Dane Hillard

PYTHON

Dobre praktyki profesjonalistów

Helion

Tytuł oryginału: Practices of the Python Pro

Tłumaczenie: Michał Sternik

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-283-6869-9

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

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

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

Printed in Poland.

Spis treści

4 Projektowanie pod kątem wysokiej wydajności

W tym rozdziale

- Zrozumienie złożoności czasu i przestrzeni
- Mierzenie złożoności kodu
- Wybór typów danych dla różnych działań w Pythonie

Zwykle po napisaniu działającego kodu jest jeszcze coś dodatkowego do zrobienia. Kod, poza tym, że ma wykonać swoje zadanie, ma także wykonać je szybko. *Wydajność* kodu jest miarą tego, jak dobrze wykorzystuje on zasoby pamięci i czas. Oprogramowanie, którego wydajność jest na akceptowalnym poziomie, które wykorzystuje zasoby efektywniej i reaguje na zadania w pożądanym czasie, uważa się za *wydajne*.

Z wydajnością oprogramowania każdego dnia stykają się prawdziwi ludzie, nieważne, czy „wrzucają selfie" na Instagram, czy przeprowadzają analizę rynku w czasie rzeczywistym przed kupnem akcji. To, jak wydajne powinno być oprogramowanie, często sprowadza się do percepcji użytkownika. Jeśli coś *wydaje się* być natychmiastowe, to prawdopodobnie jest wystarczająco szybkie.

Wydajność oprogramowania może również wpływać na niższe warstwy. Jeśli oprogramowanie wymaga zapisania czegoś na dysku lub w bazie danych, minimalizacja ilości wymaganej pamięci pozwala zaoszczędzić pieniądze. Jeśli oprogramowanie, które dostarcza informacji potrzebnych do podejmowania decyzji biznesowych, działa szybciej, może pomóc w zdobyciu większej ilości pieniędzy. Wydajność oprogramowania ma wpływ na funkcjonowanie świata rzeczywistego.

Ludzka percepcja

Zazwyczaj zmiany szybsze niż 100 ms ludzie postrzegają jako natychmiastowe. Są zadowoleni, jeżeli klikną przycisk, a ekran odpowie w 50 ms. Jeśli reakcja wynosi więcej niż 100 ms, ludzie zaczynają dostrzegać opóźnienie.

Nie zawsze można coś poradzić na opóźnienie, dzieje się tak w przypadku długo trwających czynności, takich jak pobieranie dużych plików. Wtedy ważne są dokładne aktualizacje postępu, ponieważ to zmienia postrzeganie, a zadanie wydaje się przebiegać szybciej.

4.1. Pędząc przez czas i przestrzeń

Jeśli czytałeś o wysokiej wydajności oprogramowania, prawdopodobnie napotkałeś frazę o *złożoności czasu* i *złożoności przestrzeni*. Terminy te brzmią tak, jakby wywodziły się prosto z mechaniki kwantowej i astrofizyki, ale mają też zastosowanie w oprogramowaniu.

Złożoności czasu i przestrzeni to miara tego, ile więcej czasu wykonywania programu, pamięci ulotnej lub pamięci dyskowej potrzebuje oprogramowanie w miarę wzrostu ilości danych wejściowych. Im szybciej oprogramowanie zużywa czas lub przestrzeń, tym większa jego złożoność.

Złożoność nie ma być *dokładną* miarą, ma raczej pomóc w zrozumieniu, jak szybkie i duże będzie oprogramowanie w najgorszym przypadku. W tym podrozdziale pomogę Ci wypracować intuicję dla pomiarów złożoności, tak abyś mógł poprawić wydajność w pracy. Istnieje formalny proces, który służy do określania złożoności oprogramowania, ale przejdę do niego za chwilę.

4.1.1. Złożoność jest trochę... złożona

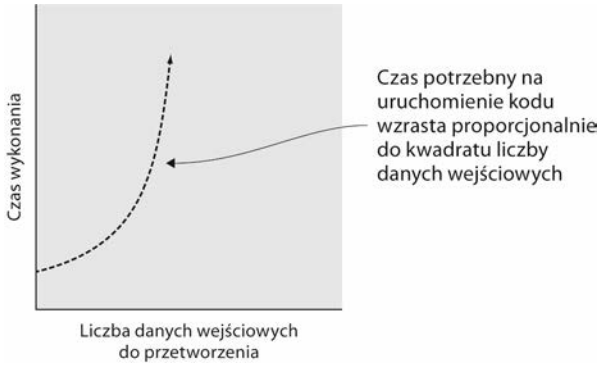
Nie będę przebierał w słowach: pomiar złożoności może być trudny i czasem mylący. Nie rozumiałem tego w szkole — tego, co wiem, nauczyłem się w praktyce. Bądź gotów zrobić tak samo.

Określenia złożoności wykonywane są w procesie zwanym *analizą asymptotyczną*, która polega na obserwacji koduuuuu i określaniu granic najgorszego scenariusza jego wydajności.

UWAGA Należy pamiętać, że pomiary złożoności są wykorzystywane do porównywania sposobów osiągnięcia konkretnego zadania; nie są przydatne do porównywania niepowiązanych zadań. Są np. przydatne do porównywania dwóch algorytmów sortowania listy liczb, ale nie można porównać algorytmu sortowania listy do drzewa wyszukiwania. Upewnij się, że porównujesz jabłka do jabłek.

Notacja stosowana w analizie asymptotycznej może wydawać się tajemnicza na pierwszy rzut oka, ale ma odzwierciedlenie w prostym języku angielskim. Często do opisu złożoności stosuje się *notację dużego* (grecka litera omikron), co oznacza najgorszy przypadek wydajności analizowanego kodu. Notacja z wielką literą może wyglądać tak: *(n2)*, gdzie *n* jest liczbą danych wejściowych, a *n*2 oznacza złożoność. Jest to skrót zdania: „ilość czasu, którego kod potrzebuje do uruchomienia, wzrasta proporcjonalnie

do kwadratu liczby danych wejściowych", co pokazałem na rysunku 4.1. O wiele łatwiej napisać *(n2)*. Notacja z dużym będzie używana dalej w tym rozdziale.



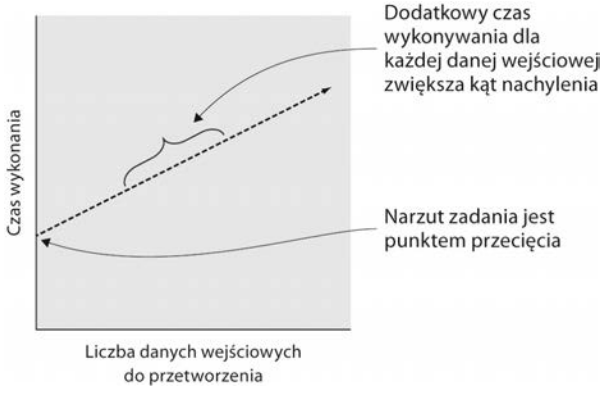
Rysunek 4.1. O(n2) to notacja z dużym O, czyli skrót od związku a ɑ x^2

4.1.2. Złożoność czasowa

Złożoność czasowa jest miarą tego, jak szybko kod może wykonywać zadania w odniesieniu do jego danych wejściowych. Ponieważ liczba danych wejściowych rośnie, złożoność czasowa podaje, w jakim stopniu kod będzie zwalniać. To może pomóc w określeniu, jak długo będzie trwało zadanie z czasem, kiedy liczba danych wejściowych będzie rosnąć.

