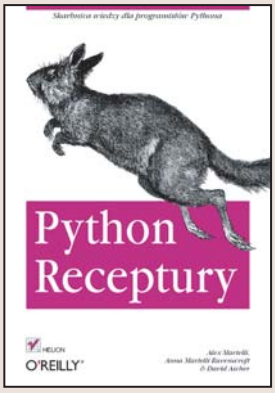
Python. Receptury



Autorzy: Alex Martelli, Anna Martelli Ravenscroft, David Ascher
Tłumaczenie: Wojciech Moch, Marek Pętlicki
ISBN: 83-246-0214-3
Tytuł oryginału: Python Cookbook
Format: B5, stron: 848

Python został opracowany na początku lat "90 i szybko zyskał uznanie programistów. Elastyczny i uniwersalny, pozwalał na stosowanie zasad programowania obiektowego, strukturalnego i funkcyjnego. Był i nadal jest wykorzystywany nie tylko do tworzenia skryptów, ale również przy dużych projektach, takich jak na przykład serwer aplikacji Zope. Decydując się na korzystanie z Pythona, stajemy się częścią niezwykłej społeczności programistów, chętnie pomagającej każdemu, kto chce doskonalić umiejętność posługiwania się tym językiem.

Książka „Python. Receptury" to zbiór rozwiązań problemów, z jakimi w codziennej pracy borykają się programiści korzystający z tego języka. Materiały do niej przygotowało ponad 300 członków społeczności Pythona odpowiadających na pytania zadawane na forum internetowym. Rozwiązania zostały przetestowane w praktyce, co ułatwia ich zaimplementowanie we własnych projektach.

W książce umówiono m.in.:

- Przetwarzanie tekstów
- Operacje na plikach
- Programowanie obiektowe
- Przeszukiwanie i sortowanie
- Łączenie skryptów z bazami danych
- Testowanie i usuwanie błędów
- Programowanie wielowątkowe
- Realizację zadań administracyjnych
- Obsługę interfejsów użytkownika
- Tworzenie aplikacji sieciowych
- Przetwarzanie dokumentów XML

Każdy programista Pythona, niezależnie od umiejętności, znajdzie w tej książce coś dla siebie

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 5.

Szukanie i sortowanie

5.0. Wprowadzenie

Pomysłodawca: Tim Peters, PythonLabs

> W latach 60. producenci komputerów szacowali, że 25% czasu pracy wszystkich sprzedanych przez nich urządzeń przeznaczane jest na zadania związane z sortowaniem. W rzeczywistości było wiele takich instalacji, w których zadanie sortowania zajmowało ponad połowę czasu pracy komputerów. Z tych danych można wywnioskować, że: a) istnieje wiele bardzo ważnych powodów sortowania, b) wielokrotnie sortuje się dane bez potrzeby lub c) powszechnie stosowane były nieefektywne algorytmy sortowania.
>
> — Donald Knuth
>
> The Art of Computer Programming, tom 3,
> *Sorting and Searching*, strona 3.

Wspaniała praca profesora Knutha na temat sortowania i wyszukiwania ma niemal 800 stron złożonego tekstu technicznego. W praktyce Pythona całą tę pracę redukuje się do dwóch imperatywów (ktoś inny przeczytał to „opasłe tomisko", dlatego Czytelnik zostanie zwolniony z tego obowiązku):

- Jeżeli musimy sortować dane, to najlepiej będzie znaleźć sposób na wykorzystanie wbudowanej w Pythona metody `sort` zajmującej się sortowaniem list.
- Jeżeli musimy przeszukiwać dane, to najlepiej będzie znaleźć sposób na wykorzystanie wbudowanego typu słowników.

Wiele receptur z niniejszego rozdziału kieruje się właśnie tymi dwoma zasadami. Najczęściej stosowanym w nich rozwiązaniem jest implementacja wzorca DSU (ang. *decorate-sort-undecorate* — dekoruj-sortuj-usuń dekorację). Jest to najogólniejsze rozwiązanie polegające na utworzeniu listy pomocniczej, którą można posortować za pomocą domyślnej, szybkiej metody `sort`. Ta technika należy do najużyteczniejszych spośród wszystkich prezentowanych w tym rozdziale. Co więcej, wzorzec DSU jest tak dalece przydatny, że w Pythonie 2.4 wprowadzono kilka nowych funkcji ułatwiających jego stosowanie. W efekcie w Pythonie 2.4 wiele z prezentowanych receptur można jeszcze bardziej uprościć, dlatego analiza starszych receptur została uzupełniona o odpowiednie instrukcje.

Wzorzec DSU polega na wykorzystaniu niezwykłych właściwości wbudowanych w Pythona porównań: sekwencje są porównywane leksykograficznie. Kolejność leksykograficzna jest uogólnieniem wszystkich znanych nam reguł porównywania ciągów znaków (czyli kolejności alfabetycznej), które rozciągnięte zostały na krotki i listy. W przypadku, gdy zmienne `s1` i `s2` są sekwencjami, wbudowana funkcja `cmp(s1, s2)` jest równoważna z następującym kodem Pythona:

```
def lexcmp(s1, s2):
    # Znajdź nierówną parę po lewej stronie.
    i = 0
    while i < len(s1) and i < len(s2):
        outcome = cmp(s1[i], s2[i])
        if outcome:
            return outcome
        i += 1
    # Wszystkie równe do momentu wyczerpania przynajmniej jednej sekwencji.
    return cmp(len(s1), len(s2))
```

Podany kod poszukuje pierwszej pary nierównych sobie elementów. Po znalezieniu takiej pary na jej podstawie określany jest wynik porównania. W przeciwnym przypadku, jeżeli jedna z sekwencji jest dokładnym przedrostkiem drugiej, to taki przedrostek uznawany jest za mniejszą sekwencję. W końcu, jeżeli nie obowiązuje żaden z wymienionych wyżej przypadków, znaczy to, że sekwencje są identyczne i w związku z tym uznawane są za równe. Oto kilka przykładów:

```
>>> cmp((1, 2, 3), (1, 2, 3))     # identyczne
0
>>> cmp((1, 2, 3), (1, 2))        # pierwsza większa, ponieważ druga jest przedrostkiem
1
>>> cmp((1, 100), (2, 1))         # pierwsza jest mniejsza, ponieważ 1<2
-1
>>> cmp((1, 2), (1, 3))           # pierwsza jest mniejsza, ponieważ 1==1, ale 2 < 3
-1
```

Bezpośrednią konsekwencją takich porównań leksykograficznych jest to, że w przypadku, gdy chcielibyśmy posortować listę obiektów według klucza głównego, a w przypadku równości wartości tego klucza — według klucza drugorzędnego, należy zbudować listę krotek, gdzie każda krotka przechowuje klucz główny, klucz drugorzędny i oryginalny obiekt, dokładnie w tej kolejności. Krotki porównywane są leksykograficznie, dlatego taka kolejność ich elementów automatycznie załatwia sprawę. W czasie porównywania krotek najpierw porównywane są klucze główne, a jeżeli są one równe, to (tylko w takim przypadku) porównywane są klucze drugorzędne.

Podawane w tym rozdziale przykłady wzorca DSU prezentują wiele zastosowań takiego postępowania. Technikę DSU można stosować z dowolną liczbą kluczy. Krotki można uzupełniać o kolejne klucze, umieszczając je w takiej kolejności, w jakiej mają być porównywane. W Pythonie 2.4 ten sam efekt można uzyskać, podając do metody `sort` opcjonalny parametr `key=`, tak jak robione jest to w niektórych recepturach. Stosowanie parametru `key=` metody `sort` jest prostsze, efektywniejsze i szybsze od ręcznego tworzenia listy krotek.

Inną z nowości wprowadzonych do Pythona 2.4 w zakresie sortowania jest bardzo wygodny skrót: wbudowana funkcja `sorted` pozwalająca na sortowanie dowolnego elementu iterowalnego. Takie sortowanie nie odbywa się „w miejscu", ale przez kopiowanie danych do nowej listy. W Pythonie 2.3 (oprócz nowego opcjonalnego parametru nazywanego, który można stosować zarówno w funkcji `sorted`, jak i w metodzie `list.sort`) to samo działanie można zaimplementować bez większego wysiłku:

```
def sorted_2_3(iterable):
    alist = list(iterable)
    alist.sort()
    return alist
```

Operacje kopiowania i sortowania listy nie są operacjami trywialnymi, a wbudowana funkcja `sorted` i tak musi je wykonać, dlatego przekształcenie funkcji `sorted` w funkcję wbudowaną nie dało praktycznie żadnego wzrostu prędkości jej działania. Jedyną zaletą jest tu po prostu wygoda. Likwidacja konieczności powtarzalnego wpisywania nawet czterech prostych wierszy kodu i świadomość, że pewne elementy mamy zawsze pod ręką, *naprawdę* stanowi ogromną poprawę wygody pracy. Z drugiej strony, zaledwie niewielka część prostych funkcji używana jest na tyle powszechnie, żeby usprawiedliwiało to rozbudowę zbioru elementów wbudowanych. W Pythonie 2.4 do tego zbioru dodane zostały funkcje `sorted` i `reversed`, ponieważ w ciągu ostatnich lat często pojawiały się prośby o dodanie ich do elementów wbudowanych.

Największa zmiana w mechanizmach sortowania stosowanych w Pythonie od czasu pierwszego wydania tej książki polegała na wprowadzeniu do Pythona 2.3 nowej implementacji algorytmu sortowania. Pierwszą widoczną konsekwencją tej zmiany był wzrost prędkości w wielu typowych przypadkach oraz fakt, że nowy algorytm jest stabilny, co oznacza, że dwa elementy, które w oryginalnej liście są sobie równe, w posortowanej liście zachowują swoją względną kolejność. Nowa implementacja była niezwykle udana, a szanse na przygotowanie lepszej były tak nikłe, że Guido dał się przekonać, że w Pythonie metoda `list.sort` już zawsze będzie stabilna. Nowa funkcja sortująca pojawiła się już w wersji 2.3, ale gwarancja stabilności algorytmu sortowania wprowadzona została dopiero w wersji 2.4. Mimo to historia algorytmów sortowania każe nam pamiętać, że zawsze mogą zostać odkryte jeszcze lepsze algorytmy sortowania. W związku z tym należałoby tutaj podać skróconą historię sortowania w Pythonie.

Krótka historia sortowania w Pythonie

We wczesnych wersjach Pythona metoda `list.sort` wykorzystywała funkcję `qsort` pochodzącą z biblioteki języka C. Takie rozwiązanie nie sprawdzało się z wielu powodów, ale przede wszystkim dlatego, że jakość funkcji `qsort` nie była jednolita na wszystkich komputerach. Niektóre wersje działały wyjątkowo wolno w przypadkach, gdy miały posortować listę z wieloma jednakowymi wartościami albo ułożoną w odwrotnej kolejności sortowania. Zdarzały się też wersje powodujące zawieszenie się procesu, ponieważ nie pozwalały na stosowanie rekursji. Zdefiniowana przez użytkownika funkcja `__cmp__` może wywoływać metodę `list.sort`, dlatego w efekcie ubocznym jedno wywołanie `list.sort` może powodować kolejne takie wywołania w związku z wykonywanymi porównaniami. Na niektórych platformach funkcja `qsort` nie była w stanie poradzić sobie z taką sytuacją. Zdefiniowana (w sposób głupi lub złośliwy) przez użytkownika funkcja `__cmp__` może też „zmutować" listę w czasie jej sortowania i dlatego na wielu platformach funkcja `qsort` może sobie nie radzić z takimi właśnie sytuacjami.

W Pythonie przygotowana została zatem specjalna implementacja algorytmu szybkiego sortowania (ang. *quicksort algorithm*). Była ona zmieniana w każdej następnej wersji języka, ponieważ znajdowane były kolejne przypadki rzeczywistych zastosowań, w których aktualna w danym momencie implementacja okazywała się niezwykle powolna. Jak się okazuje, quicksort to wyjątkowo delikatny algorytm!

W Pythonie 1.5.2 algorytm quicksort został zastąpiony hybrydą algorytmów sortowania przez wybieranie (ang. *samplesort*) i sortowania przez wstawienia (ang. *insertionsort*). Ta implementacja nie uległa zmianie przez ponad cztery lata, aż do momentu pojawienia się Pythona 2.3.

Algorytm samplesort można traktować jak wariant algorytmu quicksort, w którym używane są bardzo duże próbki do wybierania elementu rozdzielającego (metoda ta rekursywnie sortuje algorytmem samplesort duży losowy podzbiór elementów i wybiera z nich medianę). Taki wariant sprawia, że prawie nie jest możliwy wariant z czasem sortowania proporcjonalnym do kwadratu liczby elementów, a liczba porównań w typowych przypadkach jest zdecydowanie bliższa teoretycznemu minimum.

Niestety, algorytm samplesort jest na tyle skomplikowany, że jego administracja danymi okazuje się zdecydowanie zbyt rozbudowana przy pracach z niewielkimi listami. Z tego właśnie powodu małe listy (a także niewielkie wykrojenia powstające w wyniku podziałów dokonywanych przez ten algorytm) obsługiwane są za pomocą algorytmu insertionsort (jest to zwyczajny algorytm sortowania przez wstawianie, ale do określania pozycji każdego z elementów korzysta on z mechanizmów szukania binarnego). W większości tekstów na temat sortowania zaznaczane jest, że takie podziały nie są warte naszej uwagi, ale wynika to z faktu, że w tekstach tych uznaje się, że operacja porównania elementów jest mniej czasochłonna od operacji zamiany tych elementów w pamięci, co nie jest prawdą w algorytmach sortowania stosowanych w Pythonie. Przeniesienie obiektu jest operacją bardzo szybką, ponieważ kopiowana jest tylko referencja tego obiektu. Z kolei porównanie dwóch obiektów jest operacją kosztowną, ponieważ za każdym razem uruchamiany jest kod przeznaczony do wyszukiwania obiektów w pamięci oraz kod odpowiedni do wykonania porównania danych obiektów. Jak się okazało, z tego właśnie powodu w Pythonie najlepszym rozwiązaniem jest sortowanie binarne.

To hybrydowe rozwiązanie uzupełnione zostało jeszcze o obsługę kilku typowych przypadków ukierunkowaną na poprawę prędkości działania. Po pierwsze, wykrywane są listy już posortowane lub posortowane w odwrotnej kolejności i obsługiwane w czasie liniowym. W pewnych aplikacjach takie sytuacje zdarzają się bardzo często. Po drugie, dla tablicy w większości posortowanej, w której nieposortowanych jest tylko kilka ostatnich elementów, całą pracę wykonuje algorytm sortowania binarnego. Takie rozwiązanie jest znacznie szybsze od sortowania takich list algorytmem samplesort, a przedstawiona sytuacja bardzo często pojawia się w aplikacjach, które naprzemiennie sortują listę, dodają do niej nowe elementy, znowu sortują itd. W końcu specjalny kod w algorytmie samplesort wyszukuje ciągi jednakowych elementów i zajmowaną przez nie część listy od razu oznacza jako posortowaną.

W efekcie takie sortowanie w miejscu odbywa się z doskonałą wydajnością we wszystkich znanych typowych przypadkach i osiąga nienaturalnie dobrą wydajność w pewnych typowych przypadkach specjalnych. Cała implementacja zapisana została w ponad 500 wierszach skomplikowanego kodu w języku C bardzo podobnego do tego prezentowanego w recepturze 5.11.

