasy ThreadPoolExecutor, gdy wątki są potrzebne do zapewnienia współbieżności 258
Sposób 60. Zapewnij wysoką współbieżność operacji wejścia-wyjścia dzięki użyciu współprogramów 260
Sposób 61. Naucz się przekazywać do asyncio wątkowane operacje wejścia-wyjścia 264
Sposób 62. Połączenie wątków i współprogramów w celu ułatwienia konwersji na wersję stosującą asyncio 274
Sposób 63. Maksymalizuj responsywność przez unikanie blokującej pętli zdarzeń asyncio 280
Sposób 64. Rozważ użycie concurrent.futures(), aby otrzymać prawdziwą równoległość 283

Rozdział 8. Niezawodność i wydajność 289

Sposób 65. Wykorzystanie zalet wszystkich bloków w konstrukcji try-except-else-finally ... 289
Sposób 66. Rozważ użycie poleceń contextlib i with w celu uzyskania wielokrotnego użycia konstrukcji try-finally 294
Sposób 67. Podczas obsługi czasu lokalnego używaj modułu datetime zamiast time 297
Sposób 68. Niezawodne użycie pickle wraz z copyreg 301
Sposób 69. Gdy ważna jest precyzja, używaj modułu decimal 307
Sposób 70. Przed optymalizacją przeprowadzaj profilowanie 310
Sposób 71. Wybieraj typ deque podczas tworzenia kolejek typu producent – konsument ... 314
Sposób 72. Podczas wyszukiwania danych w sortowanych sekwencjach stosuj moduł bisect 321
Sposób 73. W kolejkach priorytetowych używaj modułu heapq 323
Sposób 74. Podczas kopiowania zerowego obiektów typu bytes używaj egzemplarzy memoryview i bytearray 331

Rozdział 9. Testowanie i debugowanie 337

Sposób 75. Używaj ciągów tekstowych repr do debugowania danych wyjściowych 338
Sposób 76. W podklasach klasy TestCase sprawdzaj powiązane ze sobą zachowanie 341
Sposób 77. Izoluj testy od siebie za pomocą metod setUp(), tearDown(), setUpModule() i tearDownModule() 348
Sposób 78. Podczas testowania kodu zawierającego skomplikowane zależności korzystaj z imitacji 350
Sposób 79. Hermetyzuj zależności, aby ułatwić tworzenie imitacji i testowanie 357
Sposób 80. Rozważ interaktywne usuwanie błędów za pomocą pdb 361
Sposób 81. Stosuj moduł tracemalloc, aby poznać sposób użycia pamięci i wykryć jej wycieki 365

Rozdział 10. Współpraca 369

Sposób 82. Kiedy szukać modułów opracowanych przez społeczność? 369
Sposób 83. Używaj środowisk wirtualnych dla odizolowanych i powtarzalnych zależności 370
Sposób 84. Dla każdej funkcji, klasy i modułu utwórz docstring 375
Sposób 85. Używaj pakietów do organizacji modułów i dostarczania stabilnych API 380
Sposób 86. Rozważ użycie kodu o zasięgu modułu w celu konfiguracji środowiska wdrożenia 385
Sposób 87. Zdefiniuj główny wyjątek Exception w celu odizolowania komponentu wywołującego od API 387
Sposób 88. Zobacz, jak przerwać krąg zależności 391
Sposób 89. Rozważ użycie modułu warnings podczas refaktoryzacji i migracji kodu 395
Sposób 90. Rozważ stosowanie analizy statycznej za pomocą modułu typing w celu usuwania błędów 401

7

Współbieżność i równoległość

Współbieżność występuje wtedy, gdy komputer *pozornie* wykonuje jednocześnie wiele różnych zadań. Na przykład w komputerze wyposażonym w procesor o tylko jednym rdzeniu system operacyjny będzie bardzo szybko zmieniał aktualnie wykonywany program na inny. Tym samym programy są wykonywane na przemian, co tworzy iluzję ich jednoczesnego działania.

Z kolei równoległość to *faktyczne* wykonywanie jednocześnie wielu różnych zadań. Jeżeli komputer jest wyposażony w wielordzeniowy procesor, to poszczególne rdzenie mogą jednocześnie wykonywać różne zadania. Ponieważ poszczególne rdzenie procesora wykonują polecenia innego programu, więc poszczególne aplikacje działają jednocześnie i w tym samym czasie każda z nich odnotowuje postęp w działaniu.

W ramach jednego programu współbieżność to narzędzie ułatwiające programistom rozwiązywanie pewnego rodzaju problemów. Programy współbieżne pozwalają na zastosowanie wielu różnych ścieżek działania wraz z oddzielnymi strumieniami wejścia-wyjścia, aby użytkownik miał wrażenie, że poszczególne operacje w programie odbywają się jednocześnie i niezależnie.

Kluczowa różnica między współbieżnością i równoległością to *szybkość*. Kiedy w programie są stosowane dwie oddzielne ścieżki jego wykonywania, to czas potrzebny na wykonanie całego zadania programu zmniejsza się o połowę. Współczynnik szybkości wykonywania wynosi więc dwa. Z kolei współbieżnie działające programy mogą wykonywać tysiące oddzielnych ścieżek działania, ale to nie przełoży się w ogóle na zmniejszenie ilości czasu, jaki jest potrzebny na wykonanie całej pracy.

Python ułatwia tworzenie programów współbieżnych z zastosowaniem różnych stylów. Wątki zapewniają względnie niewielką obsługę współbieżności, podczas gdy współprogramy dostarczają wiele funkcjonalności związanych z współbieżnością. Ponadto Python jest używany do równoległego wykonywania zadań za pomocą wywołań systemowych, podprocesów oraz rozszerzeń utworzonych w języku C. Jednak osiągnięcie stanu, w którym współbieżny

kod Pythona będzie faktycznie wykonywany równolegle, może być bardzo trudne. Dlatego też niezwykle ważne jest poznanie najlepszych sposobów wykorzystania Pythona w tych nieco odmiennych sytuacjach.

Sposób 52. Używaj modułu subprocess do zarządzania procesami potomnymi

Python oferuje zaprawione w bojach biblioteki przeznaczone do wykonywania procesów potomnych i zarządzania nimi. Tym samym Python staje się doskonałym językiem do łączenia ze sobą innych narzędzi, na przykład działających w powłoce. Kiedy istniejące skrypty powłoki z czasem stają się skomplikowane, jak to często się zdarza, wówczas przepisanie ich w Pythonie jest naturalnym wyborem w celu zachowania czytelności kodu i możliwości jego dalszej obsługi.

Procesy potomne uruchamiane przez Pythona mogą działać równolegle, a tym samym Python może wykorzystać wszystkie rdzenie komputera i zmaksymalizować przepustowość aplikacji. Wprawdzie sam Python może być ograniczany przez procesor (zobacz sposób 53.), ale bardzo łatwo wykorzystać ten język do koordynowania zadań obciążających procesor.

Na przestrzeni lat Python oferował wiele sposobów uruchamiania podprocesów, między innymi za pomocą wywołań `os.popen` i `os.exec*`. Obecnie najlepszym i najprostszym rozwiązaniem w zakresie zarządzania procesami potomnymi jest użycie wbudowanego modułu `subprocess`. Uruchomienie podprocesu za pomocą modułu `subprocess` jest proste. W poniższym fragmencie kodu funkcja wygodna `run()` uruchamia proces, odczytuje dane wyjściowe procesu potomnego i czeka na jego zakończenie.

```
import subprocess

result = subprocess.run(
    ['echo', 'Witaj z procesu potomnego!'],
    capture_output=True,
    encoding='utf-8')

result.check_returncode()  # Brak wyjątku oznacza poprawne zakończenie zadania
print(result.stdout)

>>>
Witaj z procesu potomnego!
```

> **Uwaga**
>
> W przykładach dla tego sposobu przyjąłem założenie, że w swoim systemie operacyjnym masz dostępne polecenia `echo`, `sleep` i `openssl`. W systemie Windows może ich nie być. Musisz dokładnie zapoznać się z pełnym kodem źródłowym w tym sposobie i ustalić, jak można go uruchamiać w systemie Windows.

Procesy potomne będą działały niezależnie od ich procesu nadrzędnego, czyli interpretera Pythona. Jeżeli proces potomny zostanie utworzony za pomocą klasy `Popen` zamiast funkcji `run()`, wówczas jego stan można okresowo sprawdzać, gdy Python wykonuje inne zadania.

```
proc = subprocess.Popen(['sleep', '1'])
while proc.poll() is None:
    print('Pracuję...')
    # Miejsce na zadania, których wykonanie wymaga dużo czasu
    # ...
print('Kod wyjścia', proc.poll())

>>>
Pracuję...
Pracuję...
Pracuję...
Pracuję...
Kod wyjścia 0
```

Oddzielenie procesów potomnego i nadrzędnego oznacza, że proces nadrzędny może równocześnie uruchomić dowolną liczbę procesów potomnych. Można to zrobić, uruchamiając jednocześnie wszystkie procesy potomne.

```
import time

start = time.time()
sleep_procs = []
for _ in range(10):
    proc = subprocess.Popen(['sleep', '1'])
    sleep_procs.append(proc)
```

Następnie można czekać na zakończenie przez nie operacji wejścia-wyjścia i zakończyć ich działanie za pomocą metody `communicate()`.

```
for proc in sleep_procs:
    proc.communicate()

end = time.time()
delta = end - start
print(f'Zakończono w ciągu {delta:.3f} sekund')

>>>
Zakończono w ciągu 1.05 sekund
```

Jeżeli wymienione procesy działają w sekwencji, to całkowite opóźnienie wynosi co najmniej 10 sekund, a nie tylko około sekundy, jak to zostało zmierzone w omawianym programie.

Istnieje również możliwość potokowania danych z programu Pythona do podprocesów oraz pobierania ich danych wyjściowych. Tym samym można wykorzystać inne programy do równoczesnego działania. Na przykład przyjmujemy założenie, że narzędzie powłoki `openssl` jest używane do szyfrowania pewnych danych. Uruchomienie procesu potomnego wraz z argumentami pochodzącymi z powłoki oraz potokowanie wejścia-wyjścia jest łatwe.

```
import os

def run_encrypt(data):
    env = os.environ.copy()
    env['password'] = 'zf7ShyBhZOraQDdE/FiZpm/m/8f9X+M1'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    proc.stdin.write(data)
    proc.stdin.flush()  # Gwarantujemy, że proces potomny otrzyma dane wejściowe
    return proc
```

W przedstawionym fragmencie kodu potokujemy losowo wygenerowane bajty do funkcji szyfrującej. W praktyce będą to dane wejściowe podane przez użytkownika, uchwyt do pliku, gniazdo sieciowe itd.

```
procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_encrypt(data)
    procs.append(proc)
```

Procesy potomne będą działały równolegle z nadrzędnym, a także będą korzystały z danych wejściowych procesów nadrzędnych. W poniższym kodzie czekamy na zakończenie działania procesów potomnych, a następnie pobieramy wygenerowane przez nie ostateczne dane wyjściowe. Zgodnie z oczekiwaniami dane wyjściowe to losowo zaszyfrowane bajty.

```
for proc in procs:
    out, _ = proc.communicate()
    print(out[-10:])
```

```
>>>
b'\x8c(\xed\xc7m1\xf0F4\xe6'
b'\x0eD\x97\xe9>\x10h{\xbd\xf0'
b'g\x93)\x14U\xa9\xdc\xdd\x04\xd2'
```

Można też tworzyć łańcuchy równocześnie działających procesów, podobnie jak potoków w systemie UNIX, używając danych wyjściowych jednego procesu potomnego jako danych wejściowych innego procesu potomnego itd. Poniżej przedstawiłem funkcję uruchamiającą proces potomny, który z kolei spowoduje, że polecenie powłoki pobierze strumień danych wejściowych:

```
def run_hash(input_stdin):
    return subprocess.Popen(
        ['openssl', 'dgst', '-whirlpool', '-binary'],
        stdin=input_stdin,
        stdout=subprocess.PIPE)
```

Teraz wykorzystujemy zbiór procesów `openssl` do szyfrowania pewnych danych, a kolejny zbiór procesów do utworzenia wartości hash na podstawie zaszyfrowanych danych. Zwróć uwagę na konieczność zachowania ostrożności co do tego, jak egzemplarz `stdout` procesu jest obsługiwany przez proces interpretera Pythona, który uruchamia ten potok procesów potomnych.

```
encrypt_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(100)

    encrypt_proc = run_encrypt(data)
    encrypt_procs.append(encrypt_proc)

    hash_proc = run_hash(encrypt_proc.stdout)
    hash_procs.append(hash_proc)
    # Trzeba zagwarantować, że proces potomny używa danych wejściowych,
    # a metoda communicate() przypadkowo nie zabierze tych danych wejściowych
    # procesowi potomnemu, ponadto trzeba pozwolić SIGPIPE na propagowanie
    # procesu upstream, jeśli proces downstream zostanie zakończony

    encrypt_proc.stdout.close()
    encrypt_proc.stdout = None
```

Operacje wejścia-wyjścia między procesami potomnymi będą zachodziły automatycznie po uruchomieniu procesów. Twoim zadaniem jest jedynie zaczekać na zakończenie działania procesów potomnych i wyświetlić ostateczne wyniki ich działania.

```
for proc in encrypt_procs:
    proc.communicate()
    assert proc.returncode == 0

for proc in hash_procs:
    out, _ = proc.communicate()
    print(out[-10:])
    assert proc.returncode == 0
```

```
>>>
b'\xe2j\x98h\xfd\xec\xe7T\xd84'
b'\xf3.i\x01\xd74|\xf2\x94E'
b'5_n\xc3-\xe6j\xeb[i'
```

Jeżeli masz obawy, że procesy potomne nigdy się nie zakończą lub coś będzie blokowało potoki danych wejściowych bądź wyjściowych, to upewnij się, czy metodzie `communicate()` został przekazany parametr `timeout`. Przekazanie tego parametru sprawi, że nastąpi zgłoszenie wyjątku, jeśli proces potomny nie udzieli odpowiedzi w podanym czasie. Tym samym zyskasz możliwość zakończenia działania nieprawidłowo zachowującego się procesu potomnego.

```
proc = subprocess.Popen(['sleep', '10'])
try:
    proc.communicate(timeout=0.1)
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Kod wyjścia', proc.poll())
```

```
>>>
Kod wyjścia -15
```

Do zapamiętania

- Używaj modułu `subprocess` do uruchamiania procesów potomnych oraz zarządzania ich strumieniami danych wejściowych i wyjściowych.
- Procesy potomne działają równolegle wraz z interpreterem Pythona, co pozwala na maksymalne wykorzystanie dostępnego procesora.
- W prostych sytuacjach używaj funkcji wygodnej `run()`, natomiast w bardziej zaawansowanych (np. potoki w stylu systemu UNIX) korzystaj z klasy `Popen`.
- Używaj parametru `timeout` w metodzie `communicate()`, aby unikać zakleszczeń i zawieszenia procesów potomnych.

Sposób 53. Użycie wątków dla operacji blokujących wejście-wyjście, unikanie równoległości

Standardowa implementacja Pythona nosi nazwę CPython. Implementacja ta uruchamia program Pythona w dwóch krokach. Pierwszy to przetworzenie i kompilacja kodu źródłowego na *kod bajtowy*, czyli niskiego poziomu reprezentacja programu mająca postać 8-bitowych instrukcji. (Wprawdzie z technicznego punktu widzenia w Pythonie 3.6 to kod *wordcode* z 16-bitowymi instrukcjami, ale idea pozostała taka sama). Drugi to uruchomienie kodu bajtowego za pomocą interpretera opartego na stosie. Wspomniany interpreter kodu bajtowego ma stan, który musi być obsługiwany i spójny podczas wykonywania programu Pythona. Język Python wymusza spójność za pomocą mechanizmu o nazwie **GIL** (ang. *global interpreter lock*).

W gruncie rzeczy mechanizm GIL to rodzaj wzajemnego wykluczania (mutex) chroniący CPython przed wpływem wywłaszczenia wielowątkowego, gdy jeden wątek przejmuje kontrolę nad programem przez przerwanie działania innego wątku. Takie przerwanie może doprowadzić do uszkodzenia stanu interpretera (np. licznika odwołań mechanizmu usuwania nieużytków), jeśli wystąpi w nieoczekiwanym czasie. Mechanizm GIL chroni przed wspomnianymi przerwaniami i gwarantuje, że każda instrukcja kodu bajtowego działa poprawnie z implementacją CPython oraz jej modułami rozszerzeń utworzonych w języku C.

Mechanizm GIL powoduje pewien ważny negatywny efekt uboczny. W przypadku programów utworzonych w językach takich jak C++ lub Java wiele wątków wykonywania oznacza, że program może jednocześnie wykorzystać wiele rdzeni procesora. Wprawdzie Python obsługuje wiele wątków wykonywania, ale mechanizm GIL powoduje, że w danej chwili tylko jeden z nich robi postęp. Dlatego też jeśli sięgasz po wątki w celu przeprowadzania równoległych obliczeń i przyśpieszenia programów Pythona, to będziesz srodze zawiedziony.

Przyjmujemy założenie, że chcesz w Pythonie wykonać zadanie wymagające dużej ilości obliczeń. Użyjemy algorytmu rozkładu liczby na czynniki.

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
```

Rozkład zbioru liczb może wymagać całkiem dużej ilości czasu.

```
import time

numbers = [2139079, 1214759, 1516637, 1852285]
start = time.time()

for number in numbers:
    list(factorize(number))

end = time.time()
delta = end - start
print(f'Operacja zabrała {delta:.3f} sekund')

>>>
Operacja zabrała 0.399 sekund
```

W innych językach programowania użycie wielu wątków będzie miało sens, ponieważ wówczas wykorzystasz wszystkie rdzenie dostępne w procesorze. Spróbujmy to zrobić w Pythonie. Poniżej zdefiniowałem wątek Pythona przeznaczony do przeprowadzenia tych samych obliczeń co wcześniej:

```
from threading import Thread

class FactorizeThread(Thread):
    def __init__(self, number):
        super().__init__()
        self.number = number

    def run(self):
        self.factors = list(factorize(self.number))
```

Teraz uruchamiam wątki w celu równoległego rozkładu poszczególnych liczb.