LINIOWOŚĆ

Złożoność liniowa jest jedną z najczęstszych złożoności wynikających z kodu. Złożoność ta jest tak nazwana, ponieważ przedstawienie liczby danych wejściowych na wykresie w funkcji czasu tworzy linię prostą. Jeśli przypomnisz sobie równania dla funkcji liniowej z matematyki, czyli $y = mx + b$, to za x można przyjąć liczbę wejść, a za y czas potrzebny program do wykonania. Program może być obciążony niezależnie od danych wejściowych programu (b lub punkt przecięcia w równaniu), a każda dodatkowa dana wejściowa dodaje pewną ilość czasu wykonania (m lub nachylenie). Przedstawiłem to na rysunku 4.2.



Rysunek 4.2. Wizualizacja zadania ze złożonością liniową

Złożoność liniowa w oprogramowaniu występuje często, gdyż wiele czynności trzeba wykonać dla każdej pozycji na liście: wyświetlić listę nazwisk, zsumować listę liczb całkowitych itd. Ponieważ lista rośnie, ilość czasu, który komputer musi poświęcić, rośnie proporcjonalnie. Zsumowanie 1000 liczb całkowitych zajmuje około połowę czasu potrzebnego na zsumowanie 2000 liczb całkowitych. Dla pewnej liczby elementów n te rodzaje czynności są liniowe z n lub w notacji z dużym O: $O(n)$.

W Pythonie kod, który prawdopodobnie jest $O(n)$, można łatwo znaleźć, szukając pętli `for`. Pojedyncza pętla na liście, w zbiorze lub innej sekwencji najprawdopodobniej jest liniowa:

```
names = ['Aliya', 'Beth', 'David', 'Kareem']
for name in names:
    print(name)
```

Jest to prawda nawet wtedy, kiedy wewnątrz pętli kod wykonuje wiele czynności:

```
names = ['Aliya', 'Beth', 'David', 'Kareem']
for name in names:
    greeting = 'Cześć, mam na imię'
    print(f'{greeting} {name}')
```

To *nadal* jest prawda nawet wtedy, kiedy pętle wykonywane są na tej samej liście określoną liczbę razy:

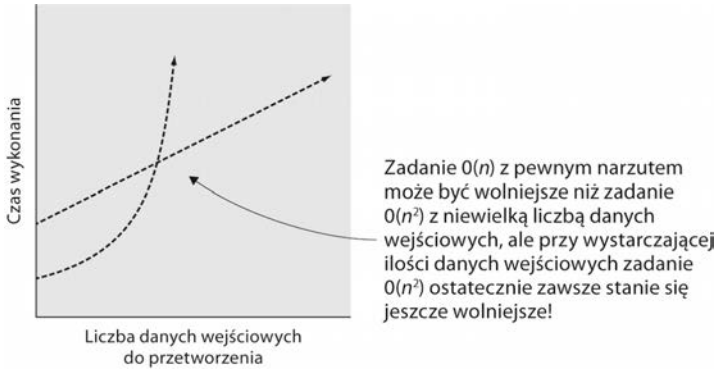
```
names = ['Aliya', 'Beth', 'David', 'Kareem']
for name in names:
    print(f'To jest {name}!')

message = 'Powitajmy '
for name in names:
    message += f'{name} '
print(message)
```

Mimo iż pętlę na tej samej liście wykona się dwukrotnie, wciąż należy o niej myśleć w kategoriach równania liniowego. Dla każdego elementu pierwsza pętla zajmuje trochę czasu (f). Druga pętla też zajmuje trochę czasu (g) dla każdego elementu. Równanie liniowe wyglądałby tak: $y = fx + gx + b$, co jest równoważne z $y = (f + g)x + b$. Wykres dla tego równania nadal będzie linią, jednak bardziej stromą.

To miejsce, w którym zaczyna się „asymptotyczna" część analizy asymptotycznej. Jeśli nawet dana czynność ma *stromą* złożoność liniową, to inne operacje mogą być jeszcze bardziej złożone, pod warunkiem że danych wejściowych jest wystarczająco dużo, co pokazałem na rysunku 4.3.

PROPORCJONALNOŚĆ DO KWADRATU

Innym rodzajem złożoności czasu jest proporcjonalność do *kwadratu* wejść ($O(n^2)$). Objawia się to w przypadkach, w których z każdej pozycji na liście trzeba spojrzeć na każdą pozycję na innej liście. Gdy dodamy więcej danych, program musi iterować po tych dodatkowych elementach, ale musi też iterować po tych dodatkowych elementach przy każdej z *tych* iteracji. Wzrost w czasie wykonywania jest spotęgowany.



Rysunek 4.3. Złożoność wyższego rzędu w dużej skali

Można to znaleźć w kodzie Pythona, szukając obecności pętli zagnieżdżonych. Poniższy kod służy do sprawdzenia, czy w liście znajdują się jakieś duplikaty:

```
def has_duplicates(sequence):
    for index1, item1 in enumerate(sequence):
        for index2, item2 in enumerate(sequence):

            if item1 == item2 and index1 != index2:
                return True
    return False
```

Zewnętrzna pętla wykonuje iterację po każdym elemencie w sekwencji

Wewnętrzna pętla ponownie wykonuje iterację po każdym elemencie dla każdego elementu w zewnętrznej pętli

Sprawdza, czy dwa elementy mają tę samą wartość, ale nie ten sam określony element z sekwencji

$O(n^2)$ to *najgorszy przypadek* dla tego kodu, ponieważ program przed zakończeniem musi wykonać iterację na wszystkich elementach, nawet jeśli tylko ostatnie elementy są duplikatami lub nie występują duplikaty. Jeśli dwie pierwsze pozycje są duplikatami, program zakończy działanie szybciej, ponieważ może zatrzymać się natychmiast, ale sensowne jest badanie najgorszego przypadku, aby uzyskać lepszy wgląd w to, do czego program jest zdolny. Z tego powodu notacja z wielkim O zawsze mierzy najgorszy przypadek złożoności kodu.

Dodatkowe oznaczenia

Przydaje się do obliczania najgorszego przypadku, ale czasami też *przeciętnego* przypadku i *najlepszego* przypadku. Notacja dużej Ω (dużej omegi) używana jest do analizy najlepszego przypadku, a notacja dużej θ (dużej thety) do wyrażania górnej i dolnej granicy specyficznej złożoności. Zazwyczaj mogą one pomóc w wyborze najbardziej odpowiedniego podejścia spośród wielu możliwych. Złożoność wielu algorytmów, takich jak „złożoność algorytmu quicksort", można znaleźć w internecie. Złożoność czasową niektórych typowych operacji można również znaleźć w dokumentacji Pythona (*https://wiki.python.org/moin/TimeComplexity)*.

CZAS STAŁY

Idealną złożonością jest czas stały ($O(1)$), który nie zależy od ilości danych wejściowych. Nic nie może być lepsze od czasu stałego, ponieważ wymagałoby, aby oprogramowanie *przyspieszało* z przyrostem danych wejściowych! Czas stały jest realizowany w niektórych typach danych w Pythonie, o których będzie mowa później.

Niektóre problemy, które normalnie byłyby liniowe (lub gorzej), mogą być przekształcone na stałe, po wykonaniu obliczeń z góry. Takie wstępne obliczenia mogą nie być stałe, ale wykonanie ich może być zyskowne, jeśli pozwalają na *przekształcenie* złożoności wielu kolejnych kroków w stałe.