Przez lata, w których używany był algorytm samplesort, cały czas oferowałem obiad temu, kto przygotuje szybszy algorytm sortujący dla Pythona. Przez cały ten czas musiałem jadać sam. Mimo to ciągle śledzę pojawiającą się literaturę, ponieważ pewne aspekty stosowanej w Pythonie hybrydy algorytmów sortujących są nieco irytujące:

- Co prawda w rzeczywistych zastosowaniach nie pojawiają się przypadki sortowania tablicy w czasie proporcjonalnym do kwadratu ilości elementów, ale wiem, że takie przypadki można sobie wyobrazić, natomiast powstanie przypadków, w których sortowanie przebiega dwa do trzech razy wolniej od średniej, jest całkiem realne.

- Specjalne przypadki przyspieszające sortowanie w sytuacjach wyjątkowych układów danych z całą pewnością były nieocenioną pomocą w normalnej praktyce, ale w czasie moich prac często spotykałem się z innymi rodzajami losowych układów danych, które można by było obsłużyć w podobny sposób. Doszedłem do wniosku, że w przypadkach rzeczywistych praktycznie nigdy nie występują całkowicie losowo ułożone elementy list wejściowych (a szczególnie poza środowiskami przygotowanymi do testowania algorytmów sortujących).
- Nie istnieje praktyczny sposób przygotowania stabilnego algorytmu samplesort bez jednoczesnego znaczącego powiększenia wykorzystania pamięci.
- Kod był niezwykle złożony, a specjalne przypadki komplikowały go jeszcze bardziej.

Aktualny algorytm sortowania

Od zawsze było wiadomo, że algorytm mergesort w pewnych przypadkach sprawuje się lepiej, w tym również w najgorszym przypadku złożoności `n log n`, a dodatkowo łatwo można przygotować jego stabilną wersję. Jak się jednak okazało, pół tuzina prób implementowania tego algorytmu w Pythonie spełzło na niczym, ponieważ przygotowane procedury działały wolniej (w algorytmie mergesort przenosi się znacznie więcej danych niż w algorytmie samplesort) i zajmowały więcej pamięci.

Coraz większa część literatury zaczyna opisywać *adaptacyjne* algorytmy sortowania, które próbują wykrywać kolejność elementów w różnych danych wejściowych. Sam przygotowałem kilka takich algorytmów, ale wszystkie okazały się być wolniejsze od pythonowej implementacji algorytmu samplesort, z wyjątkiem tych przypadków, na obsługę których były specjalnie przygotowywane. Teoretyczne podstawy tych algorytmów były po prostu zbyt złożone, żeby na ich bazie można było w praktyce utworzyć efektywny algorytm. Przeczytałem wtedy artykuł, w którym wskazywano na fakt, że dzięki sprawdzaniu liczby kolejnych „zwycięstw" poszczególnych danych wejściowych łączenie list w sposób *naturalny* wykazuje wiele cech porządku częściowego. Ta informacja była bardzo prosta i ogólna, ale w momencie gdy uświadomiłem sobie, że można by wykorzystać ją w naturalnym algorytmie mergesort, który oczywiście wykorzystałby wszystkie znane mi rodzaje porządku częściowego, moją obsesją stało się takie przygotowanie algorytmu, żeby rozwiązać problemy z prędkością sortowania danych losowych i zminimalizować zajętość pamięci.

Przygotowana „adaptacyjna, naturalna i stabilna" implementacja algorytmu mergesort dla Pythona 2.3 była wielkim sukcesem, ale związana była również z wielkim nakładem prac inżynieryjnych — po prostu diabeł tkwił w szczegółach. Implementacja ta zajmowała mniej więcej 1200 wierszy kodu w języku C. Jednak w przeciwieństwie do hybrydowej implementacji algorytmu samplesort ten kod nie zawiera obsługi żadnych przypadków specjalnych, ale jego dużą część zajmuje pewna sztuczka pozwalająca na zmniejszenie o połowę zajętości pamięci w najgorszym z możliwych przypadków. Jestem bardzo dumny z tej implementacji. Niestety, niewielka ilość miejsca przeznaczona na to wprowadzenie nie pozwala mi na opisanie tu szczegółów tego rozwiązania. Jeżeli ktoś jest ciekaw, to odsyłam go do opisu technicznego (ostrzegam, że jest długi), jaki przygotowałem i jaki dostępny jest wśród źródeł dystrybucji Pythona w pliku *Objects/listsort.txt*, w katalogu, do którego zainstalowana została dystrybucja Pythona. W poniższej liście podaję przykłady porządków częściowych, jakie wykorzystuje implementacja algorytmu mergesort w Pythonie 2.3. Na liście tej słowo „posortowany" oznacza prawidłową kolejność posortowanych elementów lub ich odwrotną kolejność:

- Dane wejściowe są już posortowane.
- Dane wejściowe są w większości posortowane, ale mają dopisane losowe dane na początku i (lub) na końcu albo wstawione w środek.
- Dane wejściowe są złożeniem dwóch lub więcej list złożonych. Najszybszym sposobem na połączenie kilku posortowanych list w Pythonie jest teraz złączenie ich w jedną wielką listę i wywołanie na jej rzecz funkcji `list.sort`.
- Dane wejściowe są w większości posortowane, ale pewne elementy nie są ułożone w prawidłowej kolejności. Typowym przykładem takiego stanu jest sytuacja, gdy użytkownicy ręcznie dopisują nowe dane do bazy danych posortowanej według nazwisk. Jak wiemy, ludzie nie najlepiej radzą sobie z takim dopisywaniem danych i utrzymywaniem ich w porządku alfabetycznym, ale często są bliscy wstawienia elementu we właściwe miejsce.
- Wśród danych wejściowych znajduje się wiele kluczy o takich samych wartościach. Na przykład w czasie sortowania bazy danych firm amerykańskich notowanych na giełdzie większość z takich firm powiązana będzie z indeksami NYSE lub NASDAQ. Taki fakt można wykorzystać w bardzo ciekawy sposób: klucze o takiej samej wartości są już posortowane, co wynika z faktu, że stosowany jest „stabilny" algorytm sortowania.
- Dane wejściowe były posortowane, ale zostały rozbite na kawałki, które później poskładano w losowej kolejności, a dodatkowo elementy niektórych kawałków zostały skutecznie przemieszane. Jest to oczywiście bardzo dziwny przykład, ale w jego wyniku powstaje porządek częściowy danych, co wskazuje na wielką ogólność prezentowanej metody.

Mówiąc krótko, w Pythonie 2.3 funkcja `timsort` (cóż, musiała dostać *jakąś* krótką nazwę) jest rozwiązaniem stabilnym, skutecznym i nienaturalnie szybkim w wielu rzeczywistych przypadkach, w związku z czym należy z niej korzystać jak najczęściej!

5.1. Sortowanie słownika

Pomysłodawca: Alex Martelli

Problem

Chcemy posortować słownik. Najprawdopodobniej oznacza to, że chcemy posortować klucze, a następnie pobierać z niego wartości w tej samej kolejności sortowania.

Rozwiązanie

Najprostsze rozwiązanie tego problemu zostało już wyrażone w opisie problemu: należy posortować klucze i wybierać powiązane z nimi wartości:

```
def sortedDictValues(adict):
    keys = adict.keys()
    keys.sort()
    return [adict[key] for key in keys]
```

Analiza

Koncepcja sortowania dotyczy wyłącznie tych kolekcji, których elementy mają jakąś kolejność (czyli sekwencję). Odwzorowania takie jak słowniki nie mają kolejności, wobec czego nie mogą być sortowane. Mimo to na listach dyskusyjnych dotyczących Pythona często pojawiają się całkowicie bezsensowne pytania „Jak mogę posortować słownik?". Najczęściej takie pytanie oznacza jednak, że osoba pytająca chciała posortować pewną sekwencję składającą się z kluczy i (lub) ich wartości pobranych ze słownika.

Jeżeli chodzi o podaną implementację, to co prawda można wymyślić bardziej złożone rozwiązania, ale jak się okazuje (w Pythonie nie powinno być to niespodzianką), kod podany w rozwiązaniu jest rozwiązaniem najprostszym, a jednocześnie najszybszym. Poprawę prędkości działania o mniej więcej 20% można w Pythonie 2.3 uzyskać przez zastąpienie w instrukcji `return` listy składanej wywołaniem funkcji `map`, na przykład:

```
return map(adict.get, keys)
```

W Pythonie 2.4 wersja podawana w rozwiązaniu jest jednak o wiele szybsza niż ta w Pythonie 2.3, dlatego z takiej zamiany nie zyskamy zbyt wiele. Inne warianty, takie jak na przykład zastąpienie metody `adict.get` metodą `adict.__getitem__` nie powodują już poniesienia prędkości działania funkcji, a na dodatek mogą spowodować pogorszenie wydajności zarówno w Pythonie 2.3, jak i w Pythonie 2.4.

Zobacz również

Recepturę 5.4, w której opisywane są sposoby sortowania słowników na podstawie przechowywanych wartości, a nie kluczy.

5.2. Sortowanie listy ciągów znaków bez uwzględniania wielkości liter

Pomysłodawcy: Kevin Altis, Robin Thomas, Guido van Rossum, Martin V. Lewis, Dave Cross

Problem

Chcemy posortować listę ciągów znaków, ignorując przy tym wszelkie różnice w wielkości liter. Oznacza to, że chcemy, aby litera `a`, mimo że jest małą literą, znalazła się przed wielką literą `B`. Niestety, domyślne porównywanie ciągów znaków uwzględnia różnice wielkości liter, co oznacza, że wszystkie wielkie litery umieszczane są przed małymi literami.

Rozwiązanie

Wzorzec DSU (decorate-sort-undecorate) sprawdza się tu doskonale, tworząc szybkie i proste rozwiązanie:

```
def case_insensitive_sort(string_list):
    auxiliary_list = [(x.lower(), x) for x in string_list]    # dekoracja
    auxiliary_list.sort()                                     # sortowanie
    return [x[1] for x in auxiliary_list]                     # usunięcie dekoracji
```

W Pythonie 2.4 wzorzec DSU obsługiwany jest w samym języku, dlatego (zakładając, że obiekty listy `string_list` rzeczywiście są ciągami znaków, a nie na przykład obiektami ciągów znaków Unikodu) można w nim zastosować poniższe, jeszcze szybsze i prostsze rozwiązanie:

```
def case_insensitive_sort(string_list):
    return sorted(string_list, key=str.lower)
```

Analiza

Dość oczywistą alternatywą dla rozwiązania podawanego w tej recepturze jest przygotowanie funkcji porównującej i przekazanie jej do metody `sort`:

```
def case_insensitive_sort_1(string_list):
    def compare(a, b): return cmp(a.lower(), b.lower())
    string_list.sort(compare)
```

Niestety, w ten sposób przy każdym porównaniu dwukrotnie wywoływana jest metoda `lower`, a liczba porównań koniecznych do posortowania listy `n`-elementowej zazwyczaj jest proporcjonalna do `n log(n)`.

Wzorzec DSU tworzy listę pomocniczą, której elementami są krotki, w których każdy element z oryginalnej listy poprzedzany jest specjalnym „kluczem". Następnie sortowanie odbywa się według tych właśnie kluczy, ponieważ Python porównuje krotki *leksykograficznie*, czyli pierwsze elementy krotek porównuje w pierwszej kolejności. Dzięki zastosowaniu wzorca DSU metoda `lower` wywoływana jest tylko `n` razy w czasie sortowania listy `n` ciągów znaków, co pozwala oszczędzić na tyle dużo czasu, że całkowicie rekompensuje konieczność początkowego *udekorowania* listy i końcowego *zdjęcia przygotowanych dekoracji*.

Wzorzec DSU czasami znany jest też pod (nie do końca poprawną) nazwą *transformacji Schwartza*, która jest nieprecyzyjną analogią do znanego idiomu języka Perl. Poprzez zastosowanie takich porównań wzorzec DSU jest bardziej zbliżony do *transformacji Guttmana-Roslera* (więcej informacji na stronie *http://www.sysarch.com/perl/sort_paper.html*).

Wzorzec DSU jest tak ważny, że w Pythonie 2.4 wprowadzono jego bezpośrednią obsługę. Do metody `sort` można opcjonalnie przekazać nazywany argument `key` będący elementem wywoływalnym, używanym w czasie sortowania do uzyskania klucza sortowania każdego elementu listy. Jeżeli do funkcji `sort` zostanie przekazany taki argument, to automatycznie zacznie ona korzystać z wzorca DSU. Oznacza to, że w Pythonie 2.4 wywołanie `string_list.sort (key=str.lower)` jest równoważne z wywołaniem funkcji `case_insensitive_sort`. Jedyna różnica polega na tym, że metoda `sort` sortuje listę w miejscu (i zwraca wartość `None`), a funkcja `case_insensitive_sort` zwraca posortowana kopię listy, nie modyfikując przy tym oryginału. Jeżeli chcielibyśmy, żeby funkcja `case_insensitive_sort` również sortowała listę w miejscu, to wystarczy wynik jej pracy przypisać do ciała wejściowej listy:

```
string_list[:] = [x[1] for x in auxiliary_list]
```

Z drugiej strony, jeżeli w Pythonie 2.4 chcielibyśmy uzyskać posortowana kopię listy, bez modyfikowania oryginału, to możemy skorzystać z wbudowanej funkcji `sorted`. Na przykład zapis:

```
for s in sorted(string_list, key=str.lower): print s
```

w Pythonie 2.4 wypisuje wszystkie ciągi znaków zapisane na liście `string_list`, która zostaje posortowana bez uwzględniania różnic w wielkości liter, a jej oryginał pozostaje bez zmian.

Wykorzystanie w Pythonie 2.4 metody `str.lower` w argumencie `key` ogranicza nas do sortowania wyłącznie ciągów znaków (nie da się tak posortować na przykład ciągów znaków Unikodu). Jeżeli wiemy, że będziemy sortować obiekty Unikodu, to należy posłużyć się parametrem `key=unicode.lower`. Jeżeli chcielibyśmy uzyskać funkcję, która działałaby tak samo wobec prostych ciągów znaków i ciągów znaków Unikodu, to można zaimportować moduł `string` i posłużyć się argumentem `key=string.lower`. Ewentualnie można też skorzystać z zapisu `key=lambda s: s.lower()`.

Skoro musimy czasem sortować listy ciągów znaków w sposób nieuwzględniający różnic wielkości liter, to równie dobrze możemy potrzebować słowników lub zbiorów stosujących klucze nieuwzględniające różnic wielkości liter, a także list, w których podobnie zachowują się metody `index` oraz `count` i inne. W takich sytuacjach potrzebny jest nam typ wywiedziony z klasy `str`, który nie uwzględniałby różnic wielkości liter w operacjach porównywania i mieszania (ang. hashing). Jest to zdecydowanie lepsze rozwiązanie w porównaniu z implementowaniem wielu różnych typów kontenerowych i funkcji obejmujących przedstawioną funkcję. Sposób implementowania takiego typu podawany był już w recepturze 1.24.

Zobacz również

Zbiór często zadawanych pytań (dostępny na stronie *http://www.python.org/cgi-bin/faqw.py?req=show&file=faq04.051.htp*. Recepturę 5.3. Podręcznik *Library Reference* Pythona 2.4 w częściach opisujących wbudowaną funkcję `sorted` oraz parametr `key` metod `sort` i `sorted`. Recepturę 1.24.

5.3. Sortowanie listy obiektów na podstawie ich atrybutów

Pomysłodawcy: Yakov Markovitch, Nick Perkins

Problem

Musimy posortować listę obiektów według wartości jednego z atrybutów tych obiektów.