```
start = time.time()

threads = []
for number in numbers:
    thread = FactorizeThread(number)
    thread.start()
    threads.append(thread)
```

Pozostało już tylko zaczekać na zakończenie działania wszystkich wątków.

```
for thread in threads:
    thread.join()

end = time.time()
delta = end - start
print(f'Operacja zabrała {delta:.3f} sekund')

>>>
Operacja zabrała 0.446 sekund
```

Zaskakujące może być, że równoległe wykonywanie metody `factorize()` trwało dłużej niż w przypadku jej szeregowego wywoływania. Przeznaczając po jednym wątku dla każdej liczby, w innych językach programowania można oczekiwać przyśpieszenia działania programu nieco mniejszego niż czterokrotne, co wynika z obciążenia związanego z tworzeniem wątków i ich koordynacją. W przypadku komputera wyposażonego w procesor dwurdzeniowy można oczekiwać jedynie około dwukrotnego przyśpieszenia wykonywania programu. Jednak nigdy nie będziesz się spodziewał, że wydajność będzie gorsza, gdy do obliczeń można wykorzystać wiele rdzeni procesora. To demonstruje wpływ mechanizmu GIL (np. konkurowanie blokad i obciążenie związane z obsługą harmonogramu zadań) na programy wykonywane przez standardowy interpreter CPython.

Istnieją różne sposoby pozwalające CPython na wykorzystanie wielu wątków, ale nie działają one ze standardową klasą `Thread` (zobacz sposób 64.) i implementacja tych rozwiązań może wymagać dość dużego wysiłku. Mając świadomość istnienia wspomnianych ograniczeń, możesz się zastanawiać, dlaczego Python w ogóle obsługuje wątki. Mamy ku temu dwa dobre powody.

Pierwszy — wiele wątków daje złudzenie, że program wykonuje jednocześnie wiele zadań. Samodzielna implementacja mechanizmu jednoczesnego wykonywania zadań jest trudna (przykład znajdziesz w sposobie 56.). Dzięki wątkom pozostawiasz Pythonowi obsługę równoległego uruchamiania funkcji. To działa, ponieważ CPython gwarantuje zachowanie równości między uruchomionymi wątkami Pythona, nawet jeśli ze względu na ograniczenie nakładane przez mechanizm GIL w danej chwili tylko jeden z nich robi postęp.

Drugi powód obsługi wątków w Pythonie to blokujące operacje wejścia-wyjścia, które zachodzą, gdy Python wykonuje określonego typu wywołania systemowe. Za pomocą wspomnianych wywołań systemowych programy Pythona proszą system operacyjny komputera o interakcję ze środowiskiem zewnętrznym. Przykłady blokujących operacji wejścia-wyjścia to odczyt i zapis plików, praca z sieciami, komunikacja z urządzeniami takimi jak monitor itd. Wątki pomagają w obsłudze blokujących operacji wejścia-wyjścia przez odizolowanie Twojego programu od czasu, jakiego system operacyjny potrzebuje na udzielenie odpowiedzi na żądania.

Załóżmy, że za pomocą portu szeregowego chcesz wysłać sygnał do zdalnie sterowanego śmigłowca. Jako proxy dla tej czynności wykorzystamy wolne wywołanie systemowe (`select`). Funkcja prosi system operacyjny o blokadę trwającą 0,1 sekundy, a następnie zwraca kontrolę z powrotem do programu. Otrzymujemy więc sytuację podobną, jaka zachodzi podczas użycia synchronicznego portu szeregowego.

```
import select
import socket

def slow_systemcall():
    select.select([socket.socket()], [], [], 0.1)
```

Szeregowe wykonywanie wywołań systemowych powoduje liniowe zwiększanie się ilości czasu niezbędnego do ich wykonania.

```
start = time.time()

for _ in range(5):
    slow_systemcall()

end = time.time()
delta = end - start
print(f'Operacja zabrała {delta:.3f} sekund')

>>>
Operacja zabrała 0.510 sekund
```

Problem polega na tym, że w trakcie wykonywania funkcji `slow_systemcall()` program nie może zrobić żadnego innego postępu. Główny wątek programu został zablokowany przez wywołanie systemowe `select`. Tego rodzaju sytuacja w praktyce jest straszna. Potrzebujesz sposobu pozwalającego na obliczanie kolejnego ruchu śmigłowca podczas wysyłania sygnału, w przeciwnym razie śmigłowiec może się rozbić. Kiedy występuje potrzeba jednoczesnego wykonania blokujących operacji wejścia-wyjścia i pewnych obliczeń, najwyższa pora rozważyć przeniesienie wywołań systemowych do wątków.

W poniższym fragmencie kodu mamy kilka wywołań funkcji `slow_systemcall()` w oddzielnych wątkach. To pozwoli na jednoczesną komunikację z wieloma portami szeregowymi (i śmigłowcami), natomiast wątek główny będzie pozostawiony do wykonywania niezbędnych obliczeń.

```
start = time.time()
threads = []
for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)
```

Po uruchomieniu wątków mamy do wykonania pewną pracę, czyli obliczenie kolejnego ruchu śmigłowca przed oczekiwaniem na zakończenie działania wątków obsługujących wywołania systemowe.

```
def compute_helicopter_location(index):
    # ...

for i in range(5):
    compute_helicopter_location(i)

for thread in threads:
    thread.join()

end = time()
delta = end - start
print(f'Operacja zabrała {delta:.3f} sekund')

>>>
Operacja zabrała 0.108 sekund
```

Całkowita ilość czasu potrzebnego na równoległe wykonanie operacji jest pięciokrotnie mniejsza niż w przypadku szeregowego wykonywania zadań. To pokazuje, że wywołania systemowe są wykonywane równocześnie w wielu wątkach Pythona, nawet pomimo ograniczeń nakładanych przez mechanizm GIL. Wprawdzie mechanizm GIL uniemożliwia równoległe wykonywanie kodu utworzonego przez programistę, ale nie ma wpływu ubocznego na wywołania systemowe. Przedstawione rozwiązanie się sprawdza, ponieważ wątki Pythona zwalniają mechanizm GIL przed wykonaniem wywołań systemowych i ponownie do niego powracają po zakończeniu wywołania systemowego.

Poza wątkami istnieje jeszcze wiele innych sposobów pracy z blokującymi operacjami wejścia-wyjścia, na przykład użycie modułu `asyncio`. Wspomniane rozwiązania alternatywne przynoszą ważne korzyści. Jednak wymagają także dodatkowej pracy w postaci konieczności refaktoryzacji kodu źródłowego, aby go dopasować do innego modelu wykonywania (zobacz sposoby 60. i 62.). Użycie wątków to najprostszy sposób na równoległe wykonywanie blokujących operacji wejścia-wyjścia i jednocześnie wymaga wprowadzania jedynie minimalnych zmian w programie.

Do zapamiętania

- Z powodu działania globalnej blokady interpretera (mechanizm GIL) wątki Pythona nie pozwalają na równoległe uruchamianie kodu bajtowego w wielu rdzeniach procesora.
- Pomimo istnienia mechanizmu GIL wątki Pythona nadal pozostają użyteczne, ponieważ oferują łatwy sposób jednoczesnego wykonywania wielu zadań.
- Używaj wątków Pythona do równoczesnego wykonywania wielu wywołań systemowych. Tym samym będzie można jednocześnie wykonywać blokujące operacje wejścia-wyjścia oraz pewne obliczenia.

Sposób 54. Używaj klasy Lock, aby unikać stanu wyścigu w wątkach

Po dowiedzeniu się o istnieniu mechanizmu GIL (zobacz sposób 53.) wielu nowych programistów Pythona przyjmuje założenie, że można zrezygnować z użycia muteksu w kodzie. Skoro mechanizm GIL uniemożliwia wątkom Pythona ich równoczesne działanie w wielu rdzeniach procesora, więc można wysnuć wniosek, że ta sama blokada musi dotyczyć także struktur danych programu, prawda? Pewne testy przeprowadzone na typach takich jak listy i słowniki mogą nawet pokazać, że przyjęte założenie jest słuszne.

Musisz mieć jednak świadomość, że niekoniecznie tak jest. Mechanizm GIL nie zapewnia ochrony programowi. Wprawdzie w danej chwili jest wykonywany tylko jeden wątek Pythona, ale między dwoma instrukcjami kodu bajtowego w interpreterze Pythona może dojść do niechcianej modyfikacji struktur danych. To jest niebezpieczne, jeśli jednocześnie z wielu wątków próbujesz uzyskać dostęp do tych samych obiektów. Na skutek wspomnianych modyfikacji, struktury danych mogą być uszkodzone w praktycznie każdej chwili, co z kolei doprowadzi do uszkodzenia programu.

Załóżmy, że tworzysz program przeprowadzający równocześnie wiele operacji, takich jak sprawdzanie poziomu światła w pewnej liczbie czujników sieciowych. Jeżeli chcesz określić całkowitą liczbę próbek, jakie miały miejsce w danym czasie, możesz je agregować za pomocą nowej klasy.

```
class Counter:
    def __init__(self):
        self.count = 0

    def increment(self, offset):
        self.count += offset
```

Wyobraź sobie, że każdy czujnik ma własny wątek roboczy, ponieważ odczyt czujnika wymaga blokującej operacji wejścia-wyjścia. Po przeprowadzeniu pomiaru wątek roboczy inkrementuje wartość licznika, cykl jest powtarzany aż do osiągnięcia maksymalnej liczby oczekiwanych operacji odczytu.

```
def worker(sensor_index, how_many, counter):
    for _ in range(how_many):
        # Odczyt danych z czujnika
        # ...
        counter.increment(1)
```

Poniżej przedstawiłem definicję funkcji uruchamiającej wątek roboczy dla poszczególnych czujników oraz oczekującej na zakończenie odczytu przez każdy z nich:

```
from threading import Thread

how_many = 10**5
counter = Counter()

threads = []
for i in range(5):
    thread = Thread(target=worker,
                    args = (i, how_many, counter))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

expected = how_many * 5
found = counter.count
print(f'Oczekiwana liczba próbek {expected}, znaleziona {found}')

>>>
Oczekiwana liczba próbek 500000, znaleziona 246760
```

Jednoczesne uruchomienie pięciu wątków wydaje się proste, a dane wyjściowe powinny być oczywiste. Jednak wynik znacznie odbiega od oczekiwanego! Co się stało? Jak coś tak prostego mogło się nie udać, zwłaszcza że w danej chwili może działać tylko jeden wątek interpretera Pythona?

Interpreter Pythona wymusza zachowanie sprawiedliwości między wykonywanymi wątkami, aby wszystkie otrzymały praktycznie taką samą ilość czasu procesora. Dlatego też Python będzie wstrzymywać działanie bieżącego wątku i wznawiać działanie kolejnego. Problem polega na tym, że dokładnie nie wiesz, kiedy Python wstrzyma działanie Twoich wątków. Wątek może być więc wstrzymany nawet w połowie operacji, która powinna pozostać niepodzielna. Tak się właśnie stało w omawianym przykładzie.

Metoda `increment()` obiektu `Counter` wygląda na prostą i z perspektywy wątku roboczego jest odpowiednikiem następującego polecenia:

```
counter.count += 1
```

Jednak operator += użyty w atrybucie obiektu tak naprawdę nakazuje Pythonowi wykonanie w tle trzech oddzielnych operacji. Powyższe polecenie jest odpowiednikiem trzech poniższych:

```
value = getattr(counter, 'count')
result = value + 1
setattr(counter, 'count', result)
```

Wątki Pythona przeprowadzające inkrementację mogą zostać wstrzymane między dwoma dowolnymi operacjami przedstawionymi powyżej. To będzie problematyczne, jeśli stara wersja `value` zostanie przypisana licznikowi. Oto przykład nieprawidłowej interakcji między dwoma wątkami A i B:

```
# Wykonywanie wątku A
value_a = getattr(counter, 'count')
# Przełączenie kontekstu do wątku B
value_b = getattr(counter, 'count')
result_b = value_b + 1
setattr(counter, 'count', result_b)
# Przełączenie kontekstu z powrotem do wątku A
result_a = value_a + 1
setattr(counter, 'count', result_a)
```

Wątek B przerwał działanie wątku A, zanim całkowicie zakończył on wykonywanie operacji. Wątek B został uruchomiony i w całości wykonał swoje zadanie, po czym nastąpiło wznowienie działania wątku A. Po przełączeniu kontekstu z wątku A do B nastąpiło usunięcie całego postępu w trakcie operacji inkrementacji licznika. Dokładnie to zdarzyło się w przedstawionym powyżej przykładzie obsługi czujników światła.

Aby zapobiec tego rodzaju sytuacji wyścigu do danych oraz innym formom uszkodzenia struktur danych, Python zawiera solidny zestaw narzędzi dostępnych we wbudowanym module `threading`. Najprostsze i najużyteczniejsze z nich to klasa `Lock` zapewniająca obsługę muteksu.

Dzięki zastosowaniu blokady klasa `Counter` może chronić jej wartość bieżącą przed jednoczesnym dostępem z wielu wątków. W danej chwili tylko jeden wątek będzie miał możliwość nałożenia blokady. W poniższym fragmencie kodu użyłem polecenia `with` do nałożenia i zwolnienia blokady. To znacznie ułatwia ustalenie, który kod jest wykonywany w trakcie trwania blokady (więcej informacji szczegółowych na ten temat znajdziesz w sposobie 66.).

```
from threading import Lock

class LockingCounter:
    def __init__(self):
        self.lock = Lock()
        self.count = 0

    def increment(self, offset):
        with self.lock:
            self.count += offset
```

Teraz podobnie jak wcześniej uruchamiam wątki robocze, ale w tym celu używam wywołania `LockingCounter()`.

```
counter = LockingCounter()

for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Oczekiwana liczba prôbek {expected}, znaleziona {found}')

>>>
Oczekiwana liczba prôbek 500000, znaleziona 500000
```

Otrzymany wynik dokładnie pokrywa się z oczekiwanym. Klasa `Lock` pozwoliła na rozwiązanie problemu.

Do zapamiętania

- Choć Python ma mechanizm GIL, nadal pozostajesz odpowiedzialny za unikanie powstawania sytuacji wyścigu do danych między wątkami używanymi przez Twój program.
- Twoje programy mogą uszkodzić stosowane w nich struktury danych, jeśli pozwolisz, aby wiele wątków jednocześnie modyfikowało te same obiekty bez nakładania na nie blokad.
- Klasa `Lock` oferowana przez wbudowany moduł `threading` to standardowa implementacja mutekstu w Pythonie.

Sposób 55. Używaj klasy Queue do koordynacji pracy między wątkami

Programy Pythona równocześnie wykonujące wiele zadań często muszą koordynować tę pracę. Jednym z najużyteczniejszych narzędzi przeznaczonych do koordynacji jednocześnie wykonywanych zadań jest potokowanie funkcji.

Potokowanie działa na zasadzie podobnej do linii montażowej w przedsiębiorstwie. Potoki mają wiele faz w serii wraz z określonymi funkcjami dla poszczególnych faz. Nowe zadania do wykonania są nieustannie umieszczane na początku potoku. Wszystkie funkcje mogą równolegle pracować nad zadaniami w obsługiwanych przez nie fazach. Cała praca przesuwa się do przodu, gdy wszystkie funkcje zakończą swoje zadanie. Cykl trwa aż do wykonania wszystkich faz. Tego rodzaju podejście jest szczególnie dobre w przypadku pracy wymagającej użycia blokujących operacji wejścia-wyjścia lub podprocesów — czyli w przypadku zadań, które mogą być łatwo wykonywane równolegle za pomocą Pythona (zobacz sposób 53.).

Na przykład chcesz zbudować system, który będzie pobierał stały strumień zdjęć z aparatu cyfrowego, zmieniał ich wielkość, a następnie przekazywał zdjęcia do galerii w internecie. Tego rodzaju program można podzielić na trzy fazy potoku. W pierwszej fazie będą pobierane nowe zdjęcia z aparatu. W drugiej fazie pobrane zdjęcia zostaną przetworzone przez funkcję odpowiedzialną za zmianę ich wielkości. Następnie w trzeciej i ostatniej fazie zmodyfikowane zdjęcia będą za pomocą odpowiedniej funkcji przekazane do galerii internetowej.

Wyobraź sobie, że już utworzyłeś funkcje Pythona przeznaczone do wykonywania poszczególnych faz: `download()`, `resize()` i `upload()`. W jaki sposób można przygotować potok, aby praca mogła być prowadzona równocześnie?

```
def download(item):
    ...
def resize(item):
    ...
def upload(item):
    ...
```

Przede wszystkim potrzebny jest sposób umożliwiający przekazywanie pracy między poszczególnymi fazami potoku. Do tego celu można wykorzystać zapewniającą bezpieczeństwo wątków kolejkę producent-konsument. (Zapoznaj się ze sposobem 54., aby zrozumieć wagę bezpieczeństwa wątków w Pythonie. Z kolei w sposobie 71. znajdziesz więcej informacji o klasie `deque`).

```
from collections import deque
from threading import Lock

class MyQueue:
    def __init__(self):
        self.items = deque()
        self.lock = Lock()
```

Producent, czyli w omawianym przykładzie aparat cyfrowy, umieszcza nowe zdjęcia na końcu listy oczekujących elementów.

```
def put(self, item):
    with self.lock:
        self.items.append(item)
```

Konsument, czyli w omawianym przykładzie pierwsza faza potoku przetwarzania, usuwa zdjęcia z początku listy oczekujących elementów.

```
def get(self):
    with self.lock:
        return self.items.popleft()
```

Poniżej poszczególne fazy potoku przedstawiłem jako wątek Pythona, który pobiera pracę z kolejki, takiej jak wcześniej wspomniana, wykonuje odpowiednią funkcję, a następnie uzyskany wynik umieszcza w innej kolejce. Ponadto monitoruje liczbę razy, jakie wątek roboczy został sprawdzony pod kątem nowych danych wejściowych oraz ilość wykonanej pracy.