4.1.3. Złożoność przestrzeni

Podobnie jak w przypadku *złożoności czasu*, złożoność przestrzeni jest miarą tego, jak kod wykorzystuje przestrzeń dyskową lub pamięć, z czasem gdy rośnie liczba danych wejściowych. Jednak złożoność przestrzeni łatwo przeoczyć, ponieważ nie zawsze jest to coś, co obserwujemy bezpośrednio. Czasami nieefektywne wykorzystanie przestrzeni dyskowej daje o sobie znać dopiero wtedy, gdy pojawia się okno pop-up mówiące, że nie ma miejsca na dysku. Aby nie wykorzystać wszystkich zasobów, warto myśleć o miejscu na dane w trakcie pisania kodu.

Śmieciarz

Innym powodem, dla którego złożoność przestrzeni jest trudniejsza w Pythonie, bywa to, że często pamięcią nie zarządza się jawnie. W niektórych językach pamięć należy jednoznacznie przydzielić i zwolnić, co zmusza do zarządzania tym, jak kod wykorzystuje zasoby. Do uwalniania pamięci przechowującej obiekty, które nie będą już używane w uruchomionym programie, Python używa automatycznego mechanizmu nazywanego *zbieraczem śmieci* (ang. *garbage collector*).

PAMIĘĆ

Często powodem używania przez programy zbyt dużej ilości pamięci jest wczytywanie całych dużych plików danych do pamięci, kiedy nie jest to konieczne. Załóżmy, że mamy plik tekstowy, który dla każdej żyjącej obecnie osoby zawiera wiersz, w jakim zapisano jej ulubiony kolor. Dla każdego koloru chcemy poznać liczbę ludzi, którzy go lubią. Można rozważyć wczytanie całego pliku w postaci listy wierszy i działać na liście:

```
color_counts = {}
with open('all-favorite-colors.txt') as favorite_colors_file:
    favorite_colors = favorite_colors_file.read().splitlines()

for color in favorite_colors:
    if color in color_counts:
        color_counts[color] += 1
    else:
        color_counts[color] = 1
```

Wczytuje cały plik do listy wierszy

Na ziemi żyje wielu ludzi. Jeśli nawet plik zawierał będzie tylko jedną kolumnę z ulubionymi kolorami, a każdy wiersz zajmował 1 bajt danych, to wielkość pliku wyniesie nieco ponad 7 GB. W komputerze może być tyle pamięci, ale zadanie nie wymaga tego, aby wczytywać wszystkie informacje dostępne w pliku jednocześnie.

W Pythonie plik można przeczytać wiersz po wierszu w pętli `for`, a w każdej iteracji pętli następny wiersz *zastąpi* w pamięci obecny. Spróbujmy zaktualizować kod tak, aby czytać jedną linię z pliku na raz.

```
color_counts = {}

with open('all-favorite-colors.txt') as favorite_colors_file:
    for color in favorite_colors_file:
        color = color.strip()

        if color in color_counts:
            color_counts[color] += 1
        else:
            color_counts[color] = 1
```

Wczytuje na raz tylko jeden wiersz

Z każdego wiersza usuwa znak nowego wiersza

Dzięki wczytaniu jednego wiersza na raz i wyrzuceniu go z pamięci po przetworzeniu użycie pamięci będzie wynosić tylko tyle, co największy wiersz w pliku. Dużo lepiej! Złożoność przestrzeni wzrosła z *O(n)* do *O(1)*.

MIEJSCE NA DYSKU

W przeszłości w długo żyjących aplikacjach spotykałem się z problemem braku miejsca na dysku. Czasami trudno to zaobserwować, ponieważ problem nie występuje zawsze od razu. Brak miejsca na dysku może wystąpić po kilku tygodniach lub miesiącach albo dlatego, że program zapisuje z czasem niewielkie ilości danych, albo po prostu dlatego, że pamięć dostępna do przechowywania danych jest bardzo duża.

Wiele dużych aplikacji internetowych emituje logi swoich działalności, aby można było w nich było szukać błędów lub je analizować. Jeśli wprowadzi się w kodzie logowanie jakiejś danej, które w wersji produkcyjnej będzie wywoływane 1000 razy na minutę, miejsce na dysku może się skończyć bardzo szybko. Można rozważyć usunięcie tego wiersza, przeniesienie go w miejsce, w którym będzie rzadziej wywoływany, lub zmianę sposobu przechowywania logów.

Znalezienie możliwości zmiany podejścia ze złożoności wyższego do niższego rzędu prawie zawsze daje lepsze zyski w wydajności niż próby wyciśnięcia jak największej wydajności z jakiejś linii kodu. Analizy złożoności należy używać w celu znajdowania takich możliwości w oprogramowaniu. Dalej w tej książce omawiam, jak można zaadresować te możliwości z wykorzystaniem niektórych funkcji wbudowanych w Pythona.

4.2. Wydajność i typy danych

Chociaż kod powinien być zaprojektowany z uwzględnieniem złożoności czasu i przestrzeni, ostatecznie zostanie zbudowany na istniejących typach danych Pythona. W następnych punktach opisałem kilka przypadków użycia, a także to, jakie typy danych są najlepiej do nich dostosowane.

4.2.1. Typy danych dla stałego czasu

Należy pamiętać, że idealna wydajność to ta w stałym czasie, który nie rośnie wraz ze zwiększeniem liczby danych wejściowych. Typy danych Pythona `dict` (słownik) i `set` (zestaw) wykazują to zachowanie podczas dodawania, usuwania i uzyskiwania dostępu do elementów. Wewnątrz są bardzo podobne, główną różnicą jest to, że *słowniki mapują klucze do wartości*, natomiast zestawy stanowią *zestaw unikalnych, indywidualnych wartości*. Iteracja na elementach każdego z tych typów danych wynosi *O(n)*, ponieważ

zależy od liczby pozycji zawartych w obiekcie. Jednak pobieranie określonych elementów lub sprawdzenie, czy istnieje specyficzna pozycja, jest tak samo szybkie, niezależnie od całkowitej liczby pozycji.

Załóżmy, że zamiast liczenia ulubionych kolorów wszystkich ludzi na świecie teraz chcemy uzyskać unikalny zbiór wszystkich ulubionych kolorów, dzięki czemu można będzie sprawdzić, czy jakieś kolory nie zostały wymienione. Tak jak wcześniej, można odczytać wartości z pliku wiersz po wierszu, ale jak przechowywać dane i szukać w nich konkretnych kolorów?

Sam spróbuj zaprezentować dane i wyszukać w nich poszczególne kolory. Potem wróć tutaj i porównaj swoją pracę z kodem umieszczonym w listingu 4.1, aby zobaczyć, jak Ci poszło.

Listing 4.1. Korzystanie z funkcji Pythona w celu zminimalizowania przestrzeni

```
all_colors = set()

with open('all-favorite-colors.txt') as favorite_colors_file:
    for line in favorite_colors_file:
        all_colors.add(line.strip())

print('Amber Waves of Grain' in all_colors)
```

Iteracja po plikach to wciąż złożoność O(n)

Dodawanie do zestawu to złożoność O(1), ale wciąż wewnątrz przestrzeni O(n)

Sprawdzenie przynależności do zestawu to złożoność O(1)

Przy użyciu typu `set` do przechowywania listy unikalnych kolorów występujących w pliku można szukać poszczególnych kolorów w kolekcji w stałym czasie (*O(1)*).

4.2.2. Typy danych w czasie liniowym

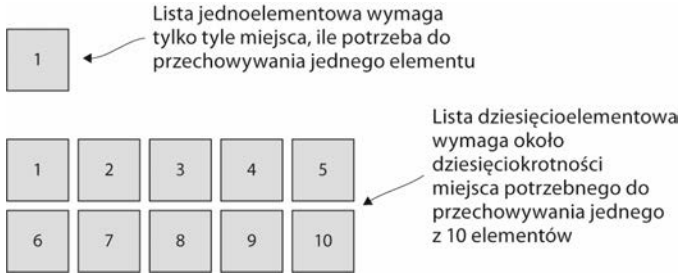
Typ danych `list` (lista) w Pythonie na ogół wykazuje złożoność operacji *O(n)*; określenie przynależności do listy lub dodanie nowego elementu do dowolnego miejsca na liście jest wolniejsze dla list składających się z wielu elementów. Dodawanie lub usuwanie z *końca* listy zajmuje *O(1)* czasu. Listy są użyteczne, gdy przechowywane elementy nie są jednoznacznie rozpoznawalne.

Typ `tuple` (krotka) jest podobny do typu `list` pod względem wydajności, z tą kluczową różnicą, że krotki nie mogą być zmienione po ich utworzeniu.

4.2.3. Złożoność przestrzeni w operacjach na typach danych

Poznałeś już złożoność czasową niektórych struktur danych wbudowanych w Pythona, teraz nauczysz się kilku sposobów na ich wykorzystanie. Wszystkie obiekty typów danych, które omawiałem do tej pory, są *iterowalne* (ang. *iterables*) — czyli są obiektami, które obsługują iterację na ich zawartości (np. w pętli `for`). Iteracja przez elementy obiektu typu `set` prawie zawsze będzie wynosić *O(n)* czasu złożoności; przejście przez wszystkie elementy zajmuje więcej czasu, jeśli elementów jest więcej. A co ze złożonością przestrzeni?

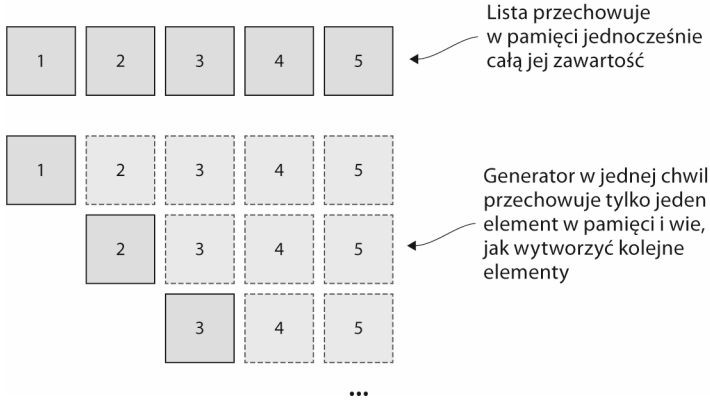
Wszystkie typy danych, które opisałem do tej pory, wszystko, co zawierają, przechowywane jest w pamięci razem. Jeżeli lista zawiera 10 elementów, zajmuje w przybliżeniu 10-krotnie więcej pamięci niż lista z jednym elementem, co pokazałem na rysunku 4.4.



Rysunek 4.4. Ślad pamięciowy list

Oznacza to, że ich złożoność *przestrzeni* też wynosi *O(n)*. Może to być problematyczne, tak samo jak problematyczne jest czytanie 7,6 mld zapisów do pamięci. Jeśli nie potrzebujemy wszystkich tych danych naraz, być może będziemy w stanie znaleźć bardziej efektywne podejście.

Generatory w Pythonie to konstrukcje, które wytwarzają jedną wartość na raz, zatrzymując się, aż wymagana jest następna wartość (patrz rysunek 4.5). To działa trochę jak podejście użyte wcześniej do odczytu pliku wiersz po wierszu. Dając jedną wartość na raz, generator unika przechowywania jednocześnie w pamięci wszystkich wartości, które może wyprodukować.



Rysunek 4.5. Oszczędzanie miejsca z wykorzystaniem generatorów

Jeśli używałeś już funkcji `range` w Pythonie, to korzystałeś z generatora. Funkcja `range` przyjmuje argumenty, które określają granice zakresu ciągu liczb całkowitych. Jeśli funkcja `range` próbowałaby przechowywać wszystkie numery z zakresu w pamięci, kod, taki jak `range(100_000_000)`, spowodowałby szybkie zajęcie całej dostępnej pamięci w krótkim czasie. Zamiast tego funkcja `range` przechowuje w pamięci tylko *granice* zakresu wartości i na jego podstawie produkuje jedną wartość na raz. Jak?

Aby efektywnie wykorzystać przestrzeń, generatory korzystają ze słowa kluczowego języka Python `yield` (daj). Po wyprodukowaniu wartości oddają sterowanie z powrotem do kodu wywołującego. Tak więc instrukcja `yield` powoduje oddanie wartości, a następnie oddanie sterowania.

Instrukcja `yield` działa podobnie do instrukcji `return`, jednak różni się tym, że *po* oddaniu wartości można jeszcze wykonywać operacje. Może to zostać wykorzystane w celu przygotowania do wyprodukowania następnej wartości. W listingu 4.2 pokazałem, jak w przybliżeniu działa funkcja `range`. Zwróć uwagę na zastosowanie słowa kluczowego `yield` i inkrementacji aktualnej wartości *po* jego użyciu.

Listing 4.2. Użycie słowa kluczowego yield do spauzowania i przygotowania

```
def range(*args):
    if len(args) == 1:  <-- Analizuje argumenty w celu ustalenia granic zakresu
        start = 0
        stop = args[0]
    else:
        start = args[0]
        stop = args[1]

    current = start

    while current < stop:  <-- Oddaje (ang. yields) każdą wartość (pojedynczo)
        yield current
        current += 1  <-- Wykonuje ustawienia niezbędne dla następnej wartości
```

W tej implementacji znajduje się wzorzec, który często będzie powtarzany w generatorach.

1. Wykonać konfigurację wymaganą do produkcji wszystkich wartości.
2. Utworzyć pętlę.
3. Oddać wartość w każdej iteracji pętli.
4. Zaktualizować stan dla następnej iteracji pętli.

Teraz sprawdź wartości z Twojego generatora `range`. Możesz np. przekształcić go w listę za pomocą funkcji `list` — `list(range(5, 10))`. Możesz także przenieść jedną wartość na raz, zapisując wartości zwracane z generatora `range(5, 10)` do zmiennej i wykonywać kolejno wywołania funkcji `next(my_range)`.

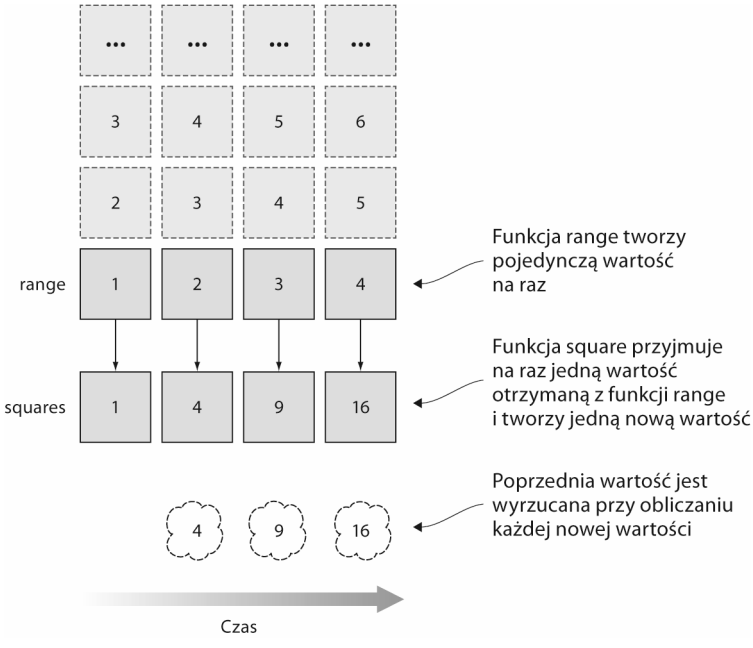
Teraz, gdy już znasz ten wzór, chciałbym, abyś napisał swój własny generator. Twoja funkcja będąca generatorem ma nazywać się `squares`, będzie przyjmować listę liczb całkowitych i produkować kwadrat każdej z nich. Spróbuj i wróć do listingu 4.3, aby zweryfikować swoją pracę.

Listing 4.3. Krótki generator, który generuje kwadraty liczb

```
def squares(items):
    for item in items:
        yield item ** 2
```

Funkcja `squares` okazuje się dość zwarta, ponieważ nie musi wykonywać żadnej konfiguracji ani zarządzać żadnym stanem. Wspomniałem, że funkcja ta przyjmuje listę, ale — co ciekawe — można zamiast tego przekazać do niej inny generator np. `squares ↪(range(100_000_000))` będzie działać tak samo dobrze. Będzie ona przechowywać tylko

jeden element z zakresu i oddawać jeden wynik podnoszenia do kwadratu na raz, dzięki czemu oszczędzi jeszcze więcej miejsca (co pokazałem na rysunku 4.6).



Rysunek 4.6. Użycie pamięci złączonych generatorów

Tam gdzie to możliwe, polecam używać generatorów zamiast list, ponieważ zawsze (jeśli trzeba) przy użyciu generatora można zbudować pełną listę w pamięci, pisząc `list(range(10000))` lub `list(squares([1, 2, 3, 4]))`. Używanie generatorów pozwoli oszczędzić pamięć, ale może także zaoszczędzić czas, gdyż kod, który będzie korzystał z wygenerowanych wartości, może wcale ich wszystkich nie potrzebować.

Wartościowanie leniwe

Ideę produkcji jednej wartości na raz i tego, że kod może nie wymagać wszystkich wartości, które można wyprodukować, często nazywamy *wartościowaniem leniwym*. Jest leniwe, bo chcemy, żeby program wykonywał tak mało pracy, jak to możliwe, i tylko wtedy, gdy dana praca zostanie jawnie wywołana. Wyobraź sobie, że generatory przesadnie wzdychają za każdym razem, kiedy zostaną użyte do uzyskania wartości.

4.3. Zrób to, zrób to dobrze, spraw, żeby było szybkie

Powiedzenie: „Zrób to, zrób to dobrze, spraw, żeby było szybkie" pochodzi od Kenta Becka, twórcy programowania ekstremalnego (ang. *Extreme Programming*). Na pierwszy rzut oka może to oznaczać, że najpierw powinieneś napisać działający kod, a *następnie* przerobić go na jasny i zwięzły, a dopiero *potem* sprawić, by był wydajny. Lubię myśleć o tych trzech zasadach jak o krokach, które można wykonać na każdym etapie procesu pisania kodu. Pamiętaj, że projektowanie, implementacja i refaktoring odbywają się w krótkich cyklach w trakcie pisania kodu.

4.3.1. Zrób to

Oto nad czym programiści spędzają najwięcej czasu. Oznacza próbę przekształcenia opisu problemu lub pomysłu w kod, który osiąga jakiś cel. Programiści (w tym ja) często rozwiązują cały problem przed przejściem do refaktoryzacji lub wydajności. A to wydaje się podobne do problemu jajka i kury: *Jak mogę zrobić, by „to" było szybkie, jeśli „to" nie jest nawet jeszcze zrobione*?

Podobnie jak dekompozycja jest przydatna dla samego oprogramowania, tak samo jest przydatna jako narzędzie w rozbijaniu celów na zarządzalne kawałki. Po drodze do osiągnięcia większego celu każdy z tych mniejszych może być stopniowo realizowany i zbadany. Jest to także o wiele łatwiejsze podejście niż „zrób to", ponieważ „to" jest bardziej szczegółowym celem. Łatwiej można naszkicować kilka pomysłów dla „obliczenia prędkości spadającego przedmiotu" niż „zrobienia silnika fizycznego".

4.3.2. Zrób to dobrze

„Zrób to" znaczy tyle, co dostanie się z punktu A do punktu B. Jeśli znamy cel, to na pytanie: „Czy to działa?", będzie można udzielić jednej z dwóch odpowiedzi.

Zrobienie tego dobrze w całości opiera się na refaktoringu. Refaktoryzacja dąży do ponownej implementacji istniejącego kodu w jaśniejszy lub lepiej dostosowany sposób, zapewniając jednocześnie ten sam spójny wynik[1].

W procesie refaktoryzacji nie ma oczywistego momentu, w którym możemy powiedzieć, że jest zakończony. W trakcie implementacji na pewno będziesz powtarzał jakieś czynności, tak samo jak wtedy, kiedy powrócisz do kodu w celu dodania jakiejś nowej funkcjonalności. Jeśli chodzi o jedną miarę dotyczącą tego, kiedy zdecydowanie *powinieneś* zrefaktoryzować kod, *reguła trzech* Martina Fowlera mówi, że jeśli trzy razy zaimplementowałeś podobną rzecz, powinieneś zrefaktoryzować swój kod tak, aby uzyskać streszczenie tego zachowania. Podoba mi się to założenie, ponieważ sugeruje równowagę wokół refaktoryzacji: nawet po dwukrotnym powieleniu nie należy niczego natychmiast wyciągać do abstrakcji, ponieważ może to być przedwczesne. Trzeba poczekać na to, jakie powstaną nowe przypadki użycia. One pozwolą na skuteczniejsze uogólnienie rozwiązania i upewnienie się, że abstrakcja jest konieczna.

Innym aspektem robienia czegoś dobrze jest używanie mocnych stron języka na swoją korzyść. Spójrz na poniższy kod, który służy do znalezienia liczby całkowitej najczęściej występującej w liście:

```
def get_number_with_highest_count(counts):
    max_count = 0
    for number, count in counts.items():
        if count > max_count:
            max_count = count
            number_with_highest_count = number
    return number_with_highest_count
```

Określa liczbę o najwyższej liczbie wystąpień w słowniku, który mapuje liczby na sumę ich wystąpień

[1] Istnieje szkoła myślenia, która mówi, że możesz tworzyć testy kodu, który piszesz, a jeśli testy są wystarczające i zwracają pozytywny wynik, możesz na nich polegać, kiedy wprowadzasz zmiany, aby upewnić się, że niczego nie zepsułeś. Istnieje wiele fantastycznych tekstów na ten temat. Przeczytaj *Test-Driven Development with Python, Harrygo Percivala*, wydanie drugie (O'Reilly, 2017).

```
def most_frequent(numbers):
    counts = {}
    for number in numbers:
        if number in counts:
            counts[number] += 1
        else:
            counts[number] = 1
    return get_number_with_highest_count(counts)
```

Sprawdza występowanie liczb w celu sprawdzenia, która występuje najczęściej

To rozwiązanie *działa*, ale w Pythonie zawarto kilka narzędzi, które mogą sprawić, że rozwiązanie będzie jeszcze łatwiejsze. Pierwsze narzędzie jest przydatne w kodzie, w którym się inkrementuje. Aby wiedzieć, czy licznik wystąpień musi być inkrementowany lub czy trzeba go zainicjować, narzędzie musi sprawdzić, czy każda liczba wystąpiła już w liście. W Python wbudowano typ danych, który pozwala uniknąć dodatkowej pracy; jest to `defaultdict`. Obiektowi klasy `defaultdict` można przekazać typ przechowywanych wartości; wtedy przyjmie on domyślnie sensowną wartość przekazanego typu, jeśli uzyskiwany jest dostęp do nowego klucza:

```
from collections import defaultdict
```

Importuje defaultdict z modułu collections

```
def get_number_with_highest_count(counts):
    max_count = 0
    for number, count in counts.items():
        if count > max_count:
            max_count = count
            number_with_highest_count = number
        return number_with_highest_count
```

```
def most_frequent(numbers):
    counts = defaultdict(int)
    for number in numbers:
        counts[number] += 1
    return get_number_with_highest_count(counts)
```

Liczby są liczbami całkowitymi, więc domyślnym typem każdej wartości w defaultdict powinien być int

Domyślna wartość dla int to 0, więc gdy liczba zostanie znaleziona pierwszy raz, jej suma wystąpień będzie wynosić 0 + 1 = 1

Nie jest źle — oszczędziliśmy sobie pisania jednego wiersza kodu, a duch funkcji jest nieco bardziej klarowny. Jednak możemy to zrobić jeszcze lepiej. W Pythonie znajduje się też narzędzie, które ułatwia zliczanie elementów w sekwencji:

```
from collections import Counter
```

Klasa Counter też znajduje się w module collections

```
def get_number_with_highest_count(counts):
    max_count = 0
    for number, count in counts.items():
        if count > max_count:
            max_count = count
            number_with_highest_count = number
    return number_with_highest_count


def most_frequent(numbers):
    counts = Counter(numbers)
    return get_number_with_highest_count(counts)
```

Działa prawie identycznie z obliczeniami wykonywanymi ręcznie

Oszczędziliśmy jeszcze kilka wierszy kodu i teraz duch funkcji `most_frequent` jest dość jasny: zlicz unikalne wartości i zwróć tę, której liczba wystąpień jest największa.

A co z funkcją `get_number_with_highest_count`? Znajduje maksymalną wartość w słowniku, w którym zmapowano liczby do ich sumy ich wystąpień. W Pythonie udostępniono dwa narzędzia, które mogą ułatwiać też tę funkcję.

Pierwszą z nich jest funkcja `max`. Funkcja `max` przyjmuje obiekt iterowalny (listy, zestawy, słowniki itd.), po czym zwraca maksymalną wartość z tego obiektu. W przypadku słownika funkcja `max` domyślnie zwraca maksymalną wartość spośród *kluczy*. Kluczami słownika `counts` są same liczby, a nie sumy ich wystąpień. Funkcja `max` przyjmuje też drugi argument, `key`, który jest funkcją mówiącą funkcji `max`, której części obiektu iterowalnego użyć.

Python jako `key` przekaże tylko jeden argument, czyli wartość z obiektu iterowalnego. W przypadku słowników Python iteruje po ich kluczach, więc funkcja przekazana do argumentu `key` funkcji `max` otrzyma tylko liczby, a nie sumy ich wystąpień. W argumencie `key` należy przekazać informację, że kiedy funkcja otrzyma liczbę, powinna ze słownika `counts` wyciągnąć wartość pod indeksem równym tej liczbie. Napisanie osobnej funkcji w module nie zadziała, ponieważ funkcja `counts` nie będzie dostępna we wszystkich przestrzeniach nazw. Jak można to obejść?

W programowaniu funkcyjnym przekazuje się powszechnie funkcje jako argumenty do innych funkcji i czasami takie przekazane funkcje są na tyle krótkie i jasne, że nie muszą być przypisane pod żadną nazwę. Prawdopodobnie w przeciwieństwie do większości funkcji, które napisałeś w Pythonie, są funkcjami *anonimowymi*. Takie funkcje nazywamy lambdami. Lambdy są w istocie funkcjami; przyjmują argumenty i zwracają wartości. Nie mają nazw i nie można się do nich odwołać bezpośrednio, ale można użyć ich w wierszu jako argumentów dla innych funkcji, co jest bardzo przydatne.

W przypadku funkcji `get_number_with_highest_count` można przekazać do funkcji `max` lambdę, która przyjmuje liczbę i zwraca wartość `counts[number]`. To rozwiązuje problem z przestrzenią nazw i oferuje zachowanie, którego potrzebujemy dla funkcji `max`. Zobacz, jak zwięzły dzięki temu będzie kod:

```
from collections import Counter


def get_number_with_highest_count(counts):
    return max(
        counts,
        key=lambda number: counts[number]
    )


def most_frequent(numbers):
    counts = Counter(numbers)
    return get_number_with_highest_count(counts)
```

Podczas iteracji po liczbach w słowniku counts jako wartości porównawczej używa wartości z counts[number] (suma wystąpień tej liczby)

To zwięzłe i nadal jasne. Zrozumienie, jakie narzędzia udostępnia język dla różnych aktywności, często pomaga w tworzeniu krótszego kodu.

Oczywiście nie zawsze krótsze znaczy lepsze. Można by pójść dalej i przenieść funkcję `max` bezpośrednio do funkcji `most_frequent`, ale kiedy używam funkcji, takich jak `max`, których zachowanie nie zawsze jest perfekcyjnie jasne, lubię zostawiać oddzielną funkcję, która ma bardziej deskryptywną nazwę.

Kiedy dotrzesz do punktu, w którym napisany kod działa, a to, *jak* działa, jest na tyle jasne, że ktoś inny może go zrozumieć i użyć, to znaczy, że „zrobiłeś to dobrze".

4.3.3. Spraw, żeby było szybkie

Optymalizacja wydajności kodu często może zająć równie dużo czasu, co samo pisanie kodu. Analiza złożoności i następne ulepszenia wymagają opieki i dokładnego przyjrzenia się typom danych i operacjom w kodzie. Nie raz będziesz musiał pójść na kompromis pomiędzy optymalizacją wydajności oprogramowania a koniecznością jak najszybszego wypuszczenia go na rynek. Jak wspomniałem na początku tego rozdziału, zawsze nadchodzi moment, kiedy trzeba podjąć decyzję, że kod jest *wystarczająco* wydajny. Lepsze jest wrogiem dobrego, więc często lepiej dostarczyć coś działającego, ale wolnego, niż nie dostarczyć niczego.

Jeśli Twoim priorytetem jest wypuszczenie czegoś na rynek, rozważ zdefiniowanie kamieni milowych dotyczących wydajności, które możesz osiągnąć w procesie iteracyjnym już po wstępnym wydaniu. W ten sposób będziesz mógł skupić się na sprawieniu, że produkt będzie działać i będzie działać dobrze, co ułatwi późniejsze usprawnienia i pozwoli na jego dostarczenie. Kiedy uruchomisz kod w produkcji, prawdopodobnie znajdziesz nowe, niespodziewane wąskie gardła.

Twój akceptowalny poziom wydajności będzie się różnił w zależności od Twoich celów. Jeśli wyświetlasz formularz logowania do witryny, po kliknięciu opcji „Zaloguj się" musi stać się to natychmiast, w przeciwnym razie możesz stracić użytkowników. Jeśli próbujesz zbudować roczny system raportowania, tak aby klienci mogli widzieć ich poziom sprzedaży, będą bardziej skorzy do tego, aby chwilę poczekać.

Na wydajność wpływa też *architektura* systemu — wszystkie serwisy, strony, interakcje itd. Większe systemy wymagają komunikacji po sieci pomiędzy różnymi API, bazami danych i buforami. Mogą też wykonywać jakieś obliczenia poza przebiegiem pracy użytkownika, takie jak nocne zbieranie metryk do analizy. Aby zdobyć punkt odniesienia, zawsze możesz przeanalizować inne serwisy w podobnej architekturze, które wykonują podobne do Twoich zadania. Tak możesz opracować świadome oczekiwania dotyczące wydajności Twojego oprogramowania i do nich dążyć. Wydajność dużych systemów wykracza poza kod.

Kiedy piszesz kod, wnoś do swojego oprogramowania to, czego nauczyłeś się o wydajności, typach danych i technikach. Możesz zacząć kształcić zmysł do wykrywania wierszy kodu, które mogą spowodować problemy. Na pewno zaczną Ci się rzucać w oczy zagnieżdżone pętle i wielkie listy przechowywane w pamięci.

4.4. Narzędzia

Prawdziwe testowanie wydajności wymaga podejścia opartego na dowodach. Jest to bezpośredni rezultat tego, że systemy, które mają prawdziwych użytkowników, doświadczają niewątpliwie innego zachowania; mogą to być kombinacja niespodziewanych danych wejściowych, zgranie w czasie, sprzęt, opóźnienia w sieci i inne, które wpływają na wydajność systemu. W związku z tym przeszukiwanie kodu w nadziei na znalezienie ogromnych zysków na wydajności może nie być najlepszym wykorzystaniem Twojego czasu.

4.4.1. timeit

Moduł Pythona timeit jest narzędziem do testowania czasu wykonywania fragmentów kodu. Może być używany z wiersza poleceń lub dla zachowania większej kontroli bezpośrednio z kodu. Moduł timeit jest przydatny do sprawdzania zasadności zmian wydajnościowych, które chcemy wprowadzić.

Wyobraź sobie, że chciałbyś dowiedzieć się, ile czasu zajmie zsumowanie liczb całkowitych z przedziału od 0 do 999. Aby zmierzyć tę czynność z poziomu wiersza poleceń, możesz za pomocą Pythona wywołać moduł timeit.

```
python -m timeit "total = sum(range(1000))"
```

To spowoduje, że moduł timeit uruchomi kod operacji sumowania wiele razy i ostatecznie wydrukuje statystyki na temat czasu wykonywania.

```
20000 loops, best of 5: 18.9 usec per loop
```

Można z tego wywnioskować, że zsumowanie liczb całkowitych z zakresu od 0 do 999 przeważnie nie zajmuje więcej niż 20 mikrosekund.

Aby zobaczyć, jak zmiana zakresu na zakres od 0 do 4999 wpłynie na wynik, możesz zmienić komendę i zwrócić ten wynik:

```
python -m timeit "total = sum(range(5000))"
2000 loops, best of 5: 105 usec per loop
```

Z tego można wywnioskować, że sumowanie liczb całkowitych od 0 do 4999 zajmuje niewiele ponad pięć razy więcej niż zakresu od 0 do 999.

Pamiętaj, że moduł timeit naprawdę uruchamia kod, a czas uruchomienia tego kodu na różnych urządzeniach może różnić się z powodu wielu zmiennych. Oprócz kodu wpływ na wynik czasowy mogą mieć inne czynniki, takie jak poziom baterii czy szybkość zegara procesora. W związku z tym warto kilkakrotnie uruchomić polecenia pomiaru czasu, aby zobaczyć, jak stabilny jest pomiar, i po wprowadzeniu zmian poszukać znaczących ulepszeń. Chociaż moduł timeit pozwala na wielokrotne pomiary, najlepiej go użyć do jakościowego porównania dwóch różnych implementacji, koncentrując się na trendach. To miejsce, w którym można zauważyć znaczne ulepszenia zauważalnie przyspieszające kod.

Interfejs wiersza poleceń dla modułu timeit jest świetny, ale może być nieporęczny do testowania dużych lub bardziej złożonych fragmentów kodu. Jeśli potrzebujesz większej kontroli nad tym, co jest mierzone, możesz użyć modułu timeit wewnątrz kodu. Jeśli chcesz zmierzyć konkretną część kodu bez mierzenia całego kodu do przygotowania danej operacji, możesz oddzielić krok przygotowujący tak, aby czas jego wykonania nie był wliczany w pomiar:

```
from timeit import timeit
setup = 'from datetime import datetime'  <-- Ten kod przygotowuje scenę dla pomiaru czasu
statement = 'datetime.now()'             <-- Ten kod wykonywany jest wewnątrz funkcji timeit
result = timeit(setup=setup, stmt=statement) <-- Funkcja timer zwraca wynik w milisekundach
print(f'Took an average of {result}ms')
```

Wynikiem będzie zmierzenie wyłącznie wywołania funkcji `datetime.now()` bez mierzenia operacji importowania, która konieczna jest do wykonania tej funkcji.

Przypuśćmy, że chciałbyś udowodnić, iż sprawdzanie, czy element znajduje się w zbiorze, jest szybsze od sprawdzania, czy jest w liście. Jak zrobiłbyś to przy użyciu modułu timeit? Utwórz dane wejściowe, używając funkcji `set(range(10000))` i `list(range(10000))` i zmierz, ile trwa znalezienie w nich wartości `300`. O ile szybsze jest wyszukiwanie w zbiorze?

Moduł timeit wiele razy uchronił mnie przed wpadnięciem w króliczą norę, mówiąc mi, że moja hipoteza o przyspieszeniu jakiegoś kodu była błędna. To prawdziwa oszczędność czasu.

4.4.2. Profilowanie CPU

Gdy używałeś modułu timeit, *profilował* on Twój kod. Profilowanie oznacza analizowanie kodu w czasie jego działania w celu zebrania metryk dotyczących jego zachowania. Moduł timeit mierzył, ile czasu zajęło Twojemu programowi wykonanie zadania. Kolejny wnikliwy sposób na pomiar wydajności oprogramowania to profilowanie CPU. Profilowanie pozwala dostrzec, w których *częściach* kodu znajdują się kosztowne obliczenia i jak często są wywoływane. Taki rodzaj informacji jest użyteczny, ponieważ pomaga zrozumieć, gdzie należy zajrzeć najpierw, szukając sposobów na zoptymalizowanie kodu.

Załóżmy, że napisałeś funkcję, której wykonanie nie jest zbyt kosztowne, ale w całym programie jest wykonywana wiele razy, Napisałeś też funkcję, której wykonanie jest kosztowne, ale wywoływana jest tylko raz. Jeśli miałbyś czas, aby naprawić tylko jedną, którą byś wybrał? Bez wykonania profilowania kodu trudno określić, która zmiana najbardziej przyspieszy kod. Można się tego dowiedzieć dzięki użyciu modułu Pythona cProfile.

UWAGA Jeśli przy próbie importowania modułu cProfile otrzymujesz błąd, zamiast niego możesz użyć modułu profile.

Moduł cProfile wyświetla kilka informacji na temat każdej wywołanej podczas wykonywania programu metody czy funkcji. Dla każdego wywołania wyświetli następujące dane.

- Ile razy nastąpiło wywołanie (`ncalls`)?
- Czas poświęcony na to konkretne wywołanie, nie wliczając wywołań kolejnych rzeczy (`tottime`).
- Średni czas wykonywania konkretnego wywołania (`percall`), liczony względem liczby wywołań (`ncalls`).
- Łączny czas wykonywania tego wywołania, wliczając w to czas wykonywania wywołań wewnętrznych (`cumtime`).

Ta informacja jest pomocna, ponieważ uwidacznia wolne miejsca — te, które mają duży `cumtime` — ale ukazuje też części, które są szybkie, jednak wykonywane wiele razy. Na poniższym listingu przedstawiam mały program, w którym wywołano funkcję 1000 razy. Wykonanie funkcji zajmuje losową ilość czasu, mniejszą niż 10 milisekund, co pokazuję w listingu 4.4.

Listing 4.4. Profilowanie wydajności CPU programu Python

```
import random
import time

def an_expensive_function():
    execution_time = random.random() / 100 <-- Wykonanie zajmuje losową ilość
    time.sleep(execution_time)                 czasu (do 10 milisekund)

if __name__ == '__main__':
    for _ in range(1000): <-- Wywołuje funkcję 1000 razy
        an_expensive_function()
```

Zapisz ten program w module *cpu_profiling.py*. Po tym możesz profilować go z wiersza poleceń przy użyciu cProfile:

```
python -m cProfile --sort cumtime cpu_profiling.py
```

W przypadku dużej liczby połączeń można oczekiwać, że funkcja, która zajmuje od 0 do 10 milisekund, potrzebuje średnio około 5 milisekund (`percall`). Wywołanie jej 1000 razy (`ncalls`) powinno trwać średnio około 5 sekund (`cumtime`). Aby sprawdzić, czy program spełnia oczekiwania, uruchom go, korzystając z cProfile. Wyświetli się wiele danych, ale posortowanie według łącznego czasu (ang. *cumulative time*) spowoduje wyświetlenie wywołań funkcji `an_expensive_function` blisko początku listy:

```
$ python -m cProfile --sort cumtime cpu_profiling.py
        5138 function calls (5095 primitive calls) in 5.644 seconds

    Ordered by: cumulative time

    ncalls tottime percall cumtime percall filename:lineno(function)
        4/1 0.000 0.000 5.644 5.644 {built-in method builtins.exec}
          1 0.002 0.002 5.644 5.644 cpu_profiling.py:1(<module>)
        1000 0.003 0.000 5.625 0.006 cpu_profiling.py:5 (an_expensive_function)
        1000 5.622 0.006 5.622 0.006 {built-in method time.sleep}
        ...
```

W powyższym uruchomieniu funkcja `an_expensie_function` wykonywała się średnio w czasie około 6 milisekund na każde wywołanie spośród 100 wywołań, a wszystkie wywołania razem trwały łącznie 5.625 sekund.

Patrząc na dane wyświetlone przez cProfile, szukać będziemy wywołań z największą wartością percall lub z dużym skokiem wartości cumtime. Charakterystyki te oznaczają, że wywołanie zajmuje sporą część czasu wykonywania programu. Przyspieszenie wolnej funkcji może znacznie poprawić prędkość całego programu, a skrócenie czasu wykonywania funkcji wywoływanej tysiące razy może być bardzo zyskowne.

4.5. Wypróbuj

Rozważmy następujący kod. Zawarto w nim funkcję `sort_expensive`, której zadaniem jest posortowanie listy 1000 liczb całkowitych z zakresu od 0 do 999999. Zawarto w nim również funkcję `sort_cheap`, która ma za zadanie posortować listę 10 liczb całkowitych z zakresu od 0 do 999.

Algorytmy sortujące z reguły są bardziej kosztowne niż *O(1)*, więc funkcja `sort_expensive` zajmie więcej czasu niż funkcja `sort_cheap`. Jeśli uruchomimy daną funkcję tylko raz, funkcja `sort_cheap` z pewnością będzie działać szybciej. Jeśli jednak trzeba by uruchomić funkcję `sort_cheal` 1000 razy, to nie wiadomo, która z funkcji okazałaby się szybsza.

```
import random

def sort_expensive():
    the_list = random.sample(range(1_000_000), 1_000)
    the_list.sort()


def sort_cheap():
    the_list = random.sample(range(1_000), 10)
    the_list.sort()


if __name__ == '__main__':
    sort_expensive()
    for i in range(1000):
        sort_cheap()
```

W celu poznania wydajności kodu musisz go sprofilować. To, jak sobie radzi każde zadanie, możesz zobaczyć, korzystając z modułów timeit i cProfile.

Podsumowanie

- Projektuj pod kątem wydajności zarówno na początku, jak i iteracyjnie w całym procesie wytwarzania oprogramowania.
- Dokładnie się zastanów, jaki typ danych jest odpowiedni do zadania.
- Aby zaoszczędzić na zużyciu pamięci, gdy nie potrzebujesz wszystkich wartości jednocześnie, używaj generatorów zamiast list.
- W celu przetestowania swoich hipotez na temat złożoności i wydajności kodu użyj modułów *timeit* oraz *cProfile/profile*.

PYTHON WYDAJE SIĘ JĘZYKIEM IDEALNYM: ma intuicyjną składnię, jest przyjemny w używaniu, umożliwia tworzenie wydajnego, elastycznego kodu. Przy tym jest wyjątkowo wszechstronny, a stosowanie go w przeróżnych celach ułatwiają liczne biblioteki tworzone przez pasjonatów. To jednak nie zmienia faktu, że aby stać się profesjonalnym programistą Pythona, trzeba nauczyć się tworzyć kod godny profesjonalisty: działający bez błędów, czysty, czytelny i łatwy w utrzymaniu. W tym celu trzeba korzystać z branżowych standardów, które określają styl kodowania, projektowania aplikacji i prowadzenie całego procesu programowania. Należy wiedzieć, kiedy i w jaki sposób modularyzować kod, jak poprawić jakość przez zmniejszenie złożoności i stosować kilka innych, koniecznych praktyk.

TA KSIĄŻKA OKAŻE SIĘ SZCZEGÓLNIE CENNA dla każdego, kto zamierza profesjonalnie tworzyć kod w Pythonie. Stanowi jasny i zrozumiały zbiór zasad wytwarzania oprogramowania o najwyższej jakości, praktyk stosowanych przez zawodowych wyjadaczy projektowania i kodowania. Poza teoretycznym omówieniem poszczególnych zagadnień znalazło się tu mnóstwo przykładów i przydatnych ćwiczeń, utrwalających prezentowany materiał. Nie zabrakło krótkiego wprowadzenia do Pythona, przedstawiono też sporo informacji o strukturach danych i różnych podejściach w kontekście osiągania dobrej wydajności kodu. Pokazano, w jaki sposób zapobiegać nadmiernemu przyrostowi kodu podczas rozwijania aplikacji i jak redukować niepożądane powiązania w aplikacji. Dodatkową wartością publikacji jest bogactwo informacji o ogólnej architekturze oprogramowania, przydatnych każdemu zawodowemu programiście.

W KSIĄŻCE MIĘDZY INNYMI:

- **podstawy projektowania w Pythonie**
- **wysokopoziomowe koncepcje rozwoju oprogramowania**
- **abstrakcje i hermetyzacja kodu**
- **różne metody testowania kodu**
- **tworzenie dużych systemów a rozszerzalność i elastyczność aplikacji**

PYTHONA PRAKTYKUJ PROFESJONALNIE!

DANE HILLARD jest głównym programistą aplikacji internetowych w ITHAKA, organizacji non profit działającej w szkolnictwie wyższym. Wcześniej zajmował się budowaniem mechanizmów wnioskowania z danych telemetrycznych i potokami ETL dla aplikacji bioinformatycznych. Stara się kodować kreatywnie i łączyć różne pasje — do muzyki, fotografii, jedzenia i programowania. Często występuje na międzynarodowych konferencjach poświęconych Pythonowi i Django.

Helion

helion.pl

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

ISBN 978-83-283-6869-9

9 788328 368699

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 59,00 zł