Rozwiązanie

Tutaj również doskonale sprawdza się wzorzec DSU:

```
def sort_by_attr(seq, attr):
    intermed = [ (getattr(x, attr), i, x) for i, x in enumerate(seq) ]
    intermed.sort()
    return [ x[-1] for x in intermed ]
def sort_by_attr_inplace(lst, attr):
    lst[:] = sort_by_attr(lst, attr)
```

W Pythonie 2.4, w którym wzorzec DSU został wbudowany w język, kod ten może być jeszcze krótszy i szybszy:

```
import operator
def sort_by_attr(seq, attr):
    return sorted(seq, key=operator.attrgetter(attr))
def sort_by_attr_inplace(lst, attr):
    lst.sort(key=operator.attrgetter(attr))
```

Analiza

Sortowanie listy obiektów według określonego atrybutu najlepiej wykonuje się za pomocą wzorca DSU omawianego w poprzedniej recepturze 5.2. W Pythonie 2.3 i 2.4 nie jest on już potrzebny do tworzenia stabilnego sortowania, co było konieczne w poprzednich wersjach języka (od wersji 2.3 algorytm sortowania stosowany w Pythonie zawsze jest stabilny), a mimo to inne zalety wzorca DSU nadal są niepodważalne.

W ogólnym przypadku i z wykorzystaniem najlepszego algorytmu sortowanie ma złożoność `O(n log n)` (w formułach matematycznych taki zapis oznacza, że wartości *n* i `log n` są mnożone). Siła wzorca DSU polega na wykorzystaniu wyłącznie wbudowanych w Pythona (i przez to najszybszych) mechanizmów porównania, przez co maksymalnie przyspieszana jest ta część wyrażenia `O(n log n)`, która zabiera najwięcej czasu w operacji sortowania sekwencji o bardzo dużej długości. Początkowy krok *dekorowania*, w którym przygotowywana jest pomocnicza lista krotek, oraz końcowy krok *usuwania dekoracji*, w którym z posortowanej listy krotek wydobywane są właściwe informacje, oba mają złożoność tylko `O(n)`. Oznacza to, że drobne niedociągnięcia w tych dwóch krokach będą miały nikły wpływ na sortowanie list z wielką liczbą elementów, a w przypadku niewielkich list wpływ tych kroków również będzie względnie niewielki.

Notacja O( )

Najbardziej użyteczny sposób określania wydajności danego algorytmu polega na wykorzystaniu tak zwanej analizy lub notacji *wielkiego* `O` (litera `O` oznacza po angielsku „order", czyli „rząd"). Dokładne objaśnienia dotyczące tej notacji znaleźć można na stronie *http://pl.wikipedia.org/wiki/Notacja_du%C5%BCego_O*. Tutaj podamy tylko krótkie podsumowanie.

Jeżeli przyjrzymy się algorytmom stosowanym na danych wejściowych o rozmiarze *N*, to dla odpowiednio dużych wartości `N` (duże ilości danych wejściowych sprawiają, że wydajność danego rozwiązania staje się sprawą krytyczną) czas ich działania może być określany jako proporcjonalny do pewnej funkcji wartości *N*. Oznacza się to za pomocą notacji takiej jak `O(N)` (czas pracy algorytmu jest proporcjonalny do *N*; przetwarzanie dwukrotnie większego zbioru danych zajmuje dwa razy więcej czasu, a przetwarzanie zbioru dziesięciokrotnie większego zajmuje dziesięć razy więcej czasu; inna nazwa tej notacji to *złożoność liniowa*), $O(N^2)$ (czas pracy algorytmu jest proporcjonalny do kwadratu *N*; przetwarzanie dwukrotnie większego zbioru danych zajmuje cztery razy więcej czasu, a przetwarzanie zbioru dziesięciokrotnie większego zajmuje sto razy więcej czasu; inna nazwa tej notacji to *złożoność kwadratowa*) itd. Często będziemy natykać się też na zapis `O(N log N)`, który oznacza algorytm szybszy niż $O(N^2)$, ale wolniejszy od algorytmu `O(N)`.

Najczęściej ignorowane są stałe proporcji (przynajmniej w rozważaniach teoretycznych), ponieważ zazwyczaj zależą one od takich czynników jak częstotliwość zegara w naszym komputerze, a nie od samego algorytmu. Jeżeli zastosujemy dwa razy szybszy komputer, to całość będzie trwała o połowę krócej, ale nie zmieni to wyników porównań poszczególnych algorytmów.

W tej recepturze w każdej krotce będącej elementem listy `intermed` umieszczamy indeks `i` przed odpowiadającym mu elementem `x` (`x` jest `i`-tym elementem sekwencji `seq`). W ten sposób upewniamy się, że dowolne dwa elementy sekwencji `seq` nie będą porównywane bezpośrednio, nawet jeżeli mają taką samą wartość atrybutu `attr`, ponieważ w takiej sytuacji ich indeksy będą miały różne wartości. W ten sposób wykonywane w Pythonie leksykograficzne

porównania krotek nigdy nie będą porównywać ostatnich elementów poszczególnych krotek (czyli właściwych elementów sekwencji `seq`). Uniknięcie porównywania samych obiektów pozwala nam uniknąć wykonywania niezwykle powolnych porównań, a nawet prób wykonania porównań niedozwolonych. Na przykład moglibyśmy sortować liczby zespolone według ich części rzeczywistej (atrybut `real`); przy próbie bezpośredniego porównania takich liczb powstałby wyjątek, ponieważ w liczbach zespolonych nie ma zdefiniowanej takiej operacji. Dzięki obostrzeniom opisywanym w tym akapicie takie porównanie nigdy nie nastąpi i w związku z tym sortowanie przebiegnie bezproblemowo.

Jak już wspominaliśmy wcześniej w recepturze 5.2, w Pythonie 2.4 wzorzec DSU obsługiwany jest w samym języku. Do metody `sort` można przekazywać opcjonalny argument nazywany `key`, który jest wywoływany na rzecz każdego elementu sortowanej listy i ma zwrócić klucz sortowania. Moduł `operator` będący częścią biblioteki standardowej udostępnia dwie nowe funkcje — `attrgetter` i `itemgetter` — przeznaczone do zwracania elementów wywoływalnych nadających się do omówionych wyżej zastosowań. Dzięki temu w Pythonie 2.4 idealnym rozwiązaniem naszego problemu staje się poniższy kod:

```
import operator
seq.sort(key=operator.attrgetter(attr))
```

Podany kod pozwala posortować listę w miejscu i to z niesamowitą prędkością — na moim komputerze sortowanie było trzykrotnie szybsze niż ta sama operacja wykonana w Pythonie 2.3 za pomocą funkcji prezentowanej w rozwiązaniu tej receptury. Jeżeli potrzebna jest nam posortowana kopia listy, bez modyfikowania jej oryginału, to w Pythonie 2.4 możemy wykorzystać nową, wbudowaną funkcję `sorted`.

```
sorted_copy = sorted(seq, key=operator.attrgetter(attr))
```

Nie jest to aż tak szybkie jak sortowanie w miejscu, ale ten ostatni kod jest nadal ponad 2,5 razy szybszy od funkcji przedstawianej w rozwiązaniu receptury. Przekazując opcjonalny argument nazywany `key` w Pythonie 2.4, uzyskujemy też pewność, że w czasie sortowania elementy listy nie zostaną porównane bezpośrednio ze sobą, więc nie musimy tworzyć żadnych dodatkowych zabezpieczeń. Co więcej, posortowana lista na pewno będzie stabilna.

Zobacz również

Recepturę 5.2. Podręcznik *Library Reference* z Pythona 2.4 w części opisującej wbudowaną funkcję `sorted`, funkcje `attrgetter` i `itemgetter` z modułu `operator` oraz parametr `key` funkcji `sort` i `sorted`.

5.4. Sortowanie kluczy lub indeksów na podstawie związanych z nimi wartości

Pomysłodawcy: John Jensen, Fred Bremmer, Nick Coghlan

Problem

Musimy zliczyć wystąpienia różnych elementów i zaprezentować te elementy w kolejności częstotliwości występowania — na przykład w celu przygotowania histogramu.

Rozwiązanie

Histogramy niezależnie od swojego zastosowania przy tworzeniu grafiki tworzone są przez zliczanie wystąpień elementów (co nietrudno jest wykonać w przypadku list lub słowników w Pythonie) i sortowanie list lub indeksów w kolejności odpowiadającej wyznaczonym wartościom. Oto klasa wywiedziona z klasy `dict`, która uzupełniona została o dwie metody:

```
class hist(dict):
    def add(self, item, increment=1):
        ''' dodaje wartość 'increment' do pozycji elementu 'item' '''
        self[item] = increment + self.get(item, 0)
    def counts(self, reverse=False):
        ''' zwraca listę kluczy posortowaną zgodnie z odpowiadającymi im wartościami '''
        aux = [ (self[k], k) for k in self ]
        aux.sort()
        if reverse: aux.reverse()
        return [k for v, k in aux]
```

Jeżeli zliczane elementy mogą być modelowane jako niewielkie liczby całkowite z określonego zakresu i wyniki zliczania elementów chcemy przechowywać na liście, to można zastosować bardzo podobne rozwiązanie:

```
class hist1(list):
    def __init__(self, n):
        ''' inicjalizacja listy do zliczania wystąpień n różnych elementów '''
        list.__init__(self, n*[0])
    def add(self, item, increment=1):
        ''' dodaje wartość 'increment' do pozycji elementu 'item' '''
        self[item] += increment
    def counts(self, reverse=False):
        ''' zwraca listę indeksów posortowaną zgodnie z odpowiadającymi im wartościami '''
        aux = [ (v, k) for k, v in enumerate(self) ]
        aux.sort()
        if reverse: aux.reverse()
        return [k for v, k in aux]
```

Analiza

Metoda `add` klasy `hist` jest przykładem wykorzystania typowego dla Pythona idiomu przeznaczonego do zliczania dowolnych (choć unikalnych) elementów. W klasie `hist1`, zbudowanej na podstawie klasy `list`, stosowane jest inne rozwiązanie, polegające na wpisywaniu w specjalnej metodzie `__init__` wartości `0` do wszystkich elementów listy, dzięki czemu metoda `add` może przyjąć prostszą postać.

Metoda `counts` tworzy listę kluczy lub indeksów posortowanych w kolejności wyznaczanej przez powiązane z nimi wartości. Problem jest bardzo podobny w obu klasach (`hist` i `hist1`), dlatego podane rozwiązania są niemal identyczne — w obu wykorzystywany jest wzorzec DSU omawiany w recepturach 5.2 i 5.3. Jeżeli w naszym programie chcielibyśmy skorzystać z obu klas, to możemy wykorzystać ich podobieństwo i wydzielić części wspólne do pomocniczej funkcji `_sorted_keys`:

```
def _sorted_keys(container, keys, reverse):
    ''' zwraca listę 'keys' posortowaną zgodnie z wartościami z parametru 'container' '''
    aux = [ (container[k], k) for k in keys ]
    aux.sort()
    if reverse: aux.reverse()
    return [k for v, k in aux]
```

Następnie metody `counts` obu klas można zaimplementować jako opakowania zaprezentowanej funkcji `_sorted_keys`:

```
class hist(dict):
    ## ...
    def counts(self, reverse=False):
        return _sorted_keys(self, self, reverse)
class hist1(list):
    ## ...
    def counts(self, reverse=False):
        return _sorted_keys(self, xrange(len(self)), reverse)
```

W Pythonie 2.4 wzorzec DSU jest tak ważny, że (jak pokazano w recepturach 5.2 i 5.3) metoda `sort` z obiektów list oraz nowa, wbudowana funkcja `sorted` oferują bardzo szybką implementację tego wzorca. Dzięki temu w Pythonie 2.4 funkcja `_sorted_keys` może być jeszcze prostsza i szybsza:

```
def _sorted_keys(container, keys, reverse):
    return sorted(keys, key=container.__getitem__, reverse=reverse)
```

Metoda powiązana `container.__getitem__` wykonuje dokładnie te same operacje co indeksowanie `container[k]` stosowane w implementacji dla Pythona 2.3, ale jest ona elementem wywoływalnym, który może być wywoływany na rzecz każdego elementu `k` z sortowanej sekwencji — dokładnie taką wartość należy przekazywać w parametrze `key` do wywoływanej funkcji `sorted`. W Pythonie 2.4 udostępniany jest też prosty i bezpośredni sposób na odczytanie listy elementów słownika posortowanych według wartości:

```
from operator import itemgetter
def dict_items_sorted_by_value(d, reverse=False):
    return sorted(d.iteritems(), key=itemgetter(1), reverse=reverse)
```

Funkcja wysokiego poziomu `operator.itemgetter` (również wprowadzona została w Pythonie 2.4) jest bardzo wygodnym sposobem dostarczania argumentu `key` w czasie sortowania kontenera, którego elementy są podkontenerami, a kluczem sortowania ma być określony element podkontenera. Dokładnie taką sytuację mamy w przedstawionym przypadku, ponieważ elementy słownika są sekwencją par (krotek dwuelementowych), a my chcemy posortować tę sekwencję według drugiego elementu każdej krotki.

Wracając do głównego tematu tej receptury, oto przykład użycia klasy `hist` prezentowanej w rozwiązaniu receptury:

```
sentence = ''' Halo! To jest test. Halo! To był test,
           ale już nim nie jest. '''
words = sentence.split()
c = hist()
for word in words: c.add(word)
print "Rosnąco:"
print c.counts()
print "Malejąco:"
print c.counts(reverse=True)
```

Podany wycinek kodu tworzy następujące dane wyjściowe:

```
Rosnąco:
[(1, 'ale'), (1, 'był'), (1, 'jest'), (1, 'jest.'), (1, 'już'), (1, 'nie'), (1, 'nim'),
(1, 'test,'), (1, 'test.'), (2, 'Halo!'), (2, 'To')]
Malejąco:
[(2, 'To'), (2, 'Halo!'), (1, 'test.'), (1, 'test,'), (1, 'nim'), (1, 'nie'), (1, 'już'),
(1, 'jest.'), (1, 'jest'), (1, 'był'), (1, 'ale')]
```

Zobacz również

Recepturę „Special Method Names" zamieszczoną w podręczniku *Library Reference* oraz rozdział o programowaniu obiektowym w podręczniku *Python in a Nutshell*, w części opisującej metodę `__getitem__`. Podręcznik *Library Reference* Pythona 2.4 w części opisującej wbudowaną funkcję `sorted` oraz parametr `key` funkcji `sort` i `sorted`.

5.5. Sortowanie ciągów znaków zawierających liczby

Pomysłodawcy: Sébastien Keim, Chui Tey, Alex Martelli

Problem

Musimy tak posortować listę ciągów znaków zawierających sekwencje cyfr (na przykład listę kodów adresowych), żeby wyglądała jak najlepiej. Na przykład tekst `'foo2.txt'` powinien znaleźć się przed tekstem `'foo10.txt'`. Niestety, w Pythonie domyślnie stosowane jest porównanie alfabetyczne, więc tekst `'foo10.txt'` znajdzie się przed `'foo2.txt'`.