```
from threading import Thread
import time

class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

Najtrudniejsza część wiąże się z tym, że wątek roboczy musi prawidłowo obsłużyć sytuację, w której kolejka danych wejściowych będzie pusta, ponieważ poprzednia faza jeszcze nie zakończyła swojego zadania. Tym zajmujemy się tam, gdzie następuje zgłoszenie wyjątku `IndexError`. Można to potraktować jako przestój na linii montażowej.

```
def run(self):
    while True:
        self.polled_count += 1
        try:
            item = self.in_queue.get()
        except IndexError:
            time.sleep(0.01)  # Brak zadania do wykonania
        else:
            result = self.func(item)
            self.out_queue.put(result)
            self.work_done += 1
```

Teraz pozostało już połączenie trzech wymienionych faz ze sobą przez utworzenie kolejek przeznaczonych do koordynacji oraz odpowiednich wątków roboczych.

```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
```

```
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]
```

Można uruchomić wątki, a następnie wstrzyknąć pewną ilość pracy do pierwszej fazy potoku. W poniższym fragmencie kodu jako proxy dla rzeczywistych danych wymaganych przez funkcję `download()` wykorzystałem zwykły egzemplarz `object`.

```
for thread in threads:
    thread.start()

for _ in range(1000):
    download_queue.put(object())
```

Pozostało już zaczekać do chwili, gdy wszystkie elementy zostaną przetworzone przez potok i znajdą się w kolejce `done_queue`.

```
while len(done_queue.items) < 1000:
    # Zrób coś użytecznego podczas oczekiwania
    # ...
```

Rozwiązanie działa prawidłowo, ale występuje interesujący efekt uboczny spowodowany przez wątki sprawdzające ich kolejki danych wejściowych pod kątem nowych zadań do wykonania. Najtrudniejsza część podczas przechwytywania wyjątków `IndexError` w metodzie `run()` jest wykonywana bardzo dużą liczbę razy.

```
processed = len(done_queue.items)
polled = sum(t.polled_count for t in threads)
print(f'Prztworzono {processed} elementów po wykonaniu',
      f' {polled} sprawdzeń')
```

```
>>>
Przetworzono 1000 elementów po wykonaniu 3035 sprawdzeń
```

Szybkość działania poszczególnych funkcji roboczych może być różna, a więc wcześniejsza faza może uniemożliwić dokonanie postępu w późniejszych fazach, tym samym korkując potok. To powoduje, że późniejsze fazy są wstrzymane i nieustannie sprawdzają ich kolejki danych wejściowych pod kątem nowych zadań do wykonania. Skutkiem będzie marnowanie przez wątki robocze czasu procesora na wykonywanie nieużytecznych zadań (będą ciągle zgłaszać i przechwytywać wyjątki `IndexError`).

To jednak dopiero początek nieodpowiednich działań podejmowanych przez tę implementację. Występują w niej jeszcze trzy kolejne błędy, których również należy unikać. Po pierwsze, operacja określenia, czy wszystkie dane wejściowe zostały przetworzone, wymaga oczekiwania w kolejce `done_queue`. Po drugie, w klasie `Worker` metoda `run()` będzie wykonywana w nieskończoność w pętli. Nie ma możliwości wskazania wątkowi roboczemu, że czas zakończyć działanie.

Po trzecie (to najpoważniejszy w skutkach z błędów), zatkanie potoku może doprowadzić do awarii programu. Jeżeli w fazie pierwszej nastąpi duży postęp, natomiast w fazie drugiej duże spowolnienie, to kolejka łącząca obie fazy będzie się nieustannie zwiększać. Druga faza po prostu nie będzie w stanie nadążyć za pierwszą z wykonywaniem swojej pracy. Przy wystarczająco dużej ilości czasu i danych wejściowych skutkiem będzie zużycie przez program całej wolnej pamięci, a następnie awaria aplikacji.

Można więc wyciągnąć wniosek, że potoki są złym rozwiązaniem. Trudno samodzielnie zbudować dobrą kolejkę producent-konsument. Dlaczego miałbyś więc nawet próbować podjąć się takiego zadania?

Ratunek w postaci klasy Queue

Klasa `Queue` z wbudowanego modułu `queue` dostarcza całą funkcjonalność, której potrzebujemy do rozwiązania przedstawionych wcześniej problemów.

Klasa `Queue` eliminuje oczekiwanie w wątku roboczym, ponieważ metoda `get()` jest zablokowana aż do chwili udostępnienia nowych danych. Na przykład poniżej przedstawiłem kod uruchamiający wątek, który oczekuje na pojawienie się w kolejce pewnych danych wejściowych.

```
from queue import Queue

my_queue = Queue()

def consumer():
    print('Konsument oczekuje')
    my_queue.get()                # Uruchomienie po metodzie put() przedstawionej poniżej
    print('Konsument zakończył pracę')

thread = Thread(target=consumer)
thread.start()
```

Wprawdzie wątek jest uruchomiony jako pierwszy, ale nie zakończy działania aż do chwili umieszczenia elementu w egzemplarzu `Queue`, gdy metoda `get()` będzie miała jakiekolwiek dane do przekazania.

```
print('Producent umieszcza dane')
my_queue.put(object())            # Uruchomienie przed metodą get() przedstawioną powyżej
print('Producent zakończył pracę')
thread.join()

>>>
Konsument oczekuje
Producent umieszcza dane
Producent zakończył pracę
Konsument zakończył pracę
```

W celu rozwiązania problemu z zatykaniem potoku, klasa `Queue` pozwala na podanie maksymalnej liczby zadań, jakie mogą między dwoma fazami oczekiwać na wykonanie. Bufor ten powoduje wywołanie metody `put()` w celu nałożenia blokady, gdy kolejka jest już zapełniona.

W poniższym fragmencie kodu przedstawiłem definicję wątku oczekującego chwilę przed użyciem kolejki:

```
my_queue = Queue(1)                 # Bufor o wielkości 1

def consumer():
    time.sleep(0.1)                 # Oczekiwanie
    my_queue.get()                  # Drugie wywołanie
    print('Konsument pobiera dane 1')
    my_queue.get()                  # Czwarte wywołanie
    print('Konsument pobiera dane 2')
    print('Konsument zakończył pracę')

thread = Thread(target=consumer)
thread.start()
```

Oczekiwanie powinno pozwolić wątkowi producenta na umieszczenie obu obiektów w kolejce, zanim wątek konsumenta w ogóle wywoła metodę `get()`. Jednak wielkość `Queue` wynosi `1`. To oznacza, że producent dodający elementy do kolejki będzie musiał zaczekać, aż wątek konsumenta przynajmniej raz wywoła metodę `get()`. Dopiero wtedy drugie wywołanie `put()` zwolni blokadę i pozwoli na dodanie drugiego elementu do kolejki.

```
my_queue.put(object())              # Pierwsze wywołanie
print('Producent umieszcza dane 1')
my_queue.put(object())              # Trzecie wywołanie
print('Producent umieszcza dane 2')
print('Producent zakończył pracę')
thread.join()

>>>
Producent umieszcza dane 1
Konsument pobiera dane 1
Producent umieszcza dane 2
Producent zakończył pracę
Konsument pobiera dane 2
Konsument zakończył pracę
```

Klasa `Queue` może również monitorować postęp pracy, używając do tego metody `task_done()`. W ten sposób można zaczekać, aż kolejka danych wejściowych fazy zostanie opróżniona, co eliminuje konieczność sprawdzania kolejki `done_queue` na końcu potoku. Na przykład poniżej zdefiniowałem wątek konsumenta wywołujący metodę `task_done()` po zakończeniu pracy nad elementem.

```
in_queue = Queue()

def consumer():
    print('Konsument oczekuje')
    work = in_queue.get()       # Zakończone jako drugie
    print('Konsument pracuje')
    # Wykonywanie pracy.
    # ...
    print('Konsument zakończył pracę')
```

```
    in_queue.task_done()        # Zakończone jako trzecie

thread = Thread(target=consumer)
thread.start()
```

Teraz kod producenta nie musi łączyć się z wątkiem konsumenta lub sprawdzać go. Producent może po prostu poczekać na zakończenie pracy przez kolejkę `in_queue`, wywołując metodę `join()` w egzemplarzu `Queue`. Nawet jeśli kolejka `in_queue` jest pusta, to nie będzie można się do niej przyłączyć, dopóki nie zostanie wywołana metoda `task_done()` dla każdego elementu, który kiedykolwiek był kolejkowany.

```
print('Producent umieszcza dane')
in_queue.put(object())          # Zakończone jako pierwsze
print('Producent oczekuje')
in_queue.join()                 # Zakończone jako czwarte
print('Producent zakończył pracę')
thread.join()

>>>
Konsument oczekuje
Producent umieszcza dane
Producent oczekuje
Konsument pracuje
Konsument zakończył pracę
Producent zakończył pracę
```

Wszystkie wymienione funkcje można umieścić razem w podklasie klasy `Queue`, która również poinformuje wątek roboczy o konieczności zakończenia przetwarzania. W poniższym fragmencie kodu znajduje się zdefiniowana metoda `close()` dodająca do kolejki *element specjalny*, który wskazuje, że po nim nie powinny znajdować się już żadne elementy danych wejściowych:

```
class ClosableQueue(Queue):
    SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)
```

Następnie definiujemy iterator dla kolejki, który wyszukuje wspomniany element specjalny i zatrzymuje iterację po znalezieniu tego elementu. Metoda iteratora `__iter__()` powoduje również wywołanie metody `task_done()` w odpowiednim momencie, co pozwala na monitorowanie postępu pracy w kolejce (zobacz sposób 31.).

```
def __iter__(self):
    while True:
        item = self.get()
        try:
            if item is self.SENTINEL:
                return  # Powoduje zakończenie działania wątku
            yield item
        finally:
            self.task_done()
```

Teraz można przedefiniować wątek roboczy, aby opierał się na funkcjonalności dostarczanej przez klasę `ClosableQueue`. Wątek zakończy działanie po zakończeniu pętli.

```
class StoppableWorker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue

    def run(self):
        for item in self.in_queue:
            result = self.func(item)
            self.out_queue.put(result)
```

Oto kod odpowiedzialny za utworzenie zbioru wątków roboczych na podstawie nowej klasy:

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()
threads = [
    StoppableWorker(download, download_queue, resize_queue),
    StoppableWorker(resize, resize_queue, upload_queue),
    StoppableWorker(upload, upload_queue, done_queue),
]
```

Po uruchomieniu wątków roboczych sygnał zatrzymania podobnie jak wcześniej jest wysyłany przez zamknięcie kolejki danych wejściowych dla pierwszej fazy po umieszczeniu w niej wszystkich elementów.

```
for thread in threads:
    thread.start()

for _ in range(1000):
    download_queue.put(object())

download_queue.close()
```

Pozostało już tylko oczekiwanie na zakończenie pracy przez połączenie poszczególnych kolejek znajdujących się między fazami. Gdy dana faza zostanie zakończona, to jest to sygnalizowane kolejnej fazie przez zamknięcie jej kolejki danych wejściowych. Na końcu kolejka `done_queue` zawiera zgodnie z oczekiwaniami wszystkie obiekty danych wyjściowych.

```
download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'elementów zostało przetworzonych')

for thread in threads:
    thread.join()
```

```
>>>
1000 elementów zostało przetworzonych
```

To podejście można rozszerzyć w celu wykorzystania wielu wątków roboczych w każdej fazie, co pozwala zwiększyć równoległość operacji wejścia-wyjścia, a także znacznie przyśpieszyć działanie takiego programu. Jeżeli zdecydujesz się na takie rozwiązanie, pracę musisz zacząć od zdefiniowania funkcji pomocniczych odpowiedzialnych za uruchamianie i zatrzymywanie wielu wątków. Funkcja `stop_threads()` działa przez wywołanie `close()` dla każdej kolejki danych wejściowych, jednokrotnie dla każdego wątku konsumenta. Dzięki temu mamy pewność, że wszystkie wątki robocze bezbłędnie i czysto zakończą pracę.

```
def start_threads(count, *args):
    threads = [StoppableWorker(*args) for _ in range(count)]
    for thread in threads:
        thread.start()
    return threads

def stop_threads(closable_queue, threads):
    for _ in threads:
        closable_queue.close()

    closable_queue.join()

    for thread in threads:
        thread.join()
```

Teraz, podobnie jak wcześniej, wszystkie elementy układanki można połączyć: obiekty przeznaczone do przetworzenia umieścić na początku potoku, dołączać kolejki i wątki podczas pracy, a na końcu użyć wygenerowanych danych.

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()

download_threads = start_threads(
    3, download, download_queue, resize_queue)
resize_threads = start_threads(
    4, resize, resize_queue, upload_queue)
upload_threads = start_threads(
    5, upload, upload_queue, done_queue)

for _ in range(1000):
    download_queue.put(object())

stop_threads(download_queue, download_threads)
stop_threads(resize_queue, resize_threads)
stop_threads(upload_queue, upload_threads)

print(done_queue.qsize(), 'elementów zostało przetworzonych')

>>>
1000 elementów zostało przetworzonych
```

Wprawdzie w omawianym przykładzie potoku liniowego egzemplarz `Queue` sprawdza się doskonale, ale w wielu innych sytuacjach istnieją znacznie lepsze narzędzia, których użycie warto rozważyć (zobacz sposób 60.).

Do zapamiętania

- Potoki to doskonały sposób organizowania sekwencji zadań (zwłaszcza w programach wykonujących ogromną ilość operacji wejścia-wyjścia) jednocześnie wykonywanych przez wiele wątków Pythona.
- Musisz być świadom, że podczas tworzenia potoków, które jednocześnie wykonują wiele zadań, pojawiają się problemy: oczekiwanie blokujące dostęp, zatrzymywanie wątków roboczych i niebezpieczeństwo zużycia całej dostępnej pamięci.
- Klasa `Queue` oferuje całą funkcjonalność, jakiej potrzebujesz do przygotowania niezawodnych potoków: obsługę blokad, bufory o wskazanej wielkości i dołączanie do kolejek.

Sposób 56. Naucz się rozpoznawać, kiedy współbieżność jest niezbędna

Nieuniknione jest, że wraz ze zwiększaniem się zakresu programu staje się on znacznie bardziej skomplikowany. Poradzenie sobie z rozszerzeniem wymagań programu w sposób zapewniający zachowanie przejrzystości, możliwości przeprowadzania testów oraz efektywności działania to jeden z najtrudniejszych aspektów programowania. Prawdopodobnie jedną z najtrudniejszych do przeprowadzenia zmian jest przejście z programu jednowątkowego do wymagającego wielu współbieżnych ścieżek wykonywania.

Pokażę Ci na przykładzie, jakie problemy możesz napotkać. Załóżmy, że zadanie polega na implementacji gry w życie, czyli klasycznego przykładu maszyny stanów skończonych. Reguły takiej gry są bardzo proste: masz dwuwymiarową siatkę o dowolnej wielkości, a każda komórka na tej siatce może być żywa lub pusta:

```
ALIVE = '*'
EMPTY = '-'
```

Postęp w grze jest ściśle związany z tyknięciem zegara. W trakcie każdego tyknięcia każda komórka sprawdza, ile z jej ośmiu sąsiadujących komórek nadal pozostaje przy życiu. Następnie na podstawie tej liczby podejmowana jest decyzja związana z komórką: pozostawienie jej przy życiu, uśmiercenie lub regeneracja. (Dokładniejsze wyjaśnienie reguł przedstawię w dalszej części rozdziału). Spójrz na przykładową siatkę do gry w życie o wielkości 5×5 pól po przeprowadzeniu czterech ruchów; czas jest przedstawiony w kolejnych kolumnach.

```
  0   |   1   |   2   |   3   |   4
 ----- | ----- | ----- | ----- | -----
-*--- | --*-- | --**- | --*-- | -----
--**- | --**- | -*--- | -*--- | -**--
---*- | --**- | --**- | --*-- | -----
----- | ----- | ----- | ----- | -----
```

Do przedstawienia stanu poszczególnych komórek można wykorzystać prostą klasę kontenera. Musi ona mieć zdefiniowane metody umożliwiające pobieranie i przypisywanie wartości w punkcie o dowolnych współrzędnych. W przypadku podania współrzędnych wykraczających poza granice powinny być wybrane najbliższe współrzędne siatki, co powoduje, że siatka działa podobnie jak przestrzeń nieskończona.

```
class Grid:
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)

    def get(self, y, x):
        return self.rows[y % self.height][x % self.width]

    def set(self, y, x, state):
        self.rows[y % self.height][x % self.width] = state

    def __str__(self):
        ...
```

Aby pokazać tę klasę w akcji, mogę utworzyć egzemplarz typu `Grid` i zdefiniować jego stan początkowy w postaci kształtu klasycznego:

```
grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)
print(grid)

>>>
---*-----
----*----
--***----
---------
---------
```

Następnym krokiem jest pobranie informacji o stanie komórek sąsiednich. Do tego celu można posłużyć się funkcją pomocniczą, która wykonuje zapytanie do siatki i zwraca liczbę żywych komórek sąsiednich. Zdecydowałem się na użycie prostej funkcji dla parametru `get` zamiast przekazywać cały egzemplarz `Grid`, ponieważ takie podejście zmniejsza poziom powiązania (zobacz sposób 38.).

```
def count_neighbors(y, x, get):
    n_ = get(y - 1, x + 0)  # Północ
    ne = get(y - 1, x + 1)  # Północny wschód
    e_ = get(y + 0, x + 1)  # Wschód
    se = get(y + 1, x + 1)  # Południowy wschód
    s_ = get(y + 1, x + 0)  # Południe
```

```
    sw = get(y + 1, x - 1)  # Południowy zachód
    w_ = get(y + 0, x - 1)  # Zachód
    nw = get(y - 1, x - 1)  # Północny zachód
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0

    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count
```

Kolejnym krokiem jest zdefiniowanie prostej logiki gry w życie na podstawie trzech reguł tej gry: komórka umiera, gdy ma mniej niż dwóch sąsiadów, umiera, gdy ma więcej niż trzech sąsiadów, ożywa, gdy ma dokładnie trzech sąsiadów.

```
def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2:
            return EMPTY     # Śmierć: za mało sąsiadów
        elif neighbors > 3:
            return EMPTY     # Śmierć: za dużo sąsiadów
    else:
        if neighbors == 3:
            return ALIVE     # Regeneracja
    return state
```

