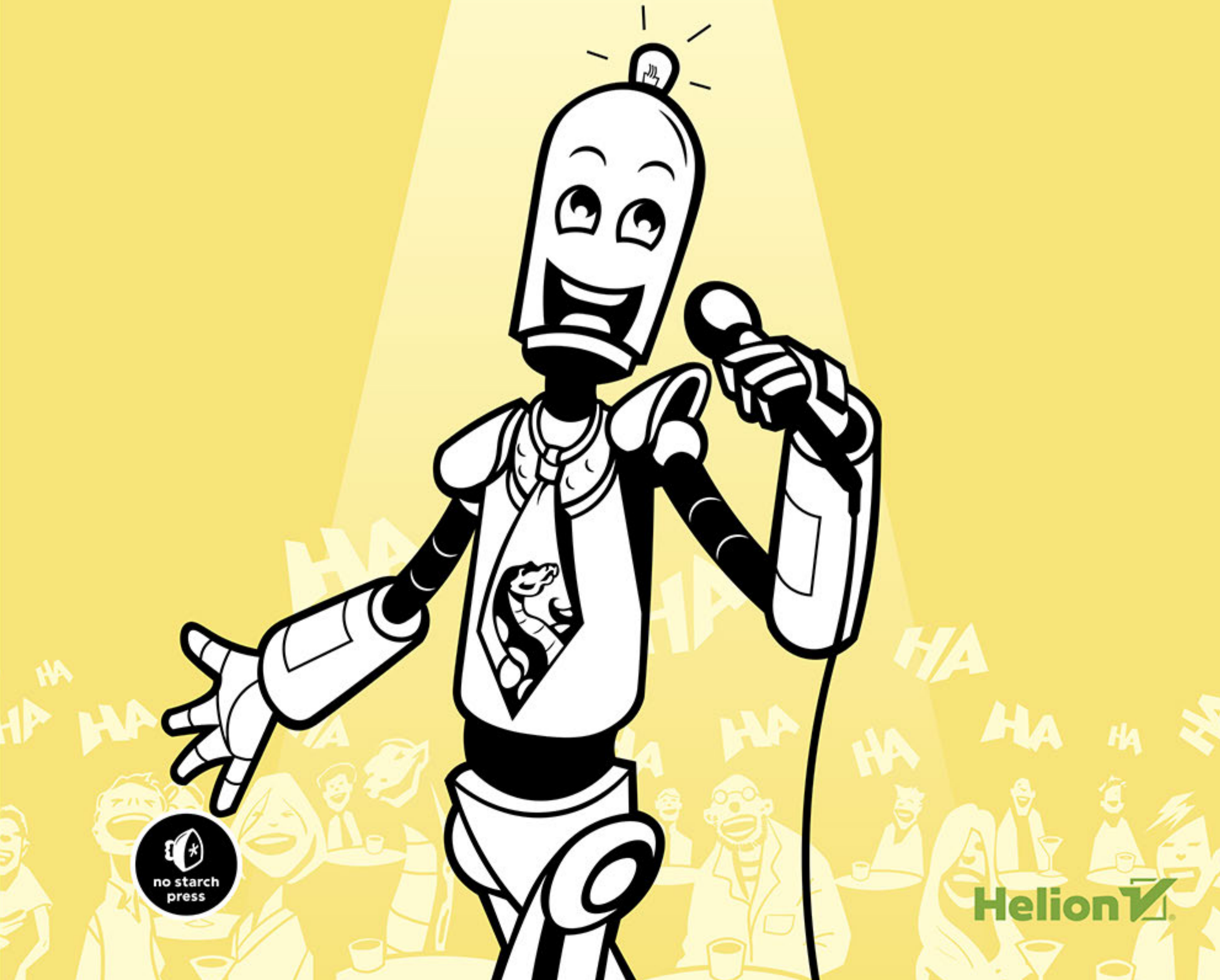
KOD PYTHONA W JEDNYM WIERSZU

JAK PROFESJONALIŚCI PISZĄ PROGRAMY DOSKONAŁE

CHRISTIAN MAYER



Tytuł oryginału: Python One-Liners: Write Concise, Eloquent Python Like a Professional

Tłumaczenie: Leszek Sagalara

ISBN: 978-83-283-7491-1

Copyright © 2020 by Christian Mayer. Title of English-language original: Python One-Liners: Write Concise, Eloquent Python Like a Professional, ISBN 978-1-7185-0050-1, published by No Starch Press.

Polish language edition copyright © 2021 by Helion SA. All rights reserved.

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

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

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

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

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

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

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

Printed in Poland.