Rozwiązanie

Musimy podzielić każdy ciąg znaków na sekwencje cyfr i niecyfr, a następnie każdą sekwencję cyfr zamienić w liczbę. W ten sposób uzyskamy listę przechowującą właściwe klucze do sortowania listy. Następnie można skorzystać ze wzorca DSU do wykonania samego sortowania. Jak widać, wystarczy nam przygotować dwie króciutkie funkcje:

```
import re
re_digits = re.compile(r'(\d+)')
def embedded_numbers(s):
    pieces = re_digits.split(s)             # dzielenie na cyfry/niecyfry
    pieces[1::2] = map(int, pieces[1::2])   # zamiana cyfr w liczby
    return pieces
def sort_strings_with_embedded_numbers(alist):
    aux = [ (embedded_numbers(s), s) for s in alist ]
    aux.sort()
    return [ s for __, s in aux ]           # konwencja: __ oznacza "ignoruj"
```

W Pythonie 2.4 można skorzystać z wbudowanej obsługi wzorca DSU (przyda się też przedstawiona wyżej funkcja `embedded_numbers`) i posortować listę za pomocą poniższej funkcji:

```
def sort_strings_with_embedded_numbers(alist):
    return sorted(alist, key=embedded_numbers)
```

Analiza

Załóżmy, że mamy nieposortowaną listę nazw plików podobną do poniższej:

```
files = 'plik3.txt plik11.txt plik7.txt plik4.txt plik15.txt'.split()
```

Jeżeli w Pythonie 2.4 podaną listę posortujemy i wypiszemy za pomocą instrukcji `print ' '.join(sorted(files))`, to na ekranie zobaczymy następujący tekst: `plik11.txt plik15.txt plik3.txt plik4.txt plik7.txt`. Taka kolejność wynika z faktu, że domyślnie ciągi znaków sortowane są alfabetycznie (a mówiąc bardziej wymyślnie — sortowane są *leksykograficznie*).

Python nie może się domyślać, że chcemy inaczej traktować ciągi znaków, ponieważ pewne części ciągów znaków dziwnym trafem opisują liczby. Musimy dokładnie określić, co chcemy zrobić, i w tej recepturze pokazujemy, jak można tego dokonać:

Korzystając z tej receptury, można uzyskać znacznie lepiej wyglądające wyniki:

```
print ' '.join(sort_strings_with_embedded_numbers(files))
```

Podana instrukcja wypisuje teraz tekst: `plik3.txt plik4.txt plik7.txt plik11.txt plik15.txt` i najprawdopodobniej o tę kolejność plików chodziło nam w tym przypadku.

Implementacja rozwiązania opiera się na wzorcu DSU. Jeżeli chcemy, żeby takie sortowanie działało w Pythonie 2.3, to wzorzec ten musimy jawnie zapisać w kodzie funkcji, natomiast kod przygotowany dla Pythona 2.4 może wykorzystywać wbudowaną implementację wzorca. Wykorzystanie wbudowanej implementacji wzorca DSU wymaga przekazania nazywanego argumentu `key` (argument ten określa funkcję, która ma być wywoływana na rzecz każdego elementu sortowanej listy w celu uzyskania klucza sortowania) do nowej wbudowanej funkcji `sorted`.

W tej recepturze właściwy klucz do porównywania poszczególnych elementów uzyskujemy za pomocą funkcji `embedded_numbers`, która zwraca listę cząstkowych ciągów znaków zawierającą zamiennie sekwencje niecyfr i wartości typu `int` uzyskanych z każdej sekwencji cyfr. Metoda `re_digits.split(s)` daje nam listę naprzemiennych sekwencji niecyfr i cyfr, przy czym sekwencje cyfr umieszczane są pod indeksami parzystymi. Następnie korzystamy z wbudowanych funkcji `map` i `int` (oraz rozbudowanych wykrojeń pobierających wszystkie elementy i ustawiających je pod indeksami nieparzystymi), aby zamienić sekwencje cyfr w liczby. Leksykograficzne porównania wykonywane w takiej liście składającej się z elementów różnych typów generują oczekiwany rezultat.

Zobacz również

Podręczniki *Library Reference* i *Python in a Nutshell* w częściach opisujących wykrojenia i moduł `re`. Podręcznik *Library Reference* z Pythona 2.4 w części opisującej wbudowaną funkcję `sorted` i parametr `key` przekazywany do funkcji `sort` i `sorted`. Receptury 5.3 i 5.2.

5.6. Przetwarzanie wszystkich elementów listy w kolejności losowej

Pomysłodawcy: Iuri Wickert, Duncan Grisby, T. Warner, Steve Holden, Alex Martelli

Problem

Wszystkie elementy długiej listy musimy obsłużyć w kolejności losowej.

Rozwiązanie

Jak to zwykle bywa w Pythonie, najlepszym rozwiązaniem jest rozwiązanie najprostsze. Jeżeli wolno nam będzie zmienić kolejność elementów w wejściowej liście, to poniższa funkcja będzie zdecydowanie najprostsza i najszybsza:

```
def process_all_in_random_order(data, process):
    # po pierwsze, listę układamy w porządku losowym
    random.shuffle(data)
    # po drugie, liniowo obsługujemy wszystkie jej elementy
    for elem in data: process(elem)
```

Jeżeli wejściowa lista musi zostać niezmieniona lub wejściowe dane zapisane są w elemencie iterowalnym niebędącym listą, to wystarczy dopisać do powyższej funkcji pierwszą instrukcję w postaci przypisania `data = list(data)`.

Analiza

Powszechnym błędem jest przywiązywanie zbytniej wagi do prędkości działania kodu, ale z drugiej strony nie można też popełniać odwrotnego błędu polegającego na ignorowaniu różnic wydajności poszczególnych algorytmów. Załóżmy, że musimy w kolejności losowej i bez powtórzeń obsłużyć wszystkie elementy długiej listy (zakładamy też, że możemy zmodyfikować, a nawet zniszczyć listę wejściową). Pierwszym pomysłem, jaki zapewne przyszedłby nam do głowy, jest losowe wybieranie jednego elementu (za pomocą funkcji `random.choice`) i po jego obsłużeniu usuwanie z listy w celu zapobieżenia powstawaniu powtórzeń:

```
import random
def process_random_removing(data, process):
    while data:
        elem = random.choice(data)
        data.remove(elem)
        process(elem)
```

Taka funkcja jest niestety bardzo powolna, nawet w przypadku list z zaledwie kilkuset elementami. Każde wywołanie metody `data.remove` wymaga liniowego przejrzenia wszystkich elementów listy w celu odnalezienia tego przeznaczonego do usunięcia. Każdy taki krok ma złożoność $O(n)$, a zatem cały proces ma złożoność $O(n^2)$ — czas obsługi listy jest proporcjonalny do *kwadratu* jej długości (a zwykle listy nie należą do krótkich).

Kolejne usprawnienia tego pierwszego pomysłu mogą polegać na: wybieraniu losowych indeksów oraz za pomocą metody `pop` pobieraniu samych elementów i jednoczesnym usuwaniu ich z listy, niskopoziomowych zabawach z indeksami mających na celu uniknięcie kosztownego usuwania elementów listy i zastępowaniu wybranego elementu ostatnim jeszcze niewybranym elementem albo zastępowaniu listy słownikami lub zbiorami. Ten ostatni pomysł może wynikać z nadziei, że uda się skorzystać z metody `popitem` obiektów słowników (lub równoważnej jej metody `pop` z klasy `sets.Set` albo wbudowanego w Pythona 2.4 typu `set`), ponieważ na pierwszy rzut oka wygląda ona na przeznaczoną do wybierania i usuwania losowych elementów, ale… zgodnie z dokumentacją Pythona metoda `dict.popitem` służyć ma do zwracania i usuwania *dowolnego* elementu słownika, który zdecydowanie nie jest elementem *losowym*. Proszę przyjrzeć się poniższemu kodowi:

```
>>> d=dict(enumerate('ciao'))
>>> while d: print d.popitem()
```

Niektórych może to zaskoczyć, ale w większości implementacji Pythona podany kod wcale nie wypisze elementów słownika `d` w kolejności losowej. Najczęściej zobaczymy najpierw `(0, 'c')`, następnie `(1, 'i')` itd. Jeżeli w Pythonie potrzebne jest nam zachowanie pseudolosowe, to obowiązkowo musimy skorzystać z modułu `random` — metoda `popitem` nie jest tutaj żadnym rozwiązaniem.

Jeżeli ktoś myślał o zamianie listy na słownik, to na pewno jest w stanie rozumować w sposób pythoniczny, mimo że w tym konkretnym problemie zastosowanie słowników nie powoduje znaczącego podniesienia prędkości działania. Jak się okazuje, istnieje jeszcze bardziej pythoniczne rozwiązanie od wybrania najwłaściwszej struktury danych. Można je podsumować następującym zdaniem: niech zrobi to biblioteka standardowa. Biblioteka standardowa Pythona jest ogromnym, bogatym zbiorem przydatnych, skutecznych i szybkich funkcji oraz klas przeznaczonych do wykonywania najróżniejszych operacji. W tym przypadku najważniejszą rzeczą jest to, żeby zaakceptować fakt, że najprostszym sposobem na obsłużenie wszystkich elementów listy w losowej kolejności jest *ułożenie* ich najpierw w kolejności losowej (proces ten nazywany jest *tasowaniem sekwencji*, przez analogię do tasowania talii kart) i liniowe obsłużenie elementów listy. Takie przetasowanie elementów listy wykonuje funkcja `random.shuffle` i dlatego wykorzystana została w rozwiązaniu tej receptury.

Wydajność danego rozwiązania zawsze musi zostać zmierzona; nigdy nie należy polegać na domysłach. Do przeprowadzania takich pomiarów najlepiej będzie wykorzystać moduł `timeit`. W połączeniu z pustą funkcją `process` i listą składającą się z tysiąca elementów funkcja `process_all_in_random_order` okazała się być prawie dziesięciokrotnie szybsza od funkcji `process_random_removing`. W przypadku listy składającej się z dwóch tysięcy elementów stosunek prędkości tych dwóch funkcji wynosi już prawie 20. Zwykle poprawę wydajności o, powiedzmy, 25% albo stały współczynnik 2 można uznać za niewartą naszej uwagi, to jednak nie można tak samo traktować algorytmu 10 lub 20 razy wolniejszego od swojego konkurenta. Tak wyjątkowo niska wydajność bardzo łatwo może spowodować, że dana część programu stanie się wąskim. Co więcej, takie ryzyko wzrasta jeszcze bardziej, gdy zaczynamy porównywać algorytm o złożoności $O(n^2)$ z algorytmem o złożoności $O(n)$. Przy takich różnicach złożoności algorytmów stosunek prędkości działania złego i dobrego algorytmu rośnie bez żadnych ograniczeń wraz ze wzrostem liczby danych wejściowych.

Zobacz również

Dokumentację modułów `random` i `timeit` dostępną w podręcznikach *Library Reference* i *Python in a Nutshell*.

5.7. Utrzymywanie porządku w sekwencji w czasie dodawania do niej nowych elementów

Pomysłodawca: John Nielsen

Problem

Chcemy utrzymać w stanie posortowania sekwencję, do której dodawane są nowe elementy, tak żeby w dowolnym momencie można było prosto sprawdzić lub usunąć najmniejszy z zapisanych aktualnie elementów.

Rozwiązanie

Załóżmy, że początkowo mamy listę nieuporządkowaną, taką jak poniższa:

```
the_list = [903, 10, 35, 69, 933, 485, 519, 379, 102, 402, 883, 1]
```

Można teraz wywołać metodę `the_list.sort()`, aby posortować listę, a następnie skorzystać z metody `the_list.pop(0)`, aby pobrać i usunąć najmniejszy element listy. Niestety, później po każdym dodaniu elementu do listy (na przykład metodą `the_list.append(0)`) konieczne jest ponownie wywołanie metody `the_list.sort()` w celu utrzymania porządku na liście.

Inne rozwiązanie polega na zastosowaniu modułu `heapq` pochodzącego ze standardowej biblioteki Pythona:

```
import heapq
heapq.heapify(the_list)
```

Tak przekształcona lista nie musi być koniecznie w pełni posortowana, ale spełnia *właściwość sterty* (ang. *heap property*) (oznacza ona, że jeżeli wszystkie indeksy są prawidłowe, to `the_list[i]<=the_list[2*i+1]` i `the_list[i]<=the_list[2*i+2]`), przez co w szczególności element `the_list[0]` jest zawsze elementem najmniejszym. W celu utrzymania na liście właściwości sterty, należy używać metody `result=heapq.heappop(the_list)`, aby pobierać i usuwać najmniejszy element listy, natomiast nowe elementy do listy należy dodawać za pomocą metody `heapq.heappush(the_list, newitem)`. Jeżeli zajdzie potrzeba wykonania obu tych operacji, czyli dodania nowego elementu z jednoczesnym pobraniem i usunięciem najmniejszego elementu, należy skorzystać z wywołania `result=heapq.heapreplace(the_list, newitem)`.

Analiza

Jeżeli musimy odbierać dane w sposób uporządkowany (przy każdym odczycie pobieramy najmniejszy spośród dostępnych aktualnie elementów), to koszt sortowania danych musimy ponieść albo w momencie odczytywania elementu albo w momencie jego dodawania. Jedno z rozwiązań polega na gromadzeniu danych w ramach listy i sortowaniu całej listy. Dzięki temu odczytywanie danych w kolejności od najmniejszego do największego elementu jest bardzo proste. Niestety, jeżeli w czasie odczytywania danych dodajemy do listy nowe elementy, to zmuszeni jesteśmy do wywoływania metody `sort`, aby mieć pewność, że po dodaniu nowego elementu odczytywać będziemy najmniejszy element na liście. W Pythonie metoda `sort` implementowana jest z wykorzystaniem mało znanego algorytmu *Natural Mergesort*, który minimalizuje koszta sortowania w takim rozwiązaniu. Mimo to rozwiązanie to może być mocno obciążające — czas każdego dodania elementu (i sortowania listy), a także każdego odczytania (i usunięcia za pomocą metody `pop`) jest proporcjonalny do liczby elementów aktualnie znajdujących się na liście (co oznacza, że algorytm ten ma złożoność `O(N)`).

Rozwiązanie alternatywne polega na wykorzystaniu sposobu organizacji danych znanego pod nazwą *sterty* (ang. *heap*). Jest to rodzaj kompaktowego drzewa binarnego, które zapewnia, że każdy węzeł-rodzic jest mniejszy od jego węzłów-dzieci. Najlepszym sposobem na utworzenie w Pythonie struktury sterty jest wykorzystanie listy i zarządzanie nią przez moduł `heapq` z biblioteki standardowej. Taka lista nie jest sortowana dokładnie, a jedynie w takim stopniu, żeby zyskać pewność, że wywołując funkcję `heappop`, otrzymamy najmniejszy z dostępnych aktualnie elementów, a wszystkie pozostałe zostaną uporządkowane tak, aby utrzymać strukturę sterty. Każde dodanie elementu funkcją `heappush`, jak również każde usunięcie elementu funkcją `heappop` zajmuje bardzo mało czasu w stosunku do długości całej listy (w ogólnym przypadku jest to `O(log N)`). Niewielkie koszta tych operacji ponosimy w trakcie pracy, a ogólny koszt pracy z tak zarządzaną listą również nie jest znaczący.

Dobrą okazją do zastosowania sterty jest na przykład sytuacja, w której mamy długą kolejkę cyklicznie dopisywanych danych, a chcemy z niej pobierać zawsze najważniejszy w danym momencie element bez konieczności ciągłego sortowania listy lub wykonywania pełnego przeszukiwania. Takie rozwiązanie nazywane jest *kolejką priorytetową* (ang. *priority queue*), a sterta jest najlepszą metodą na jej zaimplementowanie. Trzeba jednak zauważyć, że moduł `heapq` przy każdym wywołaniu funkcji `heappop` będzie dostarczał nam zawsze *najmniejszy* element sterty, dlatego należy uwzględniać tę cechę przy ustalaniu priorytetów danych dodawanych do kolejki. Na przykład otrzymywane elementy powiązane są z określonym kosztem i w związku z tym najważniejszym elementem jest ten najdroższy spośród znajdujących się aktualnie na stercie. Co więcej, spośród elementów o identycznym koszcie najważniejszym ma być ten, który został dopisany jako pierwszy. Oto sposób na przygotowanie klasy „kolejki priorytetowej" spełniającej te założenia, która korzysta z funkcji udostępnianych przez moduł `heapq`:

```
class prioq(object):
    def __init__(self):
        self.q = []
        self.i = 0
    def push(self, item, cost):
        heapq.heappush(self.q, (-cost, self.i, item))
        self.i += 1
    def pop(self):
        return heapq.heappop(self.q)[-1]
```

Najważniejszą częścią powyższego kodu jest zapisywanie na stercie krotek, w których pierwszym elementem jest koszt właściwego elementu *ze zmienionym znakiem*, dzięki czemu elementy o *najwyższym* koszcie tworzyć będą *najmniejsze* krotki (tak porównuje je Python). Zaraz za kosztem w krotce umieszczany jest postępujący indeks dodawanych elementów, przez co wśród elementów o identycznym koszcie najmniejszym będzie ten, który został dopisany jako pierwszy.