Funkcje `count_neighbors()` i `game_logic()` można połączyć w innej funkcji odpowiedzialnej za zmianę stanu komórki. Ta nowa funkcja będzie wywoływana podczas każdej generacji w celu ustalenia stanu bieżącej komórki, sprawdzenia komórek sąsiadujących z nią, ustalenia następnego stanu bieżącej komórki i odpowiedniego uaktualnienia siatki. Także tym razem zdecydowałem się wykorzystać interfejs funkcji typu `set` zamiast przekazywania egzemplarza `Grid`, ponieważ dzięki temu mamy niższy poziom powiązania kodu.

```
def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)
```

Teraz można zdefiniować funkcję odpowiedzialną za posunięcie do przodu całej siatki komórek i zwrot nowej siatki zawierającej stan dla następnej generacji. W tym miejscu trzeba wspomnieć o bardzo ważnym szczególe: wszystkie funkcje zależne muszą wywoływać metodę `get()` w egzemplarzu `Grid` reprezentującym poprzednią generację i metodę `set()` w egzemplarzu `Grid` reprezentującym następną generację. Dzięki temu mamy pewność, że stan wszystkich komórek zostanie zmieniony w jednym kroku, co ma duże znaczenie z punktu widzenia sposobu prowadzenia gry. Ten efekt można uzyskać dość łatwo, ponieważ w kodzie zostały użyte interfejsy funkcji `get()` i `set()` zamiast przekazywania egzemplarzy `Grid`.

```
def simulate(grid):
    next_grid = Grid(grid.height, grid.width)

    for y in range(grid.height):
```

```
        for x in range(grid.width):
            step_cell(y, x, grid.get, next_grid.set)
    return next_grid
```

Teraz grę można posuwać do przodu jednorazowo o jedną generację. Można zauważyć, jak kształt się zmienia, przesuwając się do dołu i w prawą stronę siatki na podstawie prostych reguł zdefiniowanych w funkcji `game_logic()`.

```
class ColumnPrinter:
    ...

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate(grid)

print(columns)
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

Takie rozwiązanie sprawdza się świetnie w przypadku programu działającego w jednym wątku na pojedynczym komputerze. Jednak wyobraź sobie zmianę wymagań programu — do czego już nawiązywałem wcześniej — który teraz musi wykonywać operacje wejścia-wyjścia (np. poprzez gniazdo) z poziomu funkcji `game_logic()`. Przykładowo takie rozwiązanie może okazać się konieczne, jeśli próbujesz zbudować ogromną, wieloosobową i prowadzoną online grę, w której zmiany stanu odbywają się na podstawie połączenia informacji o stanie siatki oraz komunikacji z innymi graczami w internecie.

Jak można rozszerzyć tę implementację o obsługę wspomnianej funkcjonalności? Najprostsze rozwiązanie polega na dodaniu blokujących operacji wejścia-wyjścia bezpośrednio w funkcji `game_logic()`:

```
def game_logic(state, neighbors):
    ...
    # Miejsce na przeprowadzenie blokujących operacji wejścia-wyjścia
    data = my_socket.recv(100)
    ...
```

Problem z takim podejściem polega na tym, że spowolni ono działanie całego programu. Jeżeli wymagane opóźnienie operacji wejścia-wyjścia wynosi 100 milisekund (jest to rozsądna wartość w przypadku prowadzonej w internecie komunikacji między użytkownikami znajdującymi się w różnych krajach), a na siatce znajduje się 45 komórek, wówczas przeprowadzenie obliczeń każdej generacji zajmie przynajmniej 4,5 sekundy, ponieważ komórki są przetwarzane pojedynczo przez funkcję `simulate()`. To jest stanowczo zbyt wolno i gra będzie praktycznie niemożliwa. Takie rozwiązanie również kiepsko się skaluje — jeżeli później

zdecydujesz się rozszerzyć siatkę do 10 000 komórek, to obliczenia związane z każdą generacją będą wymagały ponad 15 minut.

Rozwiązaniem jest równoległe przeprowadzanie operacji wejścia-wyjścia, aby obliczenia dla każdej generacji zajmowały około 100 milisekund niezależnie od wielkości siatki. Proces tworzenia współbieżnej ścieżki wykonywania dla kolejnej jednostki pracy (w omawianym przykładzie jest to komórka) nosi nazwę *fan-out*. Oczekiwanie na zakończenie działania przez wszystkie współbieżne jednostki pracy przed przejściem do następnej fazy w skoordynowanym procesie (w omawianym przykładzie jest to generacja) nosi nazwę *fan-in*.

Python oferuje wiele wbudowanych narzędzi przeznaczonych do obsługi *fan-out* i *fan-in*, choć wiążą się one z różnymi kompromisami. Należy poznać wady i zalety poszczególnych narzędzi i wybrać najlepsze do wykonania zadania w zależności od sytuacji. Szczegóły znajdziesz w kolejnych sposobach, które zostały oparte na przykładzie gry w życie (zobacz sposoby od 57. do 60.).

Do zapamiętania

- Program bardzo często rozrasta się, a wraz ze wzrostem zakresu i stopnia skomplikowania zaczyna wymagać wielu współbieżnych ścieżek wykonywania.
- Najczęściej spotykane typy współbieżności to *fan-out* (generowanie nowych jednostek współbieżności) i *fan-in* (oczekiwanie na zakończenie działania przez istniejące jednostki współbieżności).
- Python oferuje wiele różnych sposobów wykorzystania fan-out i fan-in.

Sposób 57. Unikaj tworzenia nowych egzemplarzy Thread na żądanie fan-out

Wątki to naturalne pierwsze narzędzie do wykorzystania w celu równoległego przeprowadzania operacji wejścia-wyjścia w Pythonie (zobacz sposób 53.). Jednak mają poważne wady, gdy będziesz ich używać w przypadku wielu współrzędnych ścieżek wykonywania.

Aby to zaprezentować, będę kontynuował wcześniejszy przykład gry w życie (w sposobie 56. znajdziesz więcej informacji na temat tej gry oraz implementacje używanych w niej różnych funkcji i klas). Wątki zostaną wykorzystane do rozwiązania problemu związanego z opóźnieniem spowodowanym przez wykonywanie wielu operacji wejścia-wyjścia w funkcji `game_logic()`. Trzeba zacząć od tego, że wątki wymagają koordynacji za pomocą blokad, aby w ten sposób zapewnić prawidłową obsługę struktur danych. Istnieje możliwość utworzenia podklasy klasy `Grid` zapewniającej obsługę blokad, co pozwoli na jednoczesne używanie egzemplarza przez wiele wątków.

```
from threading import Lock

ALIVE = '*'
EMPTY = '-'
```

```
class Grid:
    ...

class LockingGrid(Grid):
    def __init__(self, height, width):
        super().__init__(height, width)
        self.lock = Lock()

    def __str__(self):
        with self.lock:
            return super().__str__()

    def get(self, y, x):
        with self.lock:
            return super().get(y, x)

    def set(self, y, x, state):
        with self.lock:
            return super().set(y, x, state)
```

Następnym krokiem jest ponowna implementacja funkcji `simulate()` z wykorzystaniem typu współbieżności *fan-out* przez utworzenie wątku dla każdego wywołania funkcji `step_cell()`. Wątki będą działały równolegle bez konieczności oczekiwania na zakończenie operacji wejścia-wyjścia w innych wątkach. Następnie przed przejściem do kolejnej generacji można zastosować typ współbieżności *fan-in* przez oczekiwanie na zakończenie działania wszystkich wątków.

```
from threading import Thread

def count_neighbors(y, x, get):
    ...

def game_logic(state, neighbors):
    ...
    # Miejsce na przeprowadzenie blokujących operacji wejścia-wyjścia
    data = my_socket.recv(100)
    ...

def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)

def simulate_threaded(grid):
    next_grid = LockingGrid(grid.height, grid.width)

    threads = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            thread = Thread(target=step_cell, args=args)
            thread.start()  # Typ współbieżności fan-out
```

```
            threads.append(thread)

    for thread in threads:
        thread.join()       # Typ współbieżności fan-in

    return next_grid
```

Ten kod można uruchomić za pomocą tej samej implementacji `step_cell()` i tego samego kodu co wcześniej, po zmianie zaledwie dwóch wierszy, aby zostały użyte implementacje klasy `LockingGrid` i funkcji `simulate_threaded()`:

```
class ColumnPrinter:
    ...

grid = LockingGrid(5, 9)            # Zmiana
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_threaded(grid)  # Zmiana

print(columns)
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

To rozwiązanie działa zgodnie z oczekiwaniami, a operacje wejścia-wyjścia są teraz prowadzone równolegle między wątkami. Jednak mamy trzy poważne problemy związane z tym kodem:

- Egzemplarze `Thread` wymagają narzędzi specjalnych do wzajemnej koordynacji i zapewnienia bezpieczeństwa (zobacz sposób 54.). To powoduje, że znacznie trudniej uzasadnić tworzenie kodu używającego wątków niż kodu proceduralnego wykorzystującego tylko jeden wątek. Większy poziom skomplikowania bardzo utrudnia wraz z upływem czasu rozszerzenie kodu i jego konserwację.
- Wątki wymagają dużo pamięci — mniej więcej 8 MB dla każdego działającego wątku. W wielu komputerach ta ilość nie stanowi problemu na przykład po utworzeniu 45 wątków. Gdy siatka gry osiągnie 10 000 komórek, wystąpi konieczność utworzenia 10 000 wątków, które nie zmieszczą się w pamięci wielu komputerów, w tym także mojego. Uruchamianie wątku dla każdego współbieżnie wykonywanego zadania nie wchodzi w grę.

- Uruchomienie wątku jest operacją kosztową, same wątki zaś mają niekorzystny wpływ na wydajność działania programu, co wynika z konieczności przełączania kontekstu między wątkami. W omawianym przykładzie wszystkie wątki są uruchamiane i zatrzymywane podczas przeprowadzania obliczeń dla każdej generacji gry. To wiąże się z ogromnym obciążeniem i znacząco wydłuża opóźnienie poza oczekiwane 100 milisekund.

W przypadku wystąpienia jakichkolwiek problemów ten kod będzie również bardzo trudny do debugowania. Przykładowo wyobraź sobie, że funkcja `game_logic()` powoduje zgłoszenie wyjątku, co jest bardzo prawdopodobne ze względu na naturę operacji wejścia-wyjścia:

```
def game_logic(state, neighbors):
    ...
    raise OSError('Problem z operacjami wejścia-wyjścia')
    ...
```

Można sprawdzić zachowanie kodu po zgłoszeniu wyjątku — wystarczy utworzyć egzemplarz `Thread` wskazujący tę funkcję i przekierować dane wyjściowe `sys.stderr` programu do znajdującego się w pamięci bufora `StringIO`:

```
import contextlib
import io

fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    thread = Thread(target=game_logic, args=(ALIVE, 3))
    thread.start()
    thread.join()

print(fake_stderr.getvalue())
```

```
>>>
Exception in thread Thread-226:
Traceback (most recent call last):
  File "threading.py", line 917, in _bootstrap_inner
    self.run()
  File "threading.py", line 865, in run
    self._target(*self._args, **self._kwargs)
  File "example.py", line 193, in game_logic
    raise OSError('Problem z operacjami wejścia-wyjścia')
OSError: Problem z operacjami wejścia-wyjścia
```

Wyjątek `OSError` zostaje zgłoszony zgodnie z oczekiwaniami, natomiast w niewiadomy sposób kod tworzący egzemplarz `Thread` i wywołujący metodę `join()` pozostaje nietknięty. Jak to możliwe? Odpowiedź jest prosta: klasa `Thread` niezależnie przechwytuje wyjątki zgłaszane przez funkcję docelową, a następnie stos wywołań przekazuje do `sys.stderr`. Takie wyjątki nie są nigdy ponownie zgłaszane komponentowi wywołującemu, który uruchomił dany wątek.

Biorąc pod uwagę wszystkie wymienione dotąd problemy, nie powinno ulegać wątpliwości, że wątki nie są odpowiednim rozwiązaniem w sytuacji, w której trzeba nieustannie tworzyć i zamykać nowe współbieżnie działające funkcje. Python oferuje inne rozwiązania, które lepiej sprawdzą się w takich przypadkach (zobacz sposoby od 58. do 60.).

Do zapamiętania

- Wątki mają wiele wad: ich uruchomienie i działanie jest kosztowne, gdy potrzebujesz wielu wątków; każdy wątek wymaga znacznej ilości pamięci; wątki wymagają też koordynacji za pomocą narzędzi specjalnych, takich jak egzemplarze `Lock`.
- Wątki nie dostarczają wbudowanego mechanizmu do zgłaszania wyjątków w kodzie, który uruchomił dany wątek, lub w kodzie oczekującym na zakończenie działania danego wątku. To oznacza, że debugowanie kodu stosującego wątki jest niezwykle trudne.

Sposób 58. Pamiętaj, że stosowanie Queue do obsługi współbieżności wymaga refaktoringu

W poprzednim sposobie pokazałem wady stosowania egzemplarzy `Thread` do rozwiązywania problemów z równoczesnymi operacjami wejścia-wyjścia we wcześniejszym przykładzie gry w życie (w sposobie 56. znajdziesz więcej informacji na temat tej gry oraz implementacje używanych w niej różnych funkcji i klas).

Następne podejście do wypróbowania polega na implementacji wątkowanego potoku utworzonego za pomocą klasy `Queue` z modułu wbudowanego `queue` (zobacz sposób 55. — w kodzie opieram się na implementacjach `ClosableQueue` i `StoppableWorker` pochodzących z tego sposobu).

Oto ogólne rozwiązanie: zamiast tworzyć po jednym wątku dla komórki dla każdej generacji gry w życie, wcześniej można utworzyć stałą liczbę wątków roboczych, które później będą równolegle wykonywać operacje wejścia-wyjścia wedle potrzeb. Dzięki temu poziom użycia zasobów pozostanie pod kontrolą oraz nastąpi wyeliminowanie obciążenia wynikającego z częstego uruchamiania nowych wątków.

Rozwiązanie wymaga dwóch egzemplarzy `ClosableQueue` przeznaczonych do dwukierunkowej komunikacji z wątkami roboczymi, które wykonują funkcję `game_logic()`:

```
from queue import Queue

class ClosableQueue(Queue):
    ...
in_queue = ClosableQueue()
out_queue = ClosableQueue()
```

Istnieje możliwość uruchomienia wielu wątków wykorzystujących elementy pochodzące z `in_queue`, przetwarzających je za pomocą wywołania `game_logic()` i umieszczających wynik działania w `out_queue`. Te wątki będą działały współbieżnie, pozwalając na równoległe operacje wejścia-wyjścia i zmniejszenie opóźnienia dla każdej generacji.

```
from threading import Thread

class StoppableWorker(Thread):
    ...
```

```
def game_logic(state, neighbors):
    ...
    # Miejsce na przeprowadzenie blokujących operacji wejścia-wyjścia
    data = my_socket.recv(100)
    ...

def game_logic_thread(item):
    y, x, state, neighbors = item
    try:
        next_state = game_logic(state, neighbors)
    except Exception as e:
        next_state = e
    return (y, x, next_state)

# Uruchomienie wątków na początku
threads = []
for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, in_queue, out_queue)
    thread.start()
    threads.append(thread)
```

Teraz można ponownie zdefiniować funkcję `simulate()`, aby współdziałała z tym kolejkami podczas podejmowania decyzji o zmianie stanu i otrzymywała właściwe odpowiedzi. Dodawanie elementów do `in_queue` oznacza stosowanie trybu współbieżności *fan-out*, natomiast używanie elementów z `out_queue`, dopóki kolejka nie zostanie opróżniona, oznacza stosowanie trybu współbieżności *fan-in*.

```
ALIVE = '*'
EMPTY = '-'

class SimulationError(Exception):
    pass

class Grid:
    ...

def count_neighbors(y, x, get):
    ...

def simulate_pipeline(grid, in_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            neighbors = count_neighbors(y, x, grid.get)
            in_queue.put((y, x, state, neighbors))  # Typ współbieżności fan-out

    in_queue.join()
    out_queue.close()
    next_grid = Grid(grid.height, grid.width)
    for item in out_queue:                          # Typ współbieżności fan-in
        y, x, next_state = item
        if isinstance(next_state, Exception):
```

```
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)

    return next_grid
```

Wywołania `Grid.get()` i `Grid.set()` odbywają się z poziomu nowej funkcji `simulate_pipeline()`, co oznacza możliwość wykorzystania jednowątkowej implementacji `Grid` zamiast implementacji wymagającej egzemplarzy `Lock` do obsługi synchronizacji.

Ten kod jest również łatwiejszy do debugowania niż w przypadku opartego na egzemplarzu `Thread` podejścia zastosowanego w poprzednim sposobie. Jeżeli podczas przeprowadzania operacji wejścia-wyjścia w funkcji `game_logic()` zostanie zgłoszony wyjątek, zostanie on przechwycony, rozpropagowany do `out_queue`, a następnie ponownie zgłoszony w wątku głównym.

```
def game_logic(state, neighbors):
    ...
    raise OSError('Problem z operacjami wejścia-wyjścia w funkcji game_logic')
    ...

simulate_pipeline(Grid(1, 1), in_queue, out_queue)

>>>
Traceback ...
OSError: Problem z operacjami wejścia-wyjścia w funkcji game_logic

The above exception was the direct cause of the following
  exception:

Traceback ...
SimulationError: (0, 0)
```

Wielowątkowy potok dla powtarzanych generacji może być obsługiwany za pomocą wywołania `simulate_pipeline()` w pętli:

```
class ColumnPrinter:
    ...

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_pipeline(grid, in_queue, out_queue)

print(columns)

for thread in threads:
    in_queue.close()
```

```
for thread in threads:
    thread.join()

>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --------- | --*-*---- | --------- | ----*----
--***---- | --------- | ---**---- | --------- | --*-*----
--------- | --------- | ---*----- | --------- | ---**----
--------- | --------- | --------- | --------- | ---------
```

Wyniki są takie same jak wcześniej. Wprawdzie rozwiązałem problem dotyczący poziomu użycia pamięci, kosztu uruchamiania nowych wątków i trudności podczas debugowania kodu używającego wątków, ale wiele innych problemów pozostało:

- Funkcja `simulate_pipeline()` jest jeszcze trudniejsza do zrozumienia niż wykorzystana w poprzednim sposobie `simulate_threaded()`.
- Dodatkowe klasy pomocnicze są wymagane przez `ClosableQueue` i `StoppableWorker`, aby ułatwić zrozumienie sposobu działania kodu, choć będzie się to odbywać kosztem większego skomplikowania kodu.
- Konieczne jest wcześniejsze określenie wielkości potencjalnej równoległości — czyli liczby wątków wykonujących funkcję `game_logic_thread()` — na podstawie oczekiwanego obciążenia zamiast pozwolić systemowi na automatyczne skalowanie równoległości wedle potrzeb.
- W celu umożliwienia debugowania konieczne jest ręczne przechwytywanie wyjątków w wątkach roboczych, propagowanie ich w egzemplarzu `Queue`, a następnie ponowne zgłaszanie w wątku głównym.