• Kup książkę
• Poleć książkę
• Oceń książkę

• Księgarnia internetowa
• Lubię to! » Nasza społeczność

Spis treści

PODZIĘKOWANIA 13

WPROWADZENIE 15

Przykładowy jednowierszowiec Pythona ... 16
Uwaga na temat czytelności ... 17
Dla kogo jest ta książka? ... 18
Czego się nauczysz? ... 18
Zasoby online ... 19

1
ODŚWIEŻENIE WIADOMOŚCI O PYTHONIE 21

Podstawowe struktury danych ... 22
Typy i struktury danych liczbowych ... 22
Dane typu logicznego ... 22
Łańcuchy znaków ... 24
Słowo kluczowe None ... 26
Kontenerowe struktury danych ... 26
Listy ... 26
Stosy ... 29
Zbiory ... 30
Słowniki ... 31
Przynależność ... 32
Listy i zbiory składane ... 33
Przepływ sterowania ... 33
Słowa kluczowe if, else i elif ... 34
Pętle ... 34
Funkcje ... 36
Funkcje lambda ... 36
Podsumowanie ... 37

2
SZTUCZKI PYTHONA 39

Użycie listy składanej do wyszukiwania osób o najwyższych dochodach ... 40
Podstawy ... 40
Kod ... 42
Jak to działa? ... 42

Kup ksi k

Pole ksi k

Użycie listy składanej do wyszukiwania słów o dużej wartości informacyjnej 43
Podstawy 43
Kod 43
Jak to działa? 44
Odczytywanie pliku 44
Podstawy 44
Kod 45
Jak to działa? 45
Użycie funkcji lambda i map 46
Podstawy 46
Kod 47
Jak to działa? 48
Użycie wycinania do ekstrakcji środowisk dopasowanych łańcuchów podrzędnych 48
Podstawy 49
Kod 50
Jak to działa? 51
Połączenie listy składanej i wycinania 52
Podstawy 52
Kod 52
Jak to działa? 53
Przypisywanie do wycinków w celu skorygowania uszkodzonych list 53
Podstawy 54
Kod 54
Jak to działa? 55
Analiza danych dotyczących pracy serca za pomocą konkatenacji list 56
Podstawy 56
Kod 57
Jak to działa? 58
Użycie wyrażeń generatora do wyszukania firm, które płacą poniżej płacy minimalnej 58
Podstawy 58
Kod 59
Jak to działa? 59
Formatowanie baz danych za pomocą funkcji zip() 60
Podstawy 60
Kod 61
Jak to działa? 62
Podsumowanie 62

3
ANALIZA DANYCH 65
Podstawowe działania na tablicach dwuwymiarowych 66
Podstawy 66
Kod 68
Jak to działa? 69
Praca z tablicami NumPy: wycinanie, rozgłaszanie i typy tablic 70
Podstawy 70
Kod 75
Jak to działa? 76

Kup ksi k Pole ksi k

Warunkowe przeszukiwanie tablic, filtrowanie i rozgłaszanie
w celu wykrywania elementów odstających 78
Podstawy 78
Kod 79
Jak to działa? 80
Filtrowanie dwuwymiarowych tablic z użyciem indeksowania logicznego 81
Podstawy 82
Kod 82
Jak to działa? 83
Rozgłaszanie, przypisywanie do wycinków i przekształcanie
w celu oczyszczenia co i-tego elementu tablicy 84
Podstawy 84
Kod 87
Jak to działa? 87
Kiedy w NumPy używać funkcji sort(), a kiedy argsort()? 88
Podstawy 89
Kod 90
Jak to działa? 91
Jak wykorzystać funkcje lambda i indeksowanie logiczne do filtrowania tablic? 92
Podstawy 92
Kod 93
Jak to działa? 93
Jak tworzyć zaawansowane filtry tablic z wykorzystaniem statystyki, matematyki i logiki? 94
Podstawy 95
Kod 98
Jak to działa? 98
Prosta analiza asocjacji: klienci, którzy kupili X, kupili również Y 99
Podstawy 99
Kod 100
Jak to działa? 101
Bardziej zaawansowana analiza asocjacji w celu wyszukania
najlepiej sprzedających się pakietów 102
Podstawy 102
Kod 102
Jak to działa? 103
Podsumowanie 104

4
UCZENIE MASZYNOWE 107
Podstawy nadzorowanego uczenia maszynowego 107
Faza szkolenia 108
Faza wnioskowania 109
Regresja liniowa 109
Podstawy 109
Kod 112
Jak to działa? 113

Kup ksi k
Pole ksi k

Regresja logistyczna 115
Podstawy 115
Kod 118
Jak to działa? 119
Algorytm k-średnich 121
Podstawy 121
Kod 123
Jak to działa? 124
Algorytm k najbliższych sąsiadów 126
Podstawy 126
Kod 128
Jak to działa? 128
Analiza sieci neuronowej 130
Podstawy 130
Kod 134
Jak to działa? 135
Algorytm drzew decyzyjnych 137
Podstawy 138
Kod 139
Jak to działa? 139
Wyszukiwanie wiersza z minimalną wariancją 140
Podstawy 140
Kod 141
Jak to działa? 142
Podstawowe parametry statystyczne 143
Podstawy 143
Kod 144
Jak to działa? 145
Klasyfikacja z maszynami wektorów nośnych 146
Podstawy 146
Kod 148
Jak to działa? 149
Klasyfikacja z lasami losowymi 150
Podstawy 150
Kod 150
Jak to działa? 151
Podsumowanie 153

5
WYRAŻENIA REGULARNE **155**
Wyszukiwanie prostych wzorców tekstowych w łańcuchach znaków 155
Podstawy 156
Kod 158
Jak to działa? 159
Napisz własny scraper stron WWW z użyciem wyrażeń regularnych 159
Podstawy 159
Kod 161
Jak to działa? 161