W Pythonie 2.4 moduł `heapq` został napisany od nowa i poddany wielu optymalizacjom. Więcej informacji na jego temat podanych zostanie w recepturze 5.8.

Zobacz również

Dokumentację modułu `heapq` dostępną w podręcznikach *Library Reference* i *Python in a Nutshell*. Wśród źródeł Pythona plik *heapq.py* zawiera bardzo ciekawą dyskusję na temat stert. Recepturę 5.8, w której podawanych jest więcej informacji o module `heapq`. Recepturę 19.14, w której opisywany jest sposób łączenia posortowanych sekwencji z wykorzystaniem modułu `heapq`.

5.8. Pobieranie kilku najmniejszych elementów sekwencji

Pomysłodawcy: Matteo Dell'Amico, Raymond Hettinger, George Yoshida, Daniel Harding

Problem

Musimy pobrać kilka najmniejszych elementów sekwencji. Można oczywiście posortować sekwencję i użyć wykrojenia `seq[:n]`, ale może istnieje jakiś lepszy sposób?

Rozwiązanie

Jeżeli `n`, czyli liczba elementów wybieranych z sekwencji, jest mała w porównaniu z `L`, czyli całkowitą długością sekwencji, to problem ten da się rozwiązać lepiej. Metoda `sort` działa bardzo szybko, a mimo to zabiera `O(L log L)` czasu, natomiast pobranie z listy `n` najmniejszych elementów zajmuje `O(n)` czasu dla małych `n`. Oto prosty i bardzo praktyczny generator rozwiązujący przedstawione zadanie, działający zarówno w Pythonie 2.3, jak i w Pythonie 2.4:

```
import heapq
def isorted(data):
    data = list(data)
    heapq.heapify(data)
    while data:
        yield heapq.heappop(data)
```

W Pythonie 2.4 można skorzystać ze znacznie prostszej i szybszej metody pobierania `n` najmniejszych elementów sekwencji `data`:

```
import heapq
def smallest(n, data):
    return heapq.nsmallest(n, data)
```

Analiza

Parametr `data` może być dowolnym powiązanym elementem iterowalnym. Funkcja `isorted` podawana w rozwiązaniu receptury rozpoczyna się wywołaniem funkcji `list`, dzięki czemu warunek ten jest zawsze spełniony. Z funkcji tej instrukcję `data = list(data)` można usunąć dopiero wtedy, gdy spełnione są następujące warunki: wiemy, że parametr `data` zawsze będzie listą, nie przeszkadza nam to, że generator zmieni kolejność jej elementów, a dodatkowo chcemy usunąć z tej listy pobierane elementy.

Jak pokazaliśmy w recepturze 5.7, w bibliotece standardowej Pythona dostępny jest moduł `heapq`, który obsługuje strukturę danych znaną jako *sterta*. Generator `isorted` prezentowany w tej recepturze na początku tworzy stertę (wywołaniem funkcji `heapq.heapify`), a następnie oddaje i usuwa w każdym kroku najmniejszy element sterty (za pomocą funkcji `heapq.heappop`).

W Pythonie 2.4 moduł `heapq` uzupełniony został o dwie nowe funkcje. Funkcja `heapq.nlargest(n, data)` zwraca listę `n` największych elementów parametru `data`, natomiast funkcja `heapq.nsmallest(n, data)` zwraca listę `n` najmniejszych elementów parametru `data`. Funkcje te nie wymagają, żeby parametr `data` spełniał warunki prawidłowej sterty; co więcej, nie wymagają nawet, żeby parametr `data` był listą — całkowicie wystarczy dowolny element iterowalny z elementami pozwalającymi na porównywanie. Funkcja `smallest` prezentowana w rozwiązaniu tej receptury całość prac przekazuje zatem do funkcji `heapq.nsmallest`.

Jeżeli chcemy rozmawiać na temat prędkości działania funkcji, *zawsze* musimy ją zmierzyć — próby zgadywania względnych prędkości różnych kawałków kodu to naprawdę niebezpieczna gra. Jak w takim razie wygląda wydajność funkcji `isorted` w porównaniu z funkcją `sorted` dostępną w Pythonie 2.4, jeżeli interesuje nas tylko kilka (najmniejszych) elementów? W celu zmierzenia czasów pracy obu tych funkcji napisałem funkcję `top10`, której można użyć w połączeniu z obydwoma funkcjami. Poza tym musiałem się upewnić, że funkcja `sorted` dostępna będzie również w Pythonie 2.3, mimo że nie jest ona w tej wersji funkcją wbudowaną:

```
try:
    sorted
except:
    def sorted(data):
        data = list(data)
        data.sort()
        return data
import itertools
def top10(data, howtosort):
    return list(itertools.islice(howtosort(data), 10))
```

Na moim komputerze z Pythonem 2.4 obsłużenie listy składającej się z tysiąca dobrze przemieszanych liczb całkowitych funkcji `top10` wspomaganej przez funkcję `isorted` zajmuje 260 mikrosekund, natomiast po zmianie na współpracę z wbudowaną funkcją `sorted` ta sama operacja trwa 850 mikrosekund. Co więcej, w Pythonie 2.3 funkcje te są o wiele wolniejsze: w połączeniu z funkcją `isorted` czas testu wyniósł 12 milisekund, a z funkcją `sorted` 2,7 milisekundy. Innymi słowy, funkcja `sorted` w Pythonie 2.3 jest trzy razy wolniejsza niż w Pythonie 2.4, ale funkcja `isorted` jest *50* razy wolniejsza. Wynika z tego następująca nauka: przy okazji wprowadzania jakiejkolwiek optymalizacji należy dokonywać *pomiarów*. Nie powinno się wybierać optymalizacji na podstawie ogólnych przesłanek, ponieważ wydajność rozwiązań może różnić się znacząco nawet pomiędzy bardzo podobnymi do siebie głównymi wydaniami Pythona. Trzeba tu zaznaczyć jeszcze jedną rzecz: jeżeli komuś zależy na wydajności, powinien jak najszybciej „przesiąść się" na Pythona 2.4. Nowsza wersja została w wielu miejscach przyspieszona i zoptymalizowana w stosunku do wersji 2.3, szczególnie w zakresie wyszukiwania i sortowania.

Jeżeli mamy pewność, że nasz kod będzie działał wyłącznie w Pythonie 2.4, to tak jak pokazano w rozwiązaniu tej receptury, należy skorzystać z funkcji `nsmallest` z modułu `heapq`, ponieważ jest ona szybsza, a także prostsza od jakiegokolwiek samodzielnie przygotowanego kodu. W celu zaimplementowania funkcji `top10` w Pythonie 2.4 wystarczy tak prosty zapis:

```
import heapq
def top10(data):
    return heapq.nsmallest(10, data)
```

Podana tu wersja tę samą listę tysiąca dokładnie przemieszanych liczb całkowitych przetwarza dwa razy szybciej od prezentowanej wcześniej wersji korzystającej z funkcji `isorted`.

Zobacz również

Podręczniki *Library Reference* i *Python in a Nutshell* w częściach opisujących metodę `sort` i typ `list`, a także moduły `heapq` i `timeit`. Rozdział 19., w którym podawanych jest więcej informacji na temat iterowania w Pythonie. Podręcznik *Python in a Nutshell* w części poświęconej optymalizacji. Plik źródłowy *heapq.py*, w którym znaleźć można interesującą analizę stert. Recepturę 5.7, w której podawanych jest więcej informacji na temat modułu `heapq`.

5.9. Wyszukiwanie elementów w sekwencji posortowanej

Pomysłodawca: Noah Spurrier

Problem

W podanej sekwencji musimy odnaleźć wiele elementów.

Rozwiązanie

Jeżeli lista `L` jest posortowana, to najprostszym sposobem na sprawdzenie, czy element `x` znajduje się na liście `L` jest wykorzystanie modułu `bisect` będącego częścią standardowej biblioteki Pythona:

```
import bisect
x_insert_point = bisect.bisect_right(L, x)
x_is_present = L[x_insert_point-1:x_insert_point] == [x]
```

Analiza

W Pythonie wyszukiwanie elementu `x` na liście `L` jest wyjątkowo proste. Sprawdzenie, czy dany element jest częścią tej listy, wymaga tylko zapisania `if x in L`, natomiast uzyskanie informacji o dokładnej lokalizacji tego elementu wymaga zastosowania wywołania `L.index(x)`. Jeżeli `L` ma długość `n`, to operacje te trwają proporcjonalnie do długości `n`, co oznacza, że sprawdzają w pętli wszystkie elementy listy, porównując je z elementem `x`. Jeżeli lista `L` jest posortowana, to operacje te można wykonać zdecydowanie szybciej.

Klasyczny algorytm wyszukiwania elementu w posortowanej sekwencji znany jest pod nazwą *szukania binarnego* (ang. *binary search*). Nazwa ta wynika z tego, że w każdym kroku algorytm zmniejsza zakres poszukiwań mniej więcej o połowę, czyli w ogólnym przypadku zajmuje on $\log_2 n$ kroków. Warto o tym wiedzieć szczególnie wtedy, gdy musimy często wyszukiwać elementy w sekwencji, przez co koszt sortowania można zamortyzować w czasie wielu wyszukiwań danych. Jeżeli zdecydujemy się binarnie szukać elementu `x` na liście `L`, to po wywołaniu `L.sort()` resztę pracy bardzo ułatwi nam moduł `bisect` z biblioteki standardowej Pythona.

W szczególności przyda się nam funkcja `bisect.bisect_right`, która nie zmienia zawartości listy, ale zwraca indeks pod którym dany element *powinien* być wstawiony, aby lista pozostała posortowana. Co więcej, jeżeli szukany element znajduje się już na liście, to funkcja `bisect_right` zwraca indeks elementu znajdującego się po prawej stronie elementów o tej samej wartości. Oznacza to, że po uzyskaniu „miejsca wstawienia" elementu `x` musimy już tylko skontrolować element znajdujący się tuż *przed* tym miejscem, sprawdzając, czy element ten jest równy elementowi `x`.

Sposób wyliczenia wartości zmiennej `x_is_present`, jaki prezentowany jest w rozwiązaniu, może nie być całkiem zrozumiały. Jeżeli wiemy, że lista `L` nie jest pusta, to możemy skorzystać ze znacznie prostszego i czytelniejszego rozwiązania:

```
x_is_present = L[x_insert_point-1] == x
```

Niestety, jeżeli lista będzie pusta, to tak uproszczone indeksowanie spowoduje wyjątek. Wykrawanie działa w sposób mniej ścisły niż indeksowanie, ponieważ w przypadku nieprawidłowych granic wykrojenia tworzy tylko puste wykrojenie, ale nie wywołuje wyjątków. Mówiąc ogólnie, wyrażenie `somelist[i:i+1]` tworzy taką samą listę jednoelementową jak wyrażenie `[somelist[i]]`, pod warunkiem, że `i` jest prawidłowym indeksem na liście `somelist`. W przypadku, w którym indeksowanie powoduje wyjątek `IndexError`, wyrojenie zwraca tylko pustą listę `[]`. Przedstawiony w rozwiązaniu sposób wyliczania wartości zmiennej `x_is_present` korzysta z tej ważnej możliwości uniknięcia obsługi wyjątków i w jednakowy sposób obsługuje puste i niepuste listy `L`. Oto jeszcze inny sposób rozwiązania problemu:

```
x_is_present = L and L[x_insert_point-1] == x
```

Powyższy kod wykorzystuje zachowanie operatora `and` polegające na skróconym wyznaczaniu wyniku i w ten sposób zabezpiecza całe wyrażenie przed ewentualnymi błędami indeksowania bez konieczności używania wykrojeń.

Jeżeli elementy listy są unikalne (co oznacza, że mogą być traktowane jak klucze słownika), to pomocniczy słownik również może być bardzo ciekawym rozwiązaniem, o czym można się przekonać w recepturze 5.12. Mimo to rozwiązanie przedstawione w tej recepturze jest niezastąpione w przypadkach, w których można porównywać poszczególne elementy listy (inaczej listy nie dałoby się posortować), ale nie są one unikalne (wobec czego nie mogą być kluczami słownika).

Jeżeli lista jest już posortowana, a my musimy wyszukać w niej niezbyt dużą liczbę elementów, to zastosowanie modułu `bisect` najprawdopodobniej będzie o wiele szybsze od tworzenia pomocniczego słownika, ponieważ czas potrzebny na jego przygotowanie może zniweczyć pozostałe korzyści. Prawdopodobieństwo to wzrasta jeszcze bardziej w Pythonie 2.4, ponieważ moduł `bisect` został w nim zoptymalizowany i jest dużo szybszy niż w Pythonie 2.3. Na przykład na moim komputerze funkcja `bisect.bisect_right` wybierająca element mniej więcej ze środka listy dziesięciu tysięcy liczb całkowitych w Pythonie 2.4 jest niemal czterokrotnie szybsza niż w Pythonie 2.3.

Zobacz również

Dokumentację modułu `bisect` w podręcznikach *Library Reference* i *Python in a Nutshell*. Recepturę 5.12.

5.10. Wybieranie n-tego najmniejszego elementu w sekwencji

Pomysłodawcy: Raymond Hettinger, David Eppstein, Shane Holloway, Chris Perkins

Problem

Musimy wybrać z sekwencji n-ty element pod względem wielkości (na przykład element środkowy zwany *medianą*). Jeżeli lista byłaby posortowana, to można byłoby użyć zapisu `seq[n]`, ale nasza sekwencja *nie jest* posortowana, wiec zastanawiamy się, czy jest lepszy sposób od jej posortowania.

Rozwiązanie

Oczywiście jest lepsze rozwiązanie, sprawdzające się w sytuacji, gdy sekwencja jest duża, jej elementy są mocno przemieszane, a porównanie tych elementów jest kosztowną operacją. Sortowanie jest bardzo szybkie, ale dla dobrze przemieszanych sekwencji o *n* elementach i tak zajmuje ono `O(n log n)` czasu, natomiast dostępne są algorytmy pozwalające na odszukanie

n-tego najmniejszego elementu w czasie O(*n*). Oto funkcja będąca solidną implementacją takiego algorytmu:

```
import random
def select(data, n):
    "  Wyszukuje n-ty element pod względem wielkości.  "
    #  tworzy nową listę, sprawdza indeksy <0, szuka prawidłowych indeksów
    data = list(data)
    if n<0:
        n += len(data)
    if not 0 <= n < len(data):
        raise ValueError, "nie mogę pobrać elementu %d spośród %d" % (n, len(data))
    #  pętla główna, podobna do algorytmu quicksort, ale nie potrzebuje rekursji
    while True:
        pivot = random.choice(data)
        pcount = 0
        under, over = [], []
        uappend, oappend = under.append, over.append
        for elem in data:
            if elem < pivot:
                uappend(elem)
            elif elem > pivot:
                oappend(elem)
            else:
                pcount += 1
        numunder = len(under)
        if n < numunder:
            data = under
        elif n < numunder + pcount:
            return pivot
        else:
            data = over
            n -= numunder + pcount
```

Analiza

W prezentowanej recepturze chodzi nam o przypadki, w których *ważne są* powtórzenia. Na przykład mediana z listy `[1, 1, 1, 2, 3]` wynosi `1`, ponieważ jest to trzeci element z pięciu w kolejności rosnącej. Jeżeli z jakiegoś dziwnego powodu chcielibyśmy nie uwzględniać powtórzeń, to taką listę trzeba by najpierw zredukować do jej elementów unikalnych (na przykład stosując rozwiązania podawane w recepturze 18.1), a dopiero potem wrócić do kodu podawanego w tej recepturze.