Jednak największy problem związanym z tym kodem staje się widoczny po ponownej zmianie wymagań. Wyobraź sobie, że później trzeba przeprowadzać operacje wejścia-wyjścia także w funkcji `count_neighbors()`, a nie tylko w `game_logic()`.

```
def count_neighbors(y, x, get):
    ...
    # Miejsce na przeprowadzenie blokujących operacji wejścia- wyjścia.
    data = my_socket.recv(100)
    ...
```

Aby to rozwiązanie działało w sposób równoległy, konieczne jest dodanie kolejnego etapu do potoku zapewniającego wykonanie funkcji `count_neighbors()` w wątku. Trzeba się upewnić o prawidłowym propagowaniu wyjątków między wątkami roboczymi i wątkiem głównym. Konieczne będzie użycie egzemplarzy klasy `Lock` wraz z egzemplarzami `Grid` w celu zapewnienia bezpiecznej synchronizacji między wątkami roboczymi (zobacz sposób 54. oraz implementację `LockingGrid` w sposobie 57.).

```
def count_neighbors_thread(item):
    y, x, state, get = item
    try:
        neighbors = count_neighbors(y, x, get)
    except Exception as e:
        neighbors = e
```

```
    return (y, x, state, neighbors)

def game_logic_thread(item):
    y, x, state, neighbors = item
    if isinstance(neighbors, Exception):
        next_state = neighbors
    else:
        try:
            next_state = game_logic(state, neighbors)
        except Exception as e:
            next_state = e
    return (y, x, next_state)

class LockingGrid(Grid):
    ...
```

Trzeba utworzyć następny zbiór egzemplarzy typu `Queue` dla wątków roboczych w funkcji `count_neighbors_thread()` i odpowiadających im egzemplarzy `Thread`:

```
in_queue = ClosableQueue()
logic_queue = ClosableQueue()
out_queue = ClosableQueue()

threads = []
for _ in range(5):
    thread = StoppableWorker(
        count_neighbors_thread, in_queue, logic_queue)
    thread.start()
    threads.append(thread)

for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, logic_queue, out_queue)
    thread.start()
    threads.append(thread)
```

Ponadto należy uaktualnić funkcję `simulate_pipeline()` w celu zapewnienia koordynacji wielu faz w potoku oraz prawidłowego działania trybów współbieżności *fan-out* i *fan-in*:

```
def simulate_phased_pipeline(
        grid, in_queue, logic_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            item = (y, x, state, grid.get)
            in_queue.put(item)          # Typ współbieżności fan-out

    in_queue.join()
    logic_queue.join()                  # Sekwencja potoku
    out_queue.close()

    next_grid = LockingGrid(grid.height, grid.width)
    for item in out_queue:              # Typ współbieżności fan-in
        y, x, next_state = item
        if isinstance(next_state, Exception):
```

```
                raise SimulationError(y, x) from next_state
            next_grid.set(y, x, next_state)

    return next_grid
```

Po przygotowaniu uaktualnionych implementacji można uruchomić wieloetapowy potok od początku do końca:

```
grid = LockingGrid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)
columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_phased_pipeline(
        grid, in_queue, logic_queue, out_queue)

print(columns)

for thread in threads:
    in_queue.close()
for thread in threads:
    logic_queue.close()
for thread in threads:
    thread.join()
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

To rozwiązanie działa zgodnie z oczekiwaniami, aczkolwiek wymagało wielu zmian i dużej ilości kodu. Chciałem w tym miejscu pokazać, że egzemplarz `Queue` pozwala rozwiązać problemy z trybami współbieżności *fan-out* i *fan-in*, choć jednocześnie powoduje bardzo duże obciążenie. Wprawdzie wykorzystanie `Queue` to lepsze podejście niż używanie egzemplarzy `Thread`, ale wciąż nie tak dobre jak zastosowanie innych narzędzi Pythona (zobacz sposoby 59. i 60.).

Do zapamiętania

- Stosowanie egzemplarzy `Queue` wraz z na stałe określoną liczbą wątków roboczych poprawia skalowalność trybów współbieżności *fan-out* i *fan-in* używających wątków.
- Refaktoryzacja istniejącego kodu w celu przystosowania go do pracy z egzemplarzami `Queue` wymaga dużo pracy, zwłaszcza gdy wymaganych jest wiele etapów potoku.
- Stosowanie egzemplarzy `Queue` znacznie ogranicza całkowitą ilość równoległych operacji wejścia-wyjścia możliwych do obsługi w programie w porównaniu do rozwiązań alternatywnych oferowanych przez inne wbudowane funkcje i moduły Pythona.

Sposób 59. Rozważ użycie klasy ThreadPoolExecutor, gdy wątki są potrzebne do zapewnienia współbieżności

Python zawiera moduł wbudowany `concurrent.features` dostarczający m.in. klasę `Thread ↪PoolExecutor`. Stanowi ona połączenie najlepszych cech klas `Thread` (zobacz sposób 57.) i `Queue` (zobacz sposób 58.) pozwalających rozwiązać problem z równoległością operacji wejścia-wyjścia w przykładzie gry w życie (w sposobie 56. znajdziesz więcej informacji na temat tej gry oraz implementacje używanych w niej różnych funkcji i klas).

```
ALIVE = '*'
EMPTY = '-'

class Grid:
    ...

class LockingGrid(Grid):
    ...

def count_neighbors(y, x, get):
    ...

def game_logic(state, neighbors):
    ...
    # Miejsce na przeprowadzenie blokujących operacji wejścia-wyjścia
    data = my_socket.recv(100)
    ...

def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)
```

Zamiast uruchamiać nowy egzemplarz `Thread` dla każdego kwadratu `Grid`, można wykorzystać tryb współbieżności *fan-out* poprzez przekazanie funkcji do egzekutora, który następnie uruchomi ją w oddzielnym wątku. Później można zaczekać na wynik wykonania wszystkich zadań, aby w ten sposób zastosować tryb współbieżności *fan-in*.

```
from concurrent.futures import ThreadPoolExecutor

def simulate_pool(pool, grid):
    next_grid = LockingGrid(grid.height, grid.width)
    futures = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            future = pool.submit(step_cell, *args)  # Typ współbieżności fan-out
            futures.append(future)
```

```
    for future in futures:
        future.result()                          # Typ współbieżności fan-in

    return next_grid
```

Wątki używane przez egzekutor mogą być alokowane wcześniej, co oznacza wyeliminowanie kosztu związanego z uruchamianiem wątku w trakcie każdego wywołania funkcji `simulate_pool()`. Istnieje również możliwość określenia — za pomocą parametru `max_workers` — maksymalnej liczby wątków przeznaczonych do użycia w puli. W ten sposób nie dopuszczamy do powstania związanych z wykorzystaniem całej dostępnej pamięci problemów, które mogą się pojawiać podczas stosowania opartego na egzemplarzach `Thread` rozwiązania w zakresie obsługi równoległych operacji wejścia-wyjścia.

```
class ColumnPrinter:
    ...

grid = LockingGrid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
with ThreadPoolExecutor(max_workers=10) as pool:
    for i in range(5):
        columns.append(str(grid))
        grid = simulate_pool(pool, grid)

print(columns)
```

```
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

Najlepszą cechą klasy `ThreadPoolExecutor` jest automatyczne propagowanie wyjątków do komponentu wywołującego, gdy metoda `result()` jest wywoływana w egzemplarzu `Future` zwróconym przez metodę `submit()`:

```
def game_logic(state, neighbors):
    ...
    raise OSError('Problem z operacjami wejścia-wyjścia')
    ...

with ThreadPoolExecutor(max_workers=10) as pool:
    task = pool.submit(game_logic, ALIVE, 3)
    task.result()
```

```
>>>
Traceback ...
OSError: Problem z operacjami wejścia-wyjścia
```

Jeżeli zachodzi potrzeba zapewnienia równoległości operacji wejścia-wyjścia w funkcji `count_neighbors()`, a nie tylko w funkcji `game_logic()`, wówczas nie będzie konieczności wprowadzania żadnych zmian w programie, ponieważ `ThreadPoolExecutor` wykonuje te funkcje współbieżnie, jako część funkcji `step_cell()`. Istnieje nawet możliwość osiągnięcia równoległości działania procesora za pomocą tego samego interfejsu, o ile zachodzi potrzeba (zobacz sposób 64.).

Jednak największy z pozostałych problemów polega na ograniczonej równoległości operacji wejścia-wyjścia zapewnianej przez `ThreadPoolExecutor`. Nawet po przypisaniu parametrowi `max_workers` wartości 100 to rozwiązanie nie będzie się skalowało, gdy siatka będzie zawierała 10 000 komórek wymagających równoległych operacji wejścia-wyjścia. `ThreadPoolExecutor` to dobre rozwiązanie w sytuacji, w której nie istnieje rozwiązanie asynchroniczne (np. plikowe operacje wejścia-wyjścia), choć w wielu przypadkach mamy znacznie lepsze sposoby umożliwiające maksymalizację równoległości operacji wejścia-wyjścia (zobacz sposób 60.).

Do zapamiętania

- Egzemplarz `ThreadPoolExecutor` pozwala na prostą równoległość operacji wejścia-wyjścia, nie wymagając przy tym zbyt dużo refaktoringu. Takie rozwiązanie umożliwia łatwe uniknięcie kosztu związanego z uruchamianiem wątku w trakcie każdego wywołania funkcji współbieżności typu *fan-out*.
- Wprawdzie egzemplarz `ThreadPoolExecutor` eliminuje ryzyko zużycia całej pamięci podczas bezpośredniego stosowania wątków, ale jednocześnie ogranicza równoległość operacji wejścia-wyjścia, wymagając uprzedniego podania wartości parametru `max_workers`.

Sposób 60. Zapewnij wysoką współbieżność operacji wejścia-wyjścia dzięki użyciu współprogramów

W poprzednich sposobach próbowałem rozwiązać problem związany z równoległością operacji wejścia-wyjścia w grze w życie i osiągnąłem na tym polu różne efekty (w sposobie 56. znajdziesz więcej informacji na temat tej gry oraz implementacje używanych w niej różnych funkcji i klas). Wszystkie zaprezentowane rozwiązania okazały się niewystarczające, gdy trzeba obsłużyć jednocześnie tysiące działających funkcji (zobacz sposoby od 57. do 59.).

Python pozwala zapewnić wysoką liczbę współbieżnych operacji wejścia-wyjścia dzięki wykorzystaniu tzw. *współprogramów*. Współprogramy umożliwiają jednoczesne działanie ogromnej liczby funkcji w programie Pythona. Do implementacji współprogramu używa się słów kluczowych `async` i `await` wraz z infrastrukturą stosowaną w generatorach (zobacz sposoby 30., 34. i 35.).

Kosztem związanym z uruchomieniem współprogramu jest wywołanie funkcji. Gdy współprogram jest aktywny, wymaga poniżej 1 KB pamięci aż do chwili zakończenia jego działania. Podobnie jak wątki współprogramy to niezależne funkcje pobierające dane wejściowe z ich

środowiska i generujące pewne dane wyjściowe. Różnica między nimi polega na tym, że współprogram wstrzymuje działanie w każdym wyrażeniu `await` i wznawia działanie funkcji `async()` po zakończeniu *oczekiwania* (mamy więc zachowanie podobne do sposobu działania wyrażenia `yield` w generatorze).

Wiele oddzielnych funkcji `async()` posuwanych do przodu w jednym kroku wydaje się działać jednocześnie, odzwierciedlając w ten sposób współbieżne zachowanie wątków Pythona. Jednak w przypadku współprogramów nie mamy obciążenia związanego z wykorzystaniem pamięci, kosztem uruchomienia wątku i przełączania kontekstu, a także nie trzeba używać wymaganego przez wątki skomplikowanego kodu odpowiedzialnego za nakładanie blokad i synchronizację między wątkami. Magicznym mechanizmem napędzającym współprogram jest tzw. *pętla zdarzeń*, która może efektywnie zapewnić ogromną współbieżność operacji wejścia-wyjścia i szybkie wykonywanie na przemian odpowiednio utworzonych funkcji.

Współprogramy mogę wykorzystać do implementacji gry w życie. Moim celem jest umożliwienie przeprowadzania operacji wejścia-wyjścia w funkcji `game_logic()` i jednoczesne pokonanie problemów napotykanych w przedstawionych we wcześniejszych sposobach rozwiązaniach opartych na egzemplarzach `Thread` i `Queue`. W tym celu najpierw trzeba określić funkcję `game_logic()` jako współprogram, co oznacza konieczność zdefiniowania jej za pomocą polecenia `async def` zamiast jedynie `def`. To pozwoli stosować składnię `await` dla operacji wejścia-wyjścia, np. asynchronicznego odczytu z gniazda.

```
ALIVE = '*'
EMPTY = '-'
class Grid:
    ...

def count_neighbors(y, x, get):
    ...

async def game_logic(state, neighbors):
    ...
    # Miejsce na przeprowadzenie blokujących operacji wejścia-wyjścia
    data = await my_socket.read(50)
    ...
```

Podobnie funkcję `step_cell()` można zmienić na współprogram przez dodanie słowa kluczowego `async` do definicji oraz użycie wyrażenia `await` do wywołania funkcji `game_logic()`:

```
async def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = await game_logic(state, neighbors)
    set(y, x, next_state)
```

Funkcja `simulate()` również musi stać się współprogramem:

```
import asyncio

async def simulate(grid):
    next_grid = Grid(grid.height, grid.width)
```

```
    tasks = []
    for y in range(grid.height):
        for x in range(grid.width):
            task = step_cell(
                y, x, grid.get, next_grid.set)       # Typ współbieżności fan-out
            tasks.append(task)

    await asyncio.gather(*tasks)                     # Typ współbieżności fan-in

    return next_grid
```

Będąca współprogramem wersja funkcji `simulate()` wymaga pewnych wyjaśnień:

- Wywołanie `step_cell()` nie spowoduje natychmiastowego wykonania tej funkcji. Zamiast tego zwraca egzemplarz współprogramu, który później może zostać użyty za pomocą wyrażenia `await`. Mamy tutaj sytuację podobną do funkcjonowania generatora i jego wyrażenia `yield`, które po wywołaniu zwraca egzemplarz generatora zamiast natychmiast go uruchomić. Opóźnienie wykonania w taki sposób to mechanizm tworzący współbieżność typu *fan-out*.
- Funkcja `gather()` z biblioteki wbudowanej `asyncio` powoduje utworzenie współbieżności typu *fan-in*. Wyrażenie `await` nakazuje pętli zdarzeń współbieżne uruchomienie współprogramów `step_cell()` i wznowienie wykonywania współprogramu `simulate()`, gdy wszystkie współprogramy `step_cell()` zakończą działanie.
- Nie jest wymagane nakładanie blokad na egzemplarze `Grid`, ponieważ cały kod jest wykonywany w pojedynczym wątku. Operacje wejścia-wyjścia są przeprowadzane równolegle w ramach pętli zdarzeń dostarczanej przez `asyncio`.

Zastosowanie tego rozwiązania wymaga zmiany jednego wiersza kodu w pierwotnym przykładzie. Wykorzystujemy funkcję `asyncio.run()` do uruchomienia współprogramu `simulate()` w pętli zdarzeń i przeprowadzania przez nią niezależnych operacji wejścia-wyjścia:

```
class ColumnPrinter:
    ...

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = asyncio.run(simulate(grid))    # Uruchomienie pętli zdarzeń

print(columns)

>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
```

```
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

Wynik jest dokładnie taki sam jak wcześniej. Zostało wyeliminowane całe obciążenie związane z obsługą wątków. Podczas gdy podejścia oparte na `Queue` i `ThreadPoolExecutor` są ograniczone pod względem obsługi wyjątków — jedynie ponownie zgłaszają wyjątki w innych wątkach — to współprogramy pozwalają zastosować interaktywny debuger do analizy kodu wiersz po wierszu (zobacz sposób 80.):

```
async def game_logic(state, neighbors):
    ...
    raise OSError('Problem z operacjami wejścia-wyjścia')
    ...

asyncio.run(game_logic(ALIVE, 3))

>>>
Traceback ...
OSError: Problem z operacjami wejścia-wyjścia
```

Jeżeli później zmienią się wymagania programu i konieczne będzie przeprowadzanie operacji wejścia-wyjścia także z poziomu funkcji `count_neighbor()`,wystarczy dodać słowa kluczowe `async` i `await` do istniejących funkcji. Unikamy w ten sposób konieczności restrukturyzacji całego dotychczasowego kodu, jak ma to miejsce w przypadku zastosowania egzemplarzy `Thread` i `Queue` (kolejny przykład znajdziesz w sposobie 61.).