Kup ksi k Pole ksi k

Analizowanie hiperłączy dokumentów HTML 162
Podstawy 162
Kod 164
Jak to działa? 165
Wydobywanie z łańcucha wartości wyrażonych w dolarach 166
Podstawy 166
Kod 167
Jak to działa? 168
Wyszukiwanie adresów URL z protokołem HTTP 168
Podstawy 168
Kod 169
Jak to działa? 169
Walidacja formatu zapisu czasu wprowadzanego przez użytkownika, część I 170
Podstawy 170
Kod 171
Jak to działa? 171
Walidacja formatu zapisu czasu wprowadzanego przez użytkownika, część II 172
Podstawy 172
Kod 172
Jak to działa? 173
Wykrywanie zduplikowanych znaków w łańcuchach 174
Podstawy 174
Kod 175
Jak to działa? 175
Wykrywanie powtórzeń słów 176
Podstawy 176
Kod 176
Jak to działa? 177
Modyfikowanie wzorców wyrażeń regularnych w wielowierszowym łańcuchu znaków 178
Podstawy 178
Kod 178
Jak to działa? 179
Podsumowanie 179

6
ALGORYTMY **181**
Wyszukiwanie anagramów za pomocą funkcji lambda i sortowania 182
Podstawy 182
Kod 183
Jak to działa? 183
Wyszukiwanie palindromów za pomocą funkcji lambda i wycinania ujemnego 184
Podstawy 185
Kod 185
Jak to działa? 186
Obliczanie permutacji z użyciem rekurencyjnych funkcji silni 186
Podstawy 186
Kod 188
Jak to działa? 189

Kup ksi k Pole ksi k

Obliczanie odległości Levenshteina 190
Podstawy 190
Kod 191
Jak to działa? 191
Obliczanie zbioru potęgowego przy użyciu programowania funkcyjnego 193
Podstawy 193
Kod 195
Jak to działa? 196
Szyfrowanie szyfrem Cezara przy użyciu zaawansowanego indeksowania i listy składanej 196
Podstawy 197
Kod 197
Jak to działa? 198
Wyznaczanie liczb pierwszych za pomocą sita Eratostenesa 199
Podstawy 199
Kod 200
Jak to działa? 201
Obliczanie ciągów Fibonacciego za pomocą funkcji reduce() 205
Podstawy 206
Kod 206
Jak to działa? 206
Rekurencyjny algorytm wyszukiwania binarnego 207
Podstawy 208
Kod 210
Jak to działa? 210
Rekurencyjny algorytm sortowania szybkiego (Quicksort) 211
Podstawy 211
Kod 212
Jak to działa? 213
Podsumowanie 213

POSŁOWIE **215**

Kup ksi k Pole ksi k

2

Sztuczki Pythona

Dla naszych potrzeb *sztuczką* będziemy nazywać wykonanie jakiegoś zadania w zaskakująco szybki lub łatwy sposób. W tej książce nauczysz się wielu różnych sztuczek i technik, dzięki którym Twój kod będzie bardziej zwięzły, a jednocześnie zwiększysz szybkość jego wprowadzania. Choć wszystkie techniczne rozdziały tej książki przedstawiają sztuczki Pythona, ten rozdział poświęcony jest „łatwym łupom", czyli sztuczkom, które można zastosować szybko i bez wysiłku, ale które mają duży wpływ na produktywność kodowania.

Rozdział ten służy również jako punkt wyjścia do kolejnych, bardziej zaawansowanych rozdziałów. Musisz zrozumieć techniki wprowadzone w tych jednowierszowcach, aby zrozumieć te, które pojawią się później. Szczególną uwagę poświęcimy podstawowym funkcjonalnościom Pythona, które pomogą Ci pisać efektywny kod, takim jak: listy składane, dostęp do plików, funkcja `map()`, funkcja `lambda`, funkcja `reduce()`, wycinki, przypisywanie do wycinków, funkcje generatora i funkcja `zip()`.

Jeśli jesteś już zaawansowanym programistą, możesz jedynie pobieżnie przejrzeć ten rozdział i zdecydować, które elementy chcesz przestudiować dogłębniej, a które już dobrze znasz.

Kup ksi k Pole ksi k

Użycie listy składanej do wyszukiwania osób o najwyższych dochodach

W tej sekcji nauczysz się pięknej, skutecznej i bardzo wydajnej funkcji tworzenia list w Pythonie, jaką jest lista składana. Będziemy z niej korzystać w wielu kolejnych jednowierszowcach.

Podstawy

Powiedzmy, że pracujesz w dziale kadr dużej firmy i musisz wyszukać wszystkich pracowników, którzy zarabiają co najmniej 100 000 zł rocznie. Pożądanym wynikiem jest lista krotek, z których każda zawiera dwie wartości: imię pracownika i jego roczne wynagrodzenie. Oto kod, który opracowałeś:

```
employees = {'Alicja' : 100000,
             'Robert' : 99817,
             'Karolina' : 122908,
             'Franciszek' : 88123,
             'Ewa' : 93121}

top_earners = []
for key, val in employees.items():
    if val >= 100000:
        top_earners.append((key,val))

print(top_earners)
> [('Alicja', 100000), ('Karolina', 122908)]
```

Choć ten kod jest poprawny, istnieje łatwiejszy i znacznie bardziej zwięzły — a przez to bardziej czytelny — sposób umożliwiający osiągnięcie tego samego rezultatu. Skoro efekt będzie taki sam, rozwiązanie z *mniejszą liczbą wierszy* pozwoli czytelnikowi szybciej uchwycić znaczenie kodu.