Wejściowy parametr `data` może być dowolnym elementem iterowalnym. Kod tej receptury rozpoczyna się od wywołania na wszelki wypadek funkcji `list`. Następnie algorytm wchodzi w pętlę, w której przy każdym kroku implementuje kilka ważnych operacji: losowo wybiera *element rozdzielający* (ang. *pivot element*), dzieli listę na dwie części składające się odpowiednio z elementów „poniżej” i „ponad” elementem rozdzielającym, w następnym kroku kontynuuje pracę w jednej z części listy, ponieważ na tym etapie wiemy już, w której części znajdzie się szukany `n`-ty element. Pomysł jest bardzo zbliżony do klasycznego algorytmu znanego pod nazwą *quicksort* (różnica polega na tym, że algorytm quicksort musi obsłużyć obie części listy i w związku z tym zmuszony jest do korzystania z rekursji lub metod usuwania rekursji takich jak utrzymywanie własnego stosu zapewniającego obsługę całości listy).

Losowe wybranie elementu rozdzielającego sprawia, że algorytm lepiej sprawdza się w przypadkach niekorzystnego ułożenia danych (takich, które sieją spustoszenie w naiwnych implementacjach algorytmu quicksort). Podana implementacja mniej więcej $\log_2 N$ razy wywołuje funkcję `random.choice`. Inną wartą wymienienia cechą implementacji prezentowanej w rozwiązaniu tej receptury jest zliczanie liczby wystąpień elementu rozdzielającego. Takie działanie zapewnia dobrą wydajność nawet w niezwykłych przypadkach, w których parametr `data` zawiera wiele powtórzeń poszczególnych wartości.

Wydobywanie z list `under` oraz `over` powiązanych metod `.append` i przypisywanie ich do lokalnych zmiennych `uappend` i `oappend` na pierwszy rzut oka może wydawać się niecelowe, a na dodatek powodujące pewne komplikacje, ale w rzeczywistości jest to jedna z najważniejszych metod optymalizacji w Pythonie. W celu utrzymania prostej, solidnej i pozbawionej niespodzianek struktury kompilatora Python *nie przenosi* stałych wyliczeń poza pętle, tak jak i nie „buforuje" wyników poszukiwania metod. Jeżeli wywołujemy metody `under.append` i `over.append` w ramach pętli, to przy każdej iteracji musimy ponieść koszt wyszukania wywoływanej metody. Jeżeli chcemy, żeby coś było przechowywane, to sami musimy to przechować. Jeżeli zastanawiamy się nad pewną optymalizacją, to zawsze powinniśmy zmierzyć wydajność kodu *bez* tej optymalizacji i *z nią*. Tylko w ten sposób można ocenić, czy wprowadzona optymalizacja rzeczywiście wprowadza zauważalną różnicę. Zgodnie z moimi pomiarami usunięcie tej jednej optymalizacji powoduje mniej więcej 50% spadek wydajności w typowym zadaniu wybierania pięciotysięcznego elementu z zakresu `range(10000)`. Uwzględniając tę niewielką komplikację, jaką wprowadza stosowany zapis, z całą pewnością jest on wart dwukrotnego przyspieszenia działania tej funkcji.

Dość naturalnym pomysłem na optymalizację, z którego zrezygnowałem dopiero po wykonaniu dokładnych pomiarów, jest wywołanie w ciele pętli funkcji `cmp(elem, pivot)`, które miałoby zastąpić osobne porównania `elem < pivot` i `elem > pivot`. Niestety, pomiary wykazały, że funkcja `cmp` nie przyspiesza działania pętli, ale spowalnia ją, przynajmniej w przypadkach, gdy elementami sekwencji `data` są typy podstawowe, takie jak liczby lub ciągi znaków.

Jak w takim razie wygląda wydajność funkcji `select` w porównaniu ze znacznie prostszą funkcją przedstawioną poniżej?

```
def selsor(data, n):
    data = list(data)
    data.sort()
    return data[n]
```

Na moim komputerze wybranie mediany z dobrze przemieszanej listy składającej się z 3001 liczb całkowitych, zajmuje funkcji `select` mniej więcej 16 milisekund, podczas gdy funkcja `selsor` tę samą operację przeprowadza w 13 milisekund. Biorąc pod uwagę fakt, że metoda `sort` może wykorzystywać dowolną posortowaną już część danych, to w przypadku list podobnej długości składających się z łatwo porównywalnych typów podstawowych wykorzystanie funkcji `select` nie jest najlepszym wyborem. W przypadku list o długości 30 001 elementów wydajność obu rozwiązań jest bardzo podobna i wynosi około 170 milisekund. Dopiero w momencie, gdy zaczniemy pracować z listami o wielkości zbliżonej do 300 001 elementów, funkcja `select` zaczyna przeważać na funkcją `selsor` — czasy ich pracy wynoszą odpowiednio 2,2 sekundy i 2,5 sekundy.

Punkt zrównania wydajności porównywanych funkcji będzie znacznie niższy, jeżeli sekwencje składać się będą z elementów o bardzo złożonych i kosztownych porównaniach, ponieważ główna różnica między tymi rozwiązaniami polega na liczbie wykonywanych porównań

— funkcja `select` wykonuje `O(n)` porównań, a funkcja `selsor` wykonuje ich `O(n log n)`. Załóżmy na przykład, że musimy porównać egzemplarze klasy, w której operacje porównania są dość kosztowne (symuluje ona punkt czterowymiarowy, w którym kilka pierwszych wymiarów może się pokrywać):

```
class X(object):
    def __init__(self):
        self.a = self.b = self.c = 23.51
        self.d = random.random()
    def _dats(self):
        return self.a, self.b, self.c, self.d
    def __cmp__(self, oth):
        return cmp(self._dats, oth._dats)
```

W takiej sytuacji funkcja `select` zaczyna działać szybciej od funkcji `selsor` już w momencie wyznaczania mediany z wektora składającego się z 201 takich egzemplarzy.

Innymi słowy, co prawda funkcja `select` wykonuje więcej ogólnych operacji nadmiarowych w porównaniu z niezwykle efektywnym sposobem działania metody `sort`, to jednak w przypadku, gdy `n` jest odpowiednio duże, a każde porównanie jest wystarczająco kosztowne — zastosowanie funkcji `select` jest warte rozważenia.

Zobacz również

Podręczniki *Library Reference* i *Python in a Nutshell* w częściach opisujących metodę `sort`, typ `list` oraz moduł `random`.

5.11. Algorytm quicksort w trzech wierszach kodu

Pomysłodawcy: Nathaniel Gray, Raymond Hettinger, Christophe Delord, Jeremy Zucker

Problem

Musimy wykazać, że obsługa paradygmatu programowania funkcyjnego w Pythonie jest lepsza niż można się tego spodziewać na pierwszy rzut oka.

Rozwiązanie

Języki programowania funkcyjnego, wśród których prym wiedzie język Haskell, to wyjątkowo udane konstrukcje, ale nawet w takim towarzystwie Python prezentuje się zadziwiająco dobrze:

```
def qsort(L):
    if len(L) <= 1: return L
    return qsort([lt for lt in L[1:] if lt < L[0]]) + L[0:1] + \
           qsort([ge for ge in L[1:] if ge >= L[0]])
```

Moim skromnym zdaniem podany kod jest niemal tak piękny jak wersja zapisana w języku Haskell, pobrana ze strony *http://www.haskell.org*:

```
qsort [] = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
                where
                  elts_lt_x = [y | y <- xs, y < x]
                  elts_greq_x = [y | y <- xs, y >= x]
```

Oto funkcja testująca wersję przygotowaną w Pythonie:

```
def qs_test(length):
    import random
    joe = range(length)
    random.shuffle(joe)
    qsJoe = qsort(joe)
    for i in range(len(qsJoe)):
        assert qsJoe[i] == i, 'qsort przestał działać na pozycji %d!' %i
```

Analiza

Ta raczej naiwna implementacja algorytmu quicksort doskonale ilustruje siłę list składanych. Takiego rozwiązania nie należy jednak stosować w kodzie produkcyjnym! W Pythonie listy uzupełniane są o metodę sort, która działa o wiele szybciej od prezentowanej w tej recepturze. W Pythonie 2.4 nowa wbudowana funkcja `sorted` działa z dowolną skończoną sekwencją i zwraca nową, posortowaną listę elementów tej sekwencji. Jedynym właściwym zastosowaniem kodu z tej receptury jest pokazywanie go znajomym programistom, szczególnie tym, którzy (co zrozumiałe) bardzo entuzjastycznie traktują języki funkcyjne, a przede wszystkim język Haskell.

Podaną funkcję przygotowałem po znalezieniu na stronie *http://www.haskell.org/aboutHaskell.html* wspaniałej implementacji algorytmu quicksort w języku Haskell (podałem ją również w rozwiązaniu tej receptury). Po okresie podziwiania elegancji znalezionego kodu uświadomiłem sobie, że dokładnie takie samo rozwiązanie możliwe jest w Pythonie przy zastosowaniu list składanych. Nie na darmo zostały one „pożyczone" z języka Haskell i lekko „spythonizowane", tak żeby możliwe było wykorzystanie w nich słów kluczowych, a nie tylko operatorów.

Obie implementacje dzielą listę na jej pierwszym elemencie i dlatego ich wydajność w najgorszym przypadku, czyli w bardzo powszechnym przypadku sortowania posortowanej listy, wynosi `O(n)`. Z całą pewnością nie chcielibyśmy stosować takiego rozwiązania w kodzie produkcyjnym! Omawiana procedura jest jednak czystą „propagandówką", więc takie szczegóły nie mają znaczenia.

Można też zapisać mniej kompaktową wersję o podobnej architekturze, stosując w niej nazywane zmienne lokalne oraz funkcje poprawiające czytelność kodu:

```
def qsort(L):
    if not L: return L
    pivot = L[0]
    def lt(x): return x<pivot
    def ge(x): return x>=pivot
    return qsort(filter(lt, L[1:]))+[pivot]+qsort(filter(ge, L[1:]))
```

Skoro weszliśmy już na tę ścieżkę, to bardzo łatwo możemy podaną wersję przekształcić w wersję nieco mniej naiwną, wykorzystującą losowe wybieranie elementu rozdzielającego, przez co zmniejsza się prawdopodobieństwo wystąpienia najgorszego przypadku, oraz zliczającą elementy rozdzielające, przez co poprawia się obsługa przypadków z wieloma jednakowymi elementami:

```
import random
def qsort(L):
    if not L: return L
    pivot = random.choice(L)
    def lt(x): return x<pivot
    def gt(x): return x>pivot
    return qsort(filter(lt, L))+[pivot]*L.count(pivot)+qsort(filter(gt, L))
```

Mimo takich modyfikacji wersja ta również przeznaczona jest głównie do zabawy i celów demonstracyjnych. Porządny kod sortujący musi wyglądać zupełnie inaczej: te perełki, nad którymi się tak rozwodzimy, nigdy nie osiągną wydajności i skuteczności rozwiązań sortowania wbudowanych w Pythona.

Zamiast próbować poprawiać czytelność kodu, możemy zacząć działać w przeciwnym kierunku i próbować tworzyć przede wszystkim skuteczne rozwiązanie, pokazując przy okazji, że w Pythonie słowo kluczowe `lambda` pozwala na uzyskanie bardzo zwartego, ale i dziwnego zapisu:

```
q=lambda x:(lambda o=lambda s:[i for i in x if cmp(i,x[0])==s]:
            len(x)>1 and q(o(-1))+o(0)+q(o(1)) or x)()
```

W przypadku *tego* „potworka" (pojedynczy wiersz kodu, który z powodu swojej długości musi zostać podzielony na dwa wiersze) widać wyraźnie, że takie rozwiązania nie powinny być stosowane w rzeczywistych programach. Nawet bardziej czytelna, równoważna wersja stosująca instrukcje `def` zamiast instrukcji `lambda` będzie nie do końca zrozumiała:

```
def q(x):
    def o(s): return [i for i in x if cmp(i,x[0])==s]
    return len(x)>1 and q(o(-1))+o(0)+q(o(1)) or x
```

Nieco czytelniejszy kod można utworzyć, rozbijając bardzo zbitą instrukcję `len(x)>1 and ... or x` na instrukcje `if/else` i wprowadzając odpowiednie nazwy zmiennych:

```
def q(x):
    if len(x)>1:
        lt = [i for i in x if cmp(i,x[0]) == -1 ]
        eq = [i for i in x if cmp(i,x[0]) == 0 ]
        gt = [i for i in x if cmp(i,x[0]) == 1 ]
        return q(lt) + eq + q(gt)
    else:
        return x
```

Na szczęście prawdziwi Pythonianie są zbyt wrażliwi, żeby znosić i tworzyć takie „potworki" wypełnione instrukcjami `lambda`, jak te prezentowane w tej recepturze. W rzeczywistości wielu z nas (choć,oczywiście nie wszyscy) czuje awersję do tej instrukcji (częściowo z powodu możliwości jej nadużywania) i zdecydowanie woli stosować czytelniejsze konstrukcje z instrukcjami `def`. W efekcie umiejętność odczytywania takich „zaśmieconych wierszy" *nie* jest wymagana w świecie Pythona, jak to bywa w innych językach programowania. W podobny sposób programista próbujący zapisać „sprytny" kod może nadużyć *dowolnej* funkcji języka, dlatego część Pythonian (zdecydowana mniejszość) podobną awersją darzy inne funkcje języka, takie jak listy składane (ponieważ w wyrażeniu listy składanej można umieścić zbyt wiele niepotrzebnych elementów, podczas gdy prosta pętla `for` byłaby zdecydowanie czytelniejsza) lub wykorzystywanie warunkowości operatorów logicznych `and` i `or` (ponieważ można w ten sposób tworzyć nieczytelne wyrażenia, które dają się zastąpić znacznie czytelniejszą instrukcją `if`).

Zobacz również

Stronę języka Haskell *http://www.haskell.org*.

5.12. Wykonywanie częstych testów obecności elementów sekwencji

Pomysłodawca: Alex Martelli

Problem

Musimy wykonywać częste testy na obecność danego elementu w sekwencji. Wydajność `O(n)` często wywoływanego operatora `in` może bardzo negatywnie wpłynąć na pracę programu, ale nie możemy po prostu *zastąpić* sekwencji słownikiem lub zbiorem, ponieważ ważna jest też kolejność jej elementów.

Rozwiązanie

Załóżmy, że musimy dodawać do sekwencji elementy, ale tylko wtedy, gdy nie zostały one dodane do niej wcześniej. Poniższa funkcja jest doskonałym rozwiązaniem tego zadania:

```
def addUnique(baseList, otherList):
    auxDict = dict.fromkeys(baseList)
    for item in otherList:
        if item not in auxDict:
            baseList.append(item)
            auxDict[item] = None
```

Jeżeli nasz kod ma działać wyłącznie w Pythonie 2.4, to dokładnie takie same efekty uzyskamy za pomocą dodatkowego zbioru, a nie słownika.