```
async def count_neighbors(y, x, get):
    ...

async def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = await count_neighbors(y, x, get)
    next_state = await game_logic(state, neighbors)
    set(y, x, next_state)

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = asyncio.run(simulate(grid))

print(columns)

>>>
    0     |     1     |     2     |     3     |     4
```

```
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

Piękno współprogramów polega na tym, że pozwalają uniknąć powiązania kodu obsługującego środowisko zewnętrzne (operacje wejścia-wyjścia) od implementacji wykonującej zlecone zadania (np. pętlę zdarzeń). Dzięki temu można się skoncentrować na logice programu zamiast marnować czas na ustalania, jak wykonać zadania w sposób współbieżny.

Do zapamiętania

- Funkcje zdefiniowane za pomocą słowa kluczowego `async` są nazywane współprogramami. Używając słowa kluczowego `await`, komponent wywołujący może otrzymać wynik działania współprogramu.
- Współprogramy oferują efektywny sposób jednoczesnego wykonywania dziesiątek tysięcy funkcji.
- Współprogramy mogą stosować współbieżność typów *fan-out* i *fan-in* w celu zapewnienia równoległego wykonywania operacji wejścia-wyjścia. W takim przypadku możliwe jest wyeliminowanie wszystkich problemów związanych z wykonywaniem operacji wejścia-wyjścia przez wątki.

Sposób 61. Naucz się przekazywać do asyncio wątkowane operacje wejścia-wyjścia

Gdy tylko poznasz i zrozumiesz zalety współprogramów (zobacz sposób 60.), refaktoryzacja istniejącej bazy kodu do nowego podejścia opartego na współprogramach może wydawać się żmudnym zadaniem. Na szczęście Python oferuje obsługę asynchronicznego wykonywania kodu, która została wbudowana w język. Dzięki temu kod wykorzystujący wątki do wykonywania blokujących operacji wejścia-wyjścia można bardzo łatwo przenieść do współprogramów i asynchronicznych operacji wejścia-wyjścia.

Załóżmy, że mamy serwer oparty na TCP przeznaczony do gry wymagającej odgadnięcia liczby. Taki serwer pobiera parametry `lower` i `upper` określające przedział liczb. Następnie zwraca zgadywane wartości w postaci liczb całkowitych z tego przedziału, gdy są one żądane przez klienta. Ponadto serwer pobiera od klienta informacje o tym, czy dana liczba była bliżej (ciepło) czy dalej (zimno) od liczby do odgadnięcia.

Najczęściej stosowany do utworzenia takiego systemu typu klient – serwer sposób polega na wykorzystaniu blokujących operacji wejścia-wyjścia oraz wątków (zobacz sposób 53.). To wymaga klasy pomocniczej odpowiedzialnej za obsługę wysyłania i otrzymywania komunikatów. Na potrzeby omawianego przykładu przyjmuję założenie, że każdy wysłany lub odebrany wiersz zawiera polecenie do przetworzenia.

```
class EOFError(Exception):
    pass

class ConnectionBase:
    def __init__(self, connection):
        self.connection = connection
        self.file = connection.makefile('rb')

    def send(self, command):
        line = command + '\n'
        data = line.encode()
        self.connection.send(data)

    def receive(self):
        line = self.file.readline()
        if not line:
            raise EOFError('Połączenie zostało zakończone')
        return line[:-1].decode()
```

Serwer jest zaimplementowany w postaci klasy obsługującej w danej chwili tylko jedno połączenie i zachowującej informacje o stanie sesji klienta:

```
import random

WARMER = 'cieplej'
COLDER = 'zimniej'
UNSURE = 'nie wiadomo'
CORRECT = 'prawidłowo'

class UnknownCommandError(Exception):
    pass

class Session(ConnectionBase):
    def __init__(self, *args):
        super().__init__(*args)
        self._clear_state(None, None)

    def _clear_state(self, lower, upper):
        self.lower = lower
        self.upper = upper
        self.secret = None
        self.guesses = []
```

Ten kod zawiera jedną metodę podstawową odpowiedzialną za obsługę poleceń przychodzących od klienta i przekazującą je do odpowiednich metod. Zwróć uwagę na wykorzystanie wyrażenia przypisania (wprowadzone w Pythonie 3.8 — więcej informacji na ten temat znajdziesz w sposobie 10.), co ma zapewnić zwięzłość kodu:

```
    def loop(self):
        while command := self.receive():
            parts = command.split(' ')
            if parts[0] == 'PARAMS':
                self.set_params(parts)
```

```
            elif parts[0] == 'NUMBER':
                self.send_number()
            elif parts[0] == 'REPORT':
                self.receive_report(parts)
            else:
                raise UnknownCommandError(command)
```

Pierwsze polecenie określa dolną i górną granicę przedziału liczb, z którego program uruchomiony w serwerze ma odgadnąć jedną:

```
    def set_params(self, parts):
        assert len(parts) == 3
        lower = int(parts[1])
        upper = int(parts[2])
        self._clear_state(lower, upper)
```

Drugie polecenie powoduje wybór liczby na podstawie informacji o stanie przechowywanych w egzemplarzu `Session` klienta. W szczególności ten kod ma zagwarantować, że serwer nigdy nie spróbuje więcej niż raz wskazać tej samej liczby.

```
    def next_guess(self):
        if self.secret is not None:
            return self.secret

        while True:
            guess = random.randint(self.lower, self.upper)
            if guess not in self.guesses:
                return guess

    def send_number(self):
        guess = self.next_guess()
        self.guesses.append(guess)
        self.send(format(guess))
```

Trzecie polecenie zajmuje się obsługą odpowiedzi udzielonej przez klienta (cieplej lub zimniej) oraz odpowiednim uaktualnieniem informacji przechowywanych w egzemplarzu `Session`:

```
    def receive_report(self, parts):
        assert len(parts) == 2
        decision = parts[1]

        last = self.guesses[-1]
        if decision == CORRECT:
            self.secret = last

        print(f'Serwer: {last} oznacza {decision}')
```

Do implementacji klienta również zostaje wykorzystana klasa przechowująca informacje o stanie:

```
import contextlib
import math

class Client(ConnectionBase):
```

```
    def __init__(self, *args):
        super().__init__(*args)
        self._clear_state()

    def _clear_state(self):
        self.secret = None
        self.last_distance = None
```

Parametry dla każdej sesji gry w zgadywanie liczby są określane za pomocą polecenia `with` w celu zagwarantowania, że informacje o stanie będą obsługiwane prawidłowo po stronie serwera (zobacz sposoby 63. i 66.). Kolejna metoda wysyła pierwsze polecenie do serwera:

```
    @contextlib.contextmanager
    def session(self, lower, upper, secret):
        print(f'Odgadnij liczbę z przedziału od {lower} do {upper}!'
              f' Ciii, to jest {secret}.')
        self.secret = secret
        self.send(f'PARAMS {lower} {upper}')
        try:
            yield
        finally:
            self._clear_state()
            self.send('PARAMS 0 -1')
```

Nowa wartość jest żądana z serwera za pomocą innej metody, która implementuje drugie polecenie:

```
    def request_numbers(self, count):
        for _ in range(count):
            self.send('NUMBER')
            data = self.receive()
            yield int(data)
            if self.last_distance == 0:
                return
```

Niezależnie od tego, czy ostatnio podana liczba była bliżej czy dalej zgadywanej, zostanie wskazana za pomocą trzeciego polecenia w ostatniej metodzie:

```
    def report_outcome(self, number):
        new_distance = math.fabs(number - self.secret)
        decision = UNSURE

        if new_distance == 0:
            decision = CORRECT
        elif self.last_distance is None:
            pass
        elif new_distance < self.last_distance:
            decision = WARMER
        elif new_distance > self.last_distance:
            decision = COLDER

        self.last_distance = new_distance

        self.send(f'REPORT {decision}')
        return decision
```

Teraz można uruchomić serwer używający jednego wątku do nasłuchiwania gniazda i uruchamiający dodatkowe wątki do obsługi nowych połączeń:

```
import socket
from threading import Thread

def handle_connection(connection):
    with connection:
        session = Session(connection)
        try:
            session.loop()
        except EOFError:
            pass

def run_server(address):
    with socket.socket() as listener:
        listener.bind(address)
        listener.listen()
        while True:
            connection, _ = listener.accept()
            thread = Thread(target=handle_connection,
                            args=(connection,),
                            daemon=True)
            thread.start()
```

Klient działa w wątku głównym i zwraca wynik (zgadywana liczba) komponentowi wywołującemu. W omawianym kodzie wyraźnie są używane różne funkcje języka Python (m.in. pętle `for`, polecenia `with`, generatory i konstrukcje składane), więc w kolejnym fragmencie kodu możesz zobaczyć, jak wygląda ich przeniesienie do rozwiązania opartego na współprogramach:

```
def run_client(address):
    with socket.create_connection(address) as connection:
        client = Client(connection)

        with client.session(1, 5, 3):
            results = [(x, client.report_outcome(x))
                       for x in client.request_numbers(5)]

        with client.session(10, 15, 12):
            for number in client.request_numbers(5):
                outcome = client.report_outcome(number)
                results.append((number, outcome))

    return results
```

Wreszcie można zebrać wszystkie elementy układanki w całość i potwierdzić, że gra działa zgodnie z oczekiwaniami:

```
def main():
    address = ('127.0.0.1', 1234)
    server_thread = Thread(
        target=run_server, args=(address,), daemon=True)
```

```
    server_thread.start()

    results = run_client(address)
    for number, outcome in results:
        print(f'Klient: {number} oznacza {outcome}')

main()

>>>
Odgadnij liczbę z przedziału od 1 do5! Ciiii, to jest 3.
Serwer: 4 oznacza nie wiadomo
Serwer: 1 oznacza zimniej
Serwer: 5 oznacza nie wiadomo
Serwer: 3 oznacza prawidłowo
Odgadnij liczbę z przedziału od 10 do 15! Ciiii, to jest 12.
Serwer: 11 oznacza nie wiadomo
Serwer: 10 oznacza zimniej
Serwer: 12 oznacza prawidłowo
Klient: 4 oznacza nie wiadomo
Klient: 1 oznacza zimniej
Klient: 5 oznacza nie wiadomo
Klient: 3 oznacza prawidłowo
Klient: 11 oznacza nie wiadomo
Klient: 10 oznacza zimniej
Klient: 12 oznacza prawidłowo
```

Ile wysiłku wymaga konwersja tego przykładu na wersję wykorzystującą `async`, `await` i wbudowany moduł `asyncio`?

Przede wszystkim trzeba uaktualnić klasę `ConnectionBase` w celu dostarczenia współprogramów dla `send()` i `receive()` zamiast blokujących metod operacji wejścia-wyjścia. Każdy zmieniony wiersz kodu oznaczyłem komentarzem `# Zmiana`, aby wyraźnie pokazać różnice między wersjami poprzednią i nową:

```
class AsyncConnectionBase:
    def __init__(self, reader, writer):             # Zmiana
        self.reader = reader                        # Zmiana
        self.writer = writer                        # Zmiana

    async def send(self, command):
        line = command + '\n'
        data = line.encode()
        self.writer.write(data)                     # Zmiana
        await self.writer.drain()                   # Zmiana

    async def receive(self):
        line = await self.reader.readline()         # Zmiana
        if not line:
            raise EOFError('Połączenie zostało zakończone')
        return line[:-1].decode()
```

Istnieje możliwość utworzenia kolejnej przechowującej informacje o stanie klasy, która będzie przedstawiała stan sesji dla pojedynczego połączenia. Jedyna zmiana tutaj to nazwa klasy i dziedziczenie po `AsyncConnectionBase` zamiast po `ConnectionBase`:

```
class AsyncSession(AsyncConnectionBase):              # Zmiana
    def __init__(self, *args):
        ...

    def _clear_values(self, lower, upper):
        ...
```

Podstawowy punkt wejścia do działającej w serwerze pętli przetwarzania poleceń wymaga jedynie minimalnych zmian, po których wprowadzeniu staje się współprogramem:

```
   async def loop(self):                              # Zmiana
        while command := await self.receive():        # Zmiana
            parts = command.split(' ')
            if parts[0] == 'PARAMS':
                self.set_params(parts)
            elif parts[0] == 'NUMBER':
                await self.send_number()              # Zmiana
            elif parts[0] == 'REPORT':
                self.receive_report(parts)
            else:
                raise UnknownCommandError(command)
```

Nie są wymagane żadne zmiany w celu zapewnienia obsługi pierwszego polecenia:

```
    def set_params(self, parts):
        ...
```

Jedyną zmianą wymaganą dla drugiego polecenia jest umożliwienie użycia asynchronicznych operacji wejścia-wyjścia podczas przekazywania danych do klienta:

```
    def next_guess(self):
        ...

    async def send_number(self):                      # Zmiana
        guess = self.next_guess()
        self.guesses.append(guess)
        await self.send(format(guess))                # Zmiana
```

Nie są wymagane żadne zmiany w celu zapewnienia obsługi trzeciego polecenia:

```
    def receive_report(self, parts):
        ...
```

Klasę klienta również trzeba zaimplementować ponownie, aby dziedziczyła po `AsyncConnectionBase`:

```
class AsyncClient(AsyncConnectionBase):               # Zmiana
    def __init__(self, *args):
        ...

    def _clear_state(self):
        ...
```

Metoda pierwszego polecenia dla klienta wymaga użycia kilku słów kluczowych `async` i `await`, konieczne jest również zastosowanie funkcji pomocniczej `asynccontextmanager()` z modułu wbudowanego `contextlib`:

```
    @contextlib.asynccontextmanager                # Zmiana
    async def session(self, lower, upper, secret):  # Zmiana
        print(f'Odgadnij liczbę z przedziału od {lower} do {upper}!'
              f' Ciiii, to jest {secret}.')
        self.secret = secret
        await self.send(f'PARAMS {lower} {upper}')  # Zmiana
        try:
            yield
        finally:
            self._clear_state()
            await self.send('PARAMS 0 -1')          # Zmiana
```

Metoda polecenia drugiego wymaga jedynie dodania słów kluczowych `async` i `await` wszędzie tam, gdzie oczekiwany jest sposób działania współprogramu:

```
    async def request_numbers(self, count):         # Zmiana
        for _ in range(count):
            await self.send('NUMBER')               # Zmiana
            data = await self.receive()             # Zmiana
            yield int(data)
            if self.last_distance == 0:
                return
```

Metoda polecenia trzeciego wymaga użycia kilku słów kluczowych `async` i `await`:

```
    async def report_outcome(self, number):         # Zmiana
        ...
        await self.send(f'REPORT {decision}')       # Zmiana
        ...
```

Z kolei kod definiujący serwer trzeba zaimplementować zupełnie od początku, aby wykorzystać możliwości modułu wbudowanego `asyncio` i jego funkcji `start_server()`:

```
import asyncio

async def handle_async_connection(reader, writer):
    session = AsyncSession(reader, writer)
    try:
        await session.loop()
    except EOFError:
        pass

async def run_async_server(address):
    server = await asyncio.start_server(
        handle_async_connection, *address)
    async with server:
        await server.serve_forever()
```

Funkcja `run_client()` inicjująca grę wymaga zmian w praktycznie każdym wierszu. Cały kod, który wcześniej odpowiadał za współpracę z blokującymi egzemplarzami `socket`, trzeba zastąpić oferującym podobną funkcjonalność kodem w wersji `asyncio` (te wiersze oznaczyłem komentarzem `# Nowy`). Wszystkie pozostałe wiersze w funkcji wymagające współpracy ze współprogramami muszą w odpowiedni sposób używać słów kluczowych `async` i `await`. Jeżeli zapomnisz o dodaniu choć jednego z tych słów kluczowych, w trakcie działania programu nastąpi zgłoszenie wyjątku.

```
async def run_async_client(address):
    streams = await asyncio.open_connection(*address)    # Nowy
    client = AsyncClient(*streams)                       # Nowy

    async with client.session(1, 5, 3):
        results = [(x, await client.report_outcome(x))
                   async for x in client.request_numbers(5)]

    async with client.session(10, 15, 12):
        async for number in client.request_numbers(5):
            outcome = await client.report_outcome(number)
            results.append((number, outcome))

    _, writer = streams                                  # Nowy
    writer.close()                                       # Nowy
    await writer.wait_closed()                           # Nowy

    return results
```

Najbardziej intersującym aspektem funkcji `run_async_client()` jest brak konieczności restrukturyzacji któregokolwiek z fragmentów odpowiedzialnych za współpracę z egzemplarzem `AsyncClient`, aby można było uznać, że nowa wersja funkcji używa współprogramów. Każda niezbędna funkcja języka ma odpowiadającą jej wersję asynchroniczną, co niezwykle ułatwia migrację.

Jednak nie zawsze tak będzie. Aktualnie nie istnieją asynchroniczne wersje funkcji wbudowanych `next()` i `iter()` (zobacz sposób 31.)., więc konieczne jest użycie `await` bezpośrednio w metodach `__anext__()` i `__aiter__()`. Ponadto nie istnieje asynchroniczna wersja `yield from` (zobacz sposób 33.), co utrudnia nieco tworzenie generatorów. Jednak biorąc pod uwagę tempo dodawania funkcjonalności asynchronicznej do Pythona, to tylko kwestia czasu, gdy brakujące metody trafią do tego języka.

Uaktualnić trzeba także kod łączący wszystkie komponenty, aby nowy przykład działał asynchronicznie od początku do końca. Funkcja `asyncio.create_task()` zostanie użyta do kolejkowania serwera w celu wykonania w pętli zdarzeń. Dzięki temu dotarłszy do wyrażenia `await`, kod serwera będzie działał równolegle z klientem. To jest kolejne podejście powodujące zastosowanie współbieżności *fan-out* w przypadku użycia innej funkcji niż `asyncio.gather()`.

```
async def main_async():
    address = ('127.0.0.1', 4321)

    server = run_async_server(address)
    asyncio.create_task(server)
```

```
    results = await run_async_client(address)
    for number, outcome in results:
        print(f'Klient: {number} oznacza {outcome}')

asyncio.run(main_async())

>>>
Odgadnij liczbę z przedziału od 1 do 5! Ciiii, to jest 3.
Serwer: 5 oznacza nie wiadomo
Serwer: 4 oznacza cieplej
Serwer: 2 oznacza nie wiadomo
Serwer: 1 oznacza zimniej
Serwer: 3 oznacza prawidłowo
Odgadnij liczbę z przedziału od 10 do 15! Ciiii, to jest 12.
Serwer: 14 oznacza nie wiadomo
Serwer: 10 oznacza nie wiadomo
Serwer: 15 oznacza zimniej
Serwer: 12 oznacza prawidłowo
Klient: 5 oznacza nie wiadomo
Klient: 4 oznacza cieplej
Klient: 2 oznacza nie wiadomo
Klient: 1 oznacza zimniej
Klient: 3 oznacza prawidłowo
Klient: 14 oznacza nie wiadomo
Klient: 10 oznacza nie wiadomo
Klient: 15 oznacza zimniej
Klient: 12 oznacza prawidłowo
```

To rozwiązanie działa zgodnie z oczekiwaniami. Wersja oparta na współprogramach jest bardziej zrozumiała, ponieważ zostały usunięte wszystkie interakcje zachodzące między wątkami. Moduł wbudowany `asyncio` zapewnia dostęp do wielu funkcji pomocniczych i pozwala zmniejszyć ilość kodu niezbędnego do utworzenia serwera takiego jak w omawianym przykładzie.