Python oferuje wydajny sposób tworzenia nowych list, jakim jest **lista składana**. Formuła jest następująca:

```
[ wyrażenie + kontekst ]
```

Nawiasy wskazują, że wynikiem jest nowa lista. **Kontekst** określa, które elementy listy należy wybrać. **Wyrażenie** definiuje sposób modyfikacji każdego elementu przed dodaniem wyniku do listy. Oto przykład:

```
[x * 2 for x in range(3)]
```

Pogrubiona część równania, `for x in range(3)`, jest kontekstem, a pozostała część, `x * 2`, jest wyrażeniem. Najogólniej mówiąc, wyrażenie to podwaja wartości 0, 1, 2 generowane przez kontekst. Lista składana daje zatem następujący wynik:

Kup ksi k Pole ksi k

```
> [0, 2, 4]
```

Zarówno wyrażenie, jak i kontekst mogą być dowolnie skomplikowane. Wyrażenie może być funkcją dowolnej zmiennej zdefiniowanej w kontekście i może wykonywać dowolne obliczenia — może nawet wywoływać funkcje zewnętrzne. Celem wyrażenia jest zmodyfikowanie każdego elementu listy przed dodaniem go do nowej listy.

Kontekst może składać się z jednej lub wielu zmiennych zdefiniowanych za pomocą jednej lub wielu zagnieżdżonych pętli `for`. Możesz również ograniczyć kontekst przy użyciu instrukcji `if`. W takim przypadku nowa wartość zostanie dodana do listy tylko wtedy, gdy spełniony zostanie warunek zdefiniowany przez użytkownika.

Listę składaną najlepiej wyjaśnić na przykładzie. Przestudiuj uważnie poniższe przykłady, aby dobrze zrozumieć pojęcie listy składanej:

```
print([❶x ❷for x in range(5)])
> [0, 1, 2, 3, 4]
```

Wyrażenie ❶: Funkcja tożsamościowa (nie zmienia zmiennej kontekstowej `x`)

Kontekst ❷: Zmienna kontekstowa `x` przyjmuje wszystkie wartości zwracane przez funkcję `range`: `0, 1, 2, 3, 4`

```
print([❶(x, y) ❷for x in range(3) for y in range(3)])
> [(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

Wyrażenie ❶: Utworzenie nowej krotki ze zmiennych kontekstowych `x` i `y`

Kontekst ❷: Zmienna kontekstowa `x` iteruje po wszystkich wartościach zwracanych przez funkcję `range` (`0, 1, 2`), natomiast zmienna kontekstowa `y` iteruje po wszystkich wartościach zwracanych przez drugą funkcję `range` (`0, 1, 2`). Dwie pętle `for` są zagnieżdżone, więc zmienna kontekstowa `y` powtarza swoją procedurę iteracji dla każdej z wartości zmiennej kontekstowej `x`. Mamy więc $3 \cdot 3 = 9$ kombinacji zmiennych kontekstowych

```
print([❶x ** 2 ❷2for x in range(10) if x % 2 > 0])
> [1, 9, 25, 49, 81]
```

Wyrażenie ❶: Funkcja kwadratowa na zmiennej kontekstowej `x`

Kontekst ❷: Zmienna kontekstowa `x` iteruje po wszystkich wartościach zwracanych przez funkcję `range` — `0, 1, 2, 3, 4, 5, 6, 7, 8, 9` — ale tylko wtedy, gdy są to wartości nieparzyste, tzn. gdy `x % 2 > 0`

```
print([❶x.lower() ❷for x in ['JA', 'WCALE', 'NIE', 'KRZYCZĘ']])
> ['ja', 'wcale', 'nie', 'krzyczę']
```


Kup książkę Poleć książkę

Wyrażenie ❶: Łańcuchowa funkcja zamiany na małe litery zastosowana dla zmiennej kontekstowej `x`

Kontekst ❷: Zmienna kontekstowa `x` iteruje po wszystkich wartościach łańcuchowych na liście: `'JA'`, `'WCALE'`, `'NIE'`, `'KRZYCZĘ'`

Teraz na pewno zrozumiesz kolejny fragment kodu.

Kod

Przyjrzyjmy się przedstawionemu wcześniej problemowi dotyczącemu wynagrodzeń pracowników: mając do dyspozycji słownik z kluczami łańcuchowymi i wartościami w postaci liczb całkowitych, utwórzmy nową listę krotek (klucz, wartość) tak, aby wartość powiązana z kluczem wynosiła co najmniej 100 000. Kod jest przedstawiony na listingu 2.1.

Listing 2.1. Jednowierszowe rozwiązanie dla listy składanej

```
## Dane
employees = {'Alicja' : 100000,
             'Robert' : 99817,
             'Karolina' : 122908,
             'Franciszek' : 88123,
             'Ewa' : 93121}

## Jednowierszowiec
top_earners = [(k, v) for k, v in employees.items() if v >= 100000]

## Wynik
print(top_earners)
```

Jakie będzie wyjście tego fragmentu kodu?

Jak to działa?

Przeanalizujmy ten jednowierszowiec:

```
top_earners = [ ❶(k, v) ❷for k, v in employees.items() if v >= 100000]
```

Wyrażenie ❶: Tworzy prostą (klucz, wartość) krotkę dla zmiennych kontekstowych `k` i `v`

Kontekst ❷: Metoda słownikowa `dict.items()` zapewnia, że zmienna kontekstowa `k` iteruje po wszystkich kluczach słownika, a zmienna kontekstowa `v` iteruje po wartościach powiązanych ze zmienną kontekstową `k` — ale tylko wtedy, gdy wartość zmiennej kontekstowej `v` wynosi 100 000 lub więcej, co zapewnia warunek `if`

Wynik tego jednowierszowca jest następujący:

```
print(top_earners)
> [('Alicja', 100000), ('Karolina', 122908)]
```

Ten prosty, jednowierszowy program wprowadza ważne pojęcie *listy składanej*. W niniejszej książce wielokrotnie korzystamy z list składanych, więc upewnij się, że rozumiesz przykłady przedstawione w tej sekcji, zanim przejdziesz dalej.

Użycie listy składanej do wyszukiwania słów o dużej wartości informacyjnej

W tym jednowierszowcu jeszcze dogłębniej poznasz możliwości listy składanej.

Podstawy

Wyszukiwarki klasyfikują informacje tekstowe na podstawie ich znaczenia dla zapytania użytkownika. W tym celu analizują zawartość przeszukiwanego tekstu. Cały tekst składa się ze słów. Niektóre słowa dostarczają wielu informacji o zawartości tekstu, a inne nie. Przykładami tych pierwszych są słowa takie jak *biały*, *wieloryb*, *Kapitan*, *Ahab* (znasz ten tekst?). Przykładami drugich są słowa takie jak *to*, *do*, *w*, *i* lub *jak*, ponieważ większość tekstów zawiera te słowa. Odfiltrowywanie słów, które nie wnoszą wiele znaczenia, jest powszechną praktyką przy wdrażaniu wyszukiwarek. Prosta heurystyka polega na odfiltrowaniu wszystkich słów składających się z nie więcej niż trzech znaków.

Kod

Naszym celem jest rozwiązanie następującego problemu: mając do dyspozycji wielowierszowy łańcuch, musimy utworzyć listę list — z których każda będzie zawierać wszystkie słowa w wierszu mające więcej niż trzy znaki. Dane i rozwiązanie zostały podane na listingu 2.2.

Listing 2.2. Jednowierszowe rozwiązanie umożliwiające wyszukanie słów o dużej wartości informacyjnej

```
## Dane
text = '''Imię moje: Izmael. Przed kilku laty — mniejsza o ścisłość jak dawno temu
↳— mając niewiele czy też nie mając wcale pieniędzy w sakiewce, a nie widząc nic
↳szczególnego, co by mnie interesowało na lądzie, pomyślałem sobie, że pożegluję
↳nieco po morzach i obejrzę wodną część świata. Taki mam właśnie sposób
↳odpędzania splinu i regulowania krwiobiegu. - Moby Dick'''

## Jednowierszowiec
w = [[x for x in line.split() if len(x)>3] for line in text.split('\n')]

## Wynik
print(w)
```

Jakie będzie wyjście tego kodu?

Kup ksi k Pole ksi k

Jak to działa?

Jednowierszowiec tworzy listę list przy użyciu dwóch zagnieżdżonych wyrażeń listy składanej:

- Wyrażenie wewnętrznej listy składanej `[x for x in line.split() if len(x)>3]` używa funkcji łańcuchowej `split()` do podzielenia danego wiersza na sekwencję słów. Iterujemy po wszystkich słowach `x` i dodajemy je do listy, jeśli mają więcej niż trzy znaki.
- Wyrażenie zewnętrznej listy składanej tworzy łańcuch `line` użyty w poprzednim wyrażeniu. Ponownie wykorzystuje ono funkcję `split()` w celu podzielenia tekstu zgodnie ze znakami nowego wiersza `'\n'`.

Oczywiście musisz przyzwyczaić się do myślenia w kategoriach list składanych, co może wymagać trochę czasu. Ale po przeczytaniu tej książki listy składane staną się Twoim chlebem powszednim i będziesz w stanie szybko odczytać i napisać taki kod w Pythonie.

Odczytywanie pliku

W tej sekcji odczytasz plik i zapiszesz wynik jako listę łańcuchów znaków (jeden łańcuch na wiersz). Oprócz tego usuniesz z wierszy wszystkie początkowe i końcowe znaki niedrukowane.

Podstawy

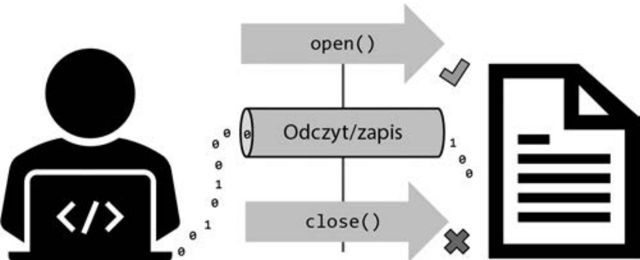
W Pythonie odczytywanie pliku jest proste, ale zrealizowanie tego wymaga zazwyczaj kilku wierszy kodu (i jednego lub dwóch wyszukiwań w Google). Standardowy sposób odczytania pliku w Pythonie jest następujący:

```
filename = "readFileDefault.py" # Ten kod

f = open(filename)
lines = []
for line in f:
    lines.append(line.strip())

print(lines)
> ['filename = "readFileDefault.py" # ten kod', '', 'f = open(filename)', 'lines =
↪[]', 'for line in f:', 'lines.append(line.strip())', '', 'print(lines)']
```

Kod zakłada, że ten fragment kodu został zapisany w folderze jako plik o nazwie *readFileDefault.py*. Następnie kod otwiera ten plik, tworzy pustą listę (`lines`) i wypełnia ją łańcuchami znaków za pomocą operacji `append()` w treści pętli `for` dokonującej iteracji po wszystkich wierszach w pliku. Użyta tu została również metoda łańcuchowa `strip()`, aby usunąć wszelkie wiodące lub końcowe znaki niedrukowane (w przeciwnym razie w łańcuchach pojawiłby się znak nowego wiersza `'\n'`).

Kup ksi k Pole ksi k

Aby uzyskać dostęp do plików na komputerze, musisz wiedzieć, jak otwierać i zamykać pliki. Dostęp do danych z pliku można uzyskać dopiero po jego otwarciu. Po zamknięciu pliku możesz mieć pewność, że dane zostały w nim zapisane. Python może utworzyć bufor i odczekać chwilę, zanim cały bufor zostanie zapisany w pliku (zobacz rysunek 2.1). Przyczyna jest prosta: dostęp do pliku jest powolny. Ze względu na wydajność Python unika zapisywania każdego pojedynczego bitu niezależnie. Zamiast tego odczekuje, aż bufor wypełni się wystarczającą ilością bajtów, a następnie zapisuje do pliku cały bufor naraz.



Rysunek 2.1. Otwieranie i zamykanie pliku w Pythonie

Dlatego też po odczytaniu pliku warto zamknąć go poleceniem `f.close()`, aby mieć pewność, że wszystkie dane zostały poprawnie zapisane w pliku, a nie znajdują się w pamięci tymczasowej. Istnieje jednak kilka wyjątków, gdzie Python zamyka plik automatycznie: jeden z nich występuje, gdy liczba referencji spadnie do zera, co ma miejsce w następnym kodzie.

Kod

Naszym celem jest otwarcie pliku, wczytanie wszystkich wierszy, usunięcie wiodących i końcowych znaków niedrukowanych oraz zapisanie wyniku w postaci listy. Jednowierszowiec wykonujący to zadanie jest przedstawiony na listingu 2.3.

Listing 2.3. Jednowierszowe rozwiązanie umożliwiające odczytywanie pliku wiersz po wierszu

```
print([line.strip() for line in open("readFile.py")])
```

Zanim zaczniesz czytać dalej, spróbuj odgadnąć wyjście tego fragmentu kodu.

Jak to działa?

Instrukcja `print()` służy do wypisania wynikowej listy do powłoki. Jest to lista składana (zobacz punkt „Użycie listy składanej do wyszukiwania osób o najwyższych dochodach”). W części listy składanej zawierającej *wyrażenie* użyta została metoda `strip()` dla obiektów łańcuchowych.

Część listy składanej zawierająca *kontekst* przeprowadza iterację po wszystkich wierszach w pliku.

Kup ksi k Pole ksi k

Wyjściem tego jednowierszowca jest jego zawartość (ponieważ odczytuje on swój własny plik z kodem źródłowym Pythona o nazwie *readFile.py*), oznaczona jako łańcuch znaków zawarty w liście:

```
print([line.strip() for line in open("readFile.py")])
> ['print([line.strip() for line in open("readFile.py")])']
```

Jak widać, krótszy i bardziej zwięzły kod jest również czytelniejszy, bez uszczerbku dla wydajności.

Użycie funkcji lambda i map

W tej sekcji przedstawiam dwie ważne funkcje Pythona: `lambda` i `map()`. Obie są cennymi narzędziami w zestawie narzędzi Pythona. Będziesz ich używał do przeszukiwania listy łańcuchów pod kątem występowania innego łańcucha.

Podstawy

Z rozdziału 1. wiesz, jak zdefiniować nową funkcję za pomocą wyrażenia `def x`, po którym następuje treść funkcji. Nie jest to jednak jedyny sposób definiowania funkcji w Pythonie. Możesz również użyć **funkcji lambda**, aby zdefiniować prostą funkcję *ze zwracaną wartością* (zwracaną wartością może być dowolny obiekt, w tym krotka, lista i zbiór). Innymi słowy, każda funkcja lambda zwraca wartość obiektu do swojego środowiska wywołującego. Zauważ, że stanowi to praktyczne ograniczenie dla funkcji lambda, ponieważ w odróżnieniu od standardowych funkcji nie są one przeznaczone do wykonywania kodu *bez* zwracania wartości obiektu do środowiska wywołującego.

WSKAZÓWKA *O funkcjach lambda była już mowa w rozdziale 1., ale ponieważ jest to ważna koncepcja, której będę często używał w niniejszej książce, w tej sekcji przyjrzymy się jej dokładniej.*

Funkcje lambda umożliwiają zdefiniowanie nowej funkcji w jednym wierszu za pomocą słowa kluczowego `lambda`. Jest to użyteczne, gdy chcesz szybko utworzyć funkcję, której użyjesz tylko raz i nie będzie Ci więcej potrzebna. Najpierw przeanalizujmy dokładnie składnię funkcji lambda:

```
lambda argumenty : wyrażenie zwrotne
```

Definicję funkcji rozpoczynasz od słowa kluczowego `lambda`, po którym następuje sekwencja argumentów funkcji. Muszą być one dostarczone przy wywoływaniu funkcji. Następnie dołączasz dwukropek (`:`) i **wyrażenie zwrotne** (ang. *return expression*), które oblicza zwracaną wartość na podstawie argumentów funkcji

Kup ksi k

Pole ksi k

lambda. Wyrażenie zwrotne, które może być dowolnym wyrażeniem Pythona, oblicza wyjście funkcji. Weźmy dla przykładu następującą definicję funkcji:

```
lambda x, y: x + y
```

Funkcja lambda ma dwa argumenty, `x` i `y`. Wartością zwrotną jest po prostu suma obu argumentów, `x + y`.

Funkcji lambda zazwyczaj używa się w sytuacji, gdy funkcja jest wywoływana tylko raz i można ją łatwo zdefiniować w jednym wierszu kodu. Typowym przykładem jest użycie funkcji lambda z funkcją `map()`, która jako argumenty wejściowe przyjmuje obiekt funkcji `f` oraz sekwencję `s`. Następnie funkcja `map()` stosuje funkcję `f` na każdym elemencie w sekwencji `s`. Oczywiście jako argument `f` *możesz* zdefiniować pełnoprawną, nazwaną funkcję. Ale często jest to niewygodne i zmniejsza czytelność — szczególnie jeśli funkcja jest krótka i potrzebujesz jej tylko raz — więc zwykle najlepiej będzie wtedy użyć funkcji lambda.

Przed zaprezentowaniem jednowierszowca pokażę szybko kolejną małą sztuczkę Pythona, która ułatwi Ci życie: sprawdzenie, czy łańcuch `x` zawiera łańcuch podrzędny `y`, poprzez użycie wyrażenia `y in x`. Wyrażenie to zwraca wynik `True`, jeśli istnieje przynajmniej jedno wystąpienie łańcucha `y` w łańcuchu `x`. Przykładowo wyrażenie `'42' in 'Odpowiedź to 42'` zostanie ocenione jako `True`, podczas gdy wyrażenie `'21' in 'Odpowiedź to 42'` zostanie ocenione jako `False`.

Przyjrzyjmy się teraz naszemu jednowierszowcowi.

Kod

Po podaniu listy łańcuchów następny jednowierszowiec (zobacz listing 2.4) tworzy nową listę krotek, z których każda składa się z wartości typu logicznego i oryginalnego łańcucha. Wartość logiczna wskazuje, czy w oryginalnym łańcuchu pojawia się łańcuch znaków `'anonimowe'`. Listę wynikową nazwaliśmy `mark` (oznaczenie), ponieważ wartości logiczne oznaczają te elementy łańcuchowe na liście, które zawierają łańcuch znaków `'anonimowe'`.

Listing 2.4. Jednowierszowe rozwiązanie do oznaczania łańcuchów, które zawierają łańcuch znaków 'anonimowe'

```
## Dane
txt = ['Funkcje lambda to funkcje anonimowe.',
       'Funkcje anonimowe nie mają nazwy.',
       'Funkcje w Pythonie są obiektami.']

## Jednowierszowiec
mark = map(lambda s: (True, s) if 'anonimowe' in s else (False, s), txt)

## Wynik
print(list(mark))
```

Jakie będzie wyjście tego kodu?

Kup ksi k

Pole ksi k

Jak to działa?

Funkcja `map()` dodaje wartość logiczną do każdego elementu łańcuchowego w oryginalnej liście `txt`. Wartość logiczna to `True`, jeśli element łańcuchowy zawiera słowo *anonimowe*. Pierwszym argumentem jest anonimowa funkcja lambda, a drugim lista łańcuchów, które chcemy sprawdzić pod kątem występowania określonego łańcucha znaków.

Użyliśmy tu wyrażenia zwrotnego funkcji lambda `(True, s) if 'anonimowe' in s else (False, s)`, aby wyszukać łańcuch znaków `'anonimowe'`. Wartość `s` jest argumentem wejściowym funkcji lambda, który w tym przykładzie jest łańcuchem znaków. Jeśli w tym łańcuchu występuje wyszukiwany łańcuch znaków `'anonimowe'`, wyrażenie zwraca krotkę `(True, s)`. W przeciwnym wypadku zwraca krotkę `(False, s)`.

Wynik działania tego jednowierszowca jest następujący:

```
## Wynik
print(list(mark))
> [(True, 'Funkcje lambda to funkcje anonimowe.'), (True, 'Funkcje anonimowe
↪nie mają nazwy.'), (False, 'Funkcje w Pythonie są obiektami.')]
```

Wartości logiczne wskazują, że tylko dwa pierwsze łańcuchy na liście zawierają łańcuch znaków `'anonimowe'`.

W następnych jednowierszowcach przekonasz się, że funkcje lambda są niezwykle użyteczne. Ponadto czynisz stały postęp na drodze do osiągnięcia celu, jakim jest zrozumienie każdego wiersza kodu Pythona, który napotkasz w praktyce.

ĆWICZENIE 2.1

Użyj listy składanej zamiast funkcji `map()`, aby uzyskać to samo wyjście (rozwiązanie znajdziesz na końcu tego rozdziału).

Użycie wycinania do ekstrakcji środowisk dopasowanych łańcuchów podrzędnych

W tej sekcji poznasz ważną koncepcję, jaką jest **wycinanie** (ang. *slicing*) — proces wyodrębniania łańcucha podrzędnego z oryginalnej pełnej sekwencji w celu przetworzenia prostych zapytań tekstowych. Przeszukamy jakiś tekst pod kątem określonego łańcucha znaków, a następnie wyodrębnimy ten łańcuch wraz z pewną liczbą otaczających go znaków, aby uzyskać kontekst.

Kup ksi k Pole ksi k

Podstawy

Wycinanie jest integralnym elementem wielu koncepcji i technik Pythona, zarówno zaawansowanych, jak i podstawowych, np. podczas korzystania z wbudowanych struktur danych Pythona, takich jak listy, krotki i łańcuchy znaków. Wycinanie jest również podstawą wielu zaawansowanych bibliotek Pythona, takich jak NumPy, Pandas, TensorFlow i *scikit-learn*. Dokładne przestudiowanie wycinania zapewni pozytywny efekt domina w Twojej karierze programisty Pythona.

Wycinanie wyodrębnia z sekwencji jej sekwencje podrzędne, np. część łańcucha znaków. Składnia jest prosta. Powiedzmy, że masz zmienną `x`, która odwołuje się do łańcucha znaków, listy lub krotki. Możesz wyodrębnić sekwencję podrzędną, używając następującego zapisu:

```
x[start:stop:krok]
```

Wynikowa sekwencja podrzędna rozpoczyna się od indeksu `start` (włącznie), a kończy na indeksie `stop` (nie obejmując go). Opcjonalnie możesz dołączyć trzeci argument, `krok`, określający, które elementy zostaną wyodrębnione, więc możesz zdecydować się na dołączanie tylko co *n*-tego elementu. Przykładowo operacja wycinania `x[1:4:1]` zastosowana na zmiennej `x = "witaj, świecie"` da w wyniku łańcuch `"ita"`. Operacja wycinania `x[1:4:2]` na tej samej zmiennej da w wyniku łańcuch `'ia'`, ponieważ do wynikowego wycinka zostanie dołączony tylko co drugi element. Jak pamiętasz z rozdziału 1., w Pythonie pierwszy element sekwencji dowolnego typu, np. łańcucha lub listy, ma indeks 0.

Jeśli nie podasz argumentu `krok`, Python przyjmuje domyślną wielkość kroku 1. Przykładowo wywołanie wycinka `x[1:4]` dałoby w rezultacie łańcuch `'ita'`.

Jeśli nie podasz argumentu rozpoczęcia lub zakończenia, Python założy, że chcesz zacząć od początku lub zakończyć na końcu. Przykładowo wywołanie wycinka `x[:4]` dałoby w wyniku łańcuch `'wita'`, a wywołanie wycinka `x[4:]` dałoby w wyniku łańcuch `'j, świecie'`.

Przestudiuj poniższe przykłady, aby jeszcze lepiej to zrozumieć:

```
s = 'Jedz więcej owoców!'

print(s[0:4])
> Jedz

❶ print(s[3:0])
> (pusty łańcuch '')

print(s[:6])
> Jedz w

print(s[6:])
> ięcej owoców!
```


Kup książkę Poleć książkę

```
❷ print(s[:100])
  > Jedz więcej owoców!

  print(s[6:9:2])
  > mr

❸ print(s[::3])
  > Jzieoc!

❹ print(s[::-1])
  > !wócowo jecęiw zdeJ

  print(s[6:1:-1])
  > iw zd
```

Te warianty podstawowego wzorca `[start:stop:krok]` wycinania w Pythonie podkreślają wiele ciekawych właściwości tej techniki:

- Jeśli `start >= stop` przy dodatniej wielkości kroku, wycinek jest pusty ❶.
- Jeśli argument `stop` jest większy niż długość sekwencji, Python wyznaczy koniec wycinka na elemencie znajdującym się najbardziej na prawo (dołączając go do wycinka) ❷.
- Jeśli wartość `krok` jest dodatnia, domyślnym początkiem jest element położony najbardziej na lewo, a domyślnym końcem element położony najbardziej na prawo (będzie on uwzględniany przy wyznaczaniu wycinka) ❸.
- Jeśli wartość `krok` jest ujemna (`krok` < 0), wyznaczanie wycinka sekwencji przebiega w odwrotnej kolejności. Przy pustych argumentach `start` i `stop` wycinanie odbywa się od skrajnego prawego elementu do skrajnego lewego elementu (są one uwzględniane przy wyznaczaniu wycinka) ❹. Zauważ, że jeśli podany jest argument `stop`, odpowiednia pozycja jest wykluczona z wycinka.

Następnie użyjemy wycinania wraz z metodą `string.find(`*`wartość`*`)`, aby znaleźć indeks argumentu łańcuchowego `wartość` w danym łańcuchu.

Kod

Naszym celem jest odnalezienie konkretnego zapytania tekstowego w obrębie wielowierszowego łańcucha znaków. Chcesz znaleźć zapytanie w tekście i zwrócić jego bezpