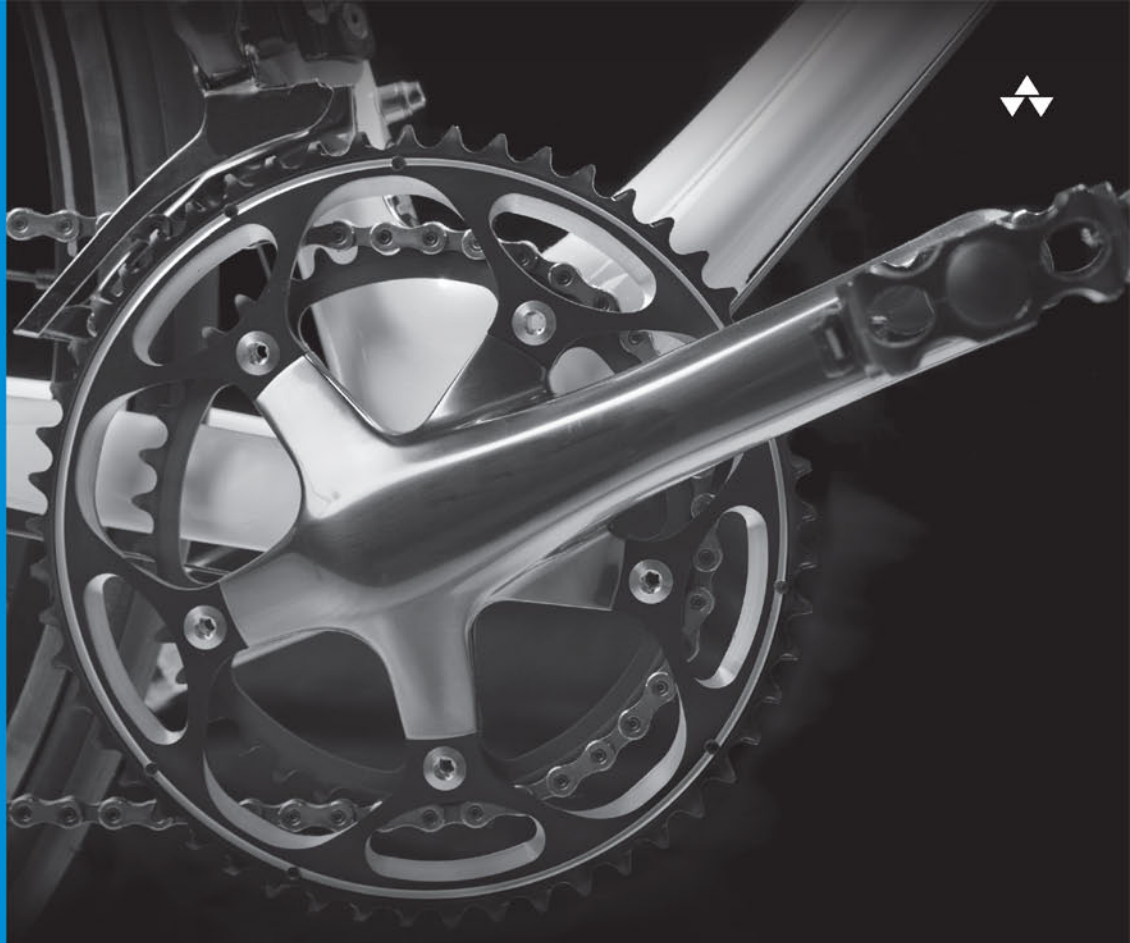

Efektywny PYTHON

59 sposobów na lepszy kod

Helion

Brett Slatkin

Tytuł oryginału: Effective Python: 59 Specific Ways to Write Better Python (Effective Software Development Series)

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-1540-2

Authorized translation from the English language edition, entitled: EFFECTIVE PYTHON: 59 SPECIFIC WAYS TO WRITE BETTER PYTHON; ISBN 0134034287; by Brett Slatkin; published by Pearson Education, Inc, publishing as Addison Wesley Professional.

Wydawnictwo HELION
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/efepyt
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

5 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, 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. Ponadto 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 36. 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 (patrz sposób 37.), 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 `popen`, `popen2` 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 konstruktor klasy `Popen` uruchamia proces. Z kolei metoda `communicate()` odczytuje dane wyjściowe procesu potomnego i czeka na jego zakończenie.

```
proc = subprocess.Popen(
    ['echo', 'Witaj z procesu potomnego!'],
    stdout=subprocess.PIPE)
out, err = proc.communicate()
print(out.decode('utf-8'))
>>>
Witaj z procesu potomnego!
```

Procesy potomne będą działały niezależnie od ich procesu nadrzędnego, czyli interpretera Pythona. Ich stan można okresowo sprawdzać, gdy Python wykonuje inne zadania.

```
proc = subprocess.Popen(['sleep', '0.3'])
while proc.poll() is None:
    print('Pracuję...')
    # Miejsce na zadania, których wykonanie wymaga dużo czasu.
```

```
    # ...
print('Kod wyjścia', proc.poll())
>>>
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.

```
def run_sleep(period):
    proc = subprocess.Popen(['sleep', str(period)])
    return proc

start = time()
procs = []
for _ in range(10):
    proc = run_sleep(0.1)
    procs.append(proc)
```

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

```
for proc in procs:
    proc.communicate()
end = time()
print('Zakończono w ciągu %.3f sekund' % (end - start))
>>>
Zakończono w ciągu 0.117 sekund
```

Wskazówka

Jeżeli wymienione procesy działają w sekwencji, to całkowite opóźnienie wynosi sekundę, a nie tylko mniej więcej 0,1 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.

```
def run_openssl(data):
    env = os.environ.copy()
    env['password'] = b'\xe24U\n\xd0Ql3S\x11'
    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_openssl(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.

```
for proc in procs:
    out, err = proc.communicate()
    print(out[-10:])
>>>
b'o4.G\x91\x95\xfe\xa0\xaa\xb7'
b'\x0b\x01\\\xb1\xb7\xfb\xb2C\xe1b'
b'ds\xc5\xf4;j\x1f\xd0c-'
```

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 `md5` pobierze strumień danych wejściowych:

```
def run_md5(input_stdin):
    proc = subprocess.Popen(
        ['md5'],
        stdin=input_stdin,
        stdout=subprocess.PIPE)
    return proc
```

Wskazówka

Wbudowany moduł Pythona o nazwie `hashlib` oferuje funkcję `md5()`, a więc uruchomienie tego rodzaju procesu potomnego nie zawsze jest konieczne. Moim celem jest tutaj pokazanie, jak podprocesy mogą potokować dane wejściowe i wyjściowe.

Teraz wykorzystujemy zbiór procesów `openssl` do szyfrowania pewnych danych, a kolejny zbiór procesów do utworzenia wartości hash na podstawie zaszyfrowanych danych.

```
input_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_openssl(data)
    input_procs.append(proc)
    hash_proc = run_md5(proc.stdout)
    hash_procs.append(hash_proc)
```

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 input_procs:
    proc.communicate()
for proc in hash_procs:
    out, err = proc.communicate()
    print(out.strip())
>>>
b'7a1822875dcf9650a5a71e5e41e77bf3'
b'd41d8cd98f00b204e9800998ecf8427e'
b'1720f581cfdc448b6273048d42621100'
```

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 = run_sleep(10)
try:
    proc.communicate(timeout=0.1)
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Kod wyjścia', proc.poll())
>>>
Kod wyjścia -15
```

Niestety, parametr `timeout` jest dostępny jedynie w Pythonie 3.3 oraz nowych wydaniach. We wcześniejszych wersjach Pythona konieczne jest użycie wbudowanego modułu `select` w `proc.stdin`, `proc.stdout` i `proc.stderr` w celu wymuszenia stosowania limitu czasu w trakcie operacji wejścia-wyjścia.

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.
- Używaj parametru `timeout` w metodzie `communicate()`, aby unikać zakleszczeń i zawieszenia procesów potomnych.

Sposób 37. 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. 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. Tego rodzaju przerwanie może doprowadzić do uszkodzenia interpretera, 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.

```
numbers = [2139079, 1214759, 1516637, 1852285]
start = time()
for number in numbers:
    list(factorize(number))
end = time()
print('Operacja zabrała %.3f sekund' % (end - start))
>>>
Operacja zabrała 1.040 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()
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()
print('Operacja zabrała %.3f sekund' % (end - start))
>>>
Operacja zabrała 1.061 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 na programy wykonywane przez standardowy interpreter CPython.

Istnieją różne sposoby pozwalające CPython na wykorzystanie wielu wątków, ale nie działają one ze standardową klasą `Thread` (patrz sposób 41.) 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 40.). 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

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

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

```
start = time()
for _ in range(5):
    slow_systemcall()
end = time()
print('Operacja zabrała %.3f sekund' % (end - start))
>>>
Operacja zabrała 0.503 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()
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()
print('Operacja zabrała %.3f sekund' % (end - start))
>>>
Operacja zabrała 0.102 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 (patrz sposób 40.). 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 38. Używaj klasy Lock, aby unikać stanu wyścigu w wątkach

Po dowiedzeniu się o istnieniu mechanizmu GIL (patrz sposób 37.) 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 operacje wątku na strukturach danych mogą być zakłócone między dwoma instrukcjami kodu bajtowego w interpreterze Pythona. To jest niebezpieczne, jeśli jednocześnie z wielu wątków próbujesz uzyskać dostęp do tych samych obiektów. Struktury danych mogą być praktycznie w każdej chwili uszkodzone na skutek wspomnianych zakłóceń, co 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(object):
    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:

```
def run_threads(func, how_many, counter):
    threads = []
    for i in range(5):
        args = (i, how_many, counter)
        thread = Thread(target=func, args=args)
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
```

Jednoczesne uruchomienie pięciu wątków wydaje się proste, a dane wyjściowe powinny być oczywiste.

```
how_many = 10**5
counter = Counter()
run_threads(worker, how_many, counter)
print('Oczekiwana liczba próbek %d, znaleziona %d' %
      (5 * how_many, counter.count))
>>>
Oczekiwana liczba próbek 500000, znaleziona 278328
```

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.

```
counter.count += offset
```

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 + offset
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)
```

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 43.).

```
class LockingCounter(object):
    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()
run_threads(worker, how_many, counter)
print('Oczekiwana liczba próbek %d, znaleziona %d' %
      (5 * how_many, counter.count))
>>>
Oczekiwana liczba próbek 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 39. 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 (patrz sposób 37.).

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?

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 38., aby zrozumieć wagę bezpieczeństwa wątków w Pythonie. Z kolei w sposobie 46. znajdziesz więcej informacji o klasie `deque`).

```
class MyQueue(object):
    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.

```
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:
            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('Przetworzono', processed, 'elementów po wykonaniu',
      polled, 'sprawdzeń')
>>>
Przetworzono 1000 elementów po wykonaniu 3030 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.

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
queue = Queue()

def consumer():
    print('Konsument oczekuje')
    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')
queue.put(object())             # Uruchomienie przed metodą get() przedstawioną powyżej.
thread.join()
print('Producent zakończył pracę')
>>>
Konsument oczekuje
Producent umieszcza dane
Konsument zakończył pracę
Producent 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 nało-

żenia blokady, gdy kolejka jest już zapełniona. W poniższym fragmencie kodu przedstawiłem definicję wątku oczekującego chwilę przed użyciem kolejki:

```
queue = Queue(1)                # Bufor o wielkości 1.

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

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.

```
queue.put(object())             # Pierwsze wywołanie.
print('Producent umieszcza dane 1')
queue.put(object())             # Trzecie wywołanie.
print('Producent umieszcza dane 2')
thread.join()
print('Producent zakończył pracę')
>>>
Producent umieszcza dane 1
Konsument pobiera dane 1
Producent umieszcza dane 2
Konsument pobiera dane 2
Producent 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(target=consumer).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.

```
in_queue.put(object())          # Zakończone jako pierwsze.
print('Producent oczekuje')
in_queue.join()                 # Zakończone jako czwarte.
print('Producent zakończył pracę')
>>>
Konsument oczekuje
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.

```
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):
        #...
```

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

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

```
download_queue = ClosableQueue()
#…
threads = [
    StoppableWorker(download, download_queue, resize_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')
>>>
1000 elementów zostało przetworzonych
```

Do zapamiętania

- Potoki to doskonały sposób organizowania sekwencji zadań 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 40. Rozważ użycie współprogramów w celu jednoczesnego wykonywania wielu funkcji

Wątki umożliwiają programistom Pythona pozornie jednoczesne wykonywanie wielu funkcji (patrz sposób 37.). Jednak z wątkami wiążą się trzy poważne problemy.

- Wymagają zastosowania specjalnych narzędzi do koordynacji bezpieczeństwa (patrz sposoby 38. i 39). Dlatego też kod oparty na wątkach jest trudniejszy do zrozumienia niż kod proceduralny wykonywany w jednym wątku. Wspomniana trudność powoduje, że kod wykorzystujący wątki staje się trudniejszy do rozbudowy i obsługi.
- Wątki wymagają dużej ilości pamięci — mniej więcej 8 MB dla każdego wykonywanego wątku. W wielu komputerach ilość dostępnej pamięci pozwala na obsługę sporej liczby wątków. Co się jednak stanie, gdy program będzie próbował wykonywać „jednocześnie" dziesiątki tysięcy funkcji? Wspomniane funkcje mogą odpowiadać żądaniom użytkowników kierowanym do serwera, pikselom na ekranie, cząsteczkom w symulacji itd. Próba uruchomienia oddzielnego wątku dla każdej unikalnej czynności się nie sprawdza.
- Uruchamianie wątków jest kosztowne. Jeżeli program ma nieustannie tworzyć nowe jednocześnie działające funkcje i kończyć ich działanie, to obciążenie związane z użyciem wątków stanie się ogromne i spowolni program.

Python pozwala na zniwelowanie wszystkich wymienionych powyżej problemów za pomocą **współprogramów**. Współprogramy pozwalają na użycie w programie Pythona wielu pozornie jednocześnie wykonywanych funkcji. Współprogramy są implementowane jako rozszerzenie generatorów (patrz sposób 16.). Kosztem uruchomienia współprogramu generatora jest wywołanie funkcji. Po uruchomieniu każdy z nich używa poniżej 1 KB pamięci.

Działanie współprogramu polega na umożliwieniu kodowi używającemu generatora na wykonanie funkcji `send()` w celu wysłania wartości z powrotem do funkcji generatora po każdym wyrażeniu `yield`. Funkcja generatora otrzymuje wartość przekazaną funkcji `send()` jako wynik wykonania odpowiedniego wyrażenia `yield`.

```
def my_coroutine():
    while True:
        received = yield
        print('Otrzymano:', received)

it = my_coroutine()
next(it)             # Wywołanie generatora.
it.send('Pierwszy')
it.send('Drugi')
```

```
>>>
Otrzymano: Pierwszy
Otrzymano: Drugi
```

Początkowe wywołanie `next()` jest wymagane do przygotowania generatora na otrzymanie pierwszego wywołania `send()` przez przejście do pierwszego wyrażenia `yield`. Razem polecenie `yield` i wywołanie `send()` zapewniają generatorowi standardowy sposób na zróżnicowanie kolejnej wartości w odpowiedzi na zewnętrzne dane wejściowe.

Na przykład chcesz zaimplementować współprogram generatora dostarczający wartość minimalną, która była dotąd użyta. W poniższym fragmencie kodu `yield` przygotowuje współprogram wraz z początkową wartością minimalną pochodzącą z zewnątrz. Następnie generator ciągle otrzymuje nowe minimum w zamian za nową wartość do rozważenia.

```
def minimize():
    current = yield
    while True:
        value = yield current
        current = min(value, current)
```

Kod wykorzystujący generator może wykonywać po jednym kroku w danej chwili i będzie wyświetlał wartość minimalną po otrzymaniu kolejnych danych wejściowych.

```
it = minimize()
next(it)            # Wywołanie generatora.
print(it.send(10))
print(it.send(4))
print(it.send(22))
print(it.send(-1))
>>>
10
4
4
-1
```

Funkcja generatora będzie pozornie działała w nieskończoność i robiła postęp wraz z każdym nowym wywołaniem `send()`. Podobnie jak wątki, współprogramy to niezależne funkcje pobierające dane wejściowe z ich środowiska i generujące dane wyjściowe. Różnica polega na pauzie po każdym wyrażeniu `yield` w funkcji generatora i wznowieniu działania po każdym wywołaniu `send()` pochodzącym z zewnątrz. Tak wygląda magiczny mechanizm współprogramów.

Przedstawione powyżej zachowanie pozwala, aby kod wykorzystujący generator podejmował działanie po każdym wyrażeniu `yield` we współprogramie. Kod może użyć wartości danych wyjściowych generatora w celu wywołania innych funkcji i uaktualnienia struktur danych. Co ważniejsze, może posunąć do przodu inne funkcje generatora, aż do ich następnego wyrażenia `yield`. Dzięki przesunięciu do przodu wielu oddzielnych generatorów wydaje

się, że wszystkie one działają jednocześnie. To pozwala w Pythonie na naśladowanie zachowania wątków.

Gra w życie

Możliwość jednoczesnego działania współprogramów zademonstruję teraz na przykładzie. Załóżmy, że chcemy je wykorzystać do implementacji gry w życie. Reguły gry są proste: mamy dwuwymiarową planszę o dowolnej wielkości. Każde pole na planszy może być żywe lub puste.

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

Postęp w grze jest oparty na jednym tyknięciu zegara. W trakcie tyknięcia następuje sprawdzenie każdego pola i ustalenie, ile z jego ośmiu sąsiednich pól nadal pozostaje żywych. Na podstawie liczby żywych sąsiadów podejmowana jest decyzja o stanie sprawdzanego pola: pozostaje żywe, umiera lub się regeneruje. Poniżej przedstawiłem przykład planszy o wymiarach 5×5 po czterech kolejkach. Każdy kolejny stan gry jest przedstawiony po prawej stronie poprzedniego. Objaśnienie konkretnych reguł znajdziesz poniżej.

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

Grę można modelować, przedstawiając poszczególne pola jako współprogram generatora działający ramię w ramię z innymi.

Aby zaimplementować grę, przede wszystkim potrzebny jest sposób na pobranie stanu sąsiednich pól. Do tego celu możemy wykorzystać współprogram o nazwie `count_neighbors()`, którego działanie polega na dostarczaniu obiektów `Query`. Wspomnianą klasę `Query` zdefiniujemy samodzielnie. Jej przeznaczeniem jest dostarczenie współprogramu generatora sprawdzającego stan otaczającego go środowiska.

```
Query = namedtuple('Query', ('y', 'x'))
```

Współprogram dostarcza obiekt `Query` dla każdego sąsiedniego pola. Wynikiem poszczególnych wyrażeń `yield` będzie wartość `ALIVE` lub `EMPTY`. Między współprogramem i korzystającym z niego kodem został zdefiniowany interfejs. Generator `count_neighbors()` sprawdza stan sąsiednich pól i zwraca liczbę pól uznawanych za żywe.

```
def count_neighbors(y, x):
    n_ = yield Query(y + 1, x + 0)  # Północ.
    ne = yield Query(y + 1, x + 1)  # Północny wschód.
    # Zdefiniowanie kolejnych kierunków e_, se, s_, sw, w_, nw ...
```

```
    # ...
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count
```

Współprogramowi `count_neighbors()` możemy teraz dostarczyć przykładowe dane, aby przetestować jego działanie. Poniżej pokazałem, jak obiekty `Query` będą dostarczane dla każdego sąsiedniego pola. Współprogram oczekuje na informacje o stanie każdego obiektu `Query` przekazywane metodą `send()` współprogramu. Ostateczna wartość licznika jest zwracana w wyjątku `StopIteration`, który jest zgłaszany, gdy generator jest wyczerpany przez polecenie `return`.

```
it = count_neighbors(10, 5)
q1 = next(it)                    # Pobranie pierwszego obiektu.
print('Pierwsze wyrażenie yield: ', q1)
q2 = it.send(ALIVE)              # Wysłanie informacji o stanie q1, pobranie q2.
print('Drugie wyrażenie yield:', q2)
q3 = it.send(ALIVE)              # Wysłanie informacji o stanie q2, pobranie q3
# ...
try:
    count = it.send(EMPTY)       # Wysłanie informacji o stanie q8, pobranie ostatecznej wartości licznika.
except StopIteration as e:
    print('Liczba: ', e.value)   # Wartość pochodząca z polecenia return.
>>>
Pierwsze wyrażenie yield:  Query(y=11, x=5)
Drugie wyrażenie yield: Query(y=11, x=6)
...
Liczba:  2
```

Teraz potrzebujemy możliwości wskazania, że pole przejdzie do nowego stanu w odpowiedzi na liczbę żywych sąsiadów zwróconą przez `count_neighbors()`. W tym celu definiujemy kolejny współprogram o nazwie `step_cell()`. Ten generator będzie wskazywał zmianę stanu pola przez dostarczanie obiektów `Transition`. To jest kolejna klasa, która podobnie jak `Query` będzie zdefiniowana.

```
Transition = namedtuple('Transition', ('y', 'x', 'state'))
```

Współprogram `step_cell()` otrzymuje argumenty w postaci danych współrzędnych pola na planszy. Pobiera obiekt `Query` w celu uzyskania początkowego stanu wspomnianych współrzędnych. Uruchomi współprogram `count_neighbors()` do sprawdzenia sąsiednich pól. Wykonuje także logikę gry w celu ustalenia, jaki stan dane pole powinno mieć dla kolejnego tyknięcia zegara. Na koniec pobierany jest obiekt `Transition`, aby wskazać środowisku następny stan pola.

```
def game_logic(state, neighbors):
    # ...

def step_cell(y, x):
    state = yield Query(y, x)
```

```
    neighbors = yield from count_neighbors(y, x)
    next_state = game_logic(state, neighbors)
    yield Transition(y, x, next_state)
```

Co ważniejsze, wywołanie `count_neighbors()` używa wyrażenia `yield from`. Wyrażenie to pozwala Pythonowi na łączenie współprogramów generatora, co ułatwia wielokrotne użycie niewielkich fragmentów funkcjonalności i tworzenie skomplikowanych współprogramów na podstawie prostych. Po wyczerpaniu `count_neighbors()` ostateczna wartość zwracana przez współprogram (za pomocą polecenia `return`) będzie przekazana do `step_cell()` jak wynik wyrażenia `yield from`.

Teraz możemy wreszcie zdefiniować prostą logikę gry w życie. Tak naprawdę mamy jedynie trzy reguły.

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

Współprogramowi `step_cell()` dostarczamy przykładowe dane, aby go przetestować.

```
it = step_cell(10, 5)
q0 = next(it)           # Obiekt Query położenia początkowego.
print('Ja:      ', q0)
q1 = it.send(ALIVE)     # Wysłanie mojego stanu, ustawienie pola sąsiada.
print('Q1:      ', q1)
# ...
t1 = it.send(EMPTY)     # Wysłanie stanu q8, podjęcie decyzji w grze.
print('Wynik: ', t1)
>>>
Ja:       Query(y=10, x=5)
Q1:       Query(y=11, x=5)
...
Wynik:  Transition(y=10, x=5, state='-')
```

Celem gry jest wykonanie tej logiki dla wszystkich pól znajdujących się na planszy. W tym celu możemy umieścić współprogram `step_cell()` we współprogramie `simulate()`. Współprogram będzie analizował kolejne pola planszy przez wielokrotne pobieranie `step_cell()`. Po sprawdzeniu wszystkich współrzędnych następuje dostarczenie obiektu `TICK`, wskazującego, że bieżąca generacja pól została zakończona.

```
TICK = object()

def simulate(height, width):
```

```
    while True:
        for y in range(height):
            for x in range(width):
                yield from step_cell(y, x)
        yield TICK
```

W przypadku współprogramu `simulate()` imponujące jest to, że pozostaje on całkowicie niezwiązany z otaczającym go środowiskiem. Nadal nie zdefiniowaliśmy sposobu przedstawienia planszy w obiektach Pythona, obsługi wartości `Query`, `Transition` i `TICK` na zewnątrz, a także tego, jak gra pobiera stan początkowy. Jednak logika pozostaje czytelna. Każde pole przeprowadzi zmianę stanu za pomocą `step_cell()`. Następnie mamy tyknięcie zegara gry. Proces będzie kontynuowany w nieskończoność, dopóki trwa postęp we współprogramie `simulate()`.

Na tym polega piękno współprogramów. Pomagają skoncentrować się na logice tego, co próbujesz osiągnąć. Pozwalają na oddzielenie poleceń kodu dla środowiska od jego implementacji, a tym samym współprogramy mogą działać równocześnie. Na przestrzeni czasu zyskujesz możliwość poprawienia implementacji wspomnianych poleceń kodu bez konieczności zmiany współprogramów.

Teraz chcemy uruchomić `simulate()` w prawdziwym środowisku. W tym celu potrzebujemy sposobu na przestawienie stanu poszczególnych pól planszy. Poniżej przedstawiłem klasę odpowiedzialną za obsługę planszy:

```
class Grid(object):
    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 __str__(self):
        # ...
```

Plansza pozwala na pobieranie i ustawianie wartości dowolnej współrzędnej. Współrzędne wykraczające poza granice będą zawijane, co powoduje, że plansza działa na zasadzie nieskończonego miejsca.

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

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

Musimy jeszcze zdefiniować funkcję interpretującą wartości otrzymane ze współprogramu `simulate()` oraz jego wszystkich wewnętrznych współprogramów. Funkcja ta zamienia instrukcje ze współprogramów na interakcje z otaczającym środowiskiem. Dla całej planszy wykonuje jeden krok do przodu, a następnie zwraca nową planszę zawierającą kolejny stan.

```
def live_a_generation(grid, sim):
    progeny = Grid(grid.height, grid.width)
    item = next(sim)
    while item is not TICK:
        if isinstance(item, Query):
            state = grid.query(item.y, item.x)
            item = sim.send(state)
        else:  # Konieczne jest przekształcenie.
            progeny.assign(item.y, item.x, item.state)
            item = next(sim)
    return progeny
```

Aby zobaczyć tę funkcję w działaniu, konieczne jest utworzenie planszy i ustawienie jej stanu początkowego. Poniżej przedstawiłem przykład utworzenia klasycznego kształtu.

```
grid = Grid(5, 9)
grid.assign(0, 3, ALIVE)
# ...
print(grid)
>>>
---*-----
----*----
--***----
---------
---------
```

Teraz możemy wykonać jeden krok naprzód. Możesz zobaczyć, że w oparciu o proste reguły zdefiniowane w funkcji `game_logic()` kształt ten zostaje przesunięty na dół i w prawą stronę.

```
class ColumnPrinter(object):
    # ...

columns = ColumnPrinter()
sim = simulate(grid.height, grid.width)
for i in range(5):
    columns.append(str(grid))
    grid = live_a_generation(grid, sim)
print(columns)
>>>
    0     |     1     |     2     |     3     |     4
---*----- | --------- | --------- | --------- | ---------
----*---- | --*-*---- | ----*---- | ---*----- | ----*----
--***---- | ---**---- | --*-*---- | ----**--- | -----*---
--------- | ---*----- | ---**---- | ---**---- | ---***---
--------- | --------- | --------- | --------- | ---------
```

Najlepsze w omawianym podejściu jest to, że można zmienić funkcję `game_logic()` bez konieczności wprowadzania jakichkolwiek modyfikacji w otaczającym ją kodzie. Istnieje więc możliwość zmiany reguł lub dodania większych sfer wpływu za pomocą istniejącej mechaniki obiektów `Query`, `Transition` i `TICK`. To pokazuje, jak współprogramy pozwalają na zachowanie podziału zadań, co jest niezwykle ważną zasadą projektową.

Współprogramy w Pythonie 2

Niestety, Python 2 nie oferuje pewnych syntaktycznych cech, dzięki którym współprogramy są tak eleganckim rozwiązaniem w Pythonie 3. W Pythonie 2 istnieją dwa poważne ograniczenia.

Pierwsze to brak wyrażenia `yield from`. Jeżeli więc chcesz łączyć współprogramy generatora w Pythonie 2, musisz zastosować dodatkową pętlę w punkcie delegacji.

```
# Python 2
def delegated():
    yield 1
    yield 2

def composed():
    yield 'A'
    for value in delegated():  # Odpowiednik wyrażenia yield from w Pythonie 3.
        yield value
    yield 'B'

print list(composed())
>>>
['A', 1, 2, 'B']
```

Drugie ograniczenie polega na braku obsługi polecenia `return` w generatorach Pythona 2. W celu uzyskania tego samego zachowania, zapewniającego prawidłowe działanie z blokami `try-except-finally`, konieczne jest zdefiniowanie własnego typu wyjątku i jego zgłaszanie, gdy ma być zwrócona wartość.

```
# Python 2
class MyReturn(Exception):
    def __init__(self, value):
        self.value = value

def delegated():
    yield 1
    raise MyReturn(2)  # Odpowiednik polecenia return 2 w Pythonie 3.
    yield 'Nie osiągnięto'

def composed():
    try:
        for value in delegated():
            yield value
    except MyReturn as e:
        output = e.value
    yield output * 4

print list(composed())
>>>
[1, 8]
```

Do zapamiętania

- Współprogramy oferują efektywny sposób wykonywania dziesiątek tysięcy funkcji pozornie w tym samym czasie.
- W przypadku generatora wartością wyrażenia `yield` będzie wartość przekazana metodzie `send()` generatora z poziomu zewnętrznego kodu.
- Współprogramy są ważnym narzędziem pozwalającym na oddzielenie podstawowej logiki programu od jego interakcji z otaczającym go środowiskiem.
- Python 2 nie obsługuje wyrażenia `yield from`, a także zwrotu wartości z generatorów.

Sposób 41. 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 (patrz sposób 58.) 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 (patrz sposób 37.), 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.

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. 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.

```
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
```

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

```
numbers = [(1963309, 2265973), (2030677, 3814172),
           (1551645, 2229620), (2039045, 2020802)]
start = time()
results = list(map(gcd, numbers))
end = time()
print('Operacja zabrała %.3f sekund' % (end - start))
>>>
Operacja zabrała 1.170 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).

```
start = time()
pool = ThreadPoolExecutor(max_workers=2)
results = list(pool.map(gcd, numbers))
end = time()
print('Operacja zabrała %.3f sekund' % (end - start))
>>>
Operacja zabrała 1.199 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.

```
start = time()
pool = ProcessPoolExecutor(max_workers=2)  # Jedyna zmiana w kodzie.
results = list(pool.map(gcd, numbers))
end = time()
print('Operacja zabrała %.3f sekund' % (end - start))
>>>
Operacja zabrała 0.663 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` (patrz sposób 44.).
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. 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`.

Skorowidz

G

H

I

J

K

L

Ł

M

N

Z

Poznaj najlepsze praktyki programowania w Pythonie!

Python to jeden z najstarszych używanych języków programowania. Co ciekawe, jego nazwa wcale nie pochodzi od zwierzęcia, a od popularnego serialu komediowego. Język ten daje programistom ogromne pole do popisu, a ponadto posiada sporo bibliotek realizujących najbardziej wymyślne zadania. Z uwagi na te atuty rozpoczęcie programowania w tym języku nie powinno przysporzyć Ci większych problemów. Jeżeli jednak chcesz robić to efektywnie, potrzebujesz tej książki.

Sięgnij po nią i poznaj 59 sposobów na tworzenie lepszego kodu w Pythonie! W kolejnych rozdziałach znajdziesz bezcenne informacje na temat programowania zgodnego z duchem Pythona, funkcji, klas i dziedziczenia oraz metaklas i atrybutów. Dalsze strony zawierają przydatną wiedzę na temat wątków i współbieżności, wbudowanych modułów oraz sposobów zarządzania kodem. Książka ta sprawdzi się w rękach każdego programisty pracującego w języku Python. Warto ją mieć!

Dzięki tej książce:

- poznasz zasady programowania zgodnego z duchem Pythona
- opanujesz zagadnienia związane z wątkami i ze współbieżnością
- dostarczysz stabilne API dzięki pakietom
- wykorzystasz potencjał języka Python

Brett Slatkin — starszy inżynier oprogramowania w Google, współzałożyciel i lider w Google Consumer Surveys. Brał udział w pracach nad infrastrukturą Google App Engine dla języka Python. Wykorzystał potencjał języka Python do zarządzania serwerami Google. Ukończył Uniwersytet Columbia w Nowym Jorku.

Addison-Wesley

Helion

36527 numer katalogowy
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ
KOD KORZYŚCI

ISBN 978-83-283-1540-2
9 788328 315402

cena: 49,00 zł

ALWAYS LEARNING

PEARSON