Twój program może z wielu różnych powodów być bardziej skomplikowany i trudniejszy do przeniesienia. Moduł `asyncio` oferuje obsługę wielu różnych operacji wejścia-wyjścia, synchronizacji i zarządzania zadaniami, dzięki którym stosowanie współprogramów stało się łatwiejsze (zobacz sposoby 62. i 63.). Zajrzyj też do dostępnej w internecie dokumentacji biblioteki `asyncio` (*https://docs.python.org/3/library/asyncio.html*), jeżeli chcesz w pełni poznać jej potencjał.

Do zapamiętania

- Python oferuje działające asynchronicznie wersje pętli `for`, poleceń `with`, generatorów, konstrukcji składanych i funkcji pomocniczych bibliotek —wszystkie można wykorzystać jako bezpośrednie zamienniki we współprogramach.
- Moduł wbudowany `asyncio` niezwykle ułatwia konwersję istniejącego kodu wykorzystującego wątki i blokujące operacje wejścia-wyjścia na wersję opartą na współprogramach i asynchronicznych operacjach wejścia-wyjścia.

Sposób 62. Połączenie wątków i współprogramów w celu ułatwienia konwersji na wersję stosującą asyncio

W poprzednim sposobie pokazałem konwersję serwera TCP używającego wątków i blokujących operacji wejścia-wyjścia na wersję wykorzystującą bibliotekę `asyncio` i współprogramy. Ta konwersja to było coś: w trakcie jednej sesji cały kod został przeniesiony do nowego stylu programowania. Jednak skonwertowanie w ten sposób ogromnego programu rzadko jest możliwe. Zamiast tego zwykle trzeba przeprowadzać konwersje fragmentami bazy kodu, uaktualniać testy według potrzeb i na każdym kroku sprawdzać, czy wszystko działa zgodnie z oczekiwaniami.

W tym celu baza kodu musi mieć możliwość jednoczesnego i kompatybilnego użycia wątków dla blokujących operacji wejścia-wyjścia (zobacz sposób 53.) i współprogramów dla asynchronicznych operacji wejścia-wyjścia (zobacz sposób 60.). W praktyce to oznacza, że wątki muszą mieć możliwość uruchamiania współprogramów, które z kolei muszą mieć możliwość uruchamiania wątków i czekania na zakończenie ich działania. Na szczęście biblioteka `asyncio` zawiera wbudowane metody pozwalające na łatwe zapewnienie takiej współpracy.

Załóżmy, że tworzymy program łączący pliki dzienników zdarzeń w jeden strumień, co ma pomóc podczas debugowania. Gdy mamy uchwyt pliku do danych wejściowych w postaci dziennika zdarzeń, może wystąpić konieczność ustalenia, czy pojawiły się nowe dane, i zwrócenia następnego wiersza danych wejściowych. Do tego celu można wykorzystać metodę `tell()` uchwytu pliku sprawdzającą, czy aktualne położenie odczytu odpowiada długości pliku. W przypadku braku nowych danych nastąpi zgłoszenie wyjątku (zobacz sposób 20.).

```
class NoNewData(Exception):
    pass

def readline(handle):
    offset = handle.tell()
    handle.seek(0, 2)
    length = handle.tell()

    if length == offset:
        raise NoNewData

    handle.seek(offset, 0)
    return handle.readline()
```

Opakowując tę funkcję pętlą `while`, można ją zmienić na wątek roboczy. Gdy pojawi się nowy wiersz, następuje wykonanie określonej funkcji wywołania zwrotnego w celu zapisu nowych danych w wyjściowym dzienniku zdarzeń (w sposobie 38. znajdziesz informacje o tym, dlaczego należy w takich przypadkach stosować interfejs funkcji zamiast klasę). Jeżeli nowe dane się jeszcze nie pojawiły, wątek będzie uśpiony, aby w ten sposób zmniejszyć obciążenie, jakie powoduje sprawdzanie źródła pod kątem nowych danych. Po zamknięciu uchwytu pliku wątek roboczy kończy działanie.

```
import time

def tail_file(handle, interval, write_func):
    while not handle.closed:
        try:
            line = readline(handle)
        except NoNewData:
            time.sleep(interval)
        else:
            write_func(line)
```

Teraz można uruchomić jeden wątek roboczy dla każdego pliku danych wejściowych i połączyć ich dane wyjściowe w jeden plik. Funkcja pomocnicza `write()` wymaga egzemplarza `Lock` (zobacz sposób 54.) w celu serializowania operacji zapisu do strumienia danych wyjściowych i zagwarantowania, że nie wystąpią żadne konflikty podczas zapisu wiersza danych wyjściowych.

```
from threading import Lock, Thread

def run_threads(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        lock = Lock()
        def write(data):
            with lock:
                output.write(data)
        threads = []
        for handle in handles:
            args = (handle, interval, write)
            thread = Thread(target=tail_file, args=args)
            thread.start()
            threads.append(thread)

        for thread in threads:
            thread.join()
```

Dopóki uchwyt pliku danych wejściowych pozostaje dostępny, dopóty będzie działał przetwarzający go wątek. To oznacza, że wystarczające jest oczekiwanie na zakończenie działania metody `join()` w każdym wątku, aby w ten sposób ustalić zakończenie całego procesu.

Mając do dyspozycji zbiór ścieżek danych wejściowych i wyjściowych, można wywołać metodę `run_threads()` i potwierdzić, że działa zgodnie z oczekiwaniami. W omawianym przykładzie sposób tworzenia lub zamykania uchwytów plików danych wejściowych nie ma znaczenia, podobnie jak funkcji sprawdzającej dane wyjściowe zdefiniowanej w `confirm_merge()`:

```
def confirm_merge(input_paths, output_path):
    ...

input_paths = ...
handles = ...
output_path = ...
run_threads(handles, 0.1, output_path)

confirm_merge(input_paths, output_path)
```

Mając tę stosującą wątki implementację jako punkt wyjścia, w jaki sposób można powoli skonwertować kod na wersję używającą współprogramów i biblioteki `asyncio`? W tym zakresie mamy dwa podejścia: od początku do końca i od końca do początku.

Podejście od początku do końca rozpoczyna się od najważniejszych fragmentów bazy kodu, np. punktu wejścia w postaci funkcji `main()`, i jest kontynuowane aż do poszczególnych funkcji i klas w hierarchii wywołań. Takie podejście okazuje się użyteczne w przypadku istnienia wielu wspólnych modułów stosowanych w odmiennych programach. Jeżeli najpierw zostaną skonwertowane punkty wejścia, wówczas z konwersją wspólnych modułów można zaczekać do chwili, gdy współprogramy będą stosowane już wszędzie.

Oto konkretne kroki do wykonania:

1. Utworzenie funkcji głównej z użyciem `async def` zamiast tylko `def`.
2. Opakowanie wszystkich jej wywołań związanych z operacjami wejścia-wyjścia — potencjalnie blokującymi pętlę zdarzeń — wywołaniami `asyncio.run_in_executor`.
3. Zagwarantowanie, że wszystkie zasoby i wywołania zwrotne używane w `run_in_executor()` są prawidłowo zsynchronizowane (np. za pomocą egzemplarzy `Lock` lub funkcji `asyncio.run_coroutine_threadsafe()`).
4. Wyeliminowanie wywołań `get_event_loop()` i `run_in_executor()` przez przejście w dół hierarchii wywołań oraz konwersję funkcji i metod na współprogramy (po trzech pierwszych krokach).

Oto przykład zastosowania trzech pierwszych kroków w funkcji `run_threads()`:

```
import asyncio

async def run_tasks_mixed(handles, interval, output_path):
    loop = asyncio.get_event_loop()

    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        def write(data):
            coro = write_async(data)
            future = asyncio.run_coroutine_threadsafe(
                coro, loop)
            future.result()

        tasks = []
        for handle in handles:
            task = loop.run_in_executor(
                None, tail_file, handle, interval, write)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

Metoda `run_in_executor()` nakazuje pętli zdarzeń wykonanie określonej funkcji — w omawianym przykładzie `tail_file()` — za pomocą konkretnego egzemplarza `ThreadPoolExecutor` (zobacz sposób 59.) lub egzemplarza domyślnego, gdy pierwszym parametrem jest `None`. Wykonując wiele żądań do `run_in_executor()` bez odpowiadających im wyrażeń `await`, współprogram `run_tasks_mixed()` (tryb współbieżności *fan-out*) ma jedną współbieżną linię wykonywania dla każdego pliku danych wejściowych. Następnie funkcja `asyncio.gather()` wraz z wyrażeniem `await` (tryb współbieżności *fan-in*) czeka w funkcji `tail_file()` na zakończenie działania wszystkich wątków (więcej informacji na temat współbieżności *fan-out* i *fan-in* znajdziesz w sposobie 56.).

Ten kod eliminuje potrzebę stosowania egzemplarza `Lock` w metodzie pomocniczej `write()` dzięki wywołaniu `asyncio.run_coroutine_threadsafe()`. Funkcja ta umożliwia zwykłym i doskonale znanym wątkom roboczym wywołanie współprogramu — w omawianym przykładzie jest to `write_async()` — i wykonanie go w pętli zdarzeń w wątku głównym (lub innych wątkach, o ile zachodzi potrzeba). To w efekcie synchronizuje wątki i gwarantuje, że wszystkie operacje zapisu pliku danych wyjściowych będą przeprowadzane jedynie przez pętlę zdarzeń w wątku głównym. Po rozwiązaniu kwestii dostępności `asyncio.gather()` można przyjąć założenie, że wszystkie operacje zapisu danych wyjściowych do pliku zostały zakończone. To oznacza możliwość zamknięcia uchwytu pliku w poleceniu `with` bez obaw, że powstanie stan wyścigu.

Poprawność działania tego kodu można sprawdzić. Funkcję `asyncio.run()` wykorzystałem do uruchomienia współprogramu i głównej pętli zdarzeń:

```
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_tasks_mixed(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

Teraz można zastosować krok 4. dla funkcji `run_tasks_mixed()` przez przejście do dołu stosu wywołań. Można ponownie zdefiniować funkcję zależną `tail_file()` jako asynchroniczny współprogram, aby nie przeprowadzała blokujących operacji wejścia-wyjścia. To oznacza konieczność wykonania kroków od 1. do 3.

```
async def tail_async(handle, interval, write_func):
    loop = asyncio.get_event_loop()

    while not handle.closed:
        try:
            line = await loop.run_in_executor(
                None, readline, handle)
        except NoNewData:
            await asyncio.sleep(interval)
        else:
            await write_func(line)
```

Ta nowa implementacja `tail_async()` pozwala przekazywać wywołania `get_event_loop()` i `run_in_executor()` do dołu stosu wywołań i całkowicie poza `run_tasks_mixed()`. Kod, który pozostał, jest znacznie bardziej przejrzysty i zrozumiały.

```
async def run_tasks(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)
        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, write_async)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

Można potwierdzić, że funkcja `run_tasks()` również działa zgodnie z oczekiwaniami:

```
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_tasks(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

Istnieje możliwość kontynuowania tego iteracyjnego wzorca refaktoryzacji i konwersji funkcji `readline()` także na asynchroniczny współprogram. Jednak ta funkcja wymaga tak wielu blokujących operacji wejścia-wyjścia, że nie warto jej konwertować, ponieważ to znacznie zmniejszy czytelność kodu źródłowego i niekorzystnie wpłynie na jej wydajność działania. W niektórych sytuacjach sensowne jest przeniesienie wszystkiego do `asyncio`, a w innych nie.

Podejście od końca do początku podczas adaptowania współprogramów również ma cztery kroki, które są podobne do kroków podejścia od początku do końca. Jednak proces oznacza poruszanie się po hierarchii wywołań w przeciwnym kierunku: od punktów końcowych do punktu wejścia.

Oto konkretne kroki do wykonania:

1. Utworzenie nowej, asynchronicznej wersji współprogramu każdej funkcji końcowej, która ma zostać skonwertowana.
2. Zamiana istniejących funkcji synchronicznych w taki sposób, aby wywoływały wersje w postaci współprogramów i uruchamiały pętle zdarzeń zamiast implementować jakiekolwiek rzeczywiste zachowanie.
3. Przejście o poziom wyżej w hierarchii wywołań, utworzenie kolejnej warstwy współprogramów i zastąpienie istniejących wywołań funkcji synchronicznych wywołaniami współprogramów zdefiniowanych w kroku 1.
4. Usunięcie synchronicznych opakowań wokół współprogramów utworzonych w kroku 2., ponieważ są już niepotrzebne do połączenia wszystkich elementów w całość.

W omawianym przykładzie rozpocząłbym od funkcji `tail_file()`, ponieważ wcześniej zdecydowałem, że funkcja `readline()` nadal powinna przeprowadzać blokujące operacje wejścia-wyjścia. Funkcję `tail_file()` można zmodyfikować w taki sposób, aby jedynie stanowiła opakowanie dla zdefiniowanego współprogramu `tail_async()`. Aby ten współprogram działał aż do końca, konieczne jest utworzenie pętli zdarzeń dla każdego wątku roboczego `tail_file`, a następnie wywoływanie jego metody `run_until_complete()`. Ta metoda będzie blokowała bieżący wątek i napędzała pętlę zdarzeń, dopóki istnieje współprogram `tail_async()`. W efekcie otrzymujemy takie samo zachowanie jak w przypadku funkcji `tail_file()` stosującej wątki i blokujące operacje wejścia-wyjścia.

```
def tail_file(handle, interval, write_func):
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)

    async def write_async(data):
        write_func(data)

    coro = tail_async(handle, interval, write_async)
    loop.run_until_complete(coro)
```

Ta nowa funkcja `tail_file()` jest bezpośrednim zamiennikiem starej. Działanie całości zgodnie z oczekiwaniami można ponownie potwierdzić za pomocą wywołania `run_threads()`.

```
input_paths = ...
handles = ...
output_path = ...
run_threads(handles, 0.1, output_path)

confirm_merge(input_paths, output_path)
```

Po opakowaniu `tail_async()` za pomocą `tail_file()` kolejnym krokiem jest konwersja funkcji `run_threads()` na współprogram. Ten krok odpowiada krokowi 4. w przedstawionym wcześniej podejściu od początku do końca, więc w tym miejscu oba style pozostają zbieżne.

Jest to doskonały początek stosowania biblioteki `asyncio`, choć można zrobić jeszcze więcej, aby poprawić responsywność tworzonych programów (zobacz sposób 63.).

Do zapamiętania

- Metoda `run_in_executor()` pętli zdarzeń `asyncio` pozwala współprogramom wykonywać funkcje synchroniczne w puli `ThreadPoolExecutor`. W ten sposób można przeprowadzić migrację do `asyncio` typu od początku do końca.
- Metoda `run_until_complete()` pętli zdarzeń `asyncio` pozwala kodowi synchronicznemu wykonywać współprogram do końca. Funkcja `asyncio.run_coroutine_threadsafe()` oferuje tę samą funkcjonalność między wątkami. Obie metody umożliwiają zastosowanie migracji do `asyncio` typu od końca do początku.

Sposób 63. Maksymalizuj responsywność przez unikanie blokującej pętli zdarzeń asyncio

W poprzednim sposobie pokazałem, jak można iteracyjne przeprowadzić migrację do `asyncio` (zobacz sposób 62., w którym znajdziesz ważne informacje i implementacje różnych funkcji). Utworzony współprogram prawidłowo pobiera pliki danych wejściowych i łączy je w pojedynczy plik danych wyjściowych:

```
import asyncio

async def run_tasks(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, write_async)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

Jednak wciąż pozostaje jeden duży problem: wywołania `open()`, `close()` i `write()` dla uchwytu pliku danych wyjściowych mają miejsce w głównej pętli zdarzeń. Te operacje wymagają wykonywania wywołań systemowych do systemu operacyjnego komputera gospodarza, w którym został uruchomiony program. To może oznaczać blokowanie pętli zdarzeń przez znaczną ilość czasu i uniemożliwiać innym współprogramom poczynienie postępów. Skutkiem może być znaczny spadek ogólnej responsywności i zwiększenie opóźnienia, zwłaszcza w przypadku programów takich jak wysoce współbieżne serwery.

Dzięki przekazaniu parametru `debug=True` do funkcji `asyncio.run()` można ustalić, kiedy ten problem występuje. W kolejnym fragmencie kodu pokazałem, jak zidentyfikować plik i wiersz nieprawidłowego współprogramu, prawdopodobnie blokującego w wolnym wywołaniu systemowym:

```
import time

async def slow_coroutine():
    time.sleep(0.5)  # Symulacja wolno wykonywanych operacji wejścia-wyjścia

asyncio.run(slow_coroutine(), debug=True)

>>>
Executing <Task finished name='Task-1' coro=<slow_coroutine()
  done, defined at example.py:29> result=None created
  at .../asyncio/base_events.py:487> took 0.503 seconds
...
```

Jeżeli chcesz mieć maksymalnie responsywny program, konieczne jest zminimalizowanie potencjalnych wywołań systemowych, które są wykonywane w pętli zdarzeń. W omawianym przykładzie mogę utworzyć nową podklasę klasy `Thread` (zobacz sposób 53.), która będzie hermetyzowała wszystko to, co jest wymagane podczas zapisu pliku danych wyjściowych za pomocą własnej pętli zdarzeń:

```
from threading import Thread

class WriteThread(Thread):
    def __init__(self, output_path):
        super().__init__()
        self.output_path = output_path
        self.output = None
        self.loop = asyncio.new_event_loop()

    def run(self):
        asyncio.set_event_loop(self.loop)
        with open(self.output_path, 'wb') as self.output:
            self.loop.run_forever()

        # Wykonanie rundy finałowej wywołań zwrotnych, aby oczekiwanie
        # na zakończenie funkcji stop() odbywało się w innej pętli zdarzeń
        self.loop.run_until_complete(asyncio.sleep(0))
```

Współprogramy w innych wątkach mogą bezpośrednio wywoływać metody `write()` w tych klasach i czekać na zakończenie ich działania, ponieważ mamy do czynienia jedynie z zapewniającym bezpieczeństwo wątków opakowaniem metody `real_write()` faktycznie odpowiedzialnej za operacje wejścia-wyjścia. To eliminuje konieczność stosowania egzemplarza `Lock` (zobacz sposób 54.).

```
    async def real_write(self, data):
        self.output.write(data)

    async def write(self, data):
        coro = self.real_write(data)
        future = asyncio.run_coroutine_threadsafe(
            coro, self.loop)
        await asyncio.wrap_future(future)
```

Inne współprogramy mogą w sposób zapewniający bezpieczeństwo wątków poinformować wątek roboczy, kiedy zakończyć pracę, używając do tego następującego kodu:

```
    async def real_stop(self):
        self.loop.stop()

    async def stop(self):
        coro = self.real_stop()
        future = asyncio.run_coroutine_threadsafe(
            coro, self.loop)
        await asyncio.wrap_future(future)
```

Istnieje również możliwość zdefiniowania metod `__aenter__()` i `__aexit__()` pozwalających tej klasie używać ich w poleceniach `with` (zobacz sposób 66.). To gwarantuje, że wątek roboczy będzie uruchamiany i zatrzymywany we właściwym czasie bez spowalniania wątku głównej pętli zdarzeń.

```
    async def __aenter__(self):
        loop = asyncio.get_event_loop()
        await loop.run_in_executor(None, self.start)
        return self

    async def __aexit__(self, *_):
        await self.stop()
```

Mając tę nową klasę `WriteThread`, można przeprowadzić refaktoryzację funkcji `run_tasks()` na wersję w pełni asynchroniczną, która będzie bardzo przejrzysta i całkowicie pozbawiona wolno wykonywanych wywołań systemowych w wątku głównej pętli zdarzeń:

```
def readline(handle):
    ...

async def tail_async(handle, interval, write_func):
    ...

async def run_fully_async(handles, interval, output_path):
    async with WriteThread(output_path) as output:
        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, output.write)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

Mając zbiór uchwytów danych wejściowych i ścieżkę pliku danych wyjściowych, można sprawdzić, czy rozwiązanie działa zgodnie z oczekiwaniami:

```
def confirm_merge(input_paths, output_path):
    ...
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_fully_async(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

Do zapamiętania

- Wykonywanie wywołań systemowych we współprogramach (obejmuje to blokujące operacje wejścia-wyjścia i uruchamianie wątków) może pogorszyć responsywność programu i spotęgować wrażenie opóźnienia w jego działaniu.
- Funkcji `asyncio.run()` przekazuj parametr `debug=True` w celu ustalenia, czy dany współprogram uniemożliwia szybkie działanie pętli zdarzeń.

Sposób 64. Rozważ użycie concurrent.futures(), aby otrzymać prawdziwą równoległość

Na pewnym etapie tworzenia programów w Pythonie możesz dotrzeć do ściany, jeśli chodzi o kwestie wydajności. Nawet po przeprowadzeniu optymalizacji kodu (zobacz sposób 70.) wykonywanie programu wciąż może okazać się za wolne w stosunku do potrzeb. W nowoczesnych komputerach, w których nieustannie zwiększa się liczba dostępnych rdzeni procesora, można przyjąć założenie, że jedynym rozsądnym rozwiązaniem jest równoległość. Co się stanie, jeżeli kod odpowiedzialny za obliczenia podzielisz na niezależne fragmenty jednocześnie działające w wielu rdzeniach procesora?

Niestety, mechanizm GIL w Pythonie uniemożliwia osiągnięcie prawdziwej równoległości w wątkach (zobacz sposób 53.), a więc tę opcję można wykluczyć. Inną często pojawiającą się propozycją jest ponowne utworzenie kodu o znaczeniu krytycznym dla wydajności. Nowy kod powinien mieć postać modułu rozszerzenia i być utworzony w języku C. Dzięki językowi C zbliżasz się bardziej do samego sprzętu, a utworzony w nim kod działa szybciej niż w Pythonie, co eliminuje konieczność zastosowania równoległości. Rozszerzenia utworzone w języku C mogą również uruchamiać rodzime wątki działające równocześnie i wykorzystujące wiele rdzeni procesora. API Pythona przeznaczone dla rozszerzeń tworzonych w języku C jest doskonale udokumentowane i stanowi doskonałe wyjście awaryjne. Warto również zapoznać się z takimi narzędziami jak SWIG (*https://github.com/swig/swig*) i CLIF (*https://github.com/google/clif*), które pomagają w tworzeniu rozszerzeń.

Jednak ponowne utworzenie kodu w języku C wiąże się z wysokim kosztem. Kod, który w Pythonie jest krótki i zrozumiały, w języku C może stać się rozwlekły i skomplikowany. Tego rodzaju kod wymaga starannego przetestowania i upewnienia się, że funkcjonalność odpowiada pierwotnej, utworzonej w Pythonie. Ponadto trzeba sprawdzić, czy nie zostały wprowadzone nowe błędy. Czasami włożony wysiłek się opłaca, co wyjaśnia istnienie w społeczności Pythona ogromnego ekosystemu modułów rozszerzeń utworzonych w języku C. Dzięki wspomnianym rozszerzeniom można przyśpieszyć operacje takie jak przetwarzanie tekstu, tworzenie obrazów i operacje na macierzach. Istnieją nawet narzędzia typu open source, na przykład Cython (*http://cython. org/*) i Numba (*http://numba.pydata.org/*) ułatwiające przejście do języka C.

Problem polega na tym, że utworzenie jednego fragmentu programu w języku C w większości przypadków okaże się niewystarczające. Zoptymalizowane programy Pythona zwykle nie mają tylko jednego źródła powolnego działania, ale raczej wiele poważnych źródeł. Aby więc wykorzystać szybkość oferowaną przez język C i wątki, konieczne będzie przepisanie dużych fragmentów programu, co drastycznie wydłuża czas potrzebny na jego przetestowanie i zwiększa ryzyko. Musi istnieć lepszy sposób pozwalający na rozwiązywanie trudnych problemów obliczeniowych w Pythonie.

Wbudowany moduł `multiprocessing`, łatwo dostępny za pomocą innego wbudowanego modułu, `concurrent.futures`, może być dokładnie tym, czego potrzebujesz (zobacz sposób 59.). Pozwala Pythonowi na jednoczesne wykorzystanie wielu rdzeni procesora dzięki uruchomieniu dodatkowych interpreterów jako procesów potomnych. Wspomniane procesy potomne są niezależne od głównego interpretera, a więc ich blokady globalne również pozostają oddzielne. Każdy proces potomny może w pełni wykorzystać jeden rdzeń procesora. Ponadto każdy z nich ma odwołanie do procesu głównego, z którego otrzymuje polecenia przeprowadzenia obliczeń i do którego zwraca wynik.

Na przykład przyjmujemy założenie, że w Pythonie ma zostać przeprowadzona operacja wykonująca intensywne obliczenia i wykorzystująca wiele rdzeni procesora. W poniższym przykładzie użyłem implementacji algorytmu wyszukującego największy wspólny mianownik dwóch liczb jako proxy dla dwóch znacznie bardziej wymagających obliczeń algorytmów, takich jak symulacja dynamiki cieczy i równania Naviera-Stokesa.

```
# my_module.py

def gcd(pair):
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
    assert False, 'Niedostępny'
```

Szeregowe wykonywanie tej funkcji oznacza liniowy wzrost czasu potrzebnego na przeprowadzenie obliczeń, ponieważ nie została użyta równoległość.

```
# run_serial.py
import my_module
import time

NUMBERS = [
    (1963309, 2265973), (2030677, 3814172),
    (1551645, 2229620), (2039045, 2020802),
    (1823712, 1924928), (2293129, 1020491),
    (1281238, 2273782), (3823812, 4237281),
    (3812741, 4729139), (1292391, 2123811),
]

def main():
    start = time.time()
    results = list(map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Operacja zabrała {delta:.3f} sekund')

if __name__ == '__main__':
    main()

>>>
Operacja zabrała 1.173 sekund
```

Jeżeli ten kod zostanie wykonany w wielu wątkach Pythona, nie spowoduje to żadnej poprawy wydajności, ponieważ mechanizm GIL uniemożliwia Pythonowi jednoczesne użycie wielu rdzeni procesora. Poniżej prezentuję, jak wygląda przeprowadzenie tych samych obliczeń za pomocą modułu `concurrent.futures`, jego klasę `ThreadPoolExecutor` i dwa wątki robocze (w celu dopasowania ich do liczby rdzeni w moim komputerze).

```
# run_threads.py
import my_module
from concurrent.futures import ThreadPoolExecutor
import time

NUMBERS = [
    ...
]

def main():
    start = time.time()
    pool = ThreadPoolExecutor(max_workers=2)
    results = list(pool.map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Operacja zabrała {delta:.3f} sekund')

if __name__ == '__main__':
    main()
```

```
>>>
Operacja zabrała 1.436 sekund
```

Jak widzisz, czas wykonania zadania jeszcze się wydłużył, co ma związek z obciążeniem dotyczącym uruchomienia puli wątków i komunikacji z nią.

Pora na coś zaskakującego: zmiana tylko jednego wiersza kodu wystarczy, aby stało się coś magicznego. Jeżeli klasę `ThreadPoolExecutor` zastąpimy klasą `ProcessPoolExecutor` z modułu `concurrent.futures`, to wszystko ulegnie przyśpieszeniu.

```
# run_parallel.py
import my_module
from concurrent.futures import ProcessPoolExecutor
import time

NUMBERS = [
    ...
]

def main():
    start = time.time()
    pool = ProcessPoolExecutor(max_workers=2)   # Jedyna zmiana w kodzie
    results = list(pool.map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Operacja zabrała {delta:.3f} sekund')
```

```
if __name__ == '__main__':
    main()
```

```
>>>
Operacja zabrała 0.683 sekund
```

Po uruchomieniu kodu na moim dwurdzeniowym komputerze widać znaczącą poprawę wydajności. Jak to możliwe? Poniżej przedstawiam faktyczny sposób działania klasy `ProcessPoolExecutor` z użyciem niskiego poziomu konstrukcji dostarczanych przez moduł `multiprocessing`:

1. Każdy element danych wejściowych `numbers` zostaje przekazany do `map`.
2. Dane są serializowane na postać danych binarnych za pomocą modułu `pickle` (zobacz sposób 68.).
3. Serializowane dane są z procesu interpretera głównego kopiowane do procesu interpretera potomnego za pomocą gniazda lokalnego.
4. Kolejnym krokiem jest deserializacja danych na postać obiektów Pythona z wykorzystaniem `pickle`. Odbywa się to w procesie potomnym.
5. Import modułu Pythona zawierającego funkcję `gcd`.
6. Uruchomienie funkcji wraz z otrzymanymi danymi wejściowymi. Inne procesy potomne wykonują tę samą funkcję, ale z innymi danymi.
7. Serializacja wyniku na postać bajtów.
8. Skopiowanie bajtów przez gniazdo lokalne do procesu nadrzędnego.
9. Deserializacja bajtów z powrotem na postać obiektów Pythona w procesie nadrzędnym.
10. Połączenie wyników z wielu procesów potomnych w pojedynczą listę będącą ostatecznym wynikiem.

Wprawdzie przedstawiony powyżej proces wydaje się prosty dla programisty, ale moduł `multiprocessing` i klasa `ProcessPoolExecutor` muszą wykonać ogromną pracę, aby równoległe wykonywanie zadań było możliwe. W większości innych języków programowania jedynym miejscem wymagającym koordynacji dwóch wątków jest pojedyncza blokada lub niepodzielna operacja (zobacz sposób 54.). Obciążenie związane z użyciem modułu `multiprocessing` jest duże z powodu konieczności przeprowadzania serializacji i deserializacji między procesami nadrzędnym i potomnymi.

Schemat ten wydaje się doskonale dopasowany do pewnego typu odizolowanych zadań, w dużej mierze opartych na dźwigni. Tutaj „*odizolowanych*” oznacza, że funkcja nie musi z innymi częściami programu współdzielić informacji o stanie. Z kolei wyrażenie „*w dużej mierze opartych na dźwigni*” oznacza tutaj sytuację, gdy między procesami nadrzędnym i potomnym musi być przekazywana jedynie niewielka ilość danych niezbędnych do przeprowadzenia dużych obliczeń. Algorytm największego wspólnego mianownika jest przykładem takiej sytuacji, choć wiele innych algorytmów matematycznych działa podobnie.

Jeżeli charakterystyka obliczeń, które chcesz przeprowadzić, jest inna od przedstawionej powyżej, to obciążenie związane z użyciem modułu `multiprocessing` może uniemożliwić zwiększenie wydajności działania programu po zastosowaniu równoległości. W takich przypadkach

moduł `multiprocessing` oferuje funkcje zaawansowane związane z pamięcią współdzieloną, blokadami między procesami, kolejkami i proxy. Jednak wszystkie wymienione funkcje są niezwykle skomplikowane. Naprawdę trudno znaleźć uzasadnienie dla umieszczania tego rodzaju narzędzi w pamięci jednego procesu współdzielonego między wątkami Pythona. Przeniesienie tego poziomu skomplikowania do innych procesów i angażowanie gniazd jeszcze bardziej utrudnia zrozumienie kodu.

Sugeruję unikanie modułu `multiprocessing` i użycie wymienionych funkcji za pomocą prostszego modułu `concurrent.futures`. Możesz rozpocząć od zastosowania klasy `ThreadPoolExecutor` w celu wykonywania odizolowanych i stanowiących duże obciążenie funkcji w wątkach. Następnie możesz przejść do klasy `ProcessPoolExecutor`, aby zwiększyć szybkość działania aplikacji. Po wyczerpaniu wszystkich opcji możesz rozważyć bezpośrednie użycie modułu `multiprocessing`.

Do zapamiętania

- Przepisanie w języku C tych fragmentów programu, które stanowią wąskie gardła, może być efektywnym sposobem poprawy wydajności działania programu i jednocześnie maksymalizacji inwestycji w kod Pythona. Jednak związany z tym koszt jest wysoki, a ponadto mogą być wprowadzone nowe błędy.
- Moduł `multiprocessing` dostarcza narzędzia, które mogą pozwolić na równoległe wykonywanie pewnego typu obliczeń w Pythonie przy minimalnym wysiłku.
- Do wykorzystania możliwości modułu `multiprocessing` najlepiej nadają się wbudowany moduł `concurrent.futures` i jego prosta klasa `ProcessPoolExecutor`.
- Należy unikać zaawansowanych funkcji modułu `multiprocessing`, ponieważ są one zbyt skomplikowane.

Python: elegancja, wydajność i ekspresja kodu!

Python słusznie cieszy się stale rosnącym uznaniem programistów: jest wszechstronny i efektywny, pozwala też na tworzenie wysokiej jakości oprogramowania. Język ten ma poza tym wiele trudniejszych do uchwycenia zalet. Aby uzyskać naprawdę imponujące efekty w zakresie wydajności kodu, jego przenaszalności i bezpieczeństwa, trzeba zagłębić się w dość subtelne niuanse kodowania. Wielu programistów, choć posiada spore doświadczenie w programowaniu w innych językach, nie dostrzega tych zależności. Z kolei osoby dopiero rozpoczynające przygodę z programowaniem mogą poczuć się zaskoczone i zdezorientowane, jeśli nie zdołają uniknąć kilku nieoczywistych błędów podczas pracy.

To drugie, zaktualizowane i uzupełnione wydanie podręcznika programowania w duchu Pythona. Zawarty tu materiał umożliwia wykorzystanie tego języka do tworzenia wyjątkowo solidnego i niezwykle wydajnego kodu źródłowego. Książka jest napisana w zwięzłym stylu i ma przemyślany układ, oparty na scenariuszach, dzięki czemu przystępnie przedstawia 90 najlepszych praktyk, wskazówek i skrótów oraz wyjaśnia ich działanie na rzeczywistych przykładach kodu. Pokazano tu szereg mało znanych, być może nieco dziwnych sztuczek i sposobów udoskonalających pracę kodu źródłowego. Przyswojenie zaprezentowanych praktyk pozwoli Ci tworzyć kod łatwy do zrozumienia, obsługi i dalszej rozbudowy. W tym wydaniu treść poszczególnych wskazówek zaktualizowano do Pythona 3, a poszczególne przykłady kodu zostały przejrzane i udoskonalone – najlepsze praktyki również ewoluują!

W tej książce:

- nowe rozwiązania dla wszystkich najważniejszych obszarów programowania w Pythonie
- techniki stosowania konstrukcji składanych i funkcji generatorów
- właściwe korzystanie z klas, obiektów, metaklas i atrybutów dynamicznych
- współbieżność, równoległość, optymalizacja i bezpieczeństwo kodu
- wbudowane moduły Pythona do debugowania i testowania
- narzędzia i najlepsze praktyki podczas wspólnej pracy nad projektami

Brett Slatkin jest starszym inżynierem oprogramowania w firmie Google. Pracował nad projektem Google Consumer Surveys, nad protokołem PubSubHubbub, a wcześniej zajmował się pierwszym w Google produktem przetwarzania w chmurze – App Engine. Ma duże doświadczenie w używaniu Pythona do zarządzania flotą serwerów Google.

ISBN 978-83-283-6732-6