Analiza

Najprostsze (naiwne?) rozwiązanie problemu z tej receptury *wygląda* całkiem dobrze:

```
def addUnique_simple(baseList, otherList):
    for item in otherList:
        if item not in baseList:
            baseList.append(item)
```

i nawet może sprawować się nieźle *pod warunkiem*, że listy będą bardzo krótkie.

Niestety, tak proste rozwiązanie okaże się bardzo powolne przy pracach z długimi listami. Jeżeli zastosujemy instrukcję `if item not in baseList`, to Python w tylko jeden sposób może zaimplementować operator `in` — za pomocą wewnętrznej pętli iterującej we wszystkich elementach listy `baseList`, która kończy się z wynikiem `True`, gdy tylko znaleziony zostanie element listy identyczny z elementem `item` albo wynikiem `False`, jeżeli żaden z elementów listy nie okazał się identyczny z elementem `item`. W typowych przypadkach zastosowanie operatora `in` zajmuje czas proporcjonalny do liczby elementów listy `baseList`. Funkcja `addUnique_simple` operator `in` wywołuje `len(otherList)` razy, więc w sumie czas jej działania jest proporcjonalny do *iloczynu* długości obu tych list.

W funkcji `addUnique` przedstawionej w rozwiązaniu tej receptury tworzymy najpierw pomocniczy słownik `auxDict` — krok ten zajmuje czas proporcjonalny do długości listy `baseList`. Następnie operator `in` umieszczony w pętli sprawdza, czy zadany element jest składnikiem

słownika. To właśnie ten krok stanowi główną różnicę między obiema funkcjami, ponieważ operator `in` zastosowany wobec słownika działa w czasie stałym, niezależnie od liczby elementów w słowniku. Oznacza to, że pętla `for` działać będzie w czasie proporcjonalnym do długości listy `otherList`, a cała funkcja działa w czasie proporcjonalnym do *sumy* długości obu list.

Taką analizę czasów działania funkcji powinniśmy przeprowadzić nieco głębiej, ponieważ w funkcji `addUnique_simple` długość listy `baseList` nie jest stała. Lista `baseList` rośnie za każdym razem, gdy badany element nie jest częścią tej listy. Wynik takiej (zaskakująco złożonej) analizy nie różniłby się jednak wcale od wyników analizy uproszczonej. Jeżeli każda z list składałaby się z 10 liczb całkowitych, z których tylko 50% byłoby ze sobą zgodnych, to prostsza wersja funkcji byłaby o mniej więcej 30% wolniejsza od funkcji prezentowanej w rozwiązaniu — taki spadek wydajności można z czystym sumieniem zignorować. Niestety, już w przypadku list zawierających sto elementów pokrywających się w 50%, prostsza wersja jest aż *dwanaście razy wolniejsza* od wersji „oficjalnej". Takich wyników na pewno nie można zignorować, tym bardziej, że różnica ta powiększa się wraz ze wzrostem wielkości list.

Czasami można uzyskać jeszcze lepszą wydajność całego programu, stosując pomocniczy słownik równolegle z samą sekwencją i zamykając je w ramach jednego obiektu. W takim jednak przypadku musimy dbać o aktualizowanie zawartości słownika w czasie modyfikowania sekwencji, zapewniając tym samym jego synchronizację z sekwencją. Zadanie takiego synchronizowania słownika z całą pewnością nie jest trywialne i może być zrealizowane na wiele różnych sposobów. Oto jeden ze sposobów synchronizowania słownika „w razie potrzeby", czyli tylko w momencie testowania obecności elementu w sekwencji, gdy istnieje prawdopodobieństwo, że słownik nie jest zsynchronizowany z sekwencją. Koszt takiej operacji jest niewielki, dlatego podana niżej klasa optymalizuje metodę `index`, jak również testy na obecność elementu:

```
class list_with_aux_dict(list):
    def __init__(self, iterable=()):
        list.__init__(self, iterable)
        self._dict_ok = False
    def _rebuild_dict(self):
        self._dict = {}
        for i, item in enumerate(self):
            if item not in self._dict:
                self._dict[item] = i
        self._dict_ok = True
    def __contains__(self, item):
        if not self._dict_ok:
            self._rebuild_dict()
        return item in self._dict
    def index(self, item):
        if not self._dict_ok:
            self._rebuild_dict()
        try: return self._dict[item]
        except KeyError: raise ValueError
def _wrapMutatorMethod(methname):
    _method = getattr(list, methname)
    def wrapper(self, *args):
        # Kasowanie znacznika 'słownik OK' i delegowanie prawdziwej metody modyfikującej
        self._dict_ok = False
        return _method(self, *args)
    # tylko w Pythonie 2.4: wrapper.__name__ = _method.__name__
    setattr(list_with_aux_dict, methname, wrapper)
for meth in 'setitem delitem setslice delslice iadd'.split():
    _wrapMutatorMethod('__%s__' % meth)
for meth in 'append insert pop remove extend'.split():
    _wrapMutatorMethod(meth)
del _wrapMutatorMethod     # usuwamy funkcję pomocniczą, nie będzie już nam potrzebna
```

Klasa `list_with_aux_dict` rozbudowuje klasę `list` i tworzy delegację wszystkich jej metod za wyjątkiem metod `__contains__` i `index`. Każda z metod, która może zmodyfikować zawartość listy, opakowywana jest domknięciem kasującym znacznik zgodności słownika z listą. W Pythonie operator `in` wywołuje w obiekcie metodę `__contains__`. Metoda ta w klasie `list_with_aux_dict` powoduje przebudowanie pomocniczego słownika, chyba że znacznik zgodności jest ustawiony (bo wtedy przebudowa słownika nie jest już konieczna). W podobny sposób działa też metoda `index`.

Zamiast budowania i instalowania domknięć opakowujących wszystkie metody modyfikujące zawartość listy za pomocą funkcji pomocniczej klasy `list_with_aux_dict`, tak jak zrobiono to w podanym kodzie, można też napisać osobne opakowanie dla każdej z metod, korzystając przy tym z instrukcji `def`. Mimo to kod zaprezentowanej klasy ma nad takim rozwiązaniem niezaprzeczalną przewagę, jako że minimalizuje potrzebę stosowania powtarzającego się, nudnego kodu o znacznej objętości, w którym bardzo często ukrywają się błędy. Możliwości, jakie Python daje nam w zakresie introspekcji i dynamicznej modyfikacji, pozwalają nam zadecydować: możemy budować opakowania metod tak jak zostało to zrobione w prezentowanej klasie, czyli spójnie i sprytnie, ale jeżeli nie opanowaliśmy jeszcze „czarnej magii" introspekcji i dynamicznej modyfikacji obiektów, to równie dobrze może zdecydować się na tworzenie powtarzającego się kodu.

Architektura klasy `list_with_aux_dict` jest doskonałym przykładem bardzo powszechnego wzorca użycia, stosowanego w sytuacjach, gdy operacje modyfikujące sekwencje zdarzają się w „paczkach". Pomiędzy takimi „paczkami modyfikacji" następują okresy, w których sekwencja nie jest poddawana żadnym modyfikacjom, ale wykonywane są testy na obecność elementów. Niestety, prezentowana wcześniej funkcja `addUnique_simple` nie zyskałaby na prędkości, jeżeli w parametrze `baseList` zamiast zwykłego obiektu `list` otrzymałaby egzemplarz klasy `list_with_aux_dict`, ponieważ funkcja ta przeplata ze sobą testy na obecność elementu i modyfikacje zawartości listy. W takich warunkach zbyt wiele operacji przebudowywania pomocniczego słownika bardzo źle wpływałoby na prędkość działania funkcji (chyba że w znakomitej większości przypadków elementy listy `otherList` są już częścią listy `baseList`, a zatem modyfikacji listy będzie o wiele mniej niż operacji sprawdzania obecności).

Bardzo ważną częścią wszystkich takich optymalizacji testów obecności jest wymóg, żeby elementy sekwencji były unikalne (jeżeli tak nie będzie, to nie mogą one być oczywiście kluczami słownika, ani elementami zbioru). Podane w tej recepturze funkcje mogłyby być na przykład wykorzystane do obsługi listy krotek, ale zupełnie nie nadawałyby się do obsługi listy list.

Zobacz również

Podręczniki *Library Reference* i *Python in a Nutshell* w częściach opisujących typy sekwencji i typy odwzorowań.

5.13. Wyszukiwanie podsekwencji

Pomysłodawcy: David Eppstein, Alexander Semenov

Problem

Musimy znaleźć wystąpienia pewnej podsekwencji w ramach większej sekwencji.

Rozwiązanie

Jeżeli sekwencjami są ciągi znaków (proste lub Unikodu), to zdecydowanie najlepszym wyjściem jest metoda `find` oraz standardowy moduł `re`. W przypadku innych sekwencji należy posłużyć się algorytmem Knutha-Morrisa-Pratta (KMP):

```
def KnuthMorrisPratt(text, pattern):
    ''' Zwraca wszystkie pozycje początków kopii podsekwencji 'pattern'
        w ramach sekwencji 'text' -- każdy z parametrów może być dowolnym
        elementem iterowalnym. Przy każdym zwróceniu elementu parametr 'text'
        zostaje odczytany do samego końca znalezionej podsekwencji 'pattern'. '''
    # musimy zapewnić sobie możliwość indeksowania w parametrze pattern,
    # a jednocześnie utworzyć jego kopię na wypadek wprowadzenia do niego zmian
    # w czasie, gdy funkcja jest wstrzymana przez instrukcję `yield'
    pattern = list(pattern)
    length = len(pattern)
    # budujemy "tablicę wartości przesunięć" i nazywamy ją 'shifts'
    shifts = [1] * (length + 1)
    shift = 1
    for pos, pat in enumerate(pattern):
        while shift <= pos and pat != pattern[pos-shift]:
            shift += shifts[pos-shift]
        shifts[pos+1] = shift
    # wykonanie właściwego wyszukiwania
    startPos = 0
    matchLen = 0
    for c in text:
        while matchLen == length or matchLen >= 0 and pattern[matchLen] != c:
            startPos += shifts[matchLen]
            matchLen -= shifts[matchLen]
        matchLen += 1
        if matchLen == length: yield startPos
```

Analiza

W niniejszej recepturze implementujemy algorytm Knutha-Morrisa-Pratta przeznaczony do wyszukiwania kopii danego wzorca w ramach ciągłej podsekwencji większego tekstu. Algorytm KMP korzysta z tekstu w sposób sekwencyjny, dlatego bardzo naturalnym rozwiązaniem jest zezwolenie na stosowanie tekstu w postaci dowolnego elementu iterowalnego. Po zakończeniu fazy przygotowań wstępnych, w której budowana jest tabela wartości przesunięć i która zajmuje czas proporcjonalny do długości szukanego wzorca, każdy z symboli przetwarzany jest w czasie stałym. Wyjaśnienia dotyczące pracy algorytmu KMP można znaleźć w dowolnej dobrej książce opisującej algorytmy (rekomendacje podajemy w punkcie „Zobacz również”).

Jeżeli parametry `text` i `pattern` są ciągami znaków, to można zastosować zdecydowanie szybsze rozwiązanie, wykorzystując przy tym metody wbudowane Pythona:

```
def finditer(text, pattern):
    pos = -1
    while True:
        pos = text.find(pattern, pos+1)
        if pos < 0: break
        yield pos
```

Na przykład, korzystając z alfabetu o długości 4 (`'ACGU'`), odnalezienie wszystkich wystąpień danego wzorca o długości 8 w ramach tekstu o długości 100 000 na moim komputerze zajmuje funkcji `finditer` mniej więcej 4,3 milisekundy, natomiast funkcja `KnuthMorrisPratt` to samo

zadanie realizuje w ciągu 540 milisekund (te wyniki dotyczą Pythona 2.3, w Pythonie 2.4 algorytm KMP działa nieco szybciej, a wspomniane zadanie realizowane jest w ciągu 480 milisekund, ale mimo to jest to wynik ponad stukrotnie gorszy od wyniku funkcji `finditer`). Należy zatem pamiętać: kod podany w tej recepturze nadaje się do przeszukiwania *dowolnych* sekwencji, włącznie z tymi, których nie da się w całości przechowywać w pamięci, ale przy przeszukiwaniu ciągów znaków zdecydowanie lepiej sprawdzają się metody wbudowane Pythona.

Zobacz również

Podstawy algorytmów, które są opisywane w wielu doskonałych książkach. Wśród nich jedną z najbardziej polecanych jest pozycja Thomasa H. Cormena, Charlesa E. Leisersona, Ronalda L Rivesta i Clifforda Steina — *Introduction to Algorithms*, wydanie drugie (MIT Press).

5.14. Wzbogacanie typu dict o możliwość wprowadzania ocen

Pomysłodawcy: Dmitry Vasiliev, Alex Martelli

Problem

Chcemy użyć słownika do przechowywania odwzorowań kluczy i aktualnych wartości ocen tych kluczy. Niejednokrotnie musimy uzyskać dostęp do kluczy i ocen w kolejności naturalnej (co oznacza malejące wartości ocen) albo sprawdzać aktualną pozycję danego klucza w takim rankingu. To wszystko sugeruje, że samo zastosowanie słownika nie jest wystarczające.

Rozwiązanie

Możemy utworzyć klasę wywiedzioną z klasy `dict` i dodać do niej lub pokryć potrzebne nam metody. W ramach dziedziczenia wielobazowego możemy jako pierwszą klasę *bazową* oznaczyć klasę `UserDict.DictMixin` i dopiero za nią dopisać klasę `dict`, a w nowej klasie można ostrożnie przygotować różne delegacje i pokrywania metod. W ten sposób uzyskamy równowagę pomiędzy niezłą wydajnością klasy a koniecznością tworzenia powtarzalnego kodu.

Wzbogacając naszą klasę o wiele przykładów zapisanych w jej dokumentacji, możemy skorzystać też z modułu biblioteki standardowej `doctest`, przez co klasa zostanie uzupełniona o funkcje testów modułowych, a my uzyskamy pewność, że przykłady podane w dokumentacji będą zgodne z prawdą:

```
#!/usr/bin/env python
''' Wzbogacony słownik przechowujący klucze powiązane z ich ocenami '''
from bisect import bisect_left, insort_left
import UserDict
class Ratings(UserDict.DictMixin, dict):
    """ klasa Ratings jest bardzo zbliżona do słownika uzupełnionego
        o kilka dodatkowych funkcji: Wartość powiązana z każdym kluczem
        jest traktowana jak jego 'ocena', a wszystkie klucze układane są według
        tych ocen. Wartości muszą być porównywalne, natomiast klucze, oprócz
        tego, że muszą być unikalne, muszą też być porównywalne na wypadek,
```

gdyby przechowywały takie same wartości (czyli miały tę samą ocenę). Wszystkie zachowania związane z odwzorowywaniem są dokładnie takie, jakich się spodziewamy, na przykład:

```
>>> r = Ratings({"piotr": 30, "paweł": 30})
>>> len(r)
2
>>> r.has_key("rafał"), "rafał" in r
(False, False)
>>> r["paweł"] = 20
>>> r.update({"wojtek": 20, "tomek": 10})
>>> len(r)
4
>>> r.has_key("paweł"), "paweł" in r
(True, True)
>>> [r[key] for key in ["piotr", "paweł", "wojtek", "tomek"]]
[30, 20, 20, 10]
>>> r.get("nikt"), r.get("nikt", 0)
(None, 0)
```

Oprócz interfejsu słownikowego klasa udostępnia metody związane z ocenami. Metoda r.rating(key) zwraca pozycję danego klucza w skali ocen, przy czym pozycja 0 oznacza ocenę najniższą (jeżeli dwa klucze mają takie same oceny, to porównywane są bezpośrednio i mniejszy z nich otrzymuje niższą pozycję):

```
>>> [r.rating(key) for key in ["piotr", "wojtek", "paweł", "tomek"]]
[3, 2, 1, 0]
```

Metody getValueByRating(ranking) i getKeyByRating(ranking) zwracają ocenę lub klucz odpowiedniej pozycji w rankingu:

```
>>> [r.getValueByRating(rating) for rating in range(4)]
[10, 20, 20, 30]
>>> [r.getKeyByRating(rating) for rating in range(4)]
['piotr', 'wojtek', 'paweł', 'tomek']
```

Metoda keys() zwraca klucze w kolejności malejących pozycji, a wszystkie pozostałe metody zwracają listy lub iteratory w pełni zgodne z tą właśnie kolejnością:

```
>>> r.keys()
['piotr', 'wojtek', 'paweł', 'tomek']
>>> [key for key in r]
['piotr', 'wojtek', 'paweł', 'tomek']
>>> [key for key in r.iterkeys()]
['piotr', 'wojtek', 'paweł', 'tomek']
>>> r.values()
[10, 20, 20, 30]
>>> [value for value in r.itervalues()]
[10, 20, 20, 30]
>>> r.items()
[('tomek', 10), ('paweł', 20), ('wojtek', 20), ('piotr', 30)]
>>> [item for item in r.iteritems()]
[('tomek', 10), ('paweł', 20), ('wojtek', 20), ('piotr', 30)]
```

Egzemplarz klasy może być dowolnie zmodyfikowany (można dodawać, zmieniać i usuwać pary klucz-ocena), a każda metoda tego egzemplarza zawsze będzie odzwierciedlać jego aktualny stan:

```
>>> r["tomek"] = 100
>>> r.items()
[('wojtek', 20), ('paweł', 20), ('piotr', 30), ('tomek', 100)]
>>> del r["paweł"]
>>> r.items()
[('wojtek', 20), ('piotr', 30), ('tomek', 100)]
>>> r["paul"] = 25
>>> r.items()
[('wojtek', 20), ('paweł', 25), ('piotr', 30), ('tomek', 100)]
>>> r.clear()
>>> r.items()
[]
"""
```

```
    ''' Implementacja klasy miesza ze sobą dziedziczenie z delegacjami w celu osiągnięcia
        zadowalającej wydajności, minimalizacji powtarzalnego kodu i oczywiście
        uzyskania poprawnej semantyki Wszystkie niezaimplementowane metody tego
        odwzorowania są dziedziczone, przede wszystkim po klasie DictMixin, ale najważniejsze
        jest to, że metoda __getitem__ dziedziczona jest po klasie dict. '''
    def __init__(self, *args, **kwds):
        ''' Egzemplarze tej klasy tworzone są tak jak egzemplarze klasy dict '''
        dict.__init__(self, *args, **kwds)
        # self._rating jest bardzo ważną, pomocniczą strukturą danych: lista
        # wszystkich par (wartość, klucz), utrzymywana w kolejności naturalnego
        # posortowania
        self._rating = [ (v, k) for k, v in dict.iteritems(self) ]
        self._rating.sort()
    def copy(self):
        ''' Tworzy identyczną, ale niezależną kopię '''
        return Ratings(self)
    def __setitem__(self, k, v):
        ''' oprócz delegowania do klasy dict zarządzamy strukturą self._rating '''
        if k in self:
            del self._rating[self.rating(k)]
        dict.__setitem__(self, k, v)
        insort_left(self._rating, (v, k))
    def __delitem__(self, k):
        ''' oprócz delegowania do klasy dict zarządzamy strukturą self._rating '''
        del self._rating[self.rating(k)]
        dict.__delitem__(self, k)
    ''' jawnie delegujemy pewne metody do klasy dict, aby uniknąć
        stosowania wolniejszej implementacji z klasy DictMixin '''
    __len__ = dict.__len__
    __contains__ = dict.__contains__
    has_key = __contains__
    ''' Główne połączenie semantyczne pomiędzy self._rating i pozycją
        w self.keys() -- DictMixin daje nam 'za darmo' wszystkie potrzebne metody, mimo że moglibyśmy
        zaimplementować je bezpośrednio z nieco lepszą wydajnością. '''
    def __iter__(self):
        for v, k in self._rating:
            yield k
    iterkeys = __iter__
    def keys(self):
        return list(self)
    ''' trzy metody związane z ocenami '''
    def rating(self, key):
        item = self[key], key
        i = bisect_left(self._rating, item)
        if item == self._rating[i]:
            return i
        raise LookupError, "elementu nie znaleziono"
    def getValueByRating(self, rating):
        return self._rating[rating][0]
    def getKeyByRating(self, rating):
        return self._rating[rating][1]
def _test():
    ''' korzystamy z modułu doctest w celu przetestowania tego modułu, który należy nazwać
        rating.py. Testy realizowane są przez wykonanie wszystkich przykładów z dokumentacji. '''
    import doctest, rating
    doctest.testmod(rating)
if __name__ == "__main__":
    _test()
```

Analiza

Pod wieloma względami słownik jest najbardziej naturalną strukturą danych w zakresie przechowywania związków między kluczami (na przykład nazwiskami uczestników pewnego konkursu) i ich aktualnymi ocenami (na przykład liczby punktów, jakie poszczególni uczestnicy zdobyli do tej pory albo najwyższe stawki zaproponowane przez poszczególnych uczestników aukcji). Jeżeli ze słownika skorzystamy w takim właśnie celu, to najprawdopodobniej będziemy chcieli też odczytywać jego elementy w *kolejności naturalnej*, czyli w kolejności rosnących wartości ocen, a poza tym będziemy chcieli szybko uzyskać aktualną pozycję klucza (w takim rankingu) wynikającą z jego aktualnej oceny (na przykład pobrać dane uczestnika znajdującego się na trzecim miejscu lub ocenę uczestnika z drugiego miejsca).

W tej recepturze osiągamy takie możliwości dzięki klasie wywiedzionej z klasy `dict` i uzupełnionej o odpowiednie funkcje, których brakuje w klasie `dict` (metody `rating`, `getValueByRating`, `getKeyByRating`). Znacznie ważniejsze są jednak subtelne modyfikacje wprowadzane do metody `keys` i do innych podobnych metod, dzięki którym metody te zwracają listy lub iteratory o wymaganym porządku (na przykład kolejność rosnących ocen, a jeżeli dwa elementy mają identyczne oceny, to kolejność określana jest przez bezpośrednie porównanie dwóch kluczy). Większość najważniejszych informacji o klasie zapisanych zostało w jej dokumentacji umieszczonej w kodzie. Jest to niezwykle istotne, ponieważ zapisanie dokumentacji klasy i przykładów jej użycia wewnątrz jej kodu pozwala nam wykorzystać moduł `doctest` ze standardowej biblioteki Pythona w celu wprowadzenia funkcji testów modułowych i jednoczesnego zapewnienia poprawności podanych przykładów.

Najbardziej interesującym aspektem podanej implementacji jest to, że bardzo zmniejszono w niej ilość powtarzalnego i nudnego, a co za tym idzie podatnego na błędy kodu, bez jednoczesnego znaczącego zredukowania wydajności. Klasa `Ratings` dziedziczy wielobazowo po klasach `dict` *i* `DictMixin`, przy czym ta druga umieszczana jest jako *pierwsza* na liście klas bazowych, przez co wszystkie niepokryte jawnie metody klasy `Ratings` pochodzą właśnie z klasy `DictMixin`, o ile ona je udostępnia.

Klasa `DictMixin` przygotowana przez Raymonda Hettingera została pierwotnie opublikowana jako receptura w sieciowej wersji książki *Python Receptury*, a później stała się częścią biblioteki standardowej Pythona 2.3. Klasa `DictMixin` udostępnia wszystkie metody słownikowe z wyjątkiem metod `__init__`, `copy` i czterech metod podstawowych: `__getitem__`, `__setitem__`, `__delitem__` i `keys`. W czasie tworzenia klasy słownikowej, która ma udostępniać wszystkie metody w pełni funkcjonalnego słownika, powinniśmy przygotować klasę wywiedzioną z klasy `DictMixin` i dodać do niej przynajmniej wymienione wcześniej metody podstawowe (zależnie od semantyki klasy — na przykład jeżeli klasa ma mieć niezmienne egzemplarze, to nie ma potrzeby udostępniania metody modyfikujących `__setitem__` i `__delitem__`). Można też zaimplementować inne metody, pokrywając implementacje udostępniane przez klasę `DictMixin`, poprawiając w ten sposób ich wydajność. Architekturę klasy `DictMixin` można uznać za doskonały przykład klasycznego wzorca projektowego Szablonu Metody (ang. Template Method) zastosowanego pasywnie w bardzo przydatnym wariancie.

W klasie prezentowanej w tej recepturze po drugiej klasie bazowej (czyli po wbudowanym typie `dict`) dziedziczymy metodę `__getitem__`, a wszystkie inne metody delegujemy jawnie do klasy `dict` (te, które można wydelegować). Samodzielnie musimy zapisać podstawowe metody modyfikujące (`__setitem__` i `__delitem__`), ponieważ oprócz wydelegowania ich do bazowej klasy `dict` musimy dodatkowo zaktualizować w nich pomocniczą strukturę `self.f`.

`_rating` — listę par (ocena, klucz) utrzymywaną w stanie posortowania za pomocą modułu `bisect` z biblioteki standardowej. Metodę `keys` implementujemy samodzielnie (a skoro już o tym wspominamy: implementujemy też metody `__iter__` i `iterkeys`, ponieważ najprostszym sposobem na implementowanie metody `keys` jest wykorzystanie metody `__iter__`), tak aby wykorzystać w niej strukturę `self._rating` i zwracać klucze słownika w potrzebnej nam kolejności. W końcu, oprócz trzech standardowych metod obsługi ocen, dodajemy też oczywiste implementacje metod `__init__` i `copy`.

Wynik okazuje się być ciekawym przykładem kodu o dobrze wyważonej spójności i czytelności, w którym bardzo szeroko wykorzystywane są funkcje udostępniane przez standardową bibliotekę Pythona. Jeżeli z tego modułu skorzystamy w naszych aplikacjach, to dokładniejsze badania mogą wykazać, że niektóre z metod prezentowanej klasy nie mają zadowalającej wydajności. Wynika to z faktu, że natura klasy `DictMixin` wymusza stosowanie w niej bardzo ogólnych implementacji. W takiej sytuacji należy bezwzględnie uzupełnić klasę własnymi implementacjami tych wszystkich metod, które są wymagane do osiągnięcia lepszej wydajności. Na przykład, jeżeli nasza aplikacja często w pętlach przegląda wyniki wywołania `r.iteritems()`, gdzie `r` jest egzemplarzem klasy `Ratings`, to nieco lepszą wydajność osiągniemy, dodając do ciała klasy bezpośrednią implementację tej metody:

```
def iteritems(self):
        for v, k in self._rating:
            yield k, v
```

Zobacz również

Podręczniki *Library Reference* i *Python in a Nutshell* w częściach opisujących klasę `DictMixin` z modułu `UserDict` oraz moduł `bisect`.

5.15. Sortowanie nazwisk i rozdzielanie ich za pomocą inicjałów

Pomysłodawcy: Brett Cannon, Amos Newcombe

Problem

Chcemy przygotować spis osób, w którym poszczególne osoby zapisane byłyby w porządku alfabetycznym i pogrupowane według ich inicjałów utworzonych na podstawie nazwisk.

Rozwiązanie

W Pythonie 2.4 nowa funkcja `itertools.groupby` bardzo ułatwia realizację tego zadania:

```
import itertools
def groupnames(name_iterable):
    sorted_names = sorted(name_iterable, key=_sortkeyfunc)
    name_dict = {}
    for key, group in itertools.groupby(sorted_names, _groupkeyfunc):
        name_dict[key] = tuple(group)
    return name_dict
pieces_order = { 2: (-1, 0), 3: (-1, 0, 1) }
def _sortkeyfunc(name):
```

```
    ''' name jest ciągiem znaków zawierającym imię i nazwisko oraz opcjonalne
        drugie imię lub inicjał rozdzielane spacjami. Zwraca ciąg znaków w kolejności
        nazwisko-imię-drugie_imię, która wymagana jest w związku z sortowaniem. '''
    name_parts = name.split()
    return ' '.join([name_parts[n] for n in pieces_order[len(name_parts)]])
def _groupkeyfunc(name):
    ''' zwraca klucz grupowania, na przykład inicjał nazwiska '''
    return name.split()[-1][0]
```

Analiza

W niniejszej recepturze parametr `name_iterable` musi być elementem iterowalnym, którego elementy są ciągami znaków zawierającymi dane osób zapisane w formie: „pierwsze_imię — drugie_imię — nazwisko". W wyniku wywołania funkcji `groupnames` na rzecz takiego elementu iterowalnego otrzymamy słownik, którego klucze są inicjałami nazwisk, a powiązane z nimi wartości są krotkami zawierającymi wszystkie nazwiska, na podstawie których można utworzyć dany inicjał.

Pomocnicza funkcja `_sortkeyfunc` dzieli dane osoby zapisane w ramach jednego ciągu znaków zawierającego albo „imię nazwisko" albo „pierwsze_imię drugie_imię nazwisko" i tworzy na ich podstawie listę, zawierając najpierw nazwisko, a za nim imię, ewentualne drugie imię i na końcu inicjał. Następnie funkcja łączy te dane w ciąg znaków i zwraca go funkcji wywołującej. Zgodnie z problemem opisywanym w tej recepturze, wynikowy ciąg znaków jest kluczem, z jakiego chcemy skorzystać w ramach sortowania danych. Wbudowana w Pythona 2.4 funkcja `sorted` przyjmuje omawianą funkcje w swoim opcjonalnym parametrze `key` (wywołuje ją na rzecz każdego elementu w celu uzyskania klucza sortowania).

Pomocnicza funkcja `_groupkeyfunc` pobiera dane osoby w takim samym formacie, z jakim pracuje funkcja `_sortkeyfunc`, i zwraca inicjał z nazwiska będący kluczem grupowania, tak jak zostało to zapisane w opisie problemu.

W ramach rozwiązywania tego problemu główna funkcja z tej receptury — `groupnames` — wykorzystuje dwie opisane wcześniej funkcje pomocnicze, funkcję `sorted` z Pythona 2.4 oraz funkcję `itertools.groupby`. Z ich pomocą tworzy ona i zwraca opisywany wcześniej słownik.

Jeżeli prezentowany kod ma być stosowany również w Pythonie 2.3, to trzeba nieco przebudować samą funkcję `groupnames`, przy czym obie funkcje pomocnicze mogą pozostać bez zmian. Ze względu na to, że w bibliotece standardowej Pythona 2.3 nie ma funkcji `groupby`, wygodnej jest w nim najpierw wykonać grupowanie elementów i dopiero potem posortować dane w ramach poszczególnych grup.

```
def groupnames(name_iterable):
    name_dict = {}
    for name in name_iterable:
        key = _groupkeyfunc(name)
        name_dict.setdefault(key, []).append(name)
    for k, v in name_dict.iteritems():
        aux = [(_sortkeyfunc(name), name) for name in v]
        aux.sort()
        name_dict[k] = tuple([ n for __, n in aux ])
    return name_dict
```

Zobacz również

Recepturę 19.21. Podręcznik *Library Reference* z Pythona 2.4 w części opisującej moduł `itertools`.