O'REILLY®

Wydanie III

Wysoko wydajny Python

Efektywne programowanie w praktyce

Micha Gorelick, Ian Ozsvald
Słowo wstępne: Hilary Mason

Helion

Tytuł oryginału: High Performance Python, 3rd Edition

Tłumaczenie: Piotr Pilch

ISBN: 978-83-289-3124-4

Authorized Polish translation of the English edition of *High Performance Python, 3rd Edition*
ISBN 9781098165963 © 2025 Micha Gorelick and Ian Ozsvald

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/wyspy3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 2.

Użycie profilowania do znajdowania wąskich gardeł

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

- Jak można zidentyfikować w kodzie wąskie gardła związane z szybkością i pamięcią RAM?
- Jak profilowane jest wykorzystanie pamięci i procesora?
- Jaka głębokość profilowania powinna zostać użyta?
- Jak można profilować aplikację działającą długoterminowo?
- Co się dzieje pod podszewką w przypadku użycia narzędzia CPython?
- Jak zapewnić poprawność kodu podczas dostrajania wydajności?

Profilowanie umożliwia znalezienie wąskich gardeł. Dzięki temu przy minimalnym nakładzie pracy możliwe jest uzyskanie największego praktycznego wzrostu wydajności. Choć można oczekiwać ogromnego wzrostu szybkości i zmniejszenia wykorzystania zasobów przy niewielkim nakładzie pracy, w praktyce celem jest uzyskanie kodu, który działa „wystarczająco szybko i niezawodnie", aby spełnić nasze wymagania. Profilowanie pozwoli podjąć najbardziej pragmatyczne decyzje przy minimalnym wysiłku.

Profilowany może być dowolny zasób (nie tylko procesor!), dla którego można dokonać pomiaru. W tym rozdziale przyjrzymy się zarówno wykorzystaniu pamięci, jak i czasu procesora. Podobne techniki możesz też zastosować do pomiaru przepustowości sieci i dyskowych operacji wejścia-wyjścia.

Jeśli program działa zbyt wolno lub używane jest za dużo pamięci RAM, wskazane będzie poprawienie wszystkich odpowiedzialnych za to części kodu. Oczywiście możesz pominąć profilowanie i poprawić to, co, jak *wierzysz*, może stanowić problem. Trzeba jednak uważać, ponieważ często kończy się to „poprawieniem" niewłaściwej rzeczy. Zamiast kierować się intuicją, przed dokonaniem zmian w strukturze kodu lepiej przeprowadź profilowanie ze zdefiniowaną hipotezą.

Czasami pośpiech nie jest wskazany. Profilowanie przed dokonywaniem zmian pozwala szybko zidentyfikować wąskie gardła, które trzeba wyeliminować. Później możesz usunąć po prostu taką ich liczbę, jaka będzie wymagana, by osiągnąć żądaną wydajność. Jeśli pominiesz profilowanie i przejdziesz do optymalizowania, całkiem prawdopodobne jest to, że w dłuższej perspektywie będziesz musiał włożyć w to więcej pracy. Zawsze kieruj się wynikami profilowania.

Efektywne profilowanie

Pierwszym zadaniem w procesie profilowania jest testowanie reprezentacyjnego systemu w celu określenia, które elementy pracują powoli (lub zużywają zbyt wiele pamięci RAM albo wymuszają za dużo operacji wejścia-wyjścia dysku lub sieci). Profilowanie zwykle powoduje obciążenie (zazwyczaj ma miejsce spowolnienie od dziesięciu do stu razy). Kod ma nadal być używany w sposób jak najbardziej zbliżony do tego, jaki wykorzystuje się w rzeczywistych warunkach. Wyodrębnij przypadek testowy i wyizoluj część systemu, która wymaga sprawdzenia. Najlepiej byłoby, gdyby ta część została już tak utworzona, aby znajdowała się we własnym zestawie modułów.

Podstawowe techniki przedstawione w tym rozdziale jako pierwsze obejmują funkcję „magiczną" `%timeit` powłoki IPython, funkcję `time.time()` i dekorator czasu. Dzięki poznaniu tych metod zrozumiesz działanie instrukcji i funkcji.

W dalszej części rozdziału zostanie omówiony moduł `cProfile` (podrozdział „Użycie modułu cProfile"). Dowiesz się, w jaki sposób za pomocą tego wbudowanego narzędzia zidentyfikować w kodzie funkcje, których wykonanie zajmuje najwięcej czasu. Umożliwi Ci to zaznajomienie się z ogólną prezentacją problemu, dzięki czemu będziesz mógł skupić swoją uwagę na krytycznych funkcjach.

Dalej przyjrzymy się narzędziu `line_profiler` (podrozdział „Użycie narzędzia line_profiler do pomiarów dotyczących kolejnych wierszy kodu"), które przeprowadzi profilowanie wybranych funkcji dla kolejnych wierszy kodu. Wynik będzie obejmować liczbę wywołań każdego wiersza i wartość procentową czasu poświęconego na każdy wiersz. Dokładnie te informacje są niezbędne do zidentyfikowania tego, co działa wolno i z jakiego powodu.

Gdy będziesz dysponował wynikami działania narzędzia `line_profiler`, uzyskasz informacje wymagane do zastosowania kompilatora omówionego w rozdziale 8. Jeśli spróbowałeś przeprowadzić kompilację bez uprzedniego profilowania, jest całkiem możliwe, że przyspieszyłeś działanie bloku kodu bez wąskiego gardła, co ostatecznie nie wpłynie na zwiększenie ogólnej szybkości.

W rozdziale 6. dowiesz się, jak użyć polecenia `perf stat` do przeanalizowania liczby instrukcji, które są ostatecznie wykonywane w procesorze, a także jak efektywnie wykorzystywać pamięci podręczne procesora. Pozwala to na dostrajanie operacji macierzowych na zaawansowanym poziomie. Po przeczytaniu tego rozdziału warto przyjrzeć się przykładowi 6.8. Zapisano go na później, gdyż odnosi się do zastosowania tablic biblioteki NumPy.

Jeśli pracujesz z długo działającymi systemami, to po narzędziu `line_profiler` godne Twojego zainteresowania będzie narzędzie `py-spy`. Umożliwia ono analizowanie już działających procesów Python.

Aby zilustrować, dlaczego wykorzystanie pamięci RAM jest duże, zostanie omówione narzędzie `memory_profiler` (podrozdział „Użycie narzędzia memory_profiler do diagnozowania wykorzystania pamięci"). Jest ono szczególnie przydatne podczas śledzenia na wykresie z etykietami wykorzystania pamięci RAM w czasie. Dzięki temu możesz wyjaśnić współpracownikom, dlaczego określone funkcje zużywają więcej pamięci RAM, niż oczekiwano.

Aby połączyć ze sobą profilowanie procesora i pamięci RAM, wskazane będzie przeczytanie opisu narzędzia Scalene (podrozdział „Połączenie profilowania procesora i pamięci przy użyciu narzędzia Scalene" niniejszego rozdziału). Łączy ono zadania realizowane przez narzędzia `line_profiler` i `memory_profiler` z nowatorskim procesem alokowania pamięci o niskim wpływie, a także uwzględnia obsługę profilowania procesorów GPU w wersji eksperymentalnej.

Narzędzie VizTracer (podrozdział „Wyświetlanie interaktywnego stosu wywołań na osi czasu za pomocą narzędzia VizTracer" tego rozdziału) pozwala uzyskać widok z osią czasu dla wykonywanego kodu. Narzędzie przedstawia stos wywołań w pionie z osią czasu biegnącą od lewej do prawej strony. Możesz klikać w obrębie stosu wywołań, a nawet dodawać własne komunikaty i opisy sposobu działania.

Niezależnie od wybranej metody profilowania kodu trzeba pamiętać o zapewnieniu w nim odpowiedniego zakresu testów jednostkowych. Testy jednostkowe ułatwiają popełnianie prostych pomyłek, a ponadto są pomocne w zapewnieniu możliwości odtworzenia wyników. Rezygnuj z nich tylko na własne ryzyko.

Zawsze profiluj kod przed kompilowaniem lub modyfikowaniem algorytmów. Do określenia najbardziej efektywnych metod przyspieszania działania kodu niezbędny jest dowód.

W dalszej części rozdziału zamieszczono też wprowadzenie do kodu bajtowego Python w obrębie narzędzia CPython (podrozdział „Użycie modułu dis do sprawdzania kodu bajtowego narzędzia CPython"). Dzięki temu możesz zrozumieć, co się dzieje pod podszewką. W szczególności zaznajomienie się ze sposobem działania maszyny wirtualnej opartej na stosie kodu w języku Python ułatwi Ci zrozumienie, dlaczego określone style kodowania cechują się mniejszą wydajnością od innych. Narzędzie Specialist (punkt „Analizowanie specjalizacji kodu bajtowego za pomocą narzędzia Specialist" tego rozdziału) ułatwi następnie stwierdzenie, jakie części kodu bajtowego mogą zostać zidentyfikowane pod kątem zwiększenia wydajności w przypadku języka Python w wersji 3.11 lub nowszej.

Na końcu rozdziału przyjrzymy się sposobom integrowania testów jednostkowych podczas profilowania (podrozdział „Testowanie jednostkowe podczas optymalizacji w celu zachowania poprawności"), aby zachować poprawność kodu podczas dokonywania zmian mających na celu zwiększenie jego wydajności.

Korzystając z rozwiązań z generatywną sztuczną inteligencją, takich jak GitHub Copilot, pamiętaj, że *zawsze otrzymasz odpowiedź*, która *czasem* może być poprawna. W przypadku profilowania istnieje milion nieaktualnych przykładów w danych uczących, które mogą wpływać na odpowiedzi udzielane przez system generatywnej sztucznej inteligencji. Podczas pracy nad książką jej autorzy czuli się zdezorientowani, poirytowani, a czasem rozbawieni sugestiami Copilota. Zawsze bądź sceptyczny.

Rozdział zostanie zakończony omówieniem strategii profilowania (podrozdział „Strategie udanego profilowania kodu"). Dzięki temu możliwe będzie niezawodne profilowanie kodu i gromadzenie właściwych danych na potrzeby testowania określonych hipotez. Dowiesz się, jak skalowanie częstotliwości procesora oraz funkcje takie jak Turbo Boost mogą zafałszować wyniki profilowania, a także jak można je wyłączyć.

Aby wykonać wszystkie te kroki, niezbędna jest prosta do analizowania funkcja. W następnym podrozdziale zostanie przedstawiony zbiór Julii. Jest to powiązana z procesorem funkcja, która w trochę większym stopniu korzysta z pamięci RAM. Ponadto przejawia działanie nieliniowe (z tego powodu nie można z łatwością przewidzieć wyników). Oznacza to, że zamiast analizowania w trybie *offline*, funkcja ta wymaga profilowania podczas działania.

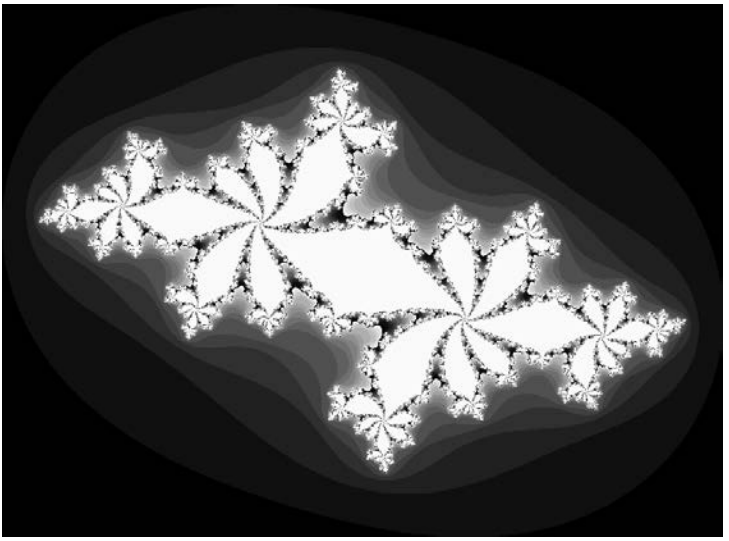
Wprowadzenie do zbioru Julii

Zbiór Julii (*http://pl.wikipedia.org/wiki/Zbi%C3%B3r_Julii*) to interesujący problem powiązany z procesorem, od którego warto zacząć. Zbiór ten jest sekwencją fraktalną, która generuje złożony obraz wyjściowy. Nazwa zbioru wywodzi się od matematyka Gastona Julii.

Zamieszczony dalej kod jest trochę dłuższy od wersji, którą możesz sam utworzyć. Zawiera on komponent powiązany z procesorem oraz wyjątkowo jawny zbiór wejść. Konfiguracja taka umożliwia profilowanie zarówno wykorzystania procesora, jak i pamięci RAM. Dzięki temu można zrozumieć, jakie części kodu zużywają dwa spośród skromnych zasobów obliczeniowych. Ponieważ taka implementacja jest *umyślnie* niezoptymalizowana, możemy zidentyfikować operacje wykorzystujące pamięć i wolne instrukcje. W dalszej części rozdziału zostanie poprawiona wolna instrukcja logiczna oraz instrukcja, która wykorzystuje dużo pamięci. W rozdziale 8. znacząco skrócony zostanie ogólny czas wykonywania tej funkcji.

Przeanalizujemy blok kodu, który w punkcie zespolonym `c=-0.62772-0.42193j` (sam spróbuj uruchomić ten wiersz, który zawiera poprawny kod w Pythonie!) generuje zarówno wykres fałszywej skali szarości (rysunek 2.1), jak i wariant zbioru Julii z czystą skalą szarości (rysunek 2.3). Zbiór Julii jest tworzony przez obliczenie każdego piksela obrazu w wyizolowaniu. Jest to „kłopotliwy problem z równoległością", gdyż między punktami nie są współużytkowane dane.

Jeśli zostanie określony inny punkt c, zostanie uzyskany inny obraz. Wybrane położenie zawiera obszary szybkie do obliczenia oraz takie, których obliczenie wymaga więcej czasu. Jest to korzystne pod kątem przykładowej analizy.



Rysunek 2.1. Wykres zbioru Julii z fałszywą skalą szarości podkreślającą szczegóły

Problem jest interesujący, ponieważ każdy piksel jest obliczany przez zastosowanie pętli, która może być wykonywana nieokreśloną liczbę razy. W każdej iteracji sprawdzane jest, czy wartość danej współrzędnej zmierza do nieskończoności, czy wydaje się wstrzymywana przez atraktor. Współrzędne powodujące niewiele iteracji mają ciemny kolor (rysunek 2.1). Z kolei współrzędne, które powodują dużą liczbę iteracji, mają biały kolor. Białe obszary są trudniejsze do obliczenia i wygenerowanie ich zajmuje więcej czasu.

Definiujemy zbiór współrzędnych *z*, które będą testowane. Obliczana funkcja określa pierwiastek kwadratowy liczby zespolonej z i dodaje punkt c:

$f(z) = z^2 + c$

Funkcja jest iterowana podczas testowania mającego na celu sprawdzenie, czy warunek zmierzania wstrzymuje użycie funkcji `abs`. Jeśli funkcja zmierzania ma wartość `False`, następuje wyjście z pętli i zarejestrowanie liczby iteracji wykonanych dla danej współrzędnej. Jeśli funkcja zmierzania nigdy nie ma wartości `False`, pętla jest kończona po liczbie iteracji określonej przez wartość `maxiter`. Wynik obliczenia liczby zespolonej z zostanie później przekształcony w kolorowy piksel, który reprezentuje położenie tej liczby zespolonej.

W pseudokodzie może to wyglądać następująco:

```
for z in coordinates:
    for iteration in range(maxiter):  # ograniczona liczba iteracji dla punktu
        if abs(z) < 2.0:  # czy warunek zmierzania został naruszony?
            z = z*z + c
        else:
            break
    # przechowanie liczby iteracji dla każdej liczby zespolonej z i wygenerowanie później wykresu
```

W celu objaśnienia tej funkcji spróbujmy użyć dwóch współrzędnych.

Użyjemy współrzędnej, która zostanie umieszczona w lewym górnym narożniku wykresu w punkcie `-1.8-1.8j`. Zanim będzie możliwe wypróbowanie reguły aktualizacji, konieczne jest sprawdzenie warunku `abs(z) < 2`:

```
z = -1.8-1.8j
print(abs(z))

2.54558441227
```

Jak widać, w przypadku lewej górnej współrzędnej test funkcji `abs(z)` da wartość `False` dla zerowej iteracji, tak jak `2.54 >= 2.0`. Oznacza to, że nie jest stosowana reguła aktualizacji. Wartość zmiennej `output` dla tej współrzędnej wynosi `0`.

Przesuńmy się do środka wykresu w punkcie `z = 0 + 0j` i sprawdźmy kilka iteracji:

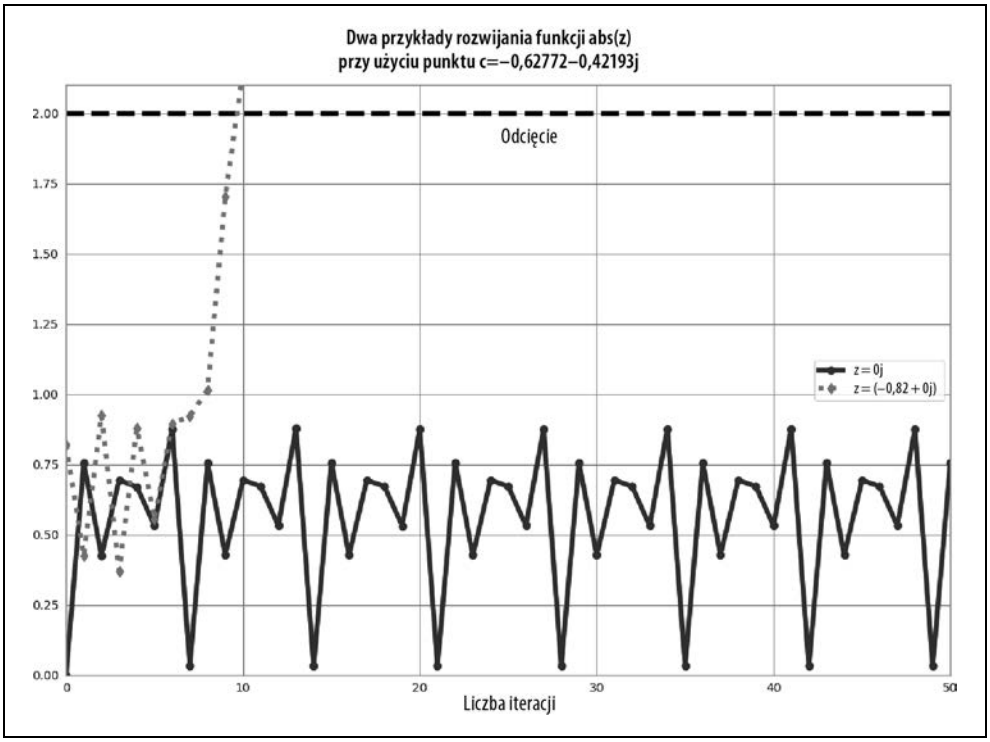
```
c = -0.62772-0.42193j
z = 0+0j
for n in range(9):
    z = z*z + c
    print(f"{n}: z={z: .5f}, abs(z)={abs(z):0.3f}, c={c: .5f}")

0: z=-0.62772-0.42193j, abs(z)=0.756, c=-0.62772-0.42193j
1: z=-0.41171+0.10778j, abs(z)=0.426, c=-0.62772-0.42193j
2: z=-0.46983-0.51068j, abs(z)=0.694, c=-0.62772-0.42193j
3: z=-0.66777+0.05793j, abs(z)=0.670, c=-0.62772-0.42193j
4: z=-0.18516-0.49930j, abs(z)=0.533, c=-0.62772-0.42193j
5: z=-0.84274-0.23703j, abs(z)=0.875, c=-0.62772-0.42193j
6: z= 0.02630-0.02242j, abs(z)=0.035, c=-0.62772-0.42193j
7: z=-0.62753-0.42311j, abs(z)=0.757, c=-0.62772-0.42193j
8: z=-0.41295+0.10910j, abs(z)=0.427, c=-0.62772-0.42193j
```

Jak widać, każda aktualizacja współrzędnej `z` dla tych kilku pierwszych iteracji powoduje uzyskanie wartości, dla której warunek `abs(z) < 2` ma wartość `True`. Po wykonaniu dla tej współrzędnej 300 iteracji w dalszym ciągu test da wartość `True`. Nie jest możliwe określenie liczby iteracji niezbędnych do uzyskania dla warunku wartości `False`. Może się to zakończyć nieskończoną sekwencją. Klauzula zatrzymania z maksymalną liczbą iteracji (wartość `maxiter`) spowoduje zakończenie potencjalnie nieskończonej iteracji.

Na rysunku 2.2 pokazano 50 pierwszych iteracji powyższej sekwencji. Dla punktu `0+0j` (linia ciągła ze znacznikami w postaci okręgów) sekwencja wydaje się powtarzać co ósmą iterację. Każda sekwencja siedmiu obliczeń ma niewielkie odchylenie względem poprzedniej sekwencji. Nie jest możliwe stwierdzenie, czy ten punkt będzie bez końca iterowany w obrębie warunku granicznego przez długi czas, czy być może zaledwie przez kilka kolejnych iteracji. Linia kreskowana `odcięcie` prezentuje granicę w punkcie `+2`.

W przypadku punktu `-0.82+0j` (linia kreskowana ze znacznikami w postaci rombów) widać, że po dziewiątej aktualizacji wynik bezwzględny przekroczył linię odcięcia `+2`, dlatego nastąpiło zatrzymanie aktualizowania tej wartości.



Rysunek 2.2. Dwa przykłady współrzędnych rozwijane dla zbioru Julii

Obliczanie pełnego zbioru Julii

W tym podrozdziale zostanie omówiony kod, który generuje zbiór Julii. Kod będzie analizowany w rozdziale na różne sposoby. Jak widać w przykładzie 2.1, na początku modułu importowany jest moduł `time` pierwszej metody profilowania i definiowanych jest kilka stałych współrzędnych.

Przykład 2.1. Definiowanie stałych globalnych dla przestrzeni współrzędnych

```
"""Generator zbioru Julii bez opcjonalnego rysowania obrazów na bazie biblioteki PIL"""
import time

# obszar przestrzeni zespolonej do przeanalizowania
x1, x2, y1, y2 = -1.8, 1.8, -1.8, 1.8
c_real, c_imag = -0.62772, -.42193
```

Aby wygenerować wykres, tworzone są dwie listy danych wejściowych. Pierwsza lista to `zs` (współrzędne zespolone *z*), a druga to `cs` (zespolony warunek początkowy). Żadna z list nie zmienia się. Listę `cs` można zoptymalizować do postaci pojedynczej wartości `c` jako stałej. Powodem utworzenia dwóch list wejściowych jest chęć uzyskania rozsądnie prezentujących się danych, które będą używane podczas profilowania wykorzystania pamięci RAM w dalszej części rozdziału.

Aby utworzyć listy zs i cs, konieczne jest określenie współrzędnych dla każdego punktu z. W przykładzie 2.2 te współrzędne są tworzone przy użyciu zmiennych xcoord i ycoord, a ponadto określono zmienne x_step i y_step. Szczegółowość takiej konfiguracji przydaje się przy przenoszeniu kodu do innych narzędzi (np. do narzędzi numpy) i środowisk opartych na kodzie w Pythonie, ponieważ ułatwia zdefiniowanie wszystkiego w *bardzo* przejrzysty sposób na potrzeby debugowania.

Przykład 2.2. Definiowanie list współrzędnych jako wejść przykładowej funkcji obliczeniowej

```
def calc_pure_python(desired_width, max_iterations):
    """Tworzenie listy współrzędnych zespolonych (zs) i parametrów
    zespolonych (cs), budowanie zbioru Julii"""
    x_step = (x2 - x1) / desired_width
    y_step = (y1 - y2) / desired_width
    x = []
    y = []
    ycoord = y2
    while ycoord > y1:
        y.append(ycoord)
        ycoord += y_step
    xcoord = x1
    while xcoord < x2:
        x.append(xcoord)
        xcoord += x_step
    # Utwórz listę współrzędnych i warunek początkowy dla każdej komórki
    # Zauważ, że warunek początkowy to stała, która z łatwością może zostać usunięta
    # Stała służy do symulowania rzeczywistego scenariusza z kilkoma wejściami
    # przekazanymi przykładowej funkcji
    zs = []
    cs = []
    for ycoord in y:
        for xcoord in x:
            zs.append(complex(xcoord, ycoord))
            cs.append(complex(c_real, c_imag))

    print("Długość dla x:", len(x))
    print("Łączna liczba elementów:", len(zs))
    start_time = time.time()
    output = calculate_z_serial_purepython(max_iterations, zs, cs)
    end_time = time.time()
    secs = end_time - start_time
    print(f"Działanie funkcji {calculate_z_serial_purepython.__name__} trwało
    ↪{secs:0.2f} s")

    # Suma ta jest oczekiwana dla siatki 1000^2 z 300 iteracjami
    # Zapewnia, że kod działa zgodnie z oczekiwaniami
    assert sum(output) == 33219980
```

Po utworzeniu list zs i cs zwracane są informacje o ich wielkości, a ponadto obliczana jest lista output za pomocą funkcji calculate_z_serial_purepython. Na końcu sumowane są dane zmiennej output i instrukcji assert, które są zgodne z oczekiwaną wartością wyjściową. Jeden z autorów posłużył się tutaj tym kodem, aby zapewnić, że w książce nie pojawi się kod zawierający błędy.

Ponieważ kod jest deterministyczny, możemy sprawdzić, czy funkcja działa zgodnie z oczekiwaniami, sumując wszystkie obliczone wartości. Przydaje się to jako kontrola poprawności. W przypadku wprowadzania zmian w kodzie numerycznym *bardzo* wskazane jest sprawdzenie, czy nie został uszkodzony algorytm. W idealnej sytuacji zostałyby użyte testy jednostkowe, a ponadto sprawdzona więcej niż jedna konfiguracja powiązana z problemem.

W przykładzie 2.3 definiowana jest następnie funkcja `calculate_z_serial_purepython`, która rozszerza omówiony wcześniej algorytm. Na początku definiowana jest też lista `output` o takiej samej długości jak w przypadku list `zs` i `cs`.

Przykład 2.3. Funkcja obliczeniowa powiązana z procesorem

```
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Obliczanie listy output przy użyciu reguły aktualizacji zbioru Julii"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while abs(z) < 2 and n < maxiter:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

W przykładzie 2.4 wywoływana jest funkcja obliczeniowa. Przez opakowanie jej za pomocą kodu kontrolnego `__main__` w przypadku niektórych metod profilowania możemy bezpiecznie zaimportować moduł bez rozpoczynania obliczeń. Nie jest tutaj prezentowana metoda używana do generowania wykresu danych wyjściowych.

Przykład 2.4. Funkcja __main__ kodu

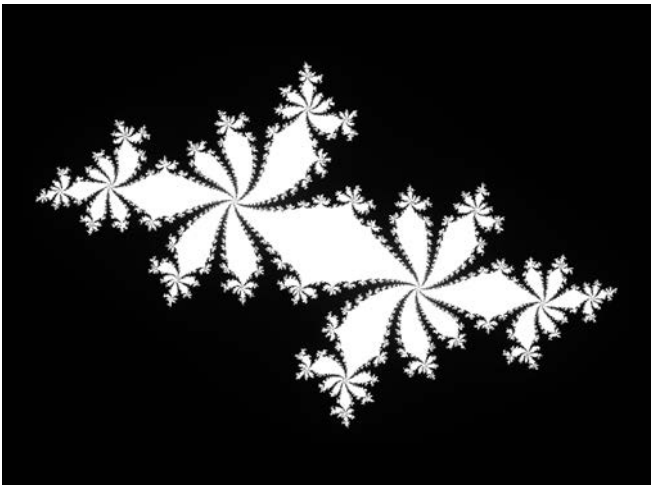
```
if __name__ == "__main__":
    # Obliczanie zbioru Julii za pomocą czystego rozwiązania opartego na języku Python
    # z wykorzystaniem wartości domyślnych rozsądnych dla laptopa
    calc_pure_python(desired_width=1000, max_iterations=300)
```

Po uruchomieniu kodu zostaną uzyskane dane wyjściowe opisujące złożoność problemu:

```
# uruchomienie powyższego kodu daje następujący wynik:
Długość dla x: 1000
Łączna liczba elementów: 1000000
Działanie funkcji calculate_z_serial_purepython trwało 5.80 s
```

W przypadku wykresu fałszywej skali szarości (rysunek 2.1) zmiany kolorów o wysokim kontraście pozwoliły nam zorientować się, gdzie koszt funkcji zmieniał się wolno lub szybko. Na rysunku 2.3 widoczna jest liniowa mapa kolorów: kolor czarny można szybko wygenerować, a generowanie koloru białego jest kosztowne.

Uwidocznienie dwóch reprezentacji tych samych danych pozwala zauważyć, że przy odwzorowaniu liniowym traconych jest wiele szczegółów. Czasem podczas analizowania kosztu funkcji przydatne może być uwzględnienie różnych reprezentacji.



Rysunek 2.3. Przykład wykresu zbioru Julii używający czystej skali szarości

Proste metody pomiaru czasu — instrukcja print i dekorator

Po wykonaniu kodu z przykładu 2.4 uzyskano dane wyjściowe wygenerowane przez kilka instrukcji `print`. Na laptopie jednego z autorów wykonanie tego kodu przy użyciu narzędzia CPython 3.12 zajęło około 5 sekund. Warto zauważyć, że czas wykonania zawsze będzie trochę inny. Podczas pomiaru czasu wykonywania kodu musisz zaobserwować standardową zmienność, ponieważ w przeciwnym razie możesz niepoprawnie przypisać wzrost wydajności w kodzie po prostu losowej zmienności czasu wykonywania.

W czasie działania kodu komputer będzie realizować inne zadania, takie jak uzyskiwanie dostępu do sieci, dysku lub pamięci RAM. Czynniki te mogą powodować zmiany czasu wykonywania programu.

Laptop jednego z autorów to Dell XPS 15 9510 z procesorem Intel Core I7-11800H (2,3 GHz, pamięć podręczna Level 3 o pojemności 24 MB i 8 rdzeni fizycznych z hiperwątkowością) oraz pamięcią RAM 64 GB i systemem Linux Mint 21.2 (opartym na systemie Ubuntu 22.04).

W funkcji `calc_pure_python` (przykład 2.2) znajduje się kilka instrukcji `print`. Jest to najprostsza metoda pomiaru czasu wykonywania porcji kodu *w obrębie* funkcji. Jest to bardzo proste rozwiązanie, które pomimo szybkości i braku przejrzystości może okazać się pomocne w początkowej fazie analizowania porcji kodu.

Użycie instrukcji `print` jest powszechne podczas debugowania i profilowania kodu. Choć metoda ta szybko staje się trudna w obsłudze, przydaje się w przypadku krótkich analiz. Spróbuj uporządkować instrukcje `print` po zakończeniu analiz, ponieważ w przeciwnym razie standardowe wyjście `stdout` nie będzie przejrzyste.

Metodą zapewniającą trochę większą czytelność jest użycie *dekoratora*. W tym przypadku powyżej interesującej nas funkcji dodajemy jeden wiersz kodu. Dekorator może być bardzo prosty i może replikować jedynie efekt działania instrukcji `print`. Później można go bardziej rozbudować.

W przykładzie 2.5 definiowana jest nowa funkcja `timefn`, która pobiera funkcję wewnętrzną `measure_time` jako argument. Funkcja ta pobiera argumenty `*args` (zmienna liczba argumentów pozycyjnych) i `**kwargs` (zmienna liczba argumentów klucz/wartość), a ponadto przekazuje je funkcji `fn` w celu wykonania.

W ramach wykonywania funkcji `fn` przechwytywana jest funkcja `time.time()`, a następnie za pomocą instrukcji `print` wyświetlane są wyniki wraz z nazwą funkcji (kod `fn.__name__`). Choć obciążenie wynikające z zastosowania tego dekoratora jest niewielkie, w przypadku wywołania funkcji `fn` miliony razy może ono stać się zauważalne. Aby ujawnić nazwę funkcji i notkę dokumentacyjną (ang. *docstring*) elementowi wywołującemu funkcji z dekoratorem, używa się kodu `@wraps(fn)` (w przeciwnym razie widoczna byłaby nazwa funkcji i notka dokumentacyjna dla dekoratora, a nie dekorowanej przez niego funkcji).

Przykład 2.5. Definiowanie dekoratora do automatyzowania pomiarów czasu

```
from functools import wraps

def timefn(fn):
    @wraps(fn)
    def measure_time(*args, **kwargs):
        t1 = time.time()
        result = fn(*args, **kwargs)
        t2 = time.time()
        print(f"@timefn: działanie funkcji {fn.__name__} trwało {(t2 - t1):0.2f} s")
        return result
    return measure_time

@timefn
def calculate_z_serial_purepython(maxiter, zs, cs):
    ...
```

Po uruchomieniu tej wersji (zachowano instrukcje `print` z wcześniejszego kodu) widać, że czas wykonywania wersji z dekoratorem jest naprawdę niewiele krótszy niż w przypadku wywołania z funkcji `calc_pure_python`. Wynika to z obciążenia związanego z wywoływaniem funkcji (różnica jest znikoma):

```
Długość elementu x: 1000
Łączna liczba elementów: 1000000
@timefn: działanie funkcji calculate_z_serial_purepython trwało 5.78 s
Działanie funkcji calculate_z_serial_purepython trwało 5.78 s
```

Dodanie informacji o profilowaniu na pewno spowolni kod. Niektóre opcje profilowania zawierają mnóstwo informacji i powodują znaczny spadek szybkości. Konieczne będzie wypracowanie kompromisu pomiędzy szczegółowością profilowania i szybkością.

Użycie modułu `timeit` to kolejny sposób przeprowadzenia zwykłego pomiaru szybkości wykonywania funkcji powiązanej z procesorem. Ta metoda jest zwykle wykorzystywana przy określaniu czasu dla różnych typów prostych wyrażeń podczas eksperymentowania ze sposobami rozwiązania problemu.

Moduł `timeit` tymczasowo wyłącza program czyszczący pamięć. Może on mieć wpływ na uzyskaną szybkość w przypadku rzeczywistych operacji, jeśli program byłby normalnie przez nie wywoływany. Pomoc z tym związana jest dostępna w dokumentacji języka Python (*https://docs.python.org/3.7/library/timeit.html*).

Z poziomu wiersza poleceń możesz uruchomić moduł `timeit` w następujący sposób:

```
$ python -m timeit -n 5 -r 1 -s "import julia1_nopil"  \
  "julia1_nopil.calc_pure_python(desired_width=1000,
  max_iterations=300)"
```

Zauważ, że w ramach kroku konfiguracyjnego musisz zaimportować moduł przy użyciu opcji `-s`, ponieważ funkcja `calc_pure_python` znajduje się wewnątrz tego modułu. Moduł `timeit` zawiera kilka praktycznych wartości domyślnych w przypadku krótkich sekcji kodu, ale na potrzeby dłużej działających funkcji sensowne może być w celu powtarzania eksperymentów określenie liczby wykonań pętli (`-n 5`) oraz liczby powtórzeń (`-r 5`). Jako odpowiedź zwracany jest wynik wykonania wszystkich powtórzeń. Dodanie flagi szczegółowych informacji (`-v`) powoduje wyświetlanie skumulowanego czasu wykonania wszystkich pętli w ramach poszczególnych powtórzeń. Może to ułatwić zapewnienie zmienności w uzyskanych wynikach.

Domyślnie uruchomienie dla tej funkcji modułu `timeit` bez określania opcji `-n` i `-r` spowoduje wykonanie 10 razy pętli z pięcioma powtórzeniami. Całość operacji zajmie 6 minut. Nadpisanie wartości domyślnych może mieć sens, jeśli wyniki mają zostać uzyskane trochę szybciej.

Interesują nas wyłącznie wyniki dla najlepszego przypadku, ponieważ inne przypadki to prawdopodobnie rezultat ingerencji innych procesów:

```
Długość dla x: 1000
Łączna liczba elementów: 1000000
Działanie funkcji calculate_z_serial_purepython trwało 5.78 s
...
5 loops, best of 1: 6.1 sec per loop
```

Spróbuj kilkakrotnie uruchomić test porównawczy, aby sprawdzić, czy uzyskiwane są różne wyniki. W celu ustalenia najkrótszego czasu przy stabilnym wyniku może być wymagana większa liczba powtórzeń. Ponieważ nie ma „właściwej” konfiguracji, jeśli występuje duża rozbieżność wyników pomiaru czasu, zwiększaj liczbę powtórzeń do momentu uzyskania stabilnego wyniku końcowego.

Uzyskane dane pokazują, że ogólny koszt wywołania funkcji `calc_pure_python` to 6,1 sekundy (dla najlepszego przypadku), pojedyncze wywołania funkcji `calculate_z_serial_purepython` zajmują natomiast około 5,8 sekundy, co zostało zmierzone przez dekorator `@timefn`. Różnica to głównie czas, jaki zajęło utworzenie list `zs` i `cs` przed zarejestrowaniem wartości zmiennej `start_time`.

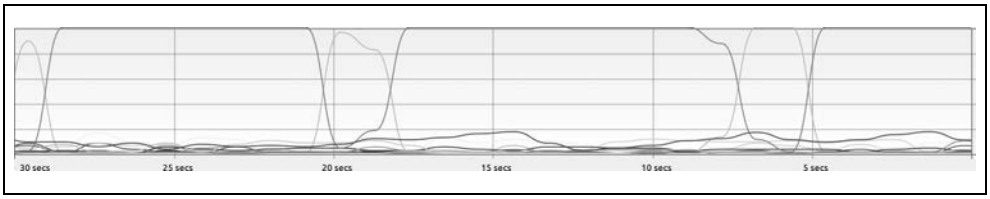
W obrębie powłoki IPython w ten sam sposób można użyć funkcji „magicznej” `%timeit`. Jeśli tworzysz kod interaktywnie w tej powłoce lub aplikacji Jupyter Notebook, możesz zastosować następujący kod:

```
In [1]: import julia1_nopil

In [2]: %timeit julia1_nopil.calc_pure_python(desired_width=1000, max_iterations=300)
```

Bądź świadom tego, że „najlepszy przypadek” jest w wypadku powłoki IPython i aplikacji Jupyter obliczany w odmienny sposób z wykorzystaniem skryptu `timeit.py` oraz funkcji `%timeit`. Skrypt używa zaprezentowanej już wartości minimalnej. W 2016 r. w powłoce IPython dokonano zmiany polegającej na użyciu średniej i odchylenia standardowego. Obie metody mają swoje wady, ale zasadniczo są w „miarę dobre”. Nie możesz jednak dokonać ich porównania. Zastosuj jedną z tych dwóch metod, lecz nie łącz ich.

Warto rozważyć zmienność obciążenia występującego w przypadku zwykłego komputera. Aktywnych jest wiele zadań w tle (np. usługa Dropbox, proces tworzenia kopii zapasowej), które w losowy sposób mogą wpływać na zasoby dyskowe i procesor. Skrypty na stronach internetowych mogą też wywoływać nieprzewidywalne wykorzystanie zasobów. Na rysunku 2.4 pokazano pojedynczy procesor obciążony w 100% przez część wcześniej wykonanych kroków pomiaru czasu. Inne rdzenie procesora tego samego komputera są nieznacznie obciążone przez inne zadania.



Rysunek 2.4. Narzędzie System Monitor w systemie Ubuntu, które prezentuje zmienne wykorzystanie procesora podczas pomiaru czasu dla przykładowej funkcji

Sporadycznie narzędzie System Monitor pokazuje dla komputera szczytową aktywność. Warto obserwować dane tego narzędzia, aby upewnić się, że nic innego nie oddziałuje na krytyczne zasoby (procesor, dysk, sieć).

Prosty pomiar czasu za pomocą polecenia time systemu Unix

Na chwilę wyjdziemy trochę poza język Python, aby opisać użycie standardowego narzędzia dostępnego w systemach uniksowych. Następujące polecenie zarejestruje różne widoki czasu wykonywania programu:

```
$ /usr/bin/time -p python julia1_nopil.py
Długość dla x: 1000
Łączna liczba elementów: 1000000
```

```
Działanie funkcji calculate_z_serial_purepython trwało 5.71 s
real 6.02
user 5.96
sys 0.05
```

Zauważ, że zamiast `time` podano dokładniejszą postać `/usr/bin/time`, aby skorzystać z systemowego narzędzia `time`, a nie z jego prostszej (i mniej przydatnej) wersji wbudowanej w powłokę. Jeśli spróbujesz użyć polecenia `time --verbose` i zostanie wygenerowany błąd, prawdopodobnie masz do czynienia z poleceniem `time` wbudowanym w powłokę, a nie z poleceniem systemowym.

Użycie flagi przenośności `-p` powoduje uzyskanie następujących trzech wyników:

- `real`. Rejestruje aktualny czas lub czas, który upłynął.
- `user`. Rejestruje czas, jaki procesor poświęcił na realizowanie zadania poza funkcjami jądra.
- `sys`. Rejestruje czas, jaki upłynął w funkcjach na poziomie jądra.

Uwzględnienie wyników `user` i `sys` pozwala zorientować się, ile czasu minęło w procesorze. Różnica między tymi wynikami i wynikiem `real` daje możliwość określenia czasu, jaki upłynął podczas oczekiwania na operację wejścia-wyjścia. Różnica może też wskazywać, że system jest zajęty przetwarzaniem innych zadań, które zakłócają pomiary.

Narzędzie `time` jest przydatne, ponieważ nie jest powiązane z językiem Python. Uwzględnia ono czas wymagany do uruchomienia programu wykonywalnego `python`, który może być znaczny, jeśli uruchamianych jest wiele nowych procesów (zamiast jednego długotrwałego procesu). Jeśli często używasz krótko działających skryptów, w przypadku których czas uruchamiania stanowi znaczącą część ogólnego czasu działania, narzędzie `time` może okazać się bardziej przydatną metodą pomiaru.

Aby uzyskać jeszcze więcej danych wyjściowych, można dodać flagę `--verbose`:

```
Długość dla x: 1000
Łączna liczba elementów: 1000000
Działanie funkcji calculate_z_serial_purepython trwało 5.76 s
        Command being timed: "python julia1_nopil.py"
        User time (seconds): 6.01
        System time (seconds): 0.05
        Percent of CPU this job got: 99%
        Elapsed (wall clock) time (h:mm:ss or m:ss): 0:06.07
        Average shared text size (kbytes): 0
        Average unshared data size (kbytes): 0
        Average stack size (kbytes): 0
        Average total size (kbytes): 0
        Maximum resident set size (kbytes): 98432
        Average resident set size (kbytes): 0
        Major (requiring I/O) page faults: 0
        Minor (reclaiming a frame) page faults: 23334
        Voluntary context switches: 1
        Involuntary context switches: 37
        Swaps: 0
        File system inputs: 0
        File system outputs: 0
        Socket messages sent: 0
        Socket messages received: 0
```

```
        Signals delivered: 0
        Page size (bytes): 4096
        Exit status: 0
```

`Maximum resident set size` to przydatny wskaźnik informujący o maksymalnej ilości pamięci RAM użytej podczas wykonywania kodu. Jeśli wartość ta zbliża się do dostępnej pojemności fizycznej pamięci RAM, jesteś blisko jej wyczerpania lub użycia dyskowego pliku wymiany, który działa bardzo wolno. W najgorszym wariancie to wykonanie kodu spowoduje zużycie 98 MB.

Kolejny przydatny wskaźnik to `Major (requiring I/O) page faults`, który określa, czy system operacyjny musi załadować z dysku strony danych, ponieważ nie ma ich już w pamięci RAM. Spowoduje to spadek szybkości. W omawianym tutaj przykładzie nie ma to miejsca, gdyż nie zarejestrowano żadnych błędów stron danych.

Ponieważ w przykładzie wymagania kodu i danych są niewielkie, nie występują błędy stronicowania. W przypadku procesu powiązanego z pamięcią lub kilku programów, które używają zmiennej i dużej ilości pamięci RAM, może to umożliwić zidentyfikowanie programu spowalnianego przez operacje dostępu do dysku na poziomie systemu operacyjnego, ponieważ części programu zostały przeniesione z pamięci RAM na dysk.

Użycie modułu cProfile

Moduł `cProfile` to wbudowane narzędzie do profilowania w bibliotece standardowej. Jest ono podłączane do maszyny wirtualnej w module `cProfile` w celu pomiaru czasu, jaki zajmuje uruchomienie każdej napotkanej funkcji. Choć powoduje to znaczne obciążenie, pozwala uzyskać odpowiednio większą ilość informacji. Czasami dodatkowe informacje mogą doprowadzić do zaskakujących wniosków dotyczących kodu.

Moduł `cProfile` to jedno z dwóch narzędzi profilujących w bibliotece standardowej. Drugie narzędzie to `profile`. `profile` to oryginalne narzędzie profilujące bazujące wyłącznie na kodzie w Pythonie. Moduł `cProfile` zawiera ten sam interfejs co narzędzie `profile`, a ponadto został napisany w języku C w celu zwiększenia wydajności. Jeśli ciekawi Cię historia tych bibliotek, zajrzyj na stronę *Armina Rigo* (*https://mail.python.org/pipermail/python-dev/2005-November/058212.html*), który w 2005 roku wystosował prośbę o dołączenie modułu `cProfile` do biblioteki standardowej.

Dobrą praktyką w czasie profilowania jest utworzenie *hipotezy* dotyczącej szybkości części kodu przed rozpoczęciem profilowania go. Jeden z autorów woli wydrukować rozpatrywany fragment kodu i dołączyć do niego adnotacje. Wcześniejsze zdefiniowanie hipotezy oznacza, że możesz stwierdzić, jak bardzo się mylisz (i nadal będziesz!), a ponadto możesz poprawić intuicję odnośnie do konkretnych stylów tworzenia kodu.

Nigdy nie należy unikać profilowania z powodu przeczucia (ostrzegamy Cię, ponieważ *zostanie* to źle przez Ciebie zrozumiane!). Zdecydowanie warto sformułować hipotezę przed rozpoczęciem profilowania, aby ułatwić sobie opanowanie umiejętności wykrywania w kodzie możliwych działań zmniejszających szybkość. Ponadto zawsze należy poprzeć dowodem podejmowane decyzje.

Zawsze kieruj się wynikami uzyskanymi przez pomiar i zaczynaj od podstawowego profilowania, aby upewnić się, że dotyczy ono właściwego obszaru. Nie ma nic bardziej upokarzającego niż zręczne optymalizowanie sekcji kodu tylko po to, aby uświadomić sobie (wiele godzin lub dni później), że pominięto najwolniejszą część procesu i w rzeczywistości w ogóle nie zajęto się zasadniczym problemem.

Przyjmijmy hipotezę, że funkcja `calculate_z_serial_purepython` będzie prawdopodobnie najwolniejszym elementem kodu. W tej funkcji przeprowadzanych jest wiele operacji usuwania odniesień i tworzenia licznych odwołań do podstawowych operatorów arytmetycznych i funkcji `abs`. Okażą się one raczej tymi, które zużywają sporo zasobów procesora.

Moduł `cProfile` zostanie użyty do uruchomienia wariantu kodu. Choć dane wyjściowe są skromne, pomagają zidentyfikować miejsce do dalszej analizy.

Flaga `-s cumulative` nakazuje modułowi `cProfile` sortowanie według łącznego czasu, jaki upłynął w obrębie każdej funkcji. Pozwala to nam uzyskać wgląd w najwolniejsze części sekcji kodu. Dane wyjściowe modułu są wyświetlane na ekranie bezpośrednio po standardowych wynikach polecenia `print`:

```
$ python -m cProfile -s cumulative julia1_nopil.py
Długość dla x: 1000
Łączna liczba elementów: 1000000
Działanie funkcji calculate_z_serial_purepython trwało 13.15 s
         36221995 function calls in 14.301 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   14.301   14.301 {built-in method builtins.exec}
        1    0.035    0.035   14.301   14.301 julia1_nopil.py:1(<module>)
        1    0.803    0.803   14.267   14.267 julia1_nopil.py:23
                                               (calc_pure_python)
        1    8.420    8.420   13.150   13.150 julia1_nopil.py:9
                                               (calculate_z_serial_purepython)
 34219980    4.730    0.000    4.730    0.000 {built-in method builtins.abs}
  2002000    0.306    0.000    0.306    0.000 {method 'append' of 'list'
                                                objects}
        1    0.007    0.007    0.007    0.007 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                                '_lsprof.Profiler' objects}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
```

Sortowanie według łącznego czasu pozwala zorientować się, w jakim miejscu upływa większość czasu wykonywania. Uzyskany wynik pokazuje, że 36 221 995 wywołania funkcji wystąpiły w czasie wynoszącym zaledwie ponad 13 sekund (tym razem uwzględnia to obciążenia związane z użyciem modułu `cProfile`). Wcześniej wykonanie kodu zajęło około 5 sekund. Po prostu dodano obciążenie wynoszące 8 sekundy podczas pomiaru czasu wykonania każdej funkcji.

Można zauważyć, że punkt wejścia do kodu skryptu `julia1_nopil.py` w wierszu 1. zajmuje łącznie 14 sekund. Jest to punkt wejścia do modułu, który rozpoczyna się od wykonania instrukcji

`import`. W kolumnie `ncalls` jest wartość 1, która wskazuje, że wiersz ten jest wykonywany tylko raz.

W obrębie funkcji `calc_pure_python` wywołanie funkcji `calculate_z_serial_purepython` zajmuje 13 sekund. Obie funkcje są wywoływane tylko raz. Można wywnioskować, że w przybliżeniu jedna sekunda jest poświęcana na wiersze kodu w obrębie funkcji `calc_pure_python`, niezależnie od wywołania funkcji `calculate_z_serial_purepython`, która intensywnie korzysta z procesora. Nie można jednak stwierdzić przy użyciu modułu `cProfile`, *jakie* wiersze zajmują czas w obrębie funkcji.

Czas poświęcany na wiersze kodu (bez wywoływania innych funkcji) wewnątrz funkcji `calculate_z_serial_purepython` wynosi 8 sekund. Funkcja tworzy 34 219 980 wywołań funkcji `abs`, co w sumie zajmuje 4 sekundy, uwzględniając kilka innych wywołań, które nie zajmują wiele czasu.

A co z wywołaniem `{abs}`? Jego wiersz służy do pomiaru czasu poszczególnych wywołań funkcji `abs` w obrębie funkcji `calculate_z_serial_purepython`. Choć czas przypadający na pojedyncze wywołanie jest nieistotny (został zarejestrowany jako 0,000 sekundy), łączny czas dla 34 219 980 wywołań to 4 sekundy. Nie możemy z góry przewidzieć, ile dokładnie zostanie wykonanych wywołań funkcji `abs`, ponieważ funkcja zbioru Julii cechuje się nieprzewidywalną dynamiką (z tego właśnie powodu jest to tak godne zainteresowania zagadnienie).

W najlepszym razie możemy określić, że funkcja będzie wywoływana co najmniej 1 000 000 razy, gdy obliczenia są przeprowadzane dla iloczynu `1000*1000` pikseli. Co najwyżej, funkcja zostanie wywołana 300 000 000 razy, jeśli obliczenia dotyczą 1 000 000 pikseli przy maksymalnej liczbie iteracji wynoszącej 300. Oznacza to, że 34 miliony wywołań to w przybliżeniu 10% najgorszego wariantu.

Jeśli przyjrzymy się oryginalnemu obrazowi skali szarości (rysunek 2.3) i wyobrazimy sobie, że jego białe części zostały ściśnięte w narożniku, możemy oszacować, że kosztowny biały obszar stanowi mniej więcej 10% reszty obrazu.

W następnym wierszu profilowanych danych wyjściowych (`{method 'append' of 'list' objects}`) podano informację o utworzeniu 2 002 000 pozycji listy.

Zadaj sobie następujące pytanie: dlaczego jest 2 002 000 pozycji? Przed dalszą lekturą zastanów się, ile jest tworzonych pozycji listy.

Tworzenie takiej liczby pozycji ma miejsce w funkcji `calc_pure_python` na etapie konfigurowania.

Listy `zs` i `cs` będą zawierać po `1000*1000` pozycji (generowanych jest 1 000 000 · 2 wywołań). Listy te są budowane przy użyciu listy 1000 współrzędnych x i 1000 współrzędnych y. Łącznie do dodania jest 2 002 000 wywołań.

Godne uwagi jest to, że te dane wyjściowe modułu `cProfile` nie są porządkowane przez funkcje nadrzędne. Moduł podsumowuje koszt wszystkich funkcji w wykonywanym bloku kodu. Stwierdzenie, co ma miejsce w poszczególnych wierszach kodu, jest bardzo trudne w przypadku modułu `cProfile`, ponieważ uzyskiwane są jedynie informacje profilowania dla samych wywołań funkcji, a nie dla każdego wiersza w obrębie funkcji.

Wewnątrz funkcji `calculate_z_serial_purepython` można przeprowadzić obliczenie dla funkcji `{abs}`, która w sumie powoduje koszt wynoszący w przybliżeniu 4,7 sekundy. Wiemy, że łączny koszt funkcji `calculate_z_serial_purepython` to 13,1 sekundy.

Blisko końca danych wyjściowych profilowania odnosi się do narzędzia `lsprof`. Jest to oryginalna nazwa narzędzia, które rozwinęło się do postaci modułu `cProfile`. Nazwa może zostać zignorowana.

Aby uzyskać większą kontrolę nad wynikami modułu `cProfile`, możesz utworzyć plik statystyk, a następnie dokonać jego analizy za pomocą języka Python:

```
$ python -m cProfile -o profile.stats julia1_nopil.py
```

Po załadowaniu w następujący sposób pliku w interpreterze języka Python zostanie uzyskany taki sam jak wcześniej raport czasu łącznego:

```
In [1]: import pstats
In [2]: p = pstats.Stats("profile.stats")
In [3]: p.sort_stats("cumulative")
Out[3]: <pstats.Stats at 0x7fe8747e8470>

In [4]: p.print_stats()
Thu Feb 22 20:38:55 2024    profile.stats

         36221995 function calls in 14.398 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000   14.398   14.398 {built-in method builtins.exec}
        1    0.036    0.036   14.398   14.398 julia1_nopil.py:1(<module>)
        1    0.799    0.799   14.363   14.363 julia1_nopil.py:23
                                               (calc_pure_python)
        1    8.453    8.453   13.252   13.252 julia1_nopil.py:9
                                               (calculate_z_serial_purepython)
 34219980    4.799    0.000    4.799    0.000 {built-in method builtins.abs}
  2002000    0.304    0.000    0.304    0.000 {method 'append' of 'list' objects}
        1    0.008    0.008    0.008    0.008 {built-in method builtins.sum}
        3    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 {method 'disable' of
                                                '_lsprof.Profiler' objects}
        2    0.000    0.000    0.000    0.000 {built-in method time.time}
        4    0.000    0.000    0.000    0.000 {built-in method builtins.len}
```

W celu śledzenia tego, jakie funkcje są profilowane, możesz wyświetlić informacje o elemencie wywołującym. Na dwóch poniższych listingach możesz zobaczyć, że funkcja `calculate_z_serial_purepython` ma największy koszt, a ponadto wywoływana jest z jednego miejsca. Jeśli ta funkcja zostałaby wywołana z wielu miejsc, listingi te mogłyby być pomocne przy zawężaniu położeń najbardziej kosztownych funkcji nadrzędnych:

```
In [5]: p.print_callers()
   Ordered by: cumulative time

Function                                  was called by...
                                              ncalls  tottime  cumtime
```

```
{built-in method builtins.exec}         <-
julia1_nopil.py:1(<module>)             <-       1    0.036   14.398
                                                  {built-in method builtins.exec}
julia1_nopil.py:23(calc_pure_python)    <-       1    0.799   14.363
                                                  :1(<module>)
julia1_nopil.py:9(...)                  <-       1    8.453   13.252
                                                  :23(calc_pure_python)
{built-in method builtins.abs}          <- 34219980    4.799    4.799
                                                  :9(calculate_z_serial_purepython)
{method 'append' of 'list' objects}     <- 2002000    0.304    0.304
                                                  :23(calc_pure_python)
{built-in method builtins.sum}          <-       1    0.008    0.008
                                                  :23(calc_pure_python)
{built-in method builtins.print}        <-       3    0.000    0.000
                                                  :23(calc_pure_python)
{built-in method time.time}             <-       2    0.000    0.000
                                                  :23(calc_pure_python)
{built-in method builtins.len}          <-       2    0.000    0.000
                                                  :9(calculate_z_serial_purepython)
```

Aby pokazać, jakie funkcje wywołują inne funkcje, powyższe dane wyjściowe można przedstawić w następujący sposób:

```
In [6]: p.print_callees()
   Ordered by: cumulative time

Function                                          called...
                                              ncalls  tottime  cumtime
{built-in method builtins.exec}         ->        1    0.036   14.398
                                                  julia1_nopil.py:1(<module>)
julia1_nopil.py:1(<module>)             ->        1    0.799   14.363
                                                  julia1_nopil.py:23(calc..._python)
julia1_nopil.py:23(calc_pure_python)    ->        1    8.453   13.252
                                                  julia1_nopil.py:9(...)
                                                            2    0.000    0.000
                                                  {built-in method builtins.len}
                                                            3    0.000    0.000
                                                  {built-in method builtins.print}
                                                            1    0.008    0.008
                                                  {built-in method builtins.sum}
                                                            2    0.000    0.000
                                                  {built-in method time.time}
                                                       2002000    0.304    0.304
                                                  {method 'append' of 'list' objects}
julia1_nopil.py:9(...)                  -> 34219980    4.799    4.799
                                                  {built-in method builtins.abs}
                                                            2    0.000    0.000
                                                  {built-in method builtins.len}
{built-in method builtins.abs}          ->
{method 'append' of 'list' objects}     ->
{built-in method builtins.sum}          ->
{built-in method builtins.print}        ->
{built-in method time.time}             ->
{built-in method builtins.len}          ->
```

Ze względu na to, że moduł `cProfile` udostępnia raczej sporo informacji, w celu wyświetlenia ich bez intensywnego korzystania z funkcji zawijania wierszy konieczny będzie szeroki ekran.

Ponieważ jednak moduł jest wbudowany, stanowi wygodne narzędzie do szybkiego identyfikowania wąskich gardeł. Takie narzędzia jak `line_profiler` i `memory_profiler`, omówione w dalszej części rozdziału, ułatwią przejście do konkretnych wierszy, na które należy zwrócić uwagę.

Użycie narzędzia SnakeViz do wizualizacji danych wyjściowych modułu cProfile

SnakeViz to narzędzie do wizualizacji, które pobiera dane wyjściowe modułu `cProfile` w postaci diagramu. Większe pola w jego obrębie identyfikują obszary kodu o dłuższym czasie wykonywania. Narzędzie zastępuje starsze narzędzie `runsnake`.

Użyj narzędzia SnakeViz do ogólnego zaznajomienia się z plikiem statystyk modułu `cProfile`, zwłaszcza wtedy, gdy analizowany jest nowy projekt, który dopiero poznajesz. Diagram ułatwi wizualizację zachowania systemu związanego z wykorzystaniem procesora, a ponadto może uwidocznić obszary, w wypadku których nie spodziewałeś się, że będą kosztowne.

Aby zainstalować narzędzie SnakeViz, wykonaj polecenie `pip install snakeviz`.

Rysunek 2.5 przedstawia wizualizację danych wyjściowych wygenerowanego właśnie pliku *profile.stats*. Punkt wejścia programu widoczny jest u samej góry diagramu. Każda kolejna niżej położona warstwa jest funkcją wywoływaną z poziomu wyżej umiejscowionej funkcji.

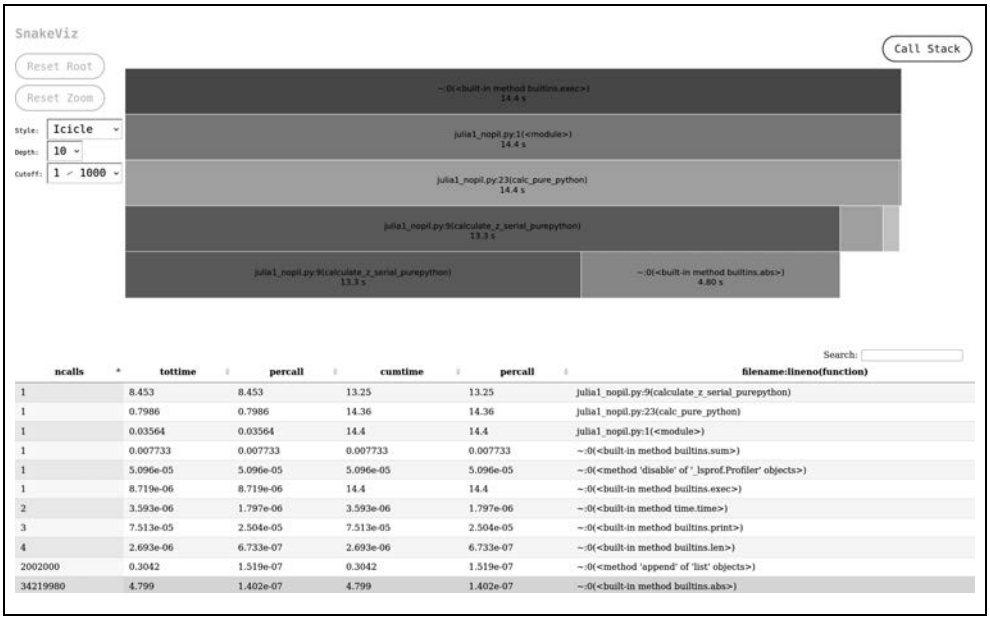
Szerokość diagramu reprezentuje całkowity czas wykonywania programu. Czwarta warstwa pokazuje, że większość czasu poświęcono na wykonywanie kodu funkcji `calculate_z_serial_purepython`. W wypadku piątej warstwy dokonano dodatkowego rozbicia — blok bez adnotacji po prawej stronie, który w przybliżeniu zajmuje 33% obszaru tej warstwy, reprezentuje czas poświęcony na wykonywanie kodu funkcji `abs`. Zaznajomienie się z tymi większymi blokami pozwala szybko zorientować się, na co poświęcany jest czas w obrębie programu.

Poniżej słupków graficznych zamieszczono tabelę, która jest ładnie zaprezentowaną wersją statystyk, jakie właśnie zostały omówione. Tabela może być sortowana przy użyciu takich kategorii jak między innymi `cumtime` (skumulowany czas), `percall` (koszt wywołania) lub `ncalls` (liczba wszystkich wywołań). Zastosowanie najpierw kategorii `cumtime` umożliwi stwierdzenie, jakie funkcje powodują ogólnie największy koszt. Wszelkie sprawdzenia naprawdę warto zacząć od tego.

Jeśli odpowiada Ci analizowanie tabel, odpowiednie mogą okazać się dane wyjściowe modułu `cProfile` wyświetlane w konsoli. Aby porozumieć się z innymi osobami, szczególnie zalecane jest korzystanie z diagramów, takich jak przedstawione wcześniej dane wyjściowe narzędzia SnakeViz. Ułatwi to innym szybkie zrozumienie podnoszonej kwestii.

Użycie narzędzia line_profiler do pomiarów dotyczących kolejnych wierszy kodu

W opinii jednego z autorów narzędzie `line_profiler` Roberta Kerna oferuje największe możliwości identyfikowania w kodzie w Pythonie przyczyny problemów powiązanych z procesorem. Działanie narzędzia polega na profilowaniu wiersz po wierszu poszczególnych funkcji. Oznacza to, że należy zacząć od modułu `cProfile` i skorzystać z ogólnego widoku, który pozwoli określić, jakie funkcje mają być profilowane za pomocą narzędzia `line_profiler`.



Rysunek 2.5. Wizualizacja pliku profile.stats przy użyciu narzędzia SnakeViz

Warto podczas modyfikowania kodu wyświetlać wersje danych wyjściowych tego narzędzia i tworzyć dla nich adnotacje, ponieważ dzięki temu uzyskuje się zapis zmian (pomyślnych lub nie), do których można szybko wrócić. Nie polegaj na pamięci przy wprowadzaniu zmian w poszczególnych wierszach.

Aby zainstalować narzędzie `line_profiler`, wykonaj polecenie `pip install line_profiler`.

Dekorator (`@profile`) służy do oznaczania wybranej funkcji. Skrypt `kernprof` jest używany do wykonywania kodu. Rejestrowany jest czas pracy procesora oraz inne statystyki dla każdego wiersza wybranej funkcji.

Argument `-l` umożliwia profilowanie wiersz po wierszu, a argument `-v` powoduje zwrócenie szczegółowych danych wyjściowych. Bez argumentu `-v` zostaną uzyskane dane wyjściowe w postaci pliku *.lprof*, które możesz później poddać analizie przy użyciu modułu `line_profiler`. Przykład 2.6 prezentuje pełne uruchomienie funkcji powiązanej z procesorem.

Przykład 2.6. Uruchomienie skryptu kernprof z danymi wyjściowymi dotyczącymi poszczególnych wierszy dla funkcji z dekoratorem w celu zarejestrowania czasu wykonywania przez procesor każdego wiersza kodu

```
$ kernprof.py -l -v julia1_lineprofiler.py
...
Wrote profile results to julia1_lineprofiler.py.lprof
Timer unit: 1e-06 s

Total time: 31.0996 s
File: julia1_lineprofiler.py
Function: calculate_z_serial_purepython at line 16

Line #      Hits   Per Hit   % Time  Line Contents
==============================================
    16                               @profile
    17                               def calculate_z_serial_purepython(maxiter,
                                                                       zs, cs):
    18                                   """Obliczanie listy output przy użyciu
                                                reguły aktualizacji zbioru Julii"""
    19         1    3720.4      0.0      output = [0] * len(zs)
    20   1000001       0.3      0.8      for i in range(len(zs)):
    21   1000000       0.2      0.6          n = 0
    22   1000000       0.2      0.7          z = zs[i]
    23   1000000       0.2      0.7          c = cs[i]
    24  34219980       0.4     44.7          while abs(z) < 2 and n < maxiter:
    25  33219980       0.3     29.2              z = z * z + c
    26  33219980       0.2     22.5              n += 1
    27   1000000       0.2      0.7          output[i] = n
    28         1       3.6      0.0      return output
```

Zastosowanie skryptu `kernprof.py` powoduje znaczne wydłużenie czasu działania. W tym przykładzie wykonanie funkcji `calculate_z_serial_purepython` zajmuje około 31 sekund. Jest to spory skok w porównaniu z 5 i 13 sekundami odpowiednio w przypadku użycia prostych instrukcji `print` i modułu `cProfile`. Korzyścią jest to, że możliwe jest przeanalizowanie wiersz po wierszu tego, jak upływa czas w obrębie funkcji.

Kolumna `% Time` jest najbardziej pomocna. Jak widać, 44% czasu zajmuje testowanie pętli `while`. Nie wiemy jednak, czy pierwsza instrukcja (`abs(z) < 2`) zajmuje więcej czasu niż druga (`n < maxiter`). Wewnątrz pętli widać, że aktualizacja liczby `z` również zajmuje sporo czasu. Nawet instrukcja `n += 1` kosztuje wiele czasu! Mechanizm dynamicznego wyszukiwania języka Python działa w przypadku każdej pętli, nawet pomimo tego, że dla każdej zmiennej w każdej pętli używane są takie same typy. W tym przypadku kompilowanie i specjalizacja typów (rozdział 8.) zapewnią ogromną poprawę. Tworzenie listy `output` i aktualizacje w wierszu 19. zajmują stosunkowo mało czasu w porównaniu z pętlą `while`.

Jeśli wcześniej nie zastanawiałeś się nad złożonością dynamicznych mechanizmów języka Python, pomyśl, co się stanie w wypadku prezentowanej operacji `n += 1`. Interpreter języka Python musi sprawdzić, czy obiekt `n` zawiera funkcję `__add__` (jeśli nie, nastąpi przejście w górę przez wszystkie dziedziczone klasy w celu stwierdzenia, czy zapewniają one tę funkcję). W dalszej kolejności przekazywany jest kolejny obiekt (`1` w tym wypadku), dzięki czemu funkcja `__add__` może zdecydować, w jaki sposób zrealizować operację. Pamiętaj, że drugi

argument może być obiektem `float` lub innym obiektem, który niekoniecznie może być zgodny. Wszystko to odbywa się dynamicznie.

Oczywistą metodą dalszego analizowania instrukcji pętli `while` jest jej rozbicie. Choć w społeczności związanej z językiem Python pojawiła się dyskusja dotycząca pomysłu ponownego tworzenia plików *.pyc* przy użyciu bardziej szczegółowych informacji na potrzeby wieloczęściowych instrukcji w postaci jednego wiersza, nieznane są nam żadne narzędzia produkcyjne, które oferują bardziej dokładną analizę niż narzędzie `line_profiler`.

W przykładzie 2.7 dokonano rozbicia na kilka instrukcji logiki pętli `while`. Taka dodatkowa złożoność zwiększy czas działania funkcji, ponieważ będzie więcej wierszy kodu do wykonania. *Może* to jednak być pomocne w zrozumieniu kosztów związanych z wykonywaniem tej części kodu.

Zanim przyjrzysz się kodowi, czy myślisz, że w ten sposób dowiemy się, jaki jest czas wykonywania fundamentalnych operacji? Czy inne czynniki mogą skomplikować analizę?

Przykład 2.7. Rozbicie złożonej instrukcji pętli while na poszczególne instrukcje w celu zarejestrowania czasu wykonania dla każdej części oryginalnej instrukcji

```
$ kernprof -l -v julia1_lineprofiler2.py
...
Wrote profile results to julia1_lineprofiler2.py.lprof
Timer unit: 1e-06 s

Total time: 63.3558 s
File: julia1_lineprofiler2.py
Function: calculate_z_serial_purepython at line 15

Line #      Hits    Per Hit   % Time  Line Contents
==============================================
    15                                @profile
    16                                def calculate_z_serial_purepython(maxiter,
                                                                        zs, cs):
    17                                    """Obliczanie listy output przy użyciu
                                             reguły aktualizacji zbioru Julii"""
    18         1     3862.9      0.0      output = [0] * len(zs)
    19   1000001        0.3      0.5      for i in range(len(zs)):
    20   1000000        0.3      0.4          n = 0
    21   1000000        0.3      0.5          z = zs[i]
    22   1000000        0.3      0.4          c = cs[i]
    23  34219980        0.2     12.3          while True:
    24  34219980        0.4     21.4              not_yet_escaped = abs(z) < 2
    25  34219980        0.3     14.8              iterations_left = n < maxiter
    26  34219980        0.3     18.6              if not_yet_escaped
                                                      and iterations_left:
    27  33219980        0.3     15.7                  z = z * z + c
    28  33219980        0.3     14.6                  n += 1
    29                                            else:
    30   1000000        0.2      0.4                  break
    31   1000000        0.3      0.4          output[i] = n
    32         1        3.0      0.0      return output
```

Wykonanie tej wersji kodu zajmuje 63 sekundy, w przypadku poprzedniej wersji było to natomiast 31 sekund. Inne czynniki *utrudniły* analizę. W tym przypadku kod jest spowalniany przez dodatkowe instrukcje, które muszą zostać wykonane 34 219 980 razy. Jeśli nie zostałby użyty skrypt `kernprof.py` do analizowania wiersz po wierszu efektu tej zmiany, z powodu braku niezbędnego dowodu być może zostałyby wyciągnięte inne wnioski na temat przyczyny spowolnienia.

Na tym etapie sensowne jest cofnięcie się o krok, do wcześniejszej metody opartej na module `timeit`, aby sprawdzić czas wykonywania poszczególnych wyrażeń:

```
Python 3.12.0 | packaged by Anaconda, Inc. | (main, Oct 2 2023, 17:29:18)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.21.0 -- An enhanced Interactive Python. Type '?' for help.
In [1]: z = 0+0j
In [2]: %timeit abs(z) < 2
66.8 ns ± 2.07 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
In [3]: n = 1
In [4]: maxiter = 300
In [5]: %timeit n < maxiter
20.7 ns ± 0.0611 ns per loop (mean ± std. dev. of 7 runs, 10000000 loops each)
```

Na podstawie tej prostej analizy można stwierdzić, że test logiki dla `n` jest ponad trzykrotnie szybszy niż wywołanie funkcji `abs`. Ponieważ wartości dla wyrażeń języka Python są określane zarówno od lewej do prawej strony, jak i oportunistycznie, sensowne jest umieszczenie najszybszego testu po lewej stronie równania. W przypadku jednego testu z każdej grupy 301 testów wykonywanych dla każdej współrzędnej test warunku `n < maxiter` da wartość `False`. Oznacza to, że interpreter języka Python nie będzie wymagał określenia wartości drugiej strony operatora `and`.

Do momentu określenia dla niego wartości nie wiemy, czy warunek `abs(z) < 2` da wartość `False`. Wcześniejsze obserwacje dla tego obszaru przestrzeni zespolonej sugerują, że jest to wartość `True` dla około 10% czasu w przypadku wszystkich 300 iteracji. Aby dobrze zrozumieć złożoność dotyczącą czasu dla tej części kodu, sensowne byłoby kontynuowanie analizy numerycznej. W tej sytuacji zależy nam jednak na prostym sprawdzeniu, czy możliwe jest szybkie zwiększenie szybkości kodu.

Możemy sformułować nową hipotezę, która brzmi: „Zmieniając kolejność operatorów w instrukcji `while`, osiągniemy solidne przyspieszenie”. Choć hipotezę tę *można* sprawdzić za pomocą skryptu `kernprof`, dodatkowe obciążenie związane z profilowaniem w ten sposób może spowodować zbyt dużo zamieszania. Możemy więc użyć wcześniejszej wersji kodu, uruchamiając test, który porównuje instrukcję `while abs(z) < 2 and n < maxiter` z instrukcją `while n < maxiter and abs(z) < 2:` (zaprezentowano to w przykładzie 2.8).

Wykonanie dwóch wariantów *poza* narzędziem `line_profiler` oznacza, że odbywa się to z podobną szybkością w przypadku zastosowanego przez nas problemu o takiej samej skali (z `x==1000`).

Wynik jest też zaburzany przez dodatkowe obciążenie generowane przez narzędzie `line_profiler`. Wyniki widoczne w wierszu 23. są podobne dla obu wariantów. W przypadku tego końcowego wyniku czas trwania jest trochę dłuższy, gdyż wynosi 33 sekundy. Można odrzucić

hipotezę, zgodnie z którą w wypadku języka Python 3.12 zmiana kolejności w logice powoduje ciągłe przyspieszenie. Nie ma na to wyraźnego dowodu.

Jeśli jednak zwiększy się poziom trudności problemu przez ustawienie wartości `x==5000`, wersja z zamienioną instrukcją `if` konsekwentnie będzie trochę szybsza. W przypadku pierwotnej kolejności po dokonaniu pomiaru jeden z autorów uzyskał czas 143 sekundy, a dla zmienionej kolejności operacji czas wyniósł 142 sekundy.

Zastosowanie odpowiedniejszej metody do rozwiązania tego problemu (np. skorzystanie z narzędzia Cython lub PyPy w sposób opisany w rozdziale 8.) dałoby większe wzrosty szybkości.

Możemy być pewni otrzymanego wyniku, ponieważ:

- Określono hipotezę prostą do sprawdzenia.
- Zmodyfikowano kod w taki sposób, aby była sprawdzana wyłącznie hipoteza (nigdy nie sprawdzaj dwóch rzeczy jednocześnie!).
- Uzyskano wystarczający dowód na poparcie wyciągniętego wniosku.

Aby wszystko było kompletne, możemy uruchomić skrypt `kernprof` dla dwóch głównych funkcji z uwzględnieniem optymalizacji w celu potwierdzenia, że dysponujemy pełnym obrazem ogólnej złożoności kodu.

Przykład 2.8. Zmiana miejsca występowania instrukcji pętli while w kodzie liczb zespolonych w celu nieznacznego przyspieszenia testu

```
$ kernprof -l -v julia1_lineprofiler3.py
...
Wrote profile results to julia1_lineprofiler3.py.lprof
Timer unit: 1e-06 s

Total time: 33.9135 s
File: julia1_lineprofiler3.py
Function: calculate_z_serial_purepython at line 15

Line #      Hits   Per Hit   % Time  Line Contents
==============================================
    15                               @profile
    16                               def calculate_z_serial_purepython(maxiter,
                                                                       zs, cs):
    17                                   """Obliczanie listy output przy użyciu
                                                reguły aktualizacji zbioru Julii"""
    18         1    4015.8      0.0      output = [0] * len(zs)
    19   1000001       0.3      1.0      for i in range(len(zs)):
    20   1000000       0.2      0.7          n = 0
    21   1000000       0.3      0.8          z = zs[i]
    22   1000000       0.2      0.7          c = cs[i]
    23  34219980       0.4     44.1          while n < maxiter and abs(z) < 2:
    24  33219980       0.3     29.0              z = z * z + c
    25  33219980       0.2     22.9              n += 1
    26   1000000       0.3      0.8          output[i] = n
    27         1       2.1      0.0      return output
```

Zgodnie z oczekiwaniami na podstawie danych wyjściowych kodu z przykładu 2.9 widać, że wykonywanie funkcji `calculate_z_serial_purepython` zajmuje większość czasu (98%) działania jej funkcji nadrzędnej. Dla porównania kroki związane z tworzeniem listy zajmują znikomą ilość czasu.

Przykład 2.9. Testowanie wiersz po wierszu czasu wykonywania programu konfiguracyjnego

```
Total time: 64.0333 s
File: julia1_lineprofiler3.py
Function: calc_pure_python at line 30

Line #      Hits   Per Hit   % Time  Line Contents
================================================
    30                               @profile
    31                               def calc_pure_python(draw_output,
                                                          desired_width,
                                                          max_iterations):
...
    52      1001       0.3      0.0      for ycoord in y:
    53   1001000       0.3      0.5          for xcoord in x:
    54   1000000       0.4      0.6              zs.append(complex(xcoord, ycoord))
    55   1000000       0.4      0.6              cs.append(complex(c_real, c_imag))
    56
    57         1      56.2      0.0      print(f"Długość dla x: {len(x):,}")
    58         1       7.4      0.0      print(f"Łączna liczba elementów: {len(zs):,}")
    59         1       5.1      0.0      start_time = time.time()
    60         1     6e+07     98.3      output = calculate_z_serial_purepython(
                                                    max_iterations, zs, cs)
    61         1       6.0      0.0      end_time = time.time()
    62         1       1.3      0.0      secs = end_time - start_time
    63         1      48.6      0.0      print(f"Działanie funkcji {calculate_
                                         ↪z_serial_purepython.__name__}
                                                     trwało {secs:0.2f} s")
    64
    65         1    7129.7      0.0      assert sum(output) == 33219980 # suma ta jest
                                           # oczekiwana w przypadku siatki 1000^2 z 300 iteracjami
```

Narzędzie `line_profiler` pozwala dokładnie zaznajomić się z kosztem związanym z wierszami kodu w obrębie pętli i wymagających funkcji. Nawet pomimo tego, że profilowanie zmniejsza szybkość, stanowi prawdziwy skarb dla projektantów i jednocześnie naukowców. Pamiętaj o stosowaniu reprezentatywnych danych, aby zapewnić, że koncentrujesz się na wierszach kodu, które pozwolą uzyskać największe korzyści.

Użycie narzędzia memory_profiler do diagnozowania wykorzystania pamięci

Tak jak narzędzie `line_profiler` Roberta Kerna dokonuje pomiaru wykorzystania procesora, tak moduł `memory_profiler` Fabiana Pedregosa i Philippe'a Gervaisa mierzy wykorzystanie pamięci dla kolejnych wierszy kodu. Zrozumienie cech kodu związanych z wykorzystaniem pamięci umożliwia zadanie sobie następujących dwóch pytań:

- Czy możliwe jest użycie *mniejszej* ilości pamięci RAM przez modyfikację funkcji pod kątem bardziej efektywnego działania?
- Czy możliwe jest za pomocą buforowania użycie *większej* ilości pamięci RAM i zyskanie cykli procesora?

Moduł `memory_profiler` działa w sposób bardzo podobny do narzędzia `line_profiler`, ale zapewnia znacznie mniejszą wydajność. Po zainstalowaniu pakietu `psutil` (opcjonalny, lecz zalecany) moduł `memory_profiler` będzie działać szybciej. Profilowanie pamięci bez trudu może sprawić, że kod będzie wykonywany od 10 do 100 razy wolniej. W praktyce moduł ten będzie używany sporadycznie, a narzędzie `line_profiler` (do profilowania wykorzystania procesora) częściej.

Moduł `memory_profiler` zainstaluj za pomocą polecenia `pip install memory_profiler` (opcjonalnie użyj polecenia `pip install psutil`).

Jak wspomniano, implementacja modułu `memory_profiler` nie zapewnia takiej wydajności jak implementacja narzędzia `line_profiler`. A zatem sensowne może być wykonywanie testów za pomocą modułu dla mniejszego problemu. Testy te zostaną zakończone w rozsądnym czasie. Całonocne uruchomienia mogą mieć sens w przypadku sprawdzania poprawności, ale do diagnozowania problemów i określania hipotez dotyczących rozwiązań niezbędne będą szybkie i rozsądne iteracje. W kodzie z przykładu 2.10 używana jest siatka 1000×1000. W przypadku laptopa jednego z autorów zgromadzenie statystyk zajęło około 2 godzin.

Przykład 2.10. Wyniki modułu memory_profiler uzyskane dla obu głównych funkcji, które uwidaczniają nieoczekiwane użycie pamięci w funkcji calculate_z_serial_purepython

```
$ python -m memory_profiler julia1_memoryprofiler.py
...
Line #    Mem usage    Increment   Line Contents
================================================
    15  123.086 MiB  123.086 MiB   @profile
    16                             def calculate_z_serial_purepython(maxiter,
                                                                     zs, cs):
    17                                 """Obliczanie listy output...
    18  130.711 MiB    7.625 MiB       output = [0] * len(zs)
    19  133.461 MiB    0.000 MiB       for i in range(len(zs)):
    20  133.461 MiB    0.000 MiB           n = 0
    21  133.461 MiB    0.000 MiB           z = zs[i]
    22  133.461 MiB    0.000 MiB           c = cs[i]
    23  133.461 MiB    2.750 MiB           while n < maxiter and abs(z) < 2:
    24  133.461 MiB    0.000 MiB               z = z * z + c
    25  133.461 MiB    0.000 MiB               n += 1
    26  133.461 MiB    0.000 MiB           output[i] = n
    27  133.461 MiB    0.000 MiB       return output
...

Line #    Mem usage    Increment   Line Contents
================================================
    30   47.137 MiB   47.137 MiB   @profile
    31                             def calc_pure_python(draw_output,
                                                        desired_width,
                                                        max_iterations):
    32                                 """Tworzenie listy liczb...
    33   47.137 MiB    0.000 MiB       x_step = (x2 - x1) / desired_width
    34   47.137 MiB    0.000 MiB       y_step = (y1 - y2) / desired_width
    35   47.137 MiB    0.000 MiB       x = []
    36   47.137 MiB    0.000 MiB       y = []
    37   47.137 MiB    0.000 MiB       ycoord = y2
    38   47.137 MiB    0.000 MiB       while ycoord > y1:
    39   47.137 MiB    0.000 MiB           y.append(ycoord)
```

```
40   47.137 MiB    0.000 MiB           ycoord += y_step
41   47.137 MiB    0.000 MiB       xcoord = x1
42   47.137 MiB    0.000 MiB       while xcoord < x2:
43   47.137 MiB    0.000 MiB           x.append(xcoord)
44   47.137 MiB    0.000 MiB           xcoord += x_step
50   47.137 MiB    0.000 MiB       zs = []
51   47.137 MiB    0.000 MiB       cs = []
52  123.086 MiB   -1.055 MiB       for ycoord in y:
53  123.086 MiB -893.922 MiB           for xcoord in x:
54  123.086 MiB -900.219 MiB               zs.append(complex(xcoord, ycoord))
55  123.086 MiB -853.844 MiB               cs.append(complex(c_real, c_imag))
56
57  123.086 MiB    0.000 MiB      print "Długość dla x:", len(x)
58  123.086 MiB    0.000 MiB      print "Łączna liczba elementów:", len(zs)
59  123.086 MiB    0.000 MiB      start_time = time.time()
60  133.461 MiB  133.461 MiB      output = calculate_z_serial...
61  133.461 MiB    0.000 MiB      end_time = time.time()
62  133.461 MiB    0.000 MiB      secs = end_time - start_time
63  133.461 MiB    0.000 MiB      print("Działanie funkcji " +
                                  ↪calculate_z_serial_purepython.__name__ + "
                                  ↪trwało", secs, "s")
64
66  133.461 MiB    0.000 MiB       assert sum(output) == 33219980
```

Wymóg modyfikowania kodu źródłowego stanowi drobny kłopot. Podobnie jak w przypadku narzędzia `line_profiler`, dekorator (`@profile`) służy do oznaczenia wybranej funkcji. Dekorator spowoduje rozbicie testów jednostkowych, chyba że zostanie utworzony fikcyjny dekorator (więcej informacji zawiera punkt „Dekorator @profile bez operacji”).

Gdy zajmujesz się przydzielaniem pamięci, musisz mieć świadomość tego, że sytuacja nie jest tak klarowna jak w przypadku wykorzystania procesora. Ogólnie rzecz biorąc, bardziej efektywne jest nadmierne przypisanie pamięci w procesie, która może być używana w dogodnym momencie, ponieważ operacje przydziału pamięci zajmują stosunkowo dużo czasu. Ponadto czyszczenie pamięci nie następuje od razu, dlatego obiekty mogą być niedostępne, ale w dalszym ciągu przez jakiś czas mogą znajdować się w puli procesu czyszczenia pamięci.

Efekt tego jest taki, że trudno w pełni zrozumieć, co dzieje się z wykorzystaniem i zwalnianiem pamięci w obrębie programu Python. Wynika to stąd, że wiersz kodu może nie przydzielić możliwej do określenia ilości pamięci, *jaką ustalono poza obrębem procesu*. Obserwowanie ogólnego trendu dla zestawu wierszy prawdopodobnie doprowadzi do lepszych wniosków niż monitorowanie zachowania tylko jednego wiersza.

Przyjrzyjmy się danym wyjściowym modułu `memory_profiler` z przykładu 2.10. Wewnątrz funkcji `calculate_z_serial_purepython` w wierszu 18. widać, że przydzielenie 1 000 000 elementów powoduje dodanie do procesu około 7 MB pamięci RAM[1]. Nie oznacza to, że lista

[1] Moduł `memory_profiler` mierzy wykorzystanie pamięci zgodnie z jednostką MiB (mebibajt) organizacji International Electrotechnical Commission. Jednostka ta odpowiada wartości 2^{20} bajtów. Różni się trochę od powszechniejszej, lecz też bardziej niejednoznacznej jednostki MB (megabajt ma dwie ogólnie akceptowane definicje!). Jeden MiB odpowiada 1,048576 (lub w przybliżeniu 1,05) MB. Jeśli nie będzie mowy o bardzo specyficznych wielkościach, na potrzeby omówienia będziemy posługiwać się jednostką MiB.

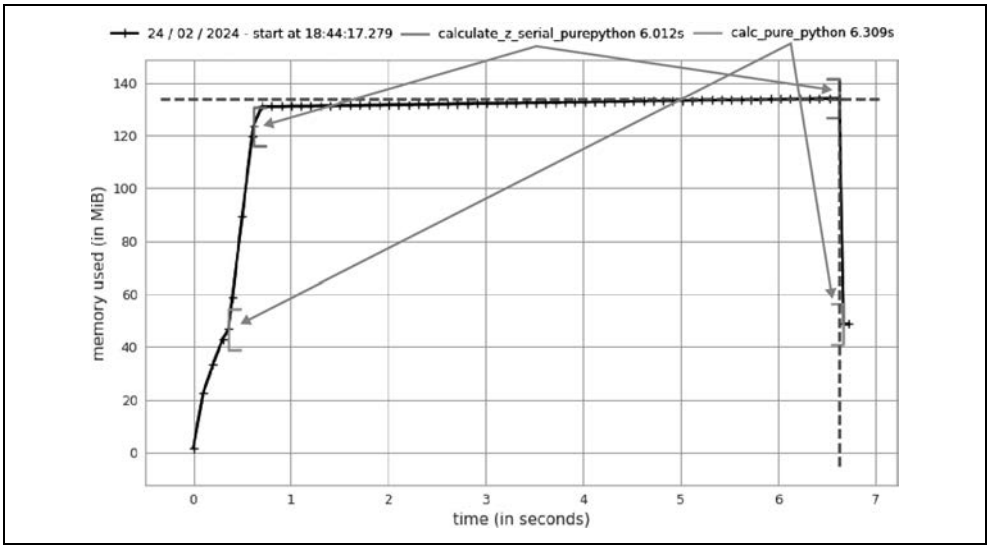
`output` na pewno ma wielkość wynoszącą 7 MB, ale jedynie to, że proces zwiększył wielkość w przybliżeniu o 7 MB w czasie wewnętrznej alokacji listy.

W funkcji nadrzędnej w wierszu 52. widoczne jest, że alokacja list `zs` i `cs` powoduje zmianę w kolumnie `Mem usage` — wykorzystanie pamięci z 47 MB do 123 MB (zmiana o 76 MB). I tym razem godne uwagi jest to, że niekoniecznie jest to rzeczywista wielkość tablic, ale jedynie wielkość, o jaką powiększył się proces po utworzeniu tych list.

Gdy pisano książkę, moduł `memory_usage` zawierał błąd polegający na tym, że wartość w kolumnie `Increment` nie zawsze była zgodna ze zmienioną wartością kolumny `Mem usage`. W przypadku pierwszego wydania książki śledzenie dla tych kolumn było poprawnie wykonywane. Status tego błędu możesz sprawdzić w serwisie GitHub (*https://github.com/pythonprofilers/memory_profiler/issues/236*). Zalecane jest użycie kolumny `Mem usage`, ponieważ w jej wypadku poprawnie śledzona jest zmiana dotycząca wielkości procesu dla poszczególnych wierszy kodu.

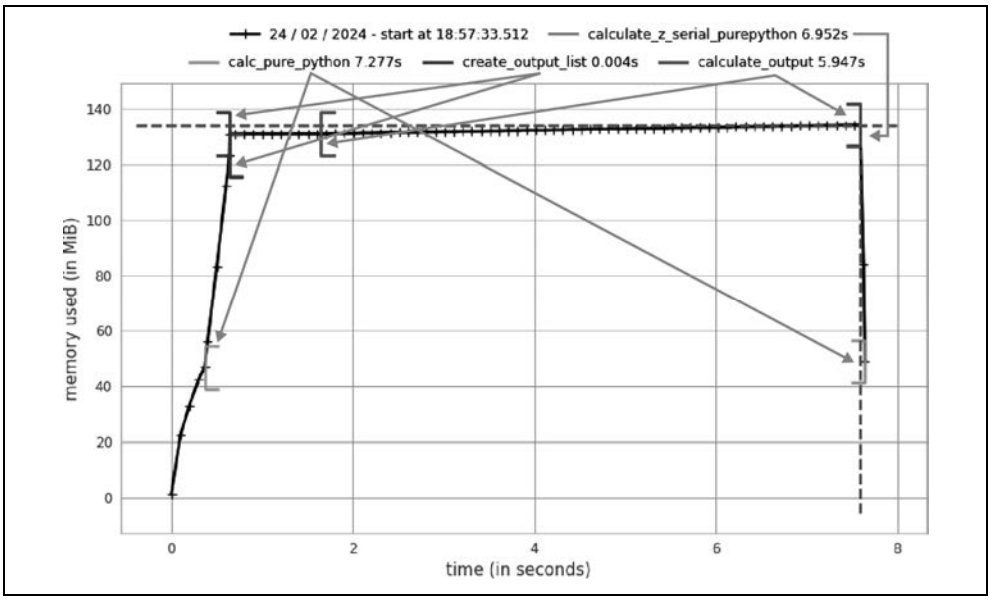
Inna metoda wizualizacji zmiany wykorzystania pamięci polega na próbkowaniu w czasie i prezentowaniu wyniku na wykresie. Narzędzie `memory_profiler` oferuje program o nazwie `mprof`, który po pierwsze, służy do próbkowania wykorzystania pamięci, a po drugie, do wizualizacji próbek. Ponieważ próbkowanie odbywa się w czasie, a nie w oparciu o wiersze kodu, ma znikomy wpływ na działanie kodu.

Rysunek 2.6 utworzono za pomocą polecenia `mprof run julia1_memoryprofiler.py`. Tworzy ono plik statystyk, który następnie jest poddawany wizualizacji przy użyciu polecenia `mprof plot`. Dwie przykładowe funkcje są zawarte w nawiasach kwadratowych. Dzięki temu widoczne jest, w jakim momencie w czasie są one aktywowane, a ponadto możliwe jest obserwowanie wzrostu wykorzystania pamięci RAM w trakcie ich działania. W obrębie funkcji `calculate_z_serial_purepython` widoczny jest ciągły wzrost wykorzystania pamięci RAM podczas jej wykonywania. Jest to spowodowane przez wszystkie niewielkie obiekty (typów `int` i `float`), które są tworzone.



Rysunek 2.6. Raport narzędzia memory_profiler wygenerowany przy użyciu programu mprof

Oprócz obserwowania zachowania na poziomie funkcji możesz dodać etykiety za pomocą menedżera kontekstów. Fragment kodu z przykładu 2.11 służy do wygenerowania wykresu z rysunku 2.7. Widoczna jest etykieta `create_output_list`, która pojawia się chwilę (około 1,5 s) po funkcji `calculate_z_serial_purepython`, powodując przydzielenie procesowi większej ilości pamięci RAM. Dalej następuje wstrzymanie na sekundę. Funkcja `time.sleep(1)` to sztuczny dodatek, którego zadaniem jest ułatwienie zrozumienia wykresu.



Rysunek 2.7. Raport narzędzia memory_profiler z etykietami wygenerowanymi przy użyciu programu mprof

Przykład 2.11. Użycie menedżera kontekstów do dodania etykiet do wykresu programu mprof

```
@profile
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Obliczanie listy output przy użyciu reguły aktualizacji zbioru Julii"""
    with profile.timestamp("create_output_list"):
        output = [0] * len(zs)
    time.sleep(1)
    with profile.timestamp("calculate_output"):
        for i in range(len(zs)):
                n = 0
                z = zs[i]
                c = cs[i]
                while n < maxiter and abs(z) < 2:
                    z = z * z + c
                    n += 1
                output[i] = n
    return output
```

W bloku etykiety `calculate_output`, którego działanie obejmuje większość wykresu, widoczny jest bardzo powolny liniowy wzrost wykorzystania pamięci RAM. Będzie to efektem użycia wszystkich liczb tymczasowych w pętlach wewnętrznych. Zastosowanie etykiet naprawdę pomaga zrozumieć, gdzie dokładnie jest wykorzystywana pamięć. Interesujące jest to, że

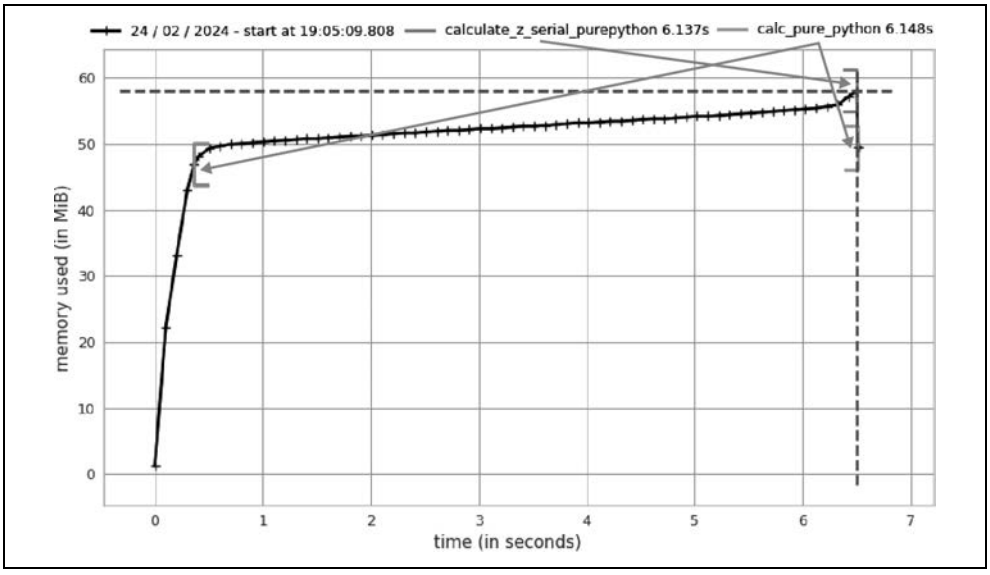
pozioma linia kreskowa szczytowego wykorzystania pamięci RAM widoczna jest tuż przed znacznikiem 7 sekund, który znajduje się przed znacznikiem momentu zakończenia programu. Potencjalnie jest to spowodowane odzyskiwaniem przez proces oczyszczania pamięci części pamięci RAM od obiektów tymczasowych używanych podczas działania funkcji `calculate_output`.

Co się stanie, jeśli uprościsz kod i wyeliminujesz tworzenie list `zs` i `cs`? W tym wypadku konieczne będzie obliczenie związanych z nimi współrzędnych w obrębie funkcji `calculate_z_serial_purepython` (oznacza to zrealizowanie tych samych działań). Dzięki temu jednak zostanie zaoszczędzona pamięć RAM, ponieważ współrzędne nie będą przechowywane na listach. Przykład 2.12 prezentuje kod.

Przykład 2.12. Dynamiczne tworzenie współrzędnych liczb zespolonych w celu zaoszczędzenia pamięci RAM

```
@profile
def calculate_z_serial_purepython(maxiter, x, y):
    """Obliczanie listy output przy użyciu reguły aktualizacji zbioru Julii"""
    output = []
    for ycoord in y:
        for xcoord in x:
            z = complex(xcoord, ycoord)
            c = complex(c_real, c_imag)
            n = 0
            while n < maxiter and abs(z) < 2:
                z = z * z + c
                n += 1
            output.append(n)
    return output
```

Na rysunku 2.8 widać odpowiedni spadek ogólnego wykorzystania pamięci RAM z 140 MB do 60 MB. Oznacza to, że wykorzystanie pamięci zmniejszyło się o połowę!



Rysunek 2.8. Raport narzędzia memory_profiler po usunięciu dwóch dużych list

Aby dokonać pomiaru ilości pamięci RAM używanej przez kilka instrukcji, można skorzystać z funkcji „magicznej” `%memit` powłoki IPython, która działa tak jak funkcja `%timeit`. W rozdziale 12. przyjrzymy się użyciu funkcji `%memit` do pomiaru ilości pamięci wykorzystywanej przez listy, a ponadto omówimy różne metody bardziej efektywnego użycia pamięci RAM.

Moduł `memory_profiler` oferuje za pośrednictwem flagi `--pdb-mmem=XXX` interesujące ułatwienie debugowania dużego procesu. Debuger `pdb` będzie aktywowany po przekroczeniu przez proces wielkości wynoszącej *XXX* MB. Spowoduje to przeniesienie bezpośrednio do miejsca w kodzie, gdzie występuje zbyt wiele alokacji w wypadku korzystania ze środowiska o ograniczonej przestrzeni.

Połączenie profilowania procesora i pamięci przy użyciu narzędzia Scalene

Narzędzie Scalene łączy profilowanie procesora, pamięci oraz procesora graficznego (GPU) w postaci jednego łatwego do uruchomienia pakietu. Działa ono na wszystkich platformach i jest instalowane za pomocą polecenia `pip install scalene`.

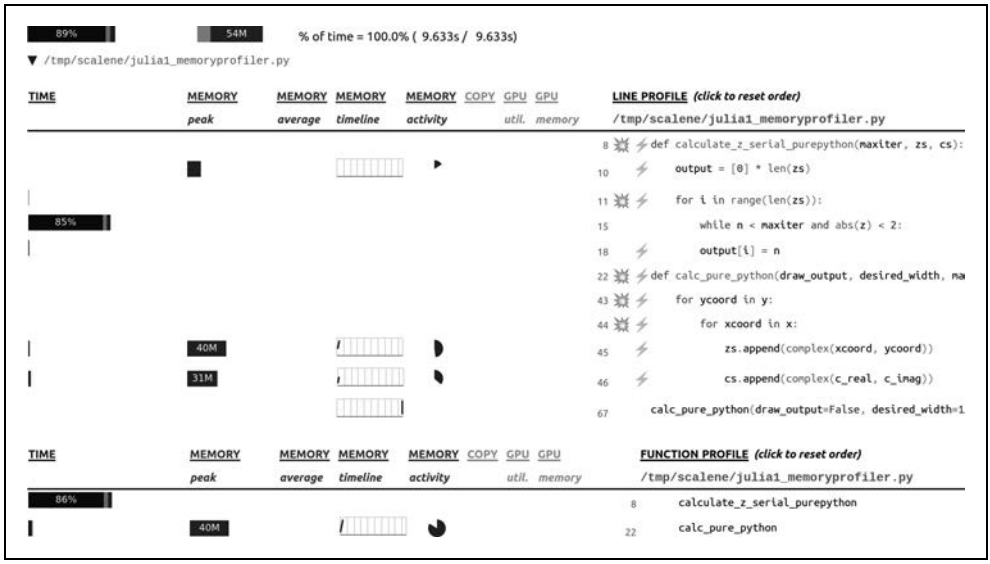
Scalene używa własnej uproszczonej biblioteki profilującej, dlatego ma bardzo niewielki wpływ na szybkość wykonywania (w przeciwieństwie do narzędzi `memory_profiler` i `line_profiler`). Narzędzie to można uruchomić poleceniem `scalene julia1_memoryprofiler.py`. Będzie ono profilować cały program bez potrzeby jakichkolwiek modyfikacji.

Scalene oferuje opcje ograniczenia profilowania za pomocą dekoratora `@profile` (podobnie do narzędzi `memory_profiler` i `line_profiler`) oraz możliwość filtrowania plików o określonej nazwie. Udostępniono rozszerzenie „magiczne” w celu użycia w programie Jupyter Notebooks z funkcją `%%scalene` umożliwiającą profilowanie całej komórki.

W najbardziej lewej kolumnie *TIME* na zrzucie ekranu z rysunku 2.9 widać, że pętla `while` zajmuje 85% czasu wykonywania, co przedstawione jest jako duży poziomy pasek. Zawiera on trzy składniki: najdłuższy (i najciemniejszy) odzwierciedla czas działania kodu natywnego w Pythonie, natomiast dwa mniejsze segmenty po prawej stronie (oba w jaśniejszych odcieniach) reprezentują czas systemowy oraz działania składników natywnych.

Czas działania kodu natywnego w Pythonie odnosi się do czasu spędzonego na wykonywaniu instrukcji tego języka, które możesz kontrolować. Czas działania składników natywnych dotyczy bibliotek (np. napisanych w języku C++ i skompilowanych), w przypadku których optymalizacja przez programistę jest mało prawdopodobna. Czas systemowy ma związek z wywołaniami systemu operacyjnego, takimi jak operacje wejścia-wyjścia, które również nie mogą być raczej optymalizowane przez programistę, ale mogą wskazać wąskie gardła w ramach tych operacji (można je wyeliminować w inny sposób).

W drugiej kolumnie *MEMORY* widoczna jest maksymalna alokacja pamięci dla poszczególnych wierszy kodu. Można zauważyć, że wiersz `zs.append(complex(xcoord, ycoord))` zużywa 40 MB, natomiast wiersz `cs.append(complex(c_real, c_imag))` wymaga 31 MB. Różne uruchomienia kodu generują odmienne wartości maksymalnej alokacji pamięci, a czasami lista `cs`



Rysunek 2.9. Połączone dane wynikowe profilowania procesora i pamięci za pomocą narzędzia Scalene

zajmuje więcej pamięci niż lista `zs`. Z tego powodu zachowaj ostrożność przy formułowaniu zdecydowanej opinii bez wcześniejszego wielokrotnego uruchomienia kodu.

Wiele wierszy kodu zawiera ikonę „wybuchu” lub „błyskawicy”. Jeśli dostarczysz klucz interfejsu API, narzędzie Scalene utworzy wywołania kierowane do modelu ChatGPT. Dzięki temu możliwe jest automatyczne proponowanie optymalizacji.

Zachowaj ostrożność przy korzystaniu ze wskazówek systemu generatywnej sztucznej inteligencji. Utworzy on coś przekonującego, ale z łatwością może opierać się na nieaktualnych sugestiach, które z racji czasu swojego istnienia uzyskały odpowiednie poparcie i ilość odwołań w cytatach. Zawsze bądź sceptyczny i wykonuj własne testy porównawcze.

Narzędzie Scalene może wygenerować zarówno tekstowe dane wyjściowe widoczne tylko w konsoli, jak i interaktywny raport w formacie HTML. Fragment wyniku w przykładzie 2.13 przedstawia raport z konsoli z dodanymi dwoma dodatkowymi wierszami tworzącymi tablice biblioteki NumPy na podstawie czystych kontenerów list języka Python.

Przykład 2.13. Dane wyjściowe narzędzia Scalene z konsoli z dodanymi dwoma tablicami biblioteki NumPy

```
Line  ...  Python | peak  | timeline/%
...
49  |  ...    20%  |  19M  | __ 15%  |  zs_np = np.array(zs)
50  | ...          | 15M   | __ 12%  |  cs_np = np.array(cs)
```

Oba wiersze powodują zduplikowanie podstawowych danych w ramach 16-bajtowego typu danych `complex128` (2 · 8 bajtów danych zmiennoprzecinkowych). Przy 1 000 000 elementów w każdej tablicy wiersze te powodują utworzenie dla każdej tablicy kopii o rozmiarze 16 MB.

Użytkownik może nie być tego świadomy podczas pisania kodu (lub w trakcie przeglądania go może nie zdawać sobie sprawy z tego, co się dzieje), natomiast narzędzie profilujące wyraźnie wskazuje, że ma miejsce duplikowanie danych w pamięci.

Możesz się spodziewać, że wyniki będą się różnić od wszelkich uzyskanych przy użyciu narzędzia `line_profiler` lub `memory_profiler`. Każde narzędzie profilujące ma inny wpływ i zazwyczaj analizuje poziom wykorzystania procesora i pamięci w nieco inny sposób i z odmienną dokładnością. Obraz w dużej skali będzie podobny, ale szczegóły będą się różnić.

W przypadku szybkości wykonywania kodu uproszczona biblioteka profilowania narzędzia Scalene nie powinna spowodować narzutu większego niż 20%, a także powinna w wiarygodny sposób mierzyć rzeczywisty czas oraz zużycie pamięci. Autorzy twierdzą, że potencjalnie biblioteka może być znacznie dokładniejsza niż inne narzędzia profilujące.

Introspekcja istniejącego procesu za pomocą narzędzia PySpy

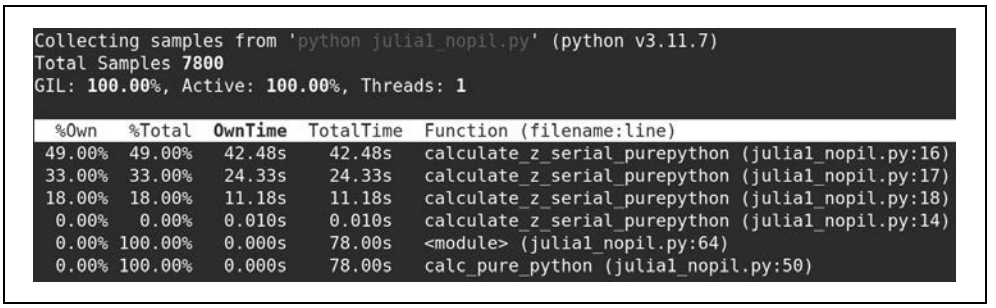
PySpy to nowe, intrygujące narzędzie próbkujące do profilowania, które zamiast wymagać jakichkolwiek zmian w kodzie, dokonuje introspekcji już działającego procesu Python, a ponadto generuje w konsoli raport w sposób podobny do narzędzia `top`. Będąc narzędziem próbkującym do profilowania, `py-spy` prawie wcale nie wpływa na wydajność działającego kodu. Narzędzie napisane w języku Rust wymaga zwiększonych uprawnień do prowadzenia introspekcji innego procesu. Może ono też profilować podprocesy i natywne rozszerzenia poddane kompilacji.

Narzędzie może okazać się bardzo przydatne w środowisku produkcyjnym z długo działającymi procesami lub mającym złożone wymagania dotyczące instalacji. Narzędzie obsługuje systemy Windows, Mac i Linux. Aby je zainstalować, użyj polecenia `pip install py-spy` (zwróć uwagę na znak `-` w nazwie narzędzia — istnieje niezwiązany z nim osobny projekt `pyspy`). Jeśli proces już działa, to w celu uzyskania jego identyfikatora (PID) możesz skorzystać z polecenia `ps`. Identyfikator może zostać następnie przekazany narzędziu `py-spy` (przykład 2.14). Narzędzie wymaga uprawnień zapewnianych przez polecenie `sudo`, które spowoduje uruchomienie go w środowisku superużytkownika. W związku z tym podczas wywoływania narzędzia `py-spy` razem z identyfikatorem procesu do monitorowania za pomocą polecenia `sudo env "PATH=$PATH"` przekazuje się wartość zmiennej `PATH` naszej nazwy logowania.

Przykład 2.14. Uruchamianie narzędzia PySpy z poziomu wiersza poleceń

```
$ ps -A -o pid,rss,cmd | ack python
...
95671 94984 python julia1_nopil.py
...
$ sudo env "PATH=$PATH" py-spy top --pid 95671
```

Na rysunku 2.10 zaprezentowano w konsoli obraz statyczny podobny jak w wypadku narzędzia `top`. Obraz jest aktualizowany co sekundę w celu uwidocznienia funkcji, które w danej chwili zajmują najwięcej czasu.



Rysunek 2.10. Introspekcja procesu Python za pomocą narzędzia PySpy

Narzędzie PySpy pozwala również na wyeksportowanie wykresu „ognistego” (ang. *flame chart*). W omawianym przykładzie opcja ta zostanie zastosowana, gdy od narzędzia zostanie zażądane bezpośrednie uruchomienie kodu bez wymogu podania identyfikatora PID. Umożliwia to polecenie `$ py-spy record -o profile.svg -- python julia1_nopil.py`. Ciąg `--` nakazuje narzędziu `py-spy` zaakceptowanie podanego dalej polecenia (`python julia1_nopil.py`), które może zawierać własne argumenty wiersza poleceń, a ponadto zignorowanie wszelkich przełączników wiersza poleceń. Polecenie zostanie wykonane wewnątrz narzędzia.

Na rysunku 2.11 widać, że szerokość wyświetlonych danych obejmuje cały czas działania programu. Każda kolejna warstwa niżej położona w obrębie obrazu reprezentuje funkcje wywołane z poziomu wyżej umiejscowionych warstw.

Rysunek 2.11. Część wykresu „ognistego” narzędzia PySpy

Narzędzie PySpy oferuje również przydatną możliwość podłączenia się do już uruchomionego procesu kodu w Pythonie. Jeśli zatem dysponujesz serwerem WWW lub długo działającym procesem ETL (*Extract-Transform-Load*), możesz się z nim połączyć i sprawdzić, na jaką część programu poświęcany jest czas. Umożliwia to lokalne profilowanie, gdy wydarzyło się coś niespodziewanego, bez konieczności uruchamiania nowego procesu profilowania.

Wyświetlanie interaktywnego stosu wywołań na osi czasu za pomocą narzędzia VizTracer

VizTracer to narzędzie składające się z dwóch części, które umożliwia profilowanie, a następnie wyświetlenie na osi czasu widoku wykonywania kodu w Pythonie. Jest to bardzo potężne narzędzie, choć może być nieco trudne do opanowania. Czas prezentowany od lewej do prawej strony, a także możliwość powiększania wykresu w stylu wykresu „ognistego" zapewniają ogólny widok całego procesu. Dzięki temu możesz zorientować się, ile czasu zajmują poszczególne części ścieżki wykonywania.

Narzędzie może śledzić proces wykonywania, argumenty, wartości zwracane i czas trwania wywołania każdej funkcji. Pozwala ono na rejestrowanie informacji niestandardowych, obsługuje wiele procesów i wątków, a ponadto nie wymaga zmian w kodzie źródłowym. Narzędzie zapewnia interfejs API oraz interaktywny interfejs graficzny.

Po zainstalowaniu narzędzia za pomocą polecenia `pip install viztracer` można uruchomić wcześniej użyty kod za pomocą polecenia `viztracer julia1_memoryprofiler.py`.

Po zakończeniu profilowania kodu za pomocą narzędzia VizTracer generowany jest plik *results.json* z danymi wyjściowymi. Używając polecenia `vizviewer results.json`, dokonuje się następnie wizualizacji tych danych. Dla kodu zbioru Julii uzyskamy wynik podobny do przedstawionego na rysunku 2.12. Zrzut ekranu zawiera następujące sekcje:

- Na samej górze znajduje się interaktywna oś czasu z uchwytami do przesuwania i powiększania widoku dla konkretnej chwili.
- Sekcja *MainProcess* przedstawia poziomą oś czasu wykonywania kodu i wywołań funkcji jako słupki opadające w dół ekranu.
- Sekcja *Current Selection* pokazuje szczegóły dowolnego elementu kliknięgo w sekcji *MainProcess*.

Rysunek 2.12. Przegląd wykonywania kodu Julii na osi czasu

Narzędzie VizTracer umożliwia śledzenie wykonywania wielu wątków i procesów. Zostaną one wyświetlone wzdłuż sekcji *MainProcess*. Narzędzie pozwala także na rejestrowanie innych zdarzeń na osi czasu, takich jak użycie instrukcji `print` oraz wywołania modułu oczyszczającego pamięć, co pozwala na tworzenie adnotacji dla konkretnych zdarzeń, które mają być obserwowane.

Dwa długie słupki na środku zrzutu ekranu poniżej sekcji *MainProcess* przedstawiają widok z osią czasu dla funkcji `calc_pure_python` oraz jej wywołania funkcji `calculate_z_serial_purepython`. Poniżej znajduje się wiele małych „szpilek", które reprezentują niektóre wywołania funkcji `abs`.

Kliknięcie funkcji `calculate_z_serial_purepython` w interfejsie graficznym powoduje otwarcie sekcji *Details* widocznej w lewej dolnej części zrzutu ekranu. Pokazuje ona nazwę funkcji, czas rozpoczęcia i czas trwania oraz szczegóły związane z wątkiem. Po prawej stronie widoczny jest kod źródłowy, który został wykonany.

Ze względu na wysoki poziom szczegółów, jakie narzędzie VizTracer potrafi śledzić, potencjalnie może ono generować duże zrzuty danych, których nie można wyświetlić. W takim przypadku może wystąpić błąd `Circular buffer is full` (bufor cykliczny jest pełny).

Narzędzie VizTracer udostępnia różne opcje filtrowania zbieranych danych. W przypadku przykładu zbioru Julii konieczne było:

- zmniejszenie wartości argumentu `desired_width` funkcji `calc_pure_python` z 1000 do 200,
- filtrowanie krótszych wywołań podczas profilowania za pomocą polecenia `viztracer --min_duration 0.1 julia1_viztracer.py`.

Inne opcje filtrowania mogą obejmować opcję `--ignore_c_function`, która uwzględnia wszystkie wbudowane funkcje (oznacza to zignorowanie każdego wywołania funkcji `abs`), lub ograniczenie głębokości stosu za pomocą opcji `--max_stack_depth 10`. W tym przypadku nie obserwujemy głębokiego stosu, lecz dużą liczbę wywołań szybkiej funkcji `abs`.

Aby uzyskać wystarczającą ilość szczegółów bez przeciążenia bufora, ograniczyliśmy tutaj profilowanie do funkcji trwających dłużej niż 0,1 milisekundy. Co ciekawe, wywołania funkcji `abs` mogą czasem przekraczać ten czas (ale często były znacznie szybsze), dlatego część z nich znalazła się w danych śledzenia. Bez tego filtra mielibyśmy miliony wywołań do śledzenia, co właśnie było przyczyną błędu `Circular buffer is full`.

Jeden z autorów uznał to narzędzie za bardzo przydatne przy analizowaniu ścieżki wykonywania w ramach takich bibliotek jak Pandas. Dzięki temu można przekonać się, ile czasu zajmują konkretne części funkcji i ile innych funkcji jest wywoływanych w tle.

Kod bajtowy od podszewki

Do tej pory dokonano przeglądu różnych metod pomiaru czasu wykonywania kodu w Pythonie (w przypadku wykorzystania procesora i pamięci RAM). Nie zajęliśmy się jednak jeszcze bazowym kodem bajtowym używanym przez maszynę wirtualną. Zrozumienie tego, co dzieje

się pod podszewką, ułatwi zbudowanie modelu pamięciowego tego, co odbywa się w powolnych funkcjach. Ponadto okaże się pomocne podczas kompilowania kodu. Zajmijmy się zatem kodem bajtowym.

Użycie modułu dis do sprawdzenia kodu bajtowego narzędzia CPython

Moduł `dis` umożliwia sprawdzanie bazowego kodu bajtowego, który jest uruchamiany w obrębie maszyny wirtualnej narzędzia CPython opartej na stosie. Zrozumienie tego, co się dzieje w maszynie wirtualnej uruchamiającej kod w Pythonie wyższego poziomu, pozwoli zorientować się, dlaczego niektóre style tworzenia kodu zapewniają większą szybkość niż inne. Ułatwi to również użycie narzędzia takiego jak Cython, które „wychodzi" poza kod Python i generuje kod C.

Moduł `dis` jest wbudowany. Po przekazaniu do niego kodu lub modułu wyświetli on wyniki dezasemblacji. W przykładzie 2.15 przeprowadzana jest dezasemblacja pętli zewnętrznej funkcji powiązanej z procesorem.

Należy spróbować dezasemblacji jednej z własnych funkcji, a następnie podjąć próbę *dokładnego* prześledzenia tego, jaka jest zgodność kodu poddanego dezasemblacji z danymi wyjściowymi po tej operacji. Czy możesz dopasować przedstawione poniżej dane wyjściowe modułu `dis` do oryginalnej funkcji?

Przykład 2.15. Użycie wbudowanego modułu dis do zrozumienia bazowej maszyny wirtualnej opartej na stosie, która wykonuje kod w Pythonie

```
In [1]: import dis
In [2]: import julia1_nopil
In [3]: dis.dis(julia1_nopil.calculate_z_serial_purepython)
  9           0 RESUME                   0

 11           2 LOAD_CONST               1 (0)
              4 BUILD_LIST                   1
              6 LOAD_GLOBAL              1 (NULL + len)
             16 LOAD_FAST                1 (zs)
             18 CALL                         1
             26 BINARY_OP                5 (*)
             30 STORE_FAST               3 (output)

 12          32 LOAD_GLOBAL              3 (NULL + range)
             42 LOAD_GLOBAL              1 (NULL + len)
             52 LOAD_FAST                       1 (zs)
             54 CALL                                 1
             62 CALL                                 1
             70 GET_ITER
        >>   72 FOR_ITER                 71 (to 218)
             76 STORE_FAST                  4 (i)

 13          78 LOAD_CONST               1 (0)
             80 STORE_FAST                  5 (n)
...
 16                 166 LOAD_GLOBAL                  5 (NULL + abs)
                    176 LOAD_FAST                          6 (z)
                    178 CALL                                       1
```

```
                  186 LOAD_CONST                2 (2)
                  188 COMPARE_OP                2 (<)
                  192 POP_JUMP_IF_FALSE       6 (to 206)
                  194 LOAD_FAST                 5 (n)
                  196 LOAD_FAST                 0 (maxiter)
                  198 COMPARE_OP                2 (<)
                  202 POP_JUMP_IF_FALSE       1 (to 206)
                  204 JUMP_BACKWARD          33 (to 140)

19        >>    206 LOAD_FAST                 5 (n)
                  208 LOAD_FAST                 3 (output)
                  210 LOAD_FAST                 4 (i)
                      212 STORE_SUBSCR
                      216 JUMP_BACKWARD        73 (to 72)

12          >>  218 END_FOR
20                220 LOAD_FAST                 3 (output)
                    222 RETURN_VALUE
```

Mimo swojej zwięzłości dane wyjściowe są dość zrozumiałe. Pierwsza kolumna zawiera numery wierszy powiązane z oryginalnym plikiem. W drugiej kolumnie znajduje się kilka symboli >>. Reprezentują one miejsca docelowe dla punktów skoku gdzieś w kodzie. Trzecia kolumna zawiera adres operacji. W czwartej kolumnie jest nazwa operacji. Piąta kolumna przechowuje parametry operacji. W szóstej kolumnie znajdują się adnotacje ułatwiające dopasowanie kodu bajtowego do oryginalnych parametrów kodu w Pythonie.

Aby dopasować kod bajtowy do odpowiedniego kodu w Pythonie, cofnij się do przykładu 2.3. Począwszy od wiersza 11., kod bajtowy umieszcza na stosie stałą wartość 0, a następnie tworzy listę jednoelementową. Dalej kod przeszukuje przestrzenie nazw w celu znalezienia funkcji `len`, umieszcza ją w stosie, ponownie przeszukuje przestrzenie nazw, aby znaleźć listę `zs`, po czym wstawia ją do stosu. Na tym etapie może zostać przeprowadzone mnożenie z użyciem elementów stosu, które są następnie zapisywane na liście `output`. Jest to pierwszy wiersz funkcji języka Python, jaką się teraz zajmowaliśmy. Prześledź następny blok kodu bajtowego, aby zrozumieć działanie drugiego wiersza kodu w Pythonie (pętla zewnętrzna `for`).

Punkty skoku (>>) dopasowują instrukcje, takie jak `JUMP_BACKWARD` i `POP_JUMP_IF_FALSE`. Przeanalizuj własną funkcję poddaną dezasemblacji i dopasuj punkty skoku do instrukcji skoku.

Analizowanie specjalizacji kodu bajtowego za pomocą narzędzia Specialist

Gdy interpreter języka Python 3.11 uruchamia Twój kod, próbuje zidentyfikować „gorący" kod, który jest wykonywany na tyle często, że warto go zoptymalizować. W miarę możliwości interpreter zastępuje bardziej ogólny kod bajtowy, który był używany aż do wersji 3.10 języka Python, wyspecjalizowanymi instrukcjami kodu bajtowego. Taki kod może działać szybciej, ponieważ wykonuje mniej operacji (być może dzięki unikaniu nadmiarowych sprawdzeń lub wykorzystaniu innych optymalizacji). Narzędzie Specialist można zainstalować za pomocą polecenia `pip install specialist`.

Tekst w danych wyjściowych narzędzia jest wyróżniany różnymi kolorami.

Specjalizacje są skuteczne, dopóki napotykane są te same typy danych. Jeśli w pętli wykonywane jest dodawanie dwóch liczb całkowitych, operacja może być wyspecjalizowana (i zostanie oznaczona kolorem zielonym). Jeżeli jednak operacja ta napotyka różne typy (w zależności od iteracji mogą to być liczby typu całkowitego lub zmiennoprzecinkowego), pozostaje ona w stanie adaptacyjnym oznaczonym kolorem czerwonym.

Za pomocą kolorów narzędzie Specialist oznacza „gorące" obszary kodu, które interpreter języka Python w wersji 3.11 lub nowszej identyfikuje podczas wykonywania, umożliwiając nam sprawdzenie, czy kod działa tak szybko, jak to możliwe. Na przykładzie zbioru Julii z rysunku 2.13 można zauważyć:

- wiersze bez cieniowania oznaczają, że nie podjęto próby specjalizacji,
- zielone wiersze (otoczone linią ciągłą w kształcie prostokąta) wskazują na pomyślną specjalizację i szybko działający kod,
- częściowej specjalizacji poddano sekcje kodu w kolorze pomarańczowym (otoczone linią przerywaną w kształcie prostokąta), który czasem zawodzi i powraca do stanu adaptacyjnego (w szczególności wiersze `abs(z) < 2` oraz `z * z + c`),
- w naszym przykładzie nie ma kodu w kolorze czerwonym (co oznacza brak specjalizacji).

```
def calculate_z_serial_purepython(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while abs(z) < 2 and n < maxiter:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Rysunek 2.13. Wyświetlone dane wyjściowe narzędzia Specialist z użyciem różnych kolorów

Wyrażenie `abs(z) < 2` można uczynić „bardziej zielonym" (czyli lepiej wyspecjalizowanym), zmieniając je do postaci `abs(z) < 2.0`, ponieważ interpreter języka Python 3.12 preferuje obecnie identyczne typy liczbowe dla niektórych operacji z typami podstawowymi. Operacja `z * z + c` uwzględniająca typ złożony nie umożliwiła poprawy.

W tym przypadku zmiana wartości na `2.0` poprawiła nieco wydajność. W obecnej wersji języka Pythona narzędzie Specialist służy bardziej do celów edukacyjnych, pomagając zrozumieć, co dzieje się w tle, ale eksperymenty mogą ujawnić możliwość zwiększenia szybkości wykonania.

Po wprowadzeniu do kodu bajtowego możemy zadać następujące pytanie: „Jak na kod bajtowy i czas wykonywania wpływa jawne tworzenie funkcji w porównaniu z użyciem w tym samym celu funkcji wbudowanych?".

Różne metody, różna złożoność

Powinna istnieć jedna, i najlepiej tylko jedna, oczywista metoda zrealizowania czegoś.
Choć początkowo metoda ta może nie być oczywista, o ile nie jesteś Holendrem…[2]

— Tim Peters, *The Zen of Python*

Istnieją różne metody wyrażania pomysłów za pomocą języka Python. Choć generalnie powinno być jasne, jaka opcja jest najbardziej rozsądna, jeśli masz doświadczenie głównie ze starszą wersją języka Python lub innego języka programowania, możesz mieć na myśli inne rozwiązania. Niektóre z tych metod mogą zapewniać mniejszą wydajność od innych.

W przypadku większości kodu prawdopodobnie bardziej zależy Ci na jego czytelności niż szybkości, aby zespół programistów mógł efektywnie tworzyć kod bez dłuższego zastanawiania się nad wydajnym, lecz zagmatwanym kodem. Czasami jednak wymagana będzie wydajność (bez utraty czytelności kodu). W tym przypadku niezbędne może być testowanie szybkości.

Przyjrzyj się dwóm fragmentom kodu z przykładu 2.16. Choć oba realizują to samo zadanie, pierwszy z nich wygeneruje mnóstwo dodatkowego kodu bajtowego Python, co spowoduje większe obciążenie.

Przykład 2.16. Naiwny i bardziej efektywny sposób rozwiązania tego samego problemu dotyczącego sumowania

```
def fn_expressive(upper=1_000_000):
    total = 0
    for n in range(upper):
        total += n
    return total

def fn_terse(upper=1_000_000):
    return sum(range(upper))

assert fn_expressive() == fn_terse(), "W wypadku obu funkcji oczekuj identycznych
wyników"
```

Obie funkcje obliczają sumę dla zakresu liczb całkowitych. Prosta zasada (*musi* jednak zostać poparta użyciem profilowania!) głosi, że większa liczba wierszy kodu bajtowego będzie wykonywana wolniej niż mniejsza liczba odpowiednich wierszy kodu bajtowego, który korzysta z wbudowanych funkcji. W przykładzie 2.17 użyto funkcji „magicznej” `%timeit` powłoki IPython do pomiaru najlepszego czasu wykonywania na podstawie zestawu uruchomień. Funkcja `fn_terse` działa ponad dwa razy szybciej niż funkcja `fn_expressive`!

Przykład 2.17. Użycie funkcji %timeit do testowania hipotezy określającej, że użycie funkcji wbudowanych powinno zapewnić większą szybkość niż w przypadku napisania własnych funkcji

```
In [2]: %timeit fn_expressive()
45.6 ms ± 963 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[2] Holenderski to język twórcy, którym jest Guido van Rossum. Nie każdy zgodził się z jego „oczywistymi” wyborami, ale ogólnie podobają nam się decyzje podjęte przez Guido!

```
In [3]: %timeit fn_terse()
16.4 ms ± 226 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Jeśli do sprawdzenia kodu dla każdej funkcji zostałby użyty moduł `dis` (przykład 2.18), okazałoby się, że maszyna wirtualna ma do wykonania 17 wierszy dla bardziej rozbudowanej funkcji i tylko 7 wierszy w przypadku bardzo czytelnej, lecz bardziej zwięzłej drugiej funkcji.

Przykład 2.18. Użycie modułu dis do wyświetlenia liczby instrukcji kodu bajtowego objętych dwiema przykładowymi funkcjami

```
In [4]: import dis

In [5]: dis.dis(fn_expressive)
  1           0 RESUME                   0

  2           2 LOAD_CONST               1 (0)
              4 STORE_FAST               1 (total)

  3           6 LOAD_GLOBAL              1 (NULL + range)
             16 LOAD_FAST                0 (upper)
             18 CALL                     1
             26 GET_ITER
        >>   28 FOR_ITER                 7 (to 46)
             32 STORE_FAST               2 (n)

  4          34 LOAD_FAST                1 (total)
             36 LOAD_FAST                2 (n)
             38 BINARY_OP               13 (+=)
             42 STORE_FAST               1 (total)
             44 JUMP_BACKWARD            9 (to 28)

  3     >>   46 END_FOR

  5          48 LOAD_FAST                1 (total)
             50 RETURN_VALUE
In [6]: dis.dis(fn_terse)
  7           0 RESUME                   0

  8           2 LOAD_GLOBAL              1 (NULL + sum)
             12 LOAD_GLOBAL              3 (NULL + range)
             22 LOAD_FAST                0 (upper)
             24 CALL                     1
             32 CALL                     1
             40 RETURN_VALUE
```

Różnica między dwoma blokami kodu jest ewidentna. Wewnątrz funkcji `fn_expressive()` utrzymywane są dwie zmienne lokalne, a ponadto wykonywana jest iteracja dla listy przy użyciu instrukcji `for`. Pętla `for` będzie sprawdzana w celu stwierdzenia, czy wyjątek `StopIteration` wystąpił w każdej pętli. Każda iteracja stosuje funkcję `total.__add__`, która sprawdzi typ drugiej zmiennej (`n`) w każdej iteracji. Wszystkie te sprawdzenia powodują niewielkie obciążenie pod kątem wydajności.

W obrębie funkcji `fn_terse()` wywoływana jest zoptymalizowana funkcja wyrażeń listowych języka C, która potrafi wygenerować wynik końcowy bez tworzenia pośrednich obiektów Python. Choć jest to znacznie szybsze, każda iteracja w dalszym ciągu musi dokonywać

sprawdzenia typów dodawanych razem obiektów (w rozdziale 4. przyjrzymy się metodom ustalania typu, dzięki czemu nie ma potrzeby sprawdzania go w każdej iteracji).

Jak wcześniej wspomniano, *konieczne* jest profilowanie kodu. Jeśli będzie się polegać jedynie na heurystyce, bez wątpienia w pewnym momencie zostanie utworzony wolniejszy kod. Zdecydowanie warto dowiedzieć się, czy język Python oferuje krótszą, a jednocześnie nadal zrozumiałą metodę rozwiązania problemu. Jeśli tak będzie, bardziej prawdopodobne jest to, że kod okaże się bardziej czytelny dla innego programisty, czyli *prawdopodobnie* będzie działać szybciej.

Testowanie jednostkowe podczas optymalizacji w celu zachowania poprawności

Jeśli jeszcze nie stosujesz dla kodu testowania jednostkowego, prawdopodobnie w dłuższej perspektywie wpłynie to niekorzystnie na Twoją produktywność. Jeden z autorów (rumieni się) wstydzi się wspomnieć, że raz spędził cały dzień na optymalizowaniu swojego kodu przy wyłączonych testach jednostkowych, dlatego że były niewygodne, tylko po to, aby odkryć, że wynikowe znaczne przyspieszenie było spowodowane rozbiciem części ulepszanego przez niego algorytmu. Ani razu nie musisz popełniać tego błędu.

W celu ułatwienia sobie pracy dodaj do kodu testy jednostkowe. Zarówno Ty, jak i Twoi znajomi zyskacie przekonanie, że kod działa. Ponadto w ten sposób zrobisz samemu sobie prezent, gdy w przyszłości konieczne będzie zajmowanie się danym kodem. Dodając testy do kodu, naprawdę w dłuższej perspektywie zaoszczędzisz wiele czasu.

Oprócz testowania jednostkowego należy też poważnie rozważyć użycie skryptu `coverage.py`. Umożliwia on stwierdzenie, jakie wiersze kodu są sprawdzane przez testy, a ponadto identyfikuje sekcje bez pokrycia. Skrypt pozwala szybko określić, czy testujesz kod, który zostanie zoptymalizowany. Dzięki temu wszelkie pomyłki, jakie mogą pojawić się w podczas procesu optymalizacji, zostaną szybko wychwycone.

Dekorator @profile bez operacji

Testy jednostkowe nie powiodą się z wygenerowanym wyjątkiem `NameError`, jeśli w kodzie używany jest dekorator `@profile` z narzędzia `line_profiler` lub `memory_profiler`. Powodem jest to, że środowisko testów jednostkowych nie wprowadzi dekoratora `@profile` do lokalnej przestrzeni nazw. Problem ten rozwiązuje przedstawiony w tym punkcie dekorator bez operacji. Najprościej, należy go dodać do testowanego bloku kodu i usunąć po zakończeniu testów.

Dzięki dekoratorowi bez operacji możesz uruchamiać testy bez modyfikowania testowanego kodu. Oznacza to, że możesz wykonywać testy po każdej profilowanej optymalizacji. Dzięki temu nigdy nie zostaniesz zaskoczony niepoprawnym krokiem optymalizacji.

Załóżmy, że istnieje trywialny moduł `ex.py` zaprezentowany w przykładzie 2.19. Moduł zawiera test (dla narzędzia `pytest`) i funkcję, która była profilowana za pomocą narzędzia `line_profiler` lub `memory_profiler`.

Przykład 2.19. Prosta funkcja i przypadek testowy, w którym ma zostać użyty dekorator @profile

```
import time

def test_some_fn():
    """Sprawdzenie dla funkcji podstawowych działań"""
    assert some_fn(2) == 4
    assert some_fn(1) == 1
    assert some_fn(-1) == 1

@profile
def some_fn(useful_input):
    """Funkcja generująca duży koszt, która ma być testowana i profilowana""")
    # "Generowane jest zmyślne i kosztowne" opóźnienie
    time.sleep(1)
    return useful_input ** 2

if __name__ == "__main__":
    print(f"Przykładowe wywołanie `some_fn(2)` == {some_fn(2)}")
```

Jeśli dla kodu zostanie uruchomione narzędzie `pytest`, zostanie wygenerowany wyjątek `NameError` (przykład 2.20).

Przykład 2.20. Pominięty dekorator powoduje podczas testowania przerwanie testów w mało użyteczny sposób

```
$ pytest test_utility.py
== test session starts ==
platform linux -- Python 3.12.0, pytest-8.0.1, pluggy-1.4.0
rootdir: .../r02/noop_profile_decorator
collected 0 items / 1 error

==ERRORS ==
__ERROR collecting test_utility.py __
test_utility.py:1: in <module>
    from utility import some_fn
utility.py:20: in <module>
    @profile
E   NameError: name 'profile' is not defined. Did you forget to import 'profile'
== short test summary info ==
ERROR test_utility.py - NameError: name 'profile' is not defined.
Did you forget to import 'profile'
!! Interrupted: 1 error during collection !!
== 1 error in 0.12s ==
```

Rozwiązanie polega na dodaniu dekoratora bez operacji na początku modułu (dekorator możesz usunąć po zakończeniu profilowania). Jeśli dekorator `@profile` nie zostanie znaleziony w jednej z przestrzeni nazw (ponieważ nie jest używane narzędzie `line_profiler` lub `memory_profiler`), zostanie dodana nowo utworzona wersja dekoratora bez operacji. Jeśli narzędzie `line_profiler` lub `memory_profiler` wprowadziło do przestrzeni nazw nową funkcję, ta wersja dekoratora zostanie zignorowana.

W przypadku narzędzi `line_profiler` i `memory_profiler` możemy dodać kod z przykładu 2.21.

Przykład 2.21. Dodanie dekoratora @profile bez operacji do przestrzeni nazw podczas testowania jednostkowego

```
# Sprawdzenie obecności narzędzia line_profiler lub memory_profiler w zasięgu lokalnym
# Oba narzędzia są wprowadzane za pomocą odpowiednich narzędzi lub są nieobecne,
# jeśli nie są one używane (w tym wypadku konieczne jest zastąpienie  fikcyjnego dekoratora @profile)
if 'line_profiler' not in dir() and 'profile' not in dir():
    def profile(func):
        def inner(*args, **kwargs):
            return func(*args, **kwargs)
        return inner
```

Po dodaniu dekoratora bez operacji możesz pomyślnie uruchomić moduł `pytest` (przykład 2.22) z uwzględnieniem narzędzi profilujących — bez wprowadzania dodatkowych zmian w kodzie.

Przykład 2.22. Po zastosowaniu dekoratora bez operacji uzyskuje się działające testy, a ponadto poprawnie działają oba narzędzia profilujące

```
$ pytest utility.py
== test session starts ==
platform linux -- Python 3.12.0, pytest-8.0.1, pluggy-1.4.0
rootdir: ...
collected 1 item
utility.py .
[100%]

== 1 passed in 3.01s ==

$ kernprof -l -v utility.py
Przykładowe wywołanie `some_fn(2)` == 4
Wyniki profilowania zapisano w pliku utility.py.lprof
Timer unit: 1e-06 s

Total time: 1.00018 s
File: utility.py
Function: some_fn at line 20

Line #      Hits         Per Hit   % Time  Line Contents
==============================
    20                                           @profile
    21                                           def some_fn(useful_input):
    22                   """Funkcja generująca duży koszt, która ma być testowana i profilowana"""
    23                                           # "Generowane jest zmyślne i kosztowne" opóźnienie
    24         1    1e+06          100.0      time.sleep(1)
    25         1          6.5          0.0      return useful_input ** 2

$ python -m memory_profiler utility.py
Przykładowe wywołanie `some_fn(2)` == 4
Filename: utility.py

Line #    Mem usage    Increment   Line Contents
==========================
    20   46.898 MiB   46.898 MiB   @profile
    21                             def some_fn(useful_input):
    22                 """Funkcja generująca duży koszt, która ma być testowana i profilowana"""
    23                                 # "Generowane jest zmyślne i kosztowne" opóźnienie
    24   46.898 MiB    0.000 MiB       time.sleep(1)
    25   46.898 MiB    0.000 MiB       return useful_input ** 2
```

Choć zrezygnowanie z użycia tych dekoratorów pozwoli zyskać kilka minut, po straceniu wielu godzin na przeprowadzenie niewłaściwej optymalizacji, która powoduje, że kod przestaje działać, postanowisz uwzględnić dekoratory w przepływie pracy.

Strategie udanego profilowania kodu

Profilowanie wymaga trochę czasu i koncentracji. Oddzielenie sekcji przeznaczonej do testowania od głównego segmentu kodu pozwoli zwiększyć szanse na zrozumienie kodu. Aby zachować poprawność, możesz następnie wykonać dla kodu test jednostkowy, a ponadto przekazać do niego dane wygenerowane w rzeczywistych warunkach w celu zidentyfikowania niewydajnych instrukcji.

Pamiętaj o wyłączeniu wszelkich akceleratorów bazujących na BIOS-ie, ponieważ spowodują one tylko niejasność uzyskanych wyników. W przypadku laptopa jednego z autorów funkcja Intel Turbo Boost może tymczasowo przyspieszyć działanie procesora ponad jego normalną, maksymalną szybkość, jeśli jest wystarczająco chłodny. Uruchomienie demonstracyjnego kodu zbioru Julii z tego rozdziału w przypadku chłodnego procesora z włączoną funkcją Turbo Boost zajęło 3,3 sekundy. Procesor nagrzałby się chwilowo, gdyby kod był uruchamiany wielokrotnie, a wraz ze wzrostem temperatury czas wykonywania by się wydłużył. Przy wyłączonej funkcji Turbo Boost czas działania tego samego kodu wyniósłby 5,6 sekundy. W tym przykładzie przy włączonej funkcji Turbo Boost jedno uruchomienie kodu zajęło 60% czasu w porównaniu z wariantem bez tej funkcji, ale różnica ta zmniejszałaby się w miarę nagrzewania procesora. Laptop zasilany akumulatorowo prawdopodobnie będzie bardziej agresywnie kontrolować szybkość procesora niż laptop podłączony do zasilania sieciowego.

Możesz sobie wyobrazić, jak może to zaburzyć wyniki Twoich testów porównawczych, gdy wielokrotnie uruchamiasz złożony kod! Firma AMD oferuje podobną technologię w postaci funkcji Turbo Core. Dotyczy ona głównie laptopów. Komputery serwerowe mogą po prostu działać z maksymalną, stałą szybkością (eliminuje się złożoność kosztem większego zużycia energii), dlatego w ich przypadku może to nie mieć zastosowania.

Aby utworzyć bardziej stabilną konfigurację do testów porównawczych, wykonaj następujące czynności:

- Wyłącz w BIOS-ie funkcję Turbo Boost.
- Wyłącz funkcję systemu operacyjnego, która nadpisuje funkcję SpeedStep (jeśli masz możliwość zarządzania BIOS-em, znajdziesz w nim tę funkcję).
- Używaj wyłącznie zasilania sieciowego (nigdy akumulatorowego).
- Podczas eksperymentowania wyłącz narzędzia działające w tle, takie jak narzędzia tworzące kopie zapasowe i Dropbox.
- Wielokrotnie przeprowadzaj eksperymenty, aby uzyskać stabilny pomiar.
- Jeśli to możliwe, przejdź na poziom uruchamiania 1 (system Unix), aby nie działały żadne inne zadania.
- By mieć całkowitą pewność wyników, zrestartuj komputer i ponownie przeprowadź eksperymenty.

Spróbuj określić hipotezę dla oczekiwanego działania kodu, a następnie potwierdź (lub obal!) jej poprawność przy użyciu wyników kroku profilowania. Choć opcje wyboru nie zmienią się (decyzje możesz podjąć tylko na podstawie wyników profilowania), zwiększy się poziom intuicyjnego rozumienia kodu, co przyniesie pozytywne efekty w przyszłych projektach, ponieważ z większym prawdopodobieństwem podejmiesz decyzje zapewniające większą wydajność. Oczywiście decyzje te będą weryfikowane w trakcie działań z wykorzystaniem profilowania.

Nie żałuj czasu na przygotowania. Jeśli spróbujesz przetestować kod pod kątem wydajności głęboko wewnątrz większego projektu bez oddzielenia od niego tego kodu, prawdopodobnie doświadczysz efektów ubocznych, które zaprzepaszczą cel starań. W przypadku wprowadzania bardziej szczegółowych zmian trudniejsze będzie użycie testu jednostkowego dla większego projektu. Może to dodatkowo komplikować działania. Efekty uboczne mogą obejmować inne wątki i procesy wpływające na wykorzystanie procesora i pamięci oraz na funkcjonowanie sieci i dysków. Spowoduje to, że wyniki nie będą do końca wiarygodne.

Oczywiście korzystasz już z kontroli kodu źródłowego (np. Git lub Mercurial), dlatego będziesz w stanie przeprowadzać wiele eksperymentów w różnych rozgałęzieniach, nie tracąc nawet dobrze działających wersji. Jeśli *nie* używasz kontroli kodu źródłowego, zrób sobie wielką przysługę i zacznij z tego rozwiązania korzystać!

W przypadku serwerów WWW skorzystaj z narzędzi `dowser` i `dozer`. Umożliwiają one wizualizację w czasie rzeczywistym działania obiektów w przestrzeni nazw. Jeśli to możliwe, zdecydowanie rozważ oddzielenie kodu przeznaczonego do przetestowania od głównej aplikacji internetowej, ponieważ znacznie przyspieszy to profilowanie.

Upewnij się, że testy jednostkowe sprawdzają wszystkie ścieżki kodu w analizowanym kodzie. Wszystko, co nie jest testowane, ale jest używane w testach porównawczych, może spowodować subtelne błędy, które spowolnią działania. Użyj skryptu `coverage.py`, aby potwierdzić, że testy obejmują wszystkie ścieżki kodu.

Utrudniony może być test jednostkowy skomplikowanej sekcji kodu, który generuje dużo numerycznych danych wyjściowych. Nie obawiaj się kierowania danych wyjściowych do pliku tekstowego wyników w celu przetworzenia go przez narzędzie `diff` lub użycia obiektu `pickled`. W przypadku problemów z optymalizacją numeryczną jeden z autorów lubi tworzyć długie pliki tekstowe z liczbami zmiennoprzecinkowymi i korzystać z narzędzia `diff`. Mniejsze błędy zaokrąglania pojawiają się natychmiast, nawet jeśli występują rzadko w danych wyjściowych.

Jeśli kodu mogą dotyczyć problemy z zaokrąglaniem liczb z powodu niewielkich zmian, lepszym rozwiązaniem jest użycie dużej ilości danych wyjściowych, które mogą zostać użyte przed porównaniem i po nim. Przyczyną błędów zaokrąglania jest różnica w precyzji liczb zmiennopozycyjnych, jaka występuje między rejestrami procesora i główną pamięcią. Uruchomienie kodu z wykorzystaniem innej ścieżki kodu może spowodować subtelne błędy zaokrąglania, które później mogą wywołać niejasności. Lepiej mieć tego świadomość od razu, gdy takie błędy się pojawią.

Oczywiście podczas profilowania i optymalizowania sensowne jest użycie narzędzia do kontrolowania kodu źródłowego. Stosowanie rozgałęzień nie jest kosztowne, a zapewnia spokój.

Podsumowanie

Po zaznajomieniu się z technikami profilowania masz do dyspozycji wszystkie narzędzia, jakie są niezbędne do identyfikowania w kodzie wąskich gardeł związanych z wykorzystaniem procesora i pamięci RAM. W następnym rozdziale dowiesz się, jak w języku Python implementowane są najbardziej typowe kontenery. Dzięki temu będziesz w stanie podejmować rozsądne decyzje dotyczące reprezentowania większych kolekcji danych.

Każdy, kto się zetknął z językiem Python, wie, że jest on prosty i przyjazny dla programistów, ale ma też swoje ograniczenia — przy pracy z dużymi wolumenami danych szybko pojawiają się problemy z wydajnością i ze skalowaniem. Niekiedy pomaga mocniejsza konfiguracja sprzętowa, jednak najczęściej kluczowe jest zastosowanie odpowiednich technik programistycznych i właściwych narzędzi.

Dzięki kolejnemu, poszerzonemu i zaktualizowanemu wydaniu tego praktycznego podręcznika zdobędziesz wszechstronną wiedzę o czynnikach wpływających na wydajność kodu. Dowiesz się, jak lokalizować „wąskie gardła" wydajności i optymalizować kod w programach, które przetwarzają duże wolumeny danych. Lepiej też zrozumiesz zasady implementacji kodu Pythona. W książce poruszono takie zagadnienia jak architektury wielordzeniowe, klastry, skalowanie systemu poza limity pamięci RAM lub z wykorzystaniem procesorów graficznych. Zaprezentowano praktyczne sposoby radzenia sobie z różnymi wyzwaniami, przybliżono również optymalizację kodu Pythona w wielu realnych scenariuszach, w tym na przykład w sytuacji wyodrębniania danych generatywnej sztucznej inteligencji i uczenia maszynowego w wersji produkcyjnej.

To lektura obowiązkowa dla każdego programisty Pythona!

— **Mikhail Timonin,** projektant, Engelhart

W książce:

- narzędzia NumPy i Cython, a także narzędzia profilujące
- wyszukiwanie „wąskich gardeł" wykorzystania czasu procesora i pamięci
- dobór odpowiednich struktur danych, macierze i wektory
- przyspieszanie sieci neuronowych i obliczeń opartych na procesorach GPU
- zarządzanie wieloma operacjami obliczeniowymi i operacjami wejścia-wyjścia
- przetwarzanie współbieżne w klastrze

Micha Gorelick jest danolożką i ekspertką w dziedzinie uczenia maszynowego. Zajmuje się też dziennikarstwem śledczym. Jest współzałożycielką firmy Fast Forward Labs.

Ian Ozsvald jest danologiem i prelegentem. Współorganizuje coroczną konferencję PyData London, uczestniczy również w innych branżowych wydarzeniach. Twórca społeczności RebelAI.

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-3124-4

Cena: 129,00 zł