O'REILLY

Wydanie II

Wysoko wydajny Python

Efektywne programowanie w praktyce

Micha Gorelick
Ian Ozsvald

Tytuł oryginału: High Performance Python: Practical Performant Programming for Humans, 2nd Edition

Tłumaczenie: Piotr Pilch

ISBN: 978-83-283-7184-2

© 2021 Helion SA

Authorized Polish translation of the English edition of High Performance Python, 2nd Edition ISBN 9781492055020 © 2020 Micha Gorelick and Ian Ozsvald

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

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

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

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

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

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pytps2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

Słowo wstępne **11**

Przedmowa **13**

1. Wydajny kod Python **19**

Podstawowy system komputerowy 19
Jednostki obliczeniowe 20
Jednostki pamięci 23
Warstwy komunikacji 26
Łączenie ze sobą podstawowych elementów 27
Porównanie wyidealizowanego przetwarzania z maszyną wirtualną języka Python 27
Dlaczego warto używać języka Python? 31
Jak zostać bardzo wydajnym programistą? 34
Sprawdzone praktyki 35
Wnioski dotyczące sprawdzonych praktyk korzystania z rozszerzenia Jupyter Notebook 37
Niech praca znów sprawia radość 38

2. Użycie profilowania do znajdowania wąskich gardeł **39**

Efektywne profilowanie 40
Wprowadzenie do zbioru Julii 41
Obliczanie pełnego zbioru Julii 45
Proste metody pomiaru czasu — instrukcja print i dekorator 48
Prosty pomiar czasu za pomocą polecenia time systemu Unix 51
Użycie modułu cProfile 52
Użycie narzędzia snakeviz do wizualizacji danych wyjściowych modułu cProfile 57
Użycie narzędzia line_profiler do pomiarów dotyczących kolejnych wierszy kodu 57
Użycie narzędzia memory_profiler do diagnozowania wykorzystania pamięci 63
Introspekcja istniejącego procesu za pomocą narzędzia py-spy 68

Kup ksi k Pole ksi k

Kod bajtowy od podszewki 69
Użycie modułu dis do sprawdzenia kodu bajtowego narzędzia CPython 69
Różne metody, różna złożoność 71
Testowanie jednostkowe podczas optymalizacji w celu zachowania poprawności 73
Dekorator @profile bez operacji 73
Strategie udanego profilowania kodu 76
Podsumowanie 77

3. Listy i krotki 79
Bardziej efektywne wyszukiwanie 82
Porównanie list i krotek 84
Listy jako tablice dynamiczne 85
Krotki w roli tablic statycznych 88
Podsumowanie 89

4. Słowniki i zbiory 91
Jak działają słowniki i zbiory? 94
Wstawianie i pobieranie 94
Usuwanie 97
Zmiana wielkości 98
Funkcje mieszania i entropia 99
Słowniki i przestrzenie nazw 102
Podsumowanie 105

5. Iteratory i generatory 107
Iteratory dla szeregów nieskończonych 111
Wartościowanie leniwe generatora 112
Podsumowanie 116

6. Obliczenia macierzowe i wektorowe 117
Wprowadzenie do problemu 118
Czy listy języka Python są wystarczająco dobre? 122
Problemy z przesadną alokacją 123
Fragmentacja pamięci 126
Narzędzie perf 128
Podejmowanie decyzji z wykorzystaniem danych wyjściowych narzędzia perf 131
Wprowadzenie do narzędzia numpy 132
Zastosowanie narzędzia numpy w przypadku problemu dotyczącego dyfuzji 135
Przydziały pamięci i operacje wewnętrzne 138
Optymalizacje selektywne: znajdowanie tego, co wymaga poprawienia 141
Moduł numexpr: przyspieszanie i upraszczanie operacji wewnętrznych 143
Przestroga: weryfikowanie „optymalizacji” (biblioteka scipy) 146

Kup ksi k Pole ksi k

Wnioski z optymalizacji macierzy 148
Narzędzie Pandas 150
Model wewnętrzny narzędzia Pandas 150
Zastosowanie funkcji dla wielu wierszy danych 152
Budowanie struktur DataFrame i szeregów z wyników częściowych, a nie przez łączenie 159
Zadanie może zostać zrealizowane na więcej niż jeden sposób (i być może szybszy) 160
Rada dotycząca efektywnego projektowania z wykorzystaniem narzędzia Pandas 161
Podsumowanie 163

7. Kompilowanie do postaci kodu C 165
Jakie wzrosty szybkości są możliwe? 166
Porównanie kompilatorów JIT i AOT 168
Dlaczego informacje o typie ułatwiają przyspieszenie działania kodu? 168
Użycie kompilatora kodu C 169
Analiza przykładu zbioru Julii 170
Cython 170
Kompilowanie czystego kodu Python za pomocą narzędzia Cython 171
pyximport 173
Użycie adnotacji kompilatora Cython do analizowania bloku kodu 173
Dodawanie adnotacji typu 175
Cython i numpy 179
Przetwarzanie równoległe rozwiązania na jednym komputerze z wykorzystaniem interfejsu OpenMP 181
Numba 183
Użycie narzędzia Numba do kompilacji kodu NumPy dla narzędzia Pandas 185
PyPy 186
Różnice związane z czyszczeniem pamięci 187
Uruchamianie interpretera PyPy i instalowanie modułów 188
Zestawienie wzrostów szybkości 189
Kiedy stosować poszczególne technologie? 190
Inne przyszłe projekty 192
Procesory graficzne (GPU) 193
Grafy dynamiczne: PyTorch 193
Podstawowe profilowanie procesora graficznego 196
Elementy wydajności procesorów graficznych 197
Kiedy stosować procesory graficzne? 199

Kup ksi k Pole ksi k

Interfejsy funkcji zewnętrznych 201
ctypes 202
cffi 204
f2py 206
Moduł narzędzia CPython 209
Podsumowanie 212

8. Asynchroniczne operacje wejścia-wyjścia 215
Wprowadzenie do programowania asynchronicznego 217
Jak działają funkcja async i instrukcja await? 219
Przeszukiwacz szeregowy 220
gevent 222
tornado 226
aiohttp 229
Wspólne obciążenie procesora i urządzeń wejścia-wyjścia 232
Proces szeregowy 233
Przetwarzanie wsadowe wyników 234
Pełna asynchronizacja 237
Podsumowanie 240

9. Moduł multiprocessing 243
Moduł multiprocessing 246
Przybliżenie liczby pi przy użyciu metody Monte Carlo 248
Przybliżanie liczby pi za pomocą procesów i wątków 249
Zastosowanie obiektów języka Python 250
Zastępowanie modułu multiprocessing biblioteką Joblib 256
Liczby losowe w systemach przetwarzania równoległego 260
Zastosowanie narzędzia numpy 261
Znajdowanie liczb pierwszych 264
Kolejki zadań roboczych 270
Weryfikowanie liczb pierwszych za pomocą komunikacji międzyprocesowej 274
Rozwiązanie z przetwarzaniem szeregowym 279
Rozwiązanie z prostym obiektem Pool 280
Rozwiązanie z bardzo prostym obiektem Pool dla mniejszych liczb 281
Użycie obiektu Manager.Value jako flagi 282
Użycie systemu Redis jako flagi 284
Użycie obiektu RawValue jako flagi 286
Użycie modułu mmap jako flagi 287
Użycie modułu mmap do odtworzenia flagi 288
Współużytkowanie danych narzędzia numpy za pomocą modułu multiprocessing 290

Kup ksi k Pole ksi k

Synchronizowanie dostępu do zmiennych i plików 297
Blokowanie plików 297
Blokowanie obiektu Value 300
Podsumowanie 303

10. Klastry i kolejki zadań 305
Zalety klastrowania 306
Wady klastrowania 307
Strata o wartości 462 milionów dolarów na giełdzie Wall Street z powodu kiepskiej strategii aktualizacji klastra 309
24-godzinny przestój usługi Skype w skali globalnej 309
Typowe projekty klastrowe 310
Metoda rozpoczęcia tworzenia rozwiązania klastrowego 311
Sposoby na uniknięcie kłopotów podczas korzystania z klastrów 312
Dwa rozwiązania klastrowe 313
Użycie modułu IPython Parallel do obsługi badań 314
Operacje równoległe narzędzia Pandas z wykorzystaniem biblioteki Dask 316
Użycie systemu NSQ dla niezawodnych klastrów produkcyjnych 321
Kolejki 321
Publikator/subskrybent 322
Rozproszone obliczenia liczb pierwszych 324
Inne warte uwagi narzędzia klastrowania 328
Docker 329
Wydajność Dockera 329
Zalety Dockera 332
Podsumowanie 334

11. Mniejsze wykorzystanie pamięci RAM 335
Obiekty typów podstawowych są kosztowne 336
Moduł array zużywa mniej pamięci do przechowywania wielu obiektów typu podstawowego 337
Mniejsze zużycie pamięci RAM w bibliotece NumPy dzięki narzędziu NumExpr 340
Analiza wykorzystania pamięci RAM w kolekcji 343
Bajty i obiekty Unicode 345
Efektywne przechowywanie zbiorów tekstowych w pamięci RAM 346
Zastosowanie metod dla 11 milionów tokenów 347
Modelowanie większej ilości tekstu za pomocą narzędzia FeatureHasher biblioteki scikit-learn 355
Wprowadzenie do narzędzi DictVectorizer i FeatureHasher 356
Porównanie narzędzi DictVectorizer i FeatureHasher w wypadku rzeczywistego problemu 358

Kup ksi k Pole ksi k

Macierze rzadkie biblioteki SciPy 360
Wskazówki dotyczące mniejszego wykorzystania pamięci RAM 363
Probabilistyczne struktury danych 363
Obliczenia o bardzo dużym stopniu przybliżenia z wykorzystaniem jednobajtowego licznika Morrisa 365
Wartości k-minimum 367
Filtry Blooma 371
Licznik LogLog 376
Praktyczny przykład 380

12. Rady specjalistów z branży ... 385
Usprawnianie potoków inżynierii cech za pomocą biblioteki Feature-engine 385
Inżynieria cech w przypadku uczenia maszynowego 386
Trudne zadanie wdrażania potoków inżynierii cech 386
Wykorzystanie możliwości bibliotek open source języka Python 387
Biblioteka Feature-engine usprawnia budowanie i wdrażanie potoków inżynierii cech 388
Ułatwienie adaptacji nowego pakietu open source 389
Projektowanie, utrzymywanie i zachęcanie do uczestnictwa w rozwoju bibliotek open source 390
Bardzo wydajne zespoły danologów 391
Ile to potrwa? 392
Poznawanie i planowanie 392
Zarządzanie oczekiwaniami i dostarczeniem produktu 393
Numba 395
Prosty przykład 395
Najlepsze praktyki i zalecenia 397
Uzyskiwanie pomocy 400
Optymalizowanie a myślenie 401
Narzędzie Social Media Analytics (SoMA) firmy Adaptive Lab (2014) 403
Język Python w firmie Adaptive Lab 404
Projekt narzędzia SoMA 404
Zastosowana metodologia projektowa 405
Serwisowanie systemu SoMA 405
Rada dla inżynierów z branży 406
Technika głębokiego uczenia prezentowana przez firmę RadimRehurek.com (2014) 406
Strzał w dziesiątkę 407
Rady dotyczące optymalizacji 409
Podsumowanie 411

Kup ksi k Pole ksi k

Uczenie maszynowe o dużej skali gotowe do zastosowań produkcyjnych w firmie Lyst.com (2014) 412
Projekt klastra 412
Ewolucja kodu w szybko rozwijającej się nowej firmie 412
Budowanie mechanizmu rekomendacji 413
Raportowanie i monitorowanie 413
Rada 414
Analiza serwisu społecznościowego o dużej skali w firmie Smesh (2014) 414
Rola języka Python w firmie Smesh 414
Platforma 415
Dopasowywanie łańcuchów w czasie rzeczywistym z dużą wydajnością 415
Raportowanie, monitorowanie, debugowanie i wdrażanie 417
Interpreter PyPy zapewniający powodzenie systemów przetwarzania danych i systemów internetowych (2014) 418
Wymagania wstępne 419
Baza danych 419
Aplikacja internetowa 420
Mechanizm OCR i tłumaczenie 420
Dystrybucja zadań i procesy robocze 421
Podsumowanie 421
Kolejki zadań w serwisie internetowym Lanyrd.com (2014) 421
Rola języka Python w serwisie Lanyrd 422
Zapewnianie odpowiedniej wydajności kolejki zadań 423
Raportowanie, monitorowanie, debugowanie i wdrażanie 423
Rada dla programistów z branży 423

Kup ksi k Pole ksi k

Kup ksi k Pole ksi k

ROZDZIAŁ 1.
Wydajny kod Python

Pytania, na jakie będziesz w stanie udzielić odpowiedzi po przeczytaniu rozdziału

- Jakie są składniki architektury komputerowej?
- Jakie są typowe alternatywne architektury komputerowe?
- Jak w języku Python przeprowadzana jest abstrakcja bazowej architektury komputerowej?
- Jakie są przeszkody na drodze do uzyskania wydajnego kodu Python?
- Jakie strategie mogą ułatwić zostanie programistą tworzącym bardzo wydajny kod?

Tworzenie oprogramowania dla komputerów można zobrazować jako proces przemieszczania porcji danych i przekształcania ich przy użyciu specjalnych sposobów, aby osiągnąć konkretny rezultat. Działania te wiążą się jednak z kosztem w postaci czasu. Oznacza to, że tworzenie *oprogramowania o dużej wydajności* polega na minimalizacji opisanych działań przez redukowanie powodowanego przez nie obciążenia (tj. przez pisanie bardziej wydajnego kodu) lub zmianę metody wykonywania działań w celu sprawienia, że każde z nich będzie bardziej konstruktywne (tj. znajdowanie bardziej odpowiedniego algorytmu).

Skoncentrujmy się na zredukowaniu obciążenia powodowanego przez kod, aby bliżej zaznajomić się z samymi urządzeniami, które są wykorzystywane do przemieszczania porcji danych. Może się wydawać, że będzie to daremny trud, ponieważ w języku Python w celu ukrycia bezpośrednich interakcji ze sprzętem w dużym zakresie stosuje się abstrakcję. Zrozumienie najlepszej metody przemieszczania bitów danych w urządzeniach oraz sposobów wymuszania przemieszczania bitów przez abstrakcje w języku Python pozwoli jednak na pisanie w tym języku lepszych programów o dużej wydajności.

Podstawowy system komputerowy

Opis bazowych elementów tworzących komputery można uprościć przez zaklasyfikowanie ich do trzech podstawowych grup: jednostek obliczeniowych, jednostek pamięci i połączeń między pierwszymi dwiema grupami. Każda z tych grup ma różne cechy, które umożliwiają jej zrozumienie.

Kup ksi k Pole ksi k

Jednostka obliczeniowa cechuje się liczbą obliczeń, jakie może wykonać w ciągu sekundy. Jednostka pamięci wyróżnia się ilością danych, jakie może przechowywać, a także szybkością odczytu i zapisu danych. Z kolei połączenia są określone przez to, jak szybko mogą przemieszczać dane z jednego miejsca w drugie.

Skoro wymieniliśmy już bloki konstrukcyjne, możemy omówić standardową stację roboczą na wielu poziomach zaawansowania. Taka stacja robocza może zawierać na przykład centralną jednostkę obliczeniową CPU (ang. *Central Processing Unit*) połączoną z pamięcią RAM (ang. *Random Access Memory*) i dyskiem twardym jako dwiema osobnymi jednostkami pamięci (każda z nich cechuje się różnymi pojemnościami oraz szybkościami odczytu/zapisu) oraz magistralę, która zapewnia połączenia między wszystkimi tymi komponentami. Można to jednak bardziej sprecyzować. Okaże się, że sam procesor CPU zawiera kilka jednostek pamięci. Są to pamięci podręczne L1, L2, a czasem nawet L3 i L4, które są bardzo szybkie, choć mają niewielkie pojemności (pojemność wynosi od kilku kilobajtów do tuzina megabajtów). Co więcej, nowe architektury komputerowe oferują zwykle nowe konfiguracje (np. w przypadku procesorów SkyLake firmy Intel magistralę FSB zastąpiono technologią Intel Ultra Path Interconnect, a ponadto przebudowano wiele połączeń). W obu przedstawionych ogólnych wariantach stacji roboczych nie doceniono znaczenia połączenia sieciowego, które może być bardzo wolne, z potencjalnymi wieloma innymi jednostkami obliczeniowymi i jednostkami pamięci!

Aby ułatwić rozwikłanie tych różnych zawiłości, dokonajmy krótkiego przeglądu wymienionych podstawowych bloków.

Jednostki obliczeniowe

Jednostka obliczeniowa komputera w największym stopniu stanowi o jego przydatności. Zapewnia ona możliwość przekształcenia dowolnych odebranych bitów w inne bity lub zmiany stanu bieżącego procesu. Jednostki CPU to najpowszechniej używane jednostki obliczeniowe. Graficzne jednostki obliczeniowe GPU (ang. *Graphics Processing Unit*), które pierwotnie były zwykle wykorzystywane do przyspieszania grafiki komputerowej, a obecnie coraz częściej są stosowane na potrzeby aplikacji numerycznych, zyskują jednak na popularności. Wynika to z ich naturalnych możliwości przetwarzania równoległego, które pozwala na jednoczesne przeprowadzenie wielu obliczeń. Niezależnie od typu jednostka obliczeniowa pobiera zestaw bitów (np. reprezentujących liczby) i zwraca kolejny zestaw bitów (np. reprezentujących sumę tych liczb). Oprócz podstawowych operacji arytmetycznych na liczbach całkowitych i rzeczywistych oraz operacji bitowych na liczbach binarnych niektóre jednostki obliczeniowe zapewniają też bardzo specjalistyczne operacje. Przykładem jest operacja FMA (ang. *Fused Multiply Add*), która pobiera trzy liczby `A`, `B` i `C`, a następnie zwraca wartość `A * B + C`.

Podstawowe interesujące nas cechy jednostki obliczeniowej to liczba operacji, jaką może ona wykonać w jednym cyklu, a także liczba cykli możliwych do zrealizowania w ciągu sekundy. Pierwsza wartość jest mierzona za pomocą instrukcji przypadających na cykl (IPC — ang. *Instructions Per Cycle*)[1], natomiast druga wartość jest określana przy użyciu szybkości zegara jednostki. Podczas projektowania

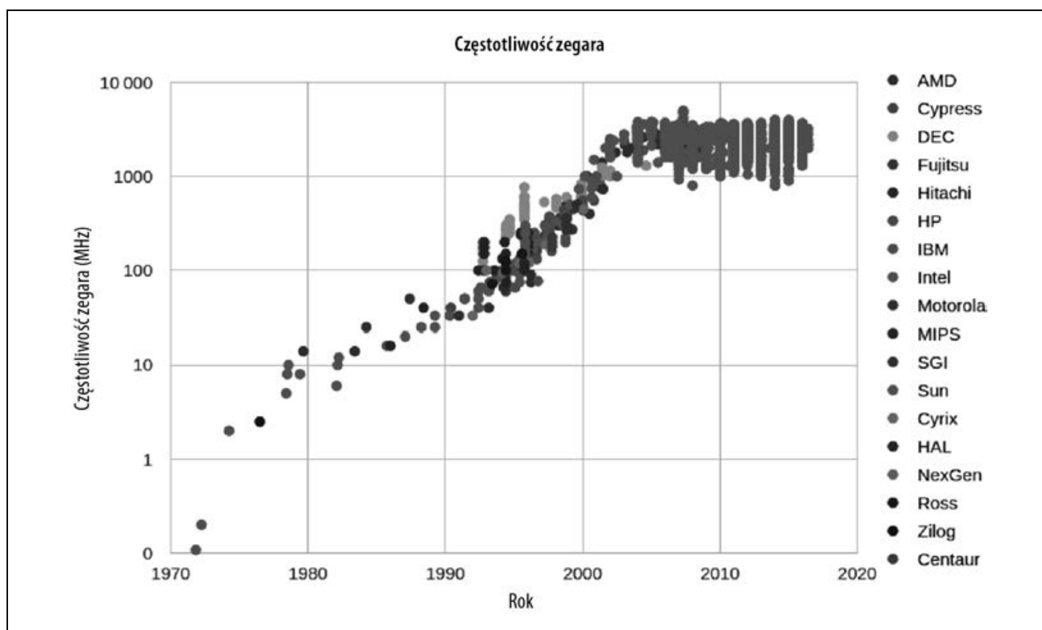
[1] Nie należy mylić z komunikacją międzyprocesową, w przypadku której używany jest taki sam skrót. Zagadnienie to zostanie omówione w rozdziale 9.

Kup ksi k Pole ksi k

nowych jednostek obliczeniowych tym dwóm parametrom zawsze towarzyszy rywalizacja. Na przykład procesory z serii Intel Core mają bardzo wysoką wartość IPC, ale mniejszą szybkość zegara. W przypadku układu Pentium 4 wygląda to odwrotnie. Z kolei jednostki GPU cechują się bardzo wysoką wartością IPC i szybkością zegara, ale dotyczą ich inne problemy, takie jak powolna komunikacja, której więcej miejsca poświęcono w punkcie „Warstwy komunikacji" w tym rozdziale.

Choć zwiększanie szybkości zegara powoduje prawie natychmiastowe przyspieszenie wszystkich programów działających na danej jednostce obliczeniowej (ze względu na możliwość wykonania wciągu sekundy większej liczby obliczeń), wyższa wartość IPC może też w znacznym stopniu wpłynąć na przetwarzanie przez zmianę możliwego poziomu *wektoryzacji*. Wektoryzacja ma miejsce wtedy, gdy jednostka CPU otrzymuje jednocześnie wiele porcji danych i ma możliwość działania na nich wszystkich w tym samym czasie. Tego rodzaju instrukcja procesora określana jest mianem instrukcji SIMD (ang. *Single Instruction, Multiple Data*).

Ogólnie rzecz biorąc, w ciągu minionej dekady jednostki obliczeniowe były dość wolno rozwijane (rysunek 1.1). Szybkości zegara i wartość IPC nie zmieniały się znacząco z powodu fizycznych ograniczeń związanych z wytwarzaniem coraz mniejszych tranzystorów. W efekcie producenci układów bazowali na innych metodach zwiększania szybkości, w tym na wielowątkowości współbieżnej (wiele wątków może działać jednocześnie), bardziej inteligentnym wykonywaniu nieuporządkowanym i architekturach wielordzeniowych.



Rysunek 1.1. Zmiana szybkości zegara procesorów z upływem czasu (dane pochodzą z serwisu CPU DB (http://cpudb.stanford.edu/))[2]

2 Kolorową wersję wykresu znajdziesz na serwerze FTP pod adresem *ftp://ftp.helion.pl/przyklady/pytps2.zip — przyp. red.*

Kup ksi k Pole ksi k

Wielowątkowość współbieżna zapewnia systemowi operacyjnemu hosta drugi wirtualny procesor CPU. Bardziej inteligentna logika sprzętowa próbuje przeplatać dwa wątki instrukcji w jednostkach wykonawczych jednego procesora CPU. W przypadku powodzenia operacji możliwe jest osiągnięcie wzrostu wydajności w porównaniu z pojedynczym wątkiem. Wzrost ten wynosi nawet 30%. Zwykle sprawdza się to, gdy jednostki robocze w obu wątkach korzystają z różnych typów jednostki wykonawczej (na przykład jeden wątek wykonuje operacje zmiennoprzecinkowe, a drugi wątek realizuje operacje całkowitoliczbowe).

Wykonywanie nieuporządkowane umożliwia kompilatorowi zidentyfikowanie wybranych części liniowych sekwencji programu, które nie są zależne od wyniku poprzedniego zadania roboczego. Dzięki temu dwa zadania robocze mogą wystąpić w dowolnej kolejności lub jednocześnie. Dopóki wyniki sekwencyjne są prezentowane w odpowiednim momencie, program poprawnie kontynuuje wykonywanie, nawet pomimo tego, że zadania robocze są przetwarzane bez zachowania kolejności określonej programowo. Umożliwia to wykonanie niektórych instrukcji, gdy inne mogą być blokowane (podczas oczekiwania na dostęp do pamięci). Dzięki temu możliwe jest lepsze ogólne wykorzystanie dostępnych zasobów.

Z punktu widzenia programisty tworzącego kod na wyższym poziomie najważniejsza jest wszechobecność architektur wielordzeniowych. Uwzględniają one wiele procesorów w tej samej jednostce. Zwiększa to ogólne możliwości bez zbliżania się do ograniczeń związanych z przyspieszaniem każdej jednostki z osobna. Z tego właśnie powodu trudno obecnie znaleźć jakikolwiek komputer wyposażony w mniej niż dwa rdzenie (w tym przypadku komputer zawiera dwie fizyczne jednostki obliczeniowe, które są ze sobą połączone). Choć powoduje to zwiększenie łącznej liczby operacji *możliwych* do wykonania w ciągu sekundy, może sprawić, że tworzenie kodu będzie trudniejsze!

Zwykłe dodanie większej liczby rdzeni do procesora nie zawsze powoduje skrócenie czasu wykonania programu. Wynika to z czegoś, co określane jest mianem *prawa Amdahla*. Mówiąc wprost, prawo to głosi, że jeśli program zaprojektowany do działania w wielu rdzeniach zawiera podprocedury, które wymagają uruchomienia tylko w jednym rdzeniu, będzie to ograniczenie dla maksymalnego przyspieszenia możliwego do osiągnięcia przez przydzielenie większej liczby rdzeni.

Jeśli na przykład miałaby zostać przeprowadzona ankieta ze stoma osobami, której wypełnienie zajęłoby minutę, zadanie to mogłoby zostać ukończone w ciągu 100 minut, gdyby pytania zadawała jedna osoba (czyli osoba ta udałaby się do uczestnika nr 1, zadała mu pytania, poczekała na odpowiedzi, a następnie udała się do uczestnika nr 2). Analogią do takiego wariantu przeprowadzania ankiety z jedną osobą zadającą pytania i czekającą na odpowiedzi jest proces szeregowy. W przypadku takich procesów operacje są realizowane po jednej naraz. Każda operacja czeka na zakończenie poprzedniej operacji.

Możliwe byłoby jednak przeprowadzenie ankiety w sposób równoległy, gdyby pytania były zadawane przez dwie osoby. Pozwoliłoby to zakończyć cały proces w zaledwie 50 minut. Jest to możliwe, ponieważ każda osoba zadająca pytania nie musi niczego wiedzieć o drugiej osobie, która zadaje pytania. W rezultacie zadanie z łatwością może zostać podzielone bez istnienia żadnej zależności między osobami zadającymi pytania.

Dodanie większej liczby osób zadających pytania zapewni dodatkowe skrócenie czasu. Będzie tak do momentu zaangażowania stu osób zadających pytania. W tym momencie cały proces zająłby minutę

Kup ksi k Pole ksi k

i byłby ograniczony jedynie przez czas, jaki zajmie uczestnikowi udzielenie odpowiedzi. Dodanie większej liczby osób, które zadają pytania, w żadnym stopniu nie przyspieszy dodatkowo procesu, ponieważ nie będą one miały żadnych zadań do wykonania — wszystkim uczestnikom są już zadawane pytania! Na tym etapie jedyną metodą skrócenia ogólnego czasu przeprowadzania ankiety jest zredukowanie czasu, jaki zajmuje wypełnienie jednej ankiety (część problemu związana z procesem szeregowym). Podobnie jest w przypadku procesorów. W razie potrzeby możliwe jest dodanie większej liczby rdzeni, które mogą zajmować się różnymi zadaniami obliczeniowymi. Można to robić do momentu, w którym pojawi się wąskie gardło w postaci konkretnego rdzenia kończącego swoje zadanie. Innymi słowy, wąskie gardło w dowolnym przetwarzaniu równoległym zawsze ma postać mniejszych zadań szeregowych, które są rozdzielane.

Co więcej, poważną przeszkodą związaną z wykorzystaniem wielu rdzeni w kodzie Python jest stosowanie przez język Python *globalnej blokady interpretera GIL* (ang. *Global Interpreter Lock*). Blokada powoduje, że proces Python może naraz uruchomić tylko jedną instrukcję, niezależnie od liczby aktualnie używanych rdzeni. Oznacza to, że nawet pomimo tego, że część kodu Python ma w tym samym czasie dostęp do wielu rdzeni, w danej chwili instrukcja kodu Python jest wykonywana tylko przez jeden rdzeń. W przypadku zaprezentowanego wcześniej przykładu ankiety oznaczałoby to, że jeśli nawet zatrudniono by stu ankieterów, w danym momencie tylko jeden z nich mógłby zadać pytanie i wysłuchać odpowiedzi. Powoduje to utratę wszelkiego rodzaju korzyści wynikających z zaangażowania wielu ankieterów! Choć może to wyglądać na sporą przeszkodę, zwłaszcza biorąc pod uwagę to, że obecnym trendem w przetwarzaniu jest stosowanie wielu jednostek obliczeniowych, a nie szybszych jednostek, problemu tego można uniknąć dzięki wykorzystaniu innych standardowych narzędzi biblioteki (na przykład narzędzia `multiprocessing` omówionego w rozdziale 9.), technologii (na przykład `numpy` i `numexpr` zaprezentowanych w rozdziale 6. lub Cython objaśnionego w rozdziale 7.) albo rozproszonych modeli obliczeniowych (rozdział 10.).

W wypadku języka Python 3.2 dokonano też poważnej *przebudowy blokady GIL* (*https://mail.python.org/pipermail/python-dev/2009-October/093321.html*). Dzięki temu system stał się znacznie bardziej efektywny, ograniczając wiele problemów towarzyszących wydajności systemu z zastosowaną jednowątkowością. Choć nadal kod Python jest blokowany tak, że w danej chwili działa tylko jedna instrukcja, blokada GIL lepiej radzi sobie obecnie z przełączaniem instrukcji, generując przy tym mniejsze obciążenie.

Jednostki pamięci

Jednostki pamięci w komputerach są używane do przechowywania bitów. Mogą to być bity reprezentujące zmienne w programie lub piksele obrazu. A zatem abstrakcja jednostki pamięci dotyczy rejestrów na płycie głównej, a także pamięci RAM i dysku twardego. Podstawową różnicą między wszystkimi tego typu jednostkami pamięci jest szybkość, z jaką są odczytywane lub zapisywane dane. Aby wszystko jeszcze bardziej skomplikować, szybkość odczytu/zapisu w dużym stopniu zależy od sposobu odczytywania danych.

Na przykład większość jednostek pamięci działa znacznie lepiej, gdy wczytuje jedną dużą porcję danych zamiast wielu małych porcji (w odniesieniu do tego używane są pojęcia *odczytu sekwencyjnego* i *danych losowych*). Jeśli dane w takich jednostkach pamięci potraktuje się jak strony w pokaźnej książce, będzie to oznaczać, że większość jednostek pamięci oferuje większą szybkość odczytu/zapisu

Kup ksi k Pole ksi k

podczas przetwarzania książki strona po stronie niż w przypadku ciągłego przeskakiwania od jednej losowej strony do kolejnej. Chociaż generalnie dotyczy to wszystkich jednostek pamięci, skala oddziaływania dla poszczególnych typów jednostek jest diametralnie różna.

Oprócz szybkości odczytu/zapisu jednostki pamięci cechują się też *opóźnieniem*, które można opisać jako czas, jakiego urządzenie potrzebuje na znalezienie używanych danych. W przypadku obracającego się dysku twardego opóźnienie może być duże, ponieważ dysk fizycznie musi osiągnąć zadaną prędkość obrotową, a głowica odczytująca musi przemieścić się do właściwego położenia. Z kolei w przypadku pamięci RAM opóźnienie może być niewielkie, gdyż w całości jest ona urządzeniem typu SS (ang. *Solid State*). Oto krótki opis różnych jednostek pamięci, które są powszechnie spotykane w standardowej stacji roboczej (wymieniono w kolejności rosnącej szybkości odczytu/zapisu)[3]:

Obracający się dysk twardy

Stosowany od dawna magazyn danych zachowywanych nawet po wyłączeniu komputera. Ogólnie rzecz biorąc, oferuje niewielkie szybkości odczytu/zapisu, ponieważ wymaga fizycznego uzyskania prędkości obrotowej i przemieszczenia głowicy. W przypadku wzorców dostępu losowego spada wydajność dysków twardych, ale mają one bardzo dużą pojemność (rzędu 10 terabajtów).

*Dysk twardy SSD (*ang. *Solid State Drive)*

Urządzenie podobne do obracającego się dysku twardego, które oferuje większe szybkości odczytu/zapisu, ale z mniejszą pojemnością (rzędu 1 terabajta).

Pamięć RAM

Używana do przechowywania kodu i danych aplikacji (np. wszystkich wykorzystywanych zmiennych). Choć pamięć RAM cechuje się krótkim czasem odczytu/zapisu, a ponadto sprawdza się dobrze w przypadku wzorców dostępu losowego, generalnie ma ograniczoną pojemność (rzędu 64 gigabajtów).

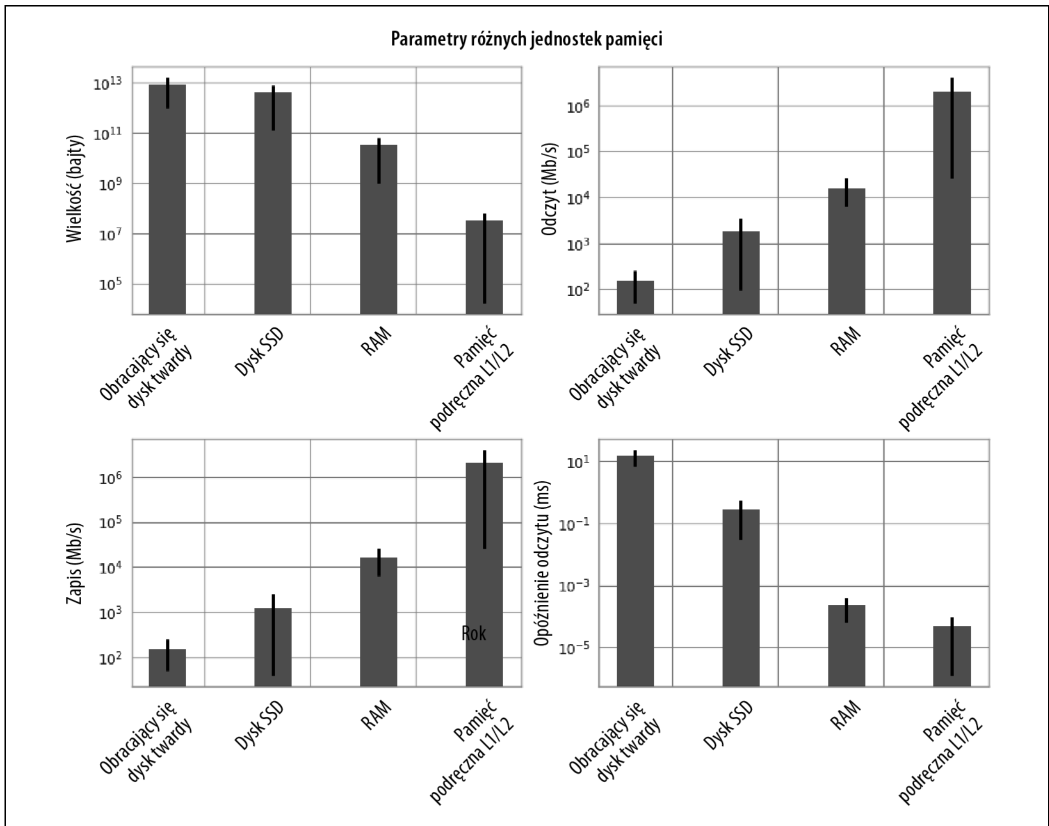
Pamięć podręczna L1/L2

Pamięć oferująca wyjątkowo duże szybkości odczytu/zapisu. Dane kierowane do procesora *muszą* po drodze trafić do tej pamięci. Pamięć ta ma bardzo niewielką pojemność (rzędu megabajtów).

Na rysunku 1.2 przedstawiono graficzną reprezentację różnic między tymi typami jednostek pamięci, analizując parametry dostępnego aktualnie sprzętu konsumenckiego.

[3] Szybkości podane w rozdziale pochodzą ze strony dostępnej pod adresem *https://colin-scott.github.io/personal_website/research/interactive_latency.html*.

Kup ksi k Pole ksi k



Rysunek 1.2. Wartości parametrów dla różnych typów jednostek pamięci (dane pochodzą z lutego 2014 r.)

Widoczny wyraźnie trend pokazuje, że szybkości odczytu/zapisu oraz pojemność są odwrotnie proporcjonalne. Przy zwiększaniu szybkości zmniejsza się pojemność. Z tego powodu wiele systemów stosuje w przypadku pamięci metodę warstwową: na początku dane w całości znajdują się na dysku twardym, ich część przenoszona jest do pamięci RAM, a następnie znacznie mniejszy podzbiór danych trafia do pamięci podręcznej L1/L2. Metoda warstwowa umożliwia programom utrzymywanie pamięci w różnych miejscach w zależności od wymagań dotyczących czasu dostępu. Przy podejmowaniu próby optymalizacji wzorców pamięci programu po prostu optymalizowane jest to, jakie dane są umieszczane w jakim miejscu, w jaki sposób są one rozmieszczane (w celu zwiększenia liczby odczytów sekwencyjnych), a także ile razy dane są przemieszczane między różnymi miejscami. Ponadto metody takie jak asynchroniczne wejście-wyjście i buforowanie z wywłaszczeniem bez konieczności marnowania dodatkowego czasu procesora pozwalają zapewnić, że dane zawsze będą tam, gdzie są wymagane. Większość takich procesów może odbywać się niezależnie podczas wykonywania innych obliczeń!

Kup ksi k Pole ksi k

Warstwy komunikacji

Przyjrzyjmy się jeszcze temu, jak przedstawione wcześniej podstawowe bloki komunikują się ze sobą. Wprawdzie istnieje wiele różnych trybów komunikacji, ale wszystkie są wariantami tego, co jest określane mianem *magistrali*.

Na przykład *magistrala FSB* (ang. *Frontside Bus*) to połączenie między pamięcią RAM i pamięcią podręczną L1/L2. Służy ona do przemieszczania danych gotowych do przekształcenia przez procesor do postaci pozwalającej na rozpoczęcie obliczeń, a także do przesyłania wyników po ich zakończeniu. Istnieją również inne magistrale, takie jak magistrala zewnętrzna pełniąca rolę głównej trasy prowadzącej od urządzeń (np. dyski twarde i karty sieciowe) do procesora i pamięci systemowej. Taka magistrala zewnętrzna jest zwykle wolniejsza od magistrali FSB.

Okazuje się, że wiele zalet pamięci podręcznej L1/L2 może być związanych z szybszą magistralą. Możliwość kolejkowania w wolnej magistrali (między pamięcią podręczną i procesorem) danych wymaganych do obliczeń w postaci dużych porcji, a następnie udostępnianie ich przy bardzo dużych szybkościach z poziomu linii pamięci podręcznej (z tej pamięci do procesora) umożliwia procesorowi wykonanie większej liczby obliczeń bez czekania przez długi czas.

Wiele mankamentów związanych z użyciem układu GPU wynika z korzystania z magistrali, z którą jest on połączony. Ponieważ GPU to zazwyczaj urządzenie peryferyjne, komunikuje się za pośrednictwem magistrali PCI, która jest znacznie wolniejsza niż magistrala FSB. W rezultacie pobieranie danych z układu GPU i wysyłanie ich do niego może być dość obciążającą operacją. Pojawienie się obliczeń heterogenicznych lub bloków obliczeniowych, które zawierają w magistrali FSB układy CPU i GPU, ma na celu zmniejszenie obciążenia związanego z transferem danych oraz zwiększenie możliwości stosowania do obliczeń układu GPU, nawet w sytuacji, gdy konieczne jest przesłanie dużej ilości danych.

Oprócz bloków komunikacji wewnątrz komputera rolę kolejnego takiego bloku może pełnić sieć. W porównaniu z wcześniej omówionymi blokami ten blok jest jednak znacznie bardziej elastyczny. Urządzenie sieciowe może zostać połączone z urządzeniem pamięciowym, takim jak magazyn danych NAS (*Network Attached Storage*), lub z innym blokiem obliczeniowym, tak jak w przypadku węzła obliczeniowego w klastrze. Komunikacja sieciowa jest jednak zwykle znacznie wolniejsza od innych, wcześniej opisanych typów komunikacji. Magistrala FSB może przesyłać dziesiątki gigabitów w ciągu sekundy, sieć natomiast jest ograniczona do transferów rzędu kilkudziesięciu megabitów.

Jasne jest zatem, że podstawową zaletą magistrali jest jej szybkość, która określa, ile danych może zostać przemieszczonych w danym czasie. Cecha ta stanowi połączenie dwóch wielkości: ilości danych przemieszczanych w ramach jednej operacji transferu (szerokość magistrali) i liczby transferów możliwych w ciągu sekundy (częstotliwość magistrali). Godne uwagi jest to, że dane przesyłane w jednym transferze zawsze są sekwencyjne. Porcja danych odczytywana jest z pamięci i przemieszczana w inne miejsce. Oznacza to, że szybkość magistrali jest rozbijana na te dwie wielkości, ponieważ osobno mogą one mieć wpływ na różne aspekty związane z obliczeniami. Duża szerokość magistrali może ułatwić działanie kodu z wektoryzacją (lub dowolnego kodu, który dokonuje sekwencyjnego odczytu z pamięci), umożliwiając przesłanie wszystkich odpowiednich danych w ramach jednej operacji transferu. Z kolei magistrala o małej szerokości, lecz bardzo dużej częstotliwości transferów może ułatwić wykonywanie kodu, który musi wykonywać wiele odczytów z losowych obszarów pamięci. Interesujące jest to, że jedna z metod modyfikowania tych cech przez projektantów

Kup ksi k Pole ksi k

komputerów polega na wprowadzaniu zmian w fizycznym układzie elementów płyty głównej. Gdy układy zostaną umieszczone blisko siebie, krótsze będą fizyczne połączenia między nimi. Pozwala to uzyskać większe szybkości transferów. Ponadto sama liczba połączeń określa szerokość magistrali (dzięki temu to pojęcie zyskuje realne znaczenie).

Ponieważ interfejsy mogą być dostrajane w celu zaoferowania właściwej wydajności na potrzeby konkretnego zastosowania, nie jest zaskoczeniem, że istnieją setki ich typów. Na rysunku 1.3 pokazano szybkości transmisji danych dla próbkowania typowych interfejsów. Zauważ, że informacje te wcale nie dotyczą opóźnienia połączeń, które określa, ile czasu zajmie utworzenie odpowiedzi dla żądania dotyczącego danych (choć opóźnienie w bardzo dużym stopniu zależy od komputera, istnieją zasadnicze ograniczenia, które są powiązane z używanymi interfejsami).

Łączenie ze sobą podstawowych elementów

Zrozumienie, jakie są podstawowe komponenty komputerów, nie wystarczy do pełnego zaznajomienia się z problemami związanymi z programowaniem pod kątem dużej wydajności. Wzajemna zależność wszystkich tych komponentów oraz sposób ich współpracy w celu rozwiązania problemu wprowadzają dodatkowe poziomy złożoności. W tym podrozdziale zostaną omówione uproszczone problemy, które ilustrują, jak działałyby idealne rozwiązania, a także w jaki sposób z problemami radzi sobie język Python.

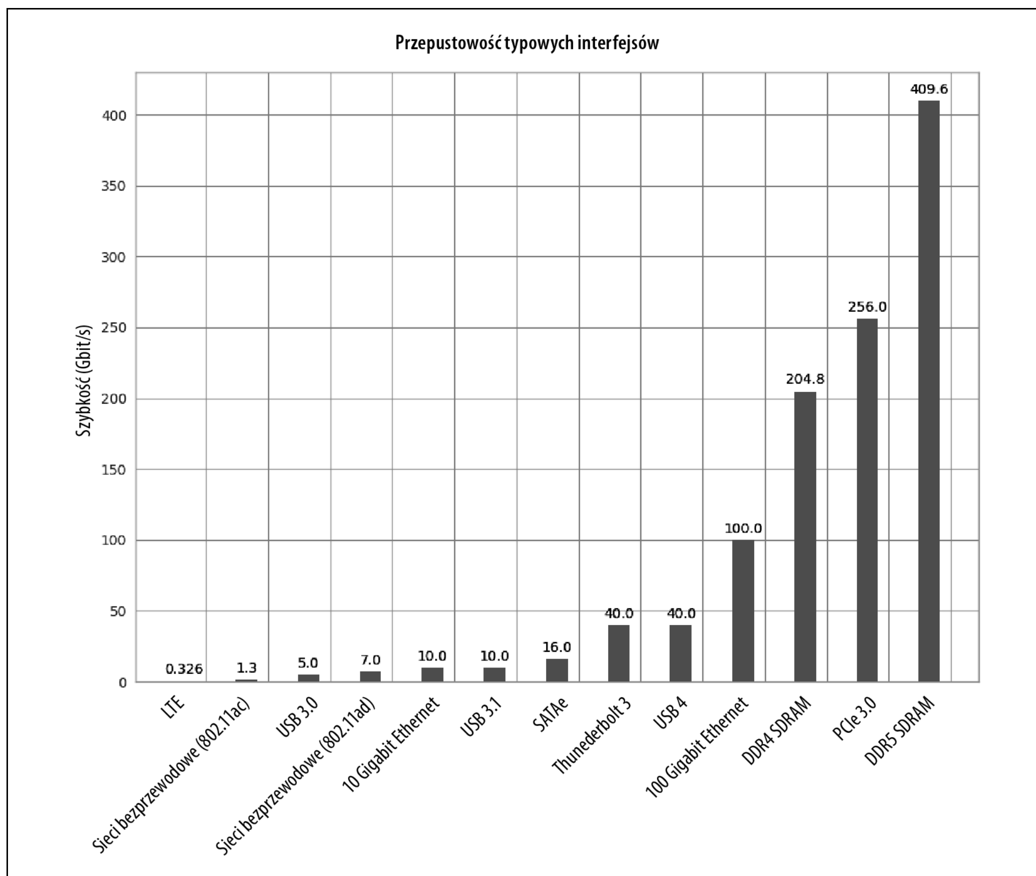
Ostrzeżenie: ten podrozdział może sprawiać wrażenie przygnębiającego — większość uwag wydaje się wskazywać, że język Python z założenia nie jest w stanie poradzić sobie z problemami dotyczącymi wydajności. Nie jest to prawdą z dwóch powodów. Po pierwsze, w przypadku tych wszystkich „elementów wydajnego przetwarzania" nie doceniono bardzo istotnego „czynnika", a mianowicie programisty. To, czego w kwestii wydajności standardowo język Python może być pozbawiony, od razu nadrabia dzięki szybkości tworzenia kodu. Po drugie, w książce zostaną przedstawione moduły i idee, które bez problemu mogą ułatwić ograniczenie skali wielu opisanych tutaj trudności. Po uwzględnieniu obu tych aspektów będziemy w stanie zachować możliwości szybkiego programowania w języku Python przy jednoczesnym usunięciu wielu ograniczeń dotyczących wydajności.

Porównanie wyidealizowanego przetwarzania z maszyną wirtualną języka Python

Aby lepiej zrozumieć szczegóły programowania pod kątem dużej wydajności, przyjrzyjmy się przykładowi prostego kodu, który sprawdza, czy dana liczba to liczba pierwsza:

```
import math
def check_prime(number):
    sqrt_number = math.sqrt(number)
    for i in range(2, int(sqrt_number) + 1):
        if (number / i).is_integer():
            return False
    return True
print(f"check_prime(10,000,000) = {check_prime(10_000_000)}")
# check_prime(10,000,000) = False
print(f"check_prime(10,000,019) = {check_prime(10_000_019)}")
# check_prime(10,000,019) = True
```


Kup ksi k Pole ksi k



Rysunek 1.3. Szybkości połączeń różnych typowych interfejsów[4]

Przeanalizujmy ten kod za pomocą abstrakcyjnego modelu obliczeniowego, a następnie porównajmy z tym, co ma miejsce podczas wykonywania tego kodu przez interpreter języka Python. Podobnie jak w przypadku dowolnej abstrakcji, pominiemy wiele subtelności dotyczących zarówno wyidealizowanego komputera, jak i sposobu wykonywania kodu przez interpreter języka Python. Ogólnie rzecz biorąc, jest to jednak odpowiednie ćwiczenie do wykonania przed rozwiązaniem problemu: pomyśl o ogólnych elementach algorytmu i zastanów się, jaki będzie najlepszy sposób połączenia ze sobą elementów obliczeniowych w celu znalezienia rozwiązania. Zrozumienie takiej idealnej sytuacji i uzyskanie wiedzy o tym, co w rzeczywistości ma miejsce wewnątrz interpretera języka Python, pozwoli w iteracyjny sposób zbliżyć tworzony kod Python do optymalnego kodu.

Wyidealizowane przetwarzanie

Po rozpoczęciu wykonywania kodu w pamięci RAM przechowywana jest wartość zmiennej `number`. W celu obliczenia wartości zmiennej `sqrt_number` konieczne jest wysłanie wartości zmiennej `number` do procesora. W idealnej sytuacji możliwe by było jednokrotne wysłanie wartości, które zostałyby za-

[4] Podane dane pochodzą ze strony dostępnej pod adresem *https://en.wikipedia.org/wiki/List_of_interface_bit_rates*.

Kup ksi k Pole ksi k

pisane w pamięci podręcznej L1/L2 procesora. Procesor przeprowadziłby obliczenia, a następnie wysłał wartości z powrotem do pamięci RAM w celu ich zapisania. Taki scenariusz jest idealny, ponieważ zminimalizowana zostałaby liczba odczytów wartości zmiennej `number` z pamięci RAM, a zamiast tego zdecydowalibyśmy się na opcję odczytów z pamięci podręcznej L1/L2, co jest znacznie szybsze. Co więcej, zminimalizowalibyśmy liczbę transferów danych za pośrednictwem magistrali FSB, używając pamięci podręcznej L1/L2 połączonej bezpośrednio z procesorem.

Taka metoda utrzymywania danych tam, gdzie są potrzebne, oraz maksymalne możliwe ograniczanie ich przesyłania odgrywają bardzo ważną rolę w przypadku optymalizacji. Pojęcie „ciężkich danych” (ang. *heavy data*) odnosi się do czasu i zasobów niezbędnych do przemieszczenia danych, a tego chcielibyśmy uniknąć.

Zamiast w przypadku pętli zawartej w kodzie wysyłać do procesora w danej chwili jedną wartość zmiennej `i`, bardziej pożądane będzie jednoczesne wysłanie do procesora zmiennej `number` oraz *kilku* wartości zmiennej `i` w celu ich sprawdzenia. Jest to możliwe, ponieważ procesor wektoryzuje operacje bez ponoszenia dodatkowego kosztu w postaci czasu. Oznacza to, że w tym samym czasie procesor może wykonywać wiele niezależnych obliczeń. A zatem chcemy wysłać do pamięci podręcznej procesora zmienną `number`, a także taką liczbę wartości zmiennej `i`, jaką może pomieścić ta pamięć. Dla każdej pary zmiennych `number` i `i` zostanie wykonane dzielenie i sprawdzenie, czy wynik jest liczbą całkowitą. Następnie zostanie wysłany z powrotem sygnał wskazujący, czy dowolna z wartości rzeczywiście okazała się liczbą całkowitą. Jeśli tak, funkcja zostanie zakończona. W przeciwnym razie operacja zostanie powtórzona. Dzięki temu dla wielu wartości zmiennej `i` konieczne jest zwrócenie tylko jednego wyniku, co eliminuje uzależnienie od wolnej magistrali dla każdej wartości. W tym przypadku wykorzystywana jest możliwość *wektoryzacji* obliczenia przez procesor lub uruchomienia jednej instrukcji dla wielu danych w jednym cyklu zegarowym.

Pojęcie wektoryzacji zostało zilustrowane w następującym kodzie:

```
import math
def check_prime(number):
    sqrt_number = math.sqrt(number)
    numbers = range(2, int(sqrt_number)+1)
    for i in range(0, len(numbers), 5):
        # poniższy wiersz nie zawiera poprawnego kodu Python
        result = (number / numbers[i:(i + 5)]).is_integer()
        if any(result):
            return False
    return True
```

W kodzie określono obliczenia, w przypadku których dzielenie i sprawdzanie liczb całkowitych realizowane jest jednocześnie dla zestawu pięciu wartości zmiennej `i`. Jeśli dokonano poprawnej wektoryzacji, procesor może wykonać te operacje w jednym kroku, zamiast przeprowadzać osobne obliczenie dla każdej wartości zmiennej `i`. W idealnej sytuacji operacja `any(result)` wystąpiłaby w procesorze bez konieczności przesyłania wyników z powrotem do pamięci RAM. W rozdziale 6. w szerszym zakresie zostanie omówiona wektoryzacja, sposób jej działania, a także to, kiedy przynosi korzyści w kodzie.

Kup ksi k Pole ksi k

Maszyna wirtualna języka Python

Interpreter języka Python wykonuje wiele działań, aby podjąć próbę utworzenia abstrakcji używanych bazowych elementów obliczeniowych. Programista wcale nie musi przejmować się przydzielaniem pamięci tablicom, sposobem zorganizowania tej pamięci lub tym, w jakiej kolejności dane są wysyłane do procesora. Jest to zaleta języka Python, gdyż pozwala skoncentrować się na implementowanych algorytmach. Kosztem tego jest jednak spory spadek wydajności.

Ważne jest zdanie sobie sprawy z tego, że w zasadzie interpreter języka Python to tak naprawdę działający zestaw bardzo dobrze zoptymalizowanych instrukcji. Zastosowany zabieg polega jednak na tym, że w tym języku instrukcje te są wykonywane w odpowiedniej kolejności w celu osiągnięcia lepszej wydajności. Dobrze widać to w poniższym przykładzie, w którym funkcja `search_fast` zadziała szybciej niż funkcja `search_slow`, nawet pomimo tego, że podczas działania obie mają złożoność obliczeniową `O(n)`. Wszystko może jednak skomplikować się przy korzystaniu z typów pochodnych, specjalnych metod języka Python lub modułów zewnętrznych. Czy możesz na przykład od razu stwierdzić, która funkcja będzie szybsza: `search_unknown1` czy `search_unknown2`?

```
def search_fast(haystack, needle):
    for item in haystack:
        if item == needle:
            return True
    return False

def search_slow(haystack, needle):
    return_value = False
    for item in haystack:
        if item == needle:
            return_value = True
    return return_value

def search_unknown1(haystack, needle):
    return any((item == needle for item in haystack))
def search_unknown2(haystack, needle):
    return any([item == needle for item in haystack])
```

Identyfikowanie obszarów kodu o małej wydajności za pomocą profilowania i znajdowania bardziej efektywnych metod przeprowadzania tych samych obliczeń przypomina wyszukiwanie tych bezwartościowych operacji i usuwanie ich. Końcowy rezultat jest identyczny, ale znacznie zmniejszona zostaje liczba obliczeń i transferów danych.

Jednym z efektów użycia takiej warstwy abstrakcji jest to, że wektoryzacja nie jest od razu uzyskiwana. Zamiast połączenia kilku iteracji w przykładzie kodu sprawdzającego początkową liczbę pierwszą dla wartości zmiennej `i` zostanie wykonana jedna iteracja pętli. Gdy jednak przyjrzysz się przykładowi abstrakcji wektoryzacji, stwierdzisz, że nie jest to poprawny kod Python, ponieważ nie jest możliwe dzielenie liczby zmiennoprzecinkowej przy użyciu listy. Zewnętrzne biblioteki, takie jak `numpy`, okażą się pomocne w takiej sytuacji, zapewniając możliwość wykonywania wektoryzowanych operacji matematycznych.

Co więcej, abstrakcja w języku Python ma negatywny wpływ na wszelkie optymalizacje, które bazują na wypełnianiu pamięci podręcznej L1/L2 odpowiednimi danymi na potrzeby następnego obliczenia. Wynika to z wielu czynników, z których najważniejszym jest to, że obiekty Python nie są rozmieszczo-

Kup ksi k Pole ksi k

ne w pamięci w najbardziej optymalny sposób. Jest to konsekwencją tego, że język Python to język dokonujący czyszczenia pamięci (jest ona automatycznie przydzielana i w razie potrzeby uwalniana). Powoduje to fragmentację pamięci, która może niekorzystnie wpłynąć na transfery do pamięci podręcznych procesora. Ponadto w żadnym momencie nie ma możliwości zmiany układu struktury danych bezpośrednio w pamięci. Oznacza to, że jedna operacja transferu w magistrali może nie zawierać wszystkich odpowiednich informacji na potrzeby obliczeń, nawet pomimo tego, że szerokość magistrali może być wystarczająca dla całych danych[5].

Poza tym fundamentalny problem wynika z typów dynamicznych języka Python, którego kod nie jest kompilowany. Jak wielu programistów używających języka C nauczyło się w ciągu wielu lat, kompilator jest czasami sprytniejszy od nas. Podczas kompilowania statycznego kodu kompilator może stosować liczne zabiegi w celu zmiany sposobu rozmieszczania elementów, a także sposobu, w jaki procesor będzie uruchamiać określone instrukcje pod kątem zoptymalizowania ich. Kod Python nie jest jednak kompilowany. Co gorsza, zawiera on typy dynamiczne. Oznacza to, że określanie jakichkolwiek ewentualnych możliwości algorytmicznych optymalizacji jest znacząco trudniejsze, ponieważ funkcjonalność kodu może być modyfikowana podczas jego wykonywania. Istnieje wiele metod zmniejszania skali tego problemu. Podstawową jest użycie narzędzia Cython, które umożliwia kompilowanie kodu Python, a ponadto pozwala użytkownikowi tworzyć „wskazówki" dla kompilatora określające rzeczywistą dynamiczność kodu.

Ponadto przy próbie równoległego wykonywania kodu wspomniana wcześniej globalna blokada interpretera GIL może spowodować spadek wydajności. Dla przykładu załóżmy, że kod jest modyfikowany w celu użycia wielu rdzeni procesora tak, aby każdy z nich otrzymał porcję liczb z zakresu od 2 do `sqrtN`. Każdy rdzeń może przeprowadzić obliczenia dla swojej porcji liczb, a następnie po ich zakończeniu rdzenie mogą porównać własne obliczenia. Choć tracimy możliwość wczesnego zakończenia wykonywania pętli, ponieważ żaden rdzeń nie może stwierdzić, czy rozwiązanie zostało znalezione, możliwe jest zmniejszenie liczby sprawdzeń, jaką każdy rdzeń musi wykonać (oznacza to, że w przypadku `M` rdzeni każdy z nich musiałby przeprowadzić `sqrtN/M` sprawdzeń). Jednak z powodu globalnej blokady interpretera GIL w danej chwili może być używany tylko jeden rdzeń. Oznacza to, że w efekcie zostałby wykonany taki sam kod jak w przypadku wersji kodu z równoległym wykonywaniem, ale bez możliwości wczesnego zakończenia. Problemu tego można uniknąć, zastępując wiele wątków wieloma procesami (z wykorzystaniem modułu `multiprocessing`) bądź za pomocą narzędzia Cython lub funkcji zewnętrznych.

Dlaczego warto używać języka Python?

Język Python cechuje się wysokim stopniem ekspresywności i jest łatwy do opanowania. Programiści, którzy zaczynają go używać, szybko stwierdzają, że w krótkim czasie mogą dzięki niemu osiągnąć całkiem sporo. Za pomocą innych języków zostało napisanych wiele narzędzi opakowujących biblioteki języka Python, które ułatwiają wywoływanie innych systemów. Na przykład system uczenia maszynowego `scikitlearn` opakowuje biblioteki LIBLINEAR i LIBSVM (obie napisano w języku C),

[5] W rozdziale 6. dowiesz się, jak możesz odzyskać nad tym kontrolę i w pełni dostosować kod do wzorców wykorzystania pamięci.

Kup ksi k

Pole ksi k

a biblioteka `numpy` zawiera bibliotekę BLAS oraz inne biblioteki języków C i Fortran. W rezultacie kod Python, który poprawnie wykorzystuje te moduły, może być naprawdę równie szybki jak porównywalny kod C.

Język Python jest określany mianem „uwzględniającego baterie”, gdyż wbudowano w niego wiele ważnych narzędzi i stabilnych bibliotek. Uwzględniają one następujące składniki:

`unicode` *i* `bytes`
: Scalone z rdzeniem języka.

`array`
: Wydajne pod względem wykorzystywanej pamięci tablice przeznaczone dla typów podstawowych.

`math`
: Podstawowe operacje matematyczne, w tym kilka prostych funkcji statystycznych.

`sqlite3`
: Biblioteka opakowująca powszechnie używany mechanizm magazynowania danych oparty na plikach SQLite3.

`collections`
: Przeróżne obiekty, w tym obiekt `deque`, licznik i warianty słownika.

`asyncio`
: Obsługa współbieżności w wypadku zadań powiązanych z operacjami wejścia-wyjścia i korzystających ze składni funkcji `async` i `await`.

Poza obrębem rdzenia języka dostępna jest ogromna liczba różnych bibliotek. Oto niektóre z nich:

`numpy`
: Numeryczna biblioteka języka Python (podstawowa biblioteka w przypadku wszystkiego, co ma związek z macierzami).

`scipy`
: Bardzo duża kolekcja zaufanych bibliotek naukowych, które często opakowują cieszące się dużym uznaniem biblioteki języków C i Fortran.

`pandas`
: Biblioteka służąca do analizy danych, która przypomina ramki danych języka R lub arkusz kalkulacyjny programu Excel. Biblioteka bazuje na bibliotekach `scipy` i `numpy`.

`scikit-learn`
: Bazująca na bibliotece `scipy` biblioteka szybko przyjmująca postać domyślnej biblioteki uczenia maszynowego.

`tornado`
: Biblioteka, która zapewnia proste powiązania na potrzeby współbieżności.

Kup ksi k Pole ksi k

PyTorch i TensorFlow

Oferowane przez firmy Facebook i Google środowiska technologii głębokiego uczenia, które zapewniają obsługę języka Python i procesorów graficznych.

NLTK, SpaCy i Gensim

Biblioteki przetwarzania języka naturalnego z rozbudowaną obsługą języka Python.

Powiązania bazy danych

Służą do komunikacji z niemal wszystkimi bazami danych, w tym Redis, MongoDB, HDF5 i SQL.

Środowiska do projektowania aplikacji internetowych

Wydajne systemy służące do tworzenia witryn internetowych, takie jak `aiohttp`, `django`, `pyramid`, `flask` i `tornado`.

`OpenCV`

Powiązania na potrzeby rozpoznawania obrazów.

Powiązania interfejsów API

Ułatwiają dostęp do popularnych interfejsów API serwisów internetowych, takich jak Google, Twitter i LinkedIn.

Dostosowanie do różnych scenariuszy wdrażania jest możliwe dzięki dużej liczbie dostępnych środowisk zarządzanych i powłok. Oto niektóre z nich:

- Standardowa dystrybucja dostępna pod adresem *http://python.org/*.
- `pipenv`, `pyenv` i `virtualenv` to proste i przenośne środowiska dla języka Python.
- Platforma Docker zapewniająca proste do uruchomienia i powielania środowiska przeznaczone do projektowania i wdrażania produkcyjnego.
- Przeznaczone dla naukowców środowisko Anaconda firmy Anaconda Inc.
- Przypominające oprogramowanie Matlab środowisko Sage, które zawiera zintegrowane środowisko programistyczne IDE (*Integrated Development Environment*).
- IPython, czyli interaktywna powłoka języka Python używana przez naukowców i projektantów.
- Oparte na przeglądarce rozszerzenie powłoki IPython o nazwie Jupyter Notebook, które jest intensywnie wykorzystywane do celów edukacyjnych i pokazowych.

Jedną z głównych zalet języka Python jest to, że pozwala na szybkie prototypowanie koncepcji. Ze względu na bogactwo bibliotek pomocniczych proste jest sprawdzenie, czy koncepcja jest możliwa do zrealizowania (nawet jeśli pierwsza implementacja może okazać się dziwna).

Aby przyspieszyć procedury matematyczne, sprawdź bibliotekę `numpy`. Jeżeli chcesz poeksperymentować z uczeniem maszynowym, wypróbuj bibliotekę `scikit-learn`. Jeśli porządkujesz i modyfikujesz dane, dobrą propozycją będzie biblioteka `pandas`.

Ogólnie rzecz biorąc, sensowne jest zadanie sobie następującego pytania: „Jeśli system działa szybciej, czy jako zespół długoterminowo będziemy pracować wolniej?”. Choć zawsze możliwe jest uzyskanie

Kup ksi k Pole ksi k

wzrostu wydajności systemu, jeśli poświęci się na to wystarczającą ilość czasu przy udziale odpowiedniej liczby osób, może to doprowadzić do uzyskania nieznacznych i źle zrozumianych optymalizacji, które ostatecznie spowodują utrudnienia w pracy zespołu.

Przykładem może być wprowadzenie narzędzia Cython (omówiono je w rozdziale 7., w podrozdziale „Cython"). Jest to oparte na kompilatorze rozwiązanie służące do tworzenia adnotacji kodu Python za pomocą typów podobnych do tych wykorzystywanych w języku C. Dzięki temu przekształcony kod może być kompilowany przy użyciu kompilatora języka C. Wprawdzie przyrost szybkości może robić wrażenie (często przy stosunkowo niewielkim nakładzie pracy osiągane są szybkości porównywalne z szybkością kodu C), ale zwiększy się koszt obsługi takiego kodu. W szczególności trudniejsza może być obsługa nowego modułu, ponieważ od członków zespołu wymagane będzie określone doświadczenie programistyczne, pozwalające zrozumieć niektóre zależności, które wystąpiły po zrezygnowaniu z maszyny wirtualnej języka Python i uzyskaniu wzrostu wydajności.

Jak zostać bardzo wydajnym programistą?

Tworzenie bardzo wydajnego kodu to tylko jeden z elementów decydujących o pomyślnym i wysoce efektywnym realizowaniu projektów w dłuższej perspektywie czasu. Ogólna wydajność zespołu jest znacznie ważniejsza niż uzyskiwanie przyspieszenia i złożone rozwiązania. Kluczem do tego jest kilka elementów, takich jak dobra struktura, dokumentacja, możliwość debugowania i wspólne standardy.

Załóżmy, że tworzysz prototyp. Nie przetestowałeś go szczegółowo, a ponadto nie został on sprawdzony przez zespół. Prototyp wydaje się być „wystarczająco dobry", dlatego kierowany jest do środowiska produkcyjnego. Ponieważ prototyp nigdy nie był tworzony w sposób strukturalny, pozbawiony jest testów i dokumentacji. Niespodziewanie w prototypie pojawiła się porcja powodującego efekt inercji kodu, który ma zostać wykorzystany przez kogoś innego. Często zarząd nie może określić kosztu związanego z pracą zespołu.

Jako że rozwiązanie to jest trudne do utrzymania, nie cieszy się sympatią — nie podlega restrukturyzowaniu, nie jest objęte testami, które ułatwiłyby zespołowi przeprowadzenie refaktoryzacji, a ponadto nikt inny nie zamierza się tym zajmować. W rezultacie za utrzymanie sprawności rozwiązania odpowiada jeden programista. W chwilach stresu może to być przyczyną powstania poważnego wąskiego gardła i spowodować znaczne ryzyko: co się stanie, gdy ten programista przestanie brać udział w projekcie?

Taki styl projektowania jest zwykle praktykowany, gdy zespół zarządzający nie rozumie bieżącego efektu inercji wywoływanego przez trudny do utrzymania kod. Pokazanie tego w testach realizowanych w dłuższej perspektywie oraz dokumentacja mogą ułatwić zespołowi utrzymanie dużej wydajności, a ponadto przekonanie menedżerów do przeznaczenia czasu na „oczyszczenie" takiego kodu prototypu.

W wypadku środowiska badawczego częstą sytuacją jest tworzenie za pomocą rozszerzenia Jupyter Notebook wielu programów, które cechują się kiepskimi praktykami pisania kodu używanymi przy korzystaniu z różnych pomysłów i zbiorów danych. Intencją zawsze jest „napisanie tego lepiej" na późniejszym etapie, który jednak nigdy nie następuje. Ostatecznie uzyskuje się wynik prac, ale brakuje infrastruktury służącej do ich odtwarzania i testowania, a także nie ma się względem niego zaufania. I tym razem czynniki ryzyka są duże, a zaufanie dotyczące wyniku będzie niewielkie.

Kup ksi k

Pole ksi k

Oto elementy ogólnego postępowania, które zapewni korzyści:

Zadbaj o to, aby działało

Najpierw tworzysz wystarczająco dobre rozwiązanie. Bardzo praktyczne jest „utworzenie rozwiązania, które może zostać zmarnowane". Pełni ono rolę rozwiązania prototypowego, które pozwala na uzyskanie lepszej struktury możliwej do zastosowania w drugiej wersji. Zawsze rozsądne jest przeprowadzenie wstępnego planowania przed rozpoczęciem tworzenia kodu. W przeciwnym razie dojdziesz do wniosku, że zaoszczędziłeś godzinę, która miała zostać poświęcona na przemyślenia, a w zamian pisanie kodu zajęło całe popołudnie. W niektórych dziedzinach jest to lepiej znane pod postacią stwierdzenia „zmierz dwa razy, a tnij raz".

Zrób to dobrze

W dalszej kolejności dodajesz rozbudowany zestaw testów wsparty dokumentacją, a także zrozumiałe instrukcje odtwarzania, tak aby mógł z nich skorzystać inny członek zespołu.

Zrób to szybko

Teraz możesz skoncentrować się na profilowaniu i kompilowaniu lub przetwarzaniu równoległym oraz zastosowaniu istniejącego zestawu testów do potwierdzenia, że nowe, szybsze rozwiązanie nadal działa zgodnie z oczekiwaniami.

Sprawdzone praktyki

Istnieje kilka kluczowych niezbędników, czyli dokumentacja, dobra struktura i testowanie.

Dokumentacja projektowa ułatwi zaznajomienie się z przejrzystą strukturą. Będzie też pomocna w przyszłości zarówno dla Ciebie, jak i współpracowników. Jeśli pominiesz tę część, nikt nie będzie za to wdzięczny (z Tobą włącznie). Umieszczenie dokumentacji w pliku *README* na najwyższym poziomie struktury stanowi sensowne miejsce początkowe. W razie potrzeby dokumentacja zawsze może później zostać przeniesiona do folderu *docs/*.

Objaśnij cel projektu, zawartość folderów i źródło pochodzenia danych. Wskaż pliki o krytycznym znaczeniu, a także wyjaśnij, jak wszystko uruchomić, w tym testy.

Jeden z autorów zaleca również zastosowanie platformy Docker. Umiejscowiony na najwyższym poziomie hierarchii plik *Dockerfile* zapewni w przyszłości informację o tym, jakie dokładnie są wymagane biblioteki systemu operacyjnego, aby można było z powodzeniem uruchomić projekt. Docker ułatwia też uruchomienie kodu na innych platformach sprzętowych lub wdrożenie go w środowisku chmury.

Dodaj folder *tests/* oraz kilka testów jednostkowych. My preferujemy `pytest` w roli nowoczesnego narzędzia do uruchamiania testów, ponieważ bazuje ono na module `unittest` wbudowanym w język Python. Zacznij od zaledwie kilku testów, a następnie je rozbudowuj. Rozpoczęcie korzystania z narzędzia `coverage`, które będzie informować, ile wierszy kodu faktycznie objętych jest testami, ułatwi uniknięcie mało przyjemnych niespodzianek.

Jeśli odziedziczyłeś starszy kod pozbawiony testów, szczególnie wskazanym działaniem będzie dodanie najpierw kilku testów. „Testy integracji" sprawdzające ogólny przepływ kodu projektu i potwierdzające,

Kup ksi k Pole ksi k

że dla określonych danych wejściowych uzyskujesz konkretne dane wynikowe, pozwolą zachować spokój przy wprowadzaniu kolejnych modyfikacji.

Dodaj test każdorazowo, gdy coś w kodzie nie zadziała. Nie ma żadnej korzyści w ponownym borykaniu się z tym samym problemem.

Zawsze pomocne będą komentarze w kodzie użyte dla każdej funkcji, klasy i modułu. Dbaj o zapewnianie wartościowych opisów tego, co jest *osiągane* w wyniku zastosowania funkcji. Gdy tylko jest to możliwe, dołącz krótki przykład demonstrujący oczekiwany wynik. Jeśli potrzebujesz inspiracji, sprawdź komentarze dokumentujące umieszczone w kodzie bibliotek NumPy i scikit-learn.

Każdorazowo, gdy kod staje się zbyt długi (np. funkcje o długości przekraczającej szerokość jednego ekranu), możesz bez obaw przeprowadzić refaktoryzację kodu w celu skrócenia go. Krótszy kod jest łatwiejszy do testowania i obsługiwania.

Opracowując testy, zastanów się nad przybliżoną tutaj metodologią projektowania bazującą na testach. Gdy wiesz dokładnie, czego potrzebujesz do przygotowania testów, a ponadto dysponujesz przykładami możliwymi do przetestowania, metoda ta staje się bardzo efektywna.

Tworzysz testy, uruchamiasz je, obserwujesz ich niepowodzenie, a *następnie* dodajesz funkcje oraz minimalną niezbędną logikę do obsługi przygotowanych testów. Gdy wszystkie testy działają, oznacza to koniec prac. Dzięki wcześniejszemu ustaleniu oczekiwanych danych wejściowych i wyjściowych funkcji dojdziesz do wniosku, że implementowanie logiki funkcji jest stosunkowo proste.

Jeżeli nie możesz wcześniej zdefiniować testów, pojawi się oczywiście następujące pytanie: czy naprawdę rozumiesz, co funkcja musi zrealizować? Jeśli nie, czy potrafisz poprawnie utworzyć funkcję w efektywny sposób? Taka metoda nie sprawdzi się dobrze, jeżeli ma miejsce proces twórczy, a ponadto analizujesz dane, których jeszcze dobrze nie rozumiesz.

Zawsze korzystaj z systemu kontroli kodu źródłowego. Będziesz wdzięczny wyłącznie sobie, gdy w nieodpowiednim momencie nadpiszesz coś o krytycznym znaczeniu. Przywyknij do częstego zatwierdzania zmian (codziennie lub nawet co 10 minut) i codziennego przesyłania danych do repozytorium.

Przestrzegaj standardu kodowania PEP8. Jeszcze lepsze będzie zastosowanie narzędzia `black` („pewne siebie” narzędzie do formatowania kodu) w połączeniu z fazą wstępnego zatwierdzania w systemie kontroli kodu źródłowego. W ten sposób narzędzie to po prostu samo zmodyfikuje kod pod kątem zgodności ze standardem. Aby uniknąć innych błędów, użyj narzędzia `flake8` do dopracowania kodu.

Tworzenie środowisk izolowanych od systemu operacyjnego ułatwi pracę. Pierwszy z autorów preferuje środowisko Anaconda, natomiast drugi autor woli używać środowiska `pipenv` w połączeniu z platformą Docker. Oba rozwiązania są sensowne, a ponadto znacznie lepsze niż stosowanie globalnego środowiska języka Python w systemie operacyjnym!

Kup ksi k Pole ksi k

Pamiętaj o tym, że automatyzacja jest pomocna. Mniejsza liczba ręcznie realizowanych operacji oznacza mniejsze ryzyko pojawienia się błędów. Systemy zautomatyzowanego procesu budowania, ciągła integracja ze zautomatyzowanymi narzędziami uruchamiającymi zestawy testów oraz systemy zautomatyzowanego wdrażania sprawiają, że żmudne i podatne na błędy zadania stają się standardowymi procesami, które mogą być przez każdego inicjowane i obsługiwane.

Pamiętaj też o tym, że zadbanie o czytelność jest znacznie ważniejsze niż wykazywanie się sprytem. Krótkie porcje złożonego i nieczytelnego kodu będą trudne w utrzymaniu zarówno dla ich twórcy, jak i dla jego współpracowników. Z tego powodu nie będzie chętnych do zajmowania się takim kodem. Zamiast tego utwórz dłuższą i bardziej czytelną funkcję, a także dołącz do niej wartościową dokumentację wyjaśniającą, co jest zwracane przez funkcję. Kod uzupełnij testami potwierdzającymi, że *działa* zgodnie z oczekiwaniami.

Wnioski dotyczące sprawdzonych praktyk korzystania z rozszerzenia Jupyter Notebook

Jeśli używasz rozszerzenia Jupyter Notebook, sprawdza się ono znakomicie w roli narzędzia do komunikacji wizualnej, ale sprzyja lenistwu. Jeżeli stwierdzisz, że wewnątrz rozszerzenia Notebook pozostawiasz długie funkcje, bez obaw możesz wyodrębnić je do modułu języka Python, a następnie dodać testy.

Rozważ prototypowanie kodu w powłoce IPython lub QTConsole. Przekształć wiersze kodu w funkcje w rozszerzeniu Notebook, a następnie przenieś je z niego do modułu uzupełnionego przez testy. Na koniec rozważ opakowanie kodu w klasie, jeśli przydatna okaże się hermetyzacja i ukrywanie danych.

Dowolnie rozmieść instrukcje `assert` wewnątrz rozszerzenia Notebook, aby sprawdzić, czy funkcje działają zgodnie z oczekiwaniami. Kod nie może być w prosty sposób testowany w obrębie tego rozszerzenia. Dopóki w wyniku refaktoryzacji nie umieścisz funkcji w osobnych modułach, sprawdzenia oparte na instrukcji `assert` stanowią prosty sposób zapewniania pewnego poziomu kontroli poprawności. Jeśli kodu nie wyodrębniono do modułu i nie utworzono sensownych testów jednostkowych, nie należy obdarzać go zaufaniem.

Zastosowanie instrukcji `assert` do sprawdzania danych w kodzie nie powinno być dobrze postrzegane. Jest to prosta metoda potwierdzenia, że są spełniane określone warunki, ale nie jest to idiomatyczna metoda języka Python. Aby utworzony kod był czytelniejszy dla innych projektantów, sprawdź oczekiwany stan danych, po czym wygeneruj odpowiedni wyjątek, gdy sprawdzenie zakończy się niepowodzeniem. Częstym wyjątkiem będzie `ValueError`, jeśli funkcja napotka nieoczekiwaną wartość. *Biblioteka Bulwark* (*https://bulwark.readthedocs.io/en/stable/*) to przykład środowiska testowania ukierunkowanego na narzędzie Pandas, które pozwala sprawdzić, czy dane spełniają określone ograniczenia.

Możesz też zdecydować się na dodanie na końcu rozszerzenia Notebook kilku sprawdzeń poprawności. Jest to kombinacja sprawdzeń logiki oraz instrukcji `raise` i `print`, które demonstrują, że właśnie zostało wygenerowane dokładnie to, co było wymagane. Gdy powrócisz do tego kodu za 6 miesięcy, podziękujesz sobie za ułatwienie całkowitego stwierdzenia, że działał poprawnie!

Kup ksi k Pole ksi k

W wypadku rozszerzenia Notebook trudnością jest współużytkowanie kodu z systemami kontroli kodu źródłowego. *nbdime* (*https://nbdime.readthedocs.io/en/latest/*) to jeden z rozwijających się zestawów nowych narzędzi, które pozwalają określać różnice między używanymi rozszerzeniami Notebook. Zestaw ten stanowi prawdziwe wybawienie, a ponadto pozwala pracować z innymi osobami.

Niech praca znów sprawia radość

Życie potrafi się komplikować. W ciągu pięciu lat, jakie minęły, odkąd autorzy książki napisali jej pierwsze wydanie, spotkało ich osobiście, a także w związku ze znajomymi i rodziną kilka nieoczekiwanych sytuacji życiowych, w tym depresja, nowotwór, przeprowadzka, zakończenie działalności biznesowej z powodzeniem, a także i z powodu porażki oraz zmiana w realizowaniu kariery. Nieuchronnie takie zewnętrzne zdarzenia będą mieć wpływ na pracę każdego i pogląd na życie.

Pamiętaj, aby cały czas szukać radości w nowych działaniach. Gdy zaczniesz poszukiwać, zawsze znajdą się interesujące szczegóły lub wymagania. Możesz zadać następujące pytanie: „Dlaczego podjęto taką decyzję?” albo „Jak mógłbym to zrobić inaczej?”. I nagle będziesz gotowy do rozpoczęcia dyskusji na temat tego, jak coś może zostać zmienione lub ulepszone.

Rejestruj rzeczy, które są warte świętowania. Tak łatwo zapomnieć o podziękowaniach, będąc zaabsorbowanym codziennością. Ludzie wypalają się, ponieważ zawsze chcą być na bieżąco i zapominają o tym, jak duży zrobili postęp.

Sugerujemy, aby utworzyć listę pozycji wartych uczczenia, a ponadto zaznaczyć, w jaki sposób to zrobić. Jeden z autorów utrzymuje taką listę. Jest on szczęśliwie zaskoczony w momencie aktualizowania listy, gdy widzi, jak wiele fajnych rzeczy się wydarzyło w ostatnim roku (być może w przeciwnym razie zostałyby zapomniane!). Nie powinny to być jedynie kamienie milowe związane z pracą. Uwzględnij zainteresowania i sport. Ciesz się osiągniętymi kamieniami milowymi. Jeden z autorów dba o to, aby jego życie prywatne miało właściwy priorytet, dlatego poświęca wiele dni z dala od komputera na zajmowanie się projektami innymi niż techniczne. Kluczowe znaczenie ma ciągłe rozwijanie posiadanych umiejętności, lecz niekoniecznie do poziomu stanu wypalenia!

Programowanie, a zwłaszcza gdy dotyczy ono poprawiania wydajności, przebiega znakomicie w sensie ciekawości i chęci do tego, aby zawsze jeszcze bardziej zagłębiać się w szczegóły techniczne. Niestety taka ciekawość to pierwsza rzecz, jaka prowadzi do stanu wypalenia. Z tego powodu zwolnij i zadbaj o to, żeby podróży towarzyszyła radość. Poza tym miej satysfakcję i kieruj się ciekawością.

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

Programowanie w Pythonie: wydajność i niezawodność!

Python jest językiem łatwym do opanowania i przyjemnym dla programisty. Jednak łatwość projektowania nie przekłada się na szybkość działania kodu. W konsekwencji przetwarzanie dużych wolumenów danych czy próba skalowania aplikacji kończą się problemami z wydajnością lub niezawodnością. Niekiedy rozwiązaniem jest zastosowanie procesów szeregowych, w innych przypadkach warto sięgnąć do architektury wielordzeniowej, klastrów lub układów GPU. Relatywnie często okazuje się, że dobre wyniki uzyskuje się w efekcie takiego zmodyfikowania technik kodowania, aby przy wykorzystaniu potencjału Pythona stosować sprawdzone metody poprawy wydajności kodu.

Dzięki drugiemu, poszerzonemu i zaktualizowanemu wydaniu tej książki zdobędziesz wszechstronną wiedzę o czynnikach wpływających na wydajność kodu. Dowiesz się, jakie procesy zachodzą w tle komputera, na jakich zasadach odbywa się przydzielanie pamięci, oraz zyskasz nowe spojrzenie na proces kompilacji do postaci kodu maszynowego. Zapoznasz się z zagadnieniem współbieżności i obliczeń klastrowych. Zaczniesz swobodnie posługiwać się najlepszymi narzędziami Pythona, takimi jak NumPy czy moduł multiprocessing. Z pewnością docenisz techniki zapewniające korzystanie z minimum zasobów, takich jak czas procesora czy pamięć RAM. Opisane tu zagadnienia zilustrowano przykładami kodu oraz poradami najlepszych specjalistów z branży.

W książce:

- narzędzia NumPy, Cython i Docker
- znajdowanie wąskich gardeł związanych z wykorzystaniem czasu procesora i pamięci
- wydajność kodu a odpowiednie struktury danych
- przyspieszanie obliczeń opartych na macierzach i wektorach
- zarządzanie wieloma operacjami obliczeniowymi i wejścia-wyjścia
- przetwarzanie współbieżne i uruchamianie kodu w klastrze

Micha Gorelick fascynuje się danologią. Jest autorem licznych prac poświęconych etycznym i praktycznym aspektom badań oraz wdrażania uczenia maszynowego. Współzałożyciel firmy Fast Forward Labs, w której realizuje się w roli szalonego naukowca.

Ian Ozsvald jest danologiem i prelegentem. Współorganizuje coroczną konferencję PyData London, bierze też udział w innych branżowych spotkaniach. W ramach firmy Mor Consulting świadczy usługi konsultingowe z zakresu danologii.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
HELIONSZKOLENIA.PL
INFORMATYKA W NAJLEPSZYM WYDANIU
KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-7184-2

9 788328 371842

Cena: 89,00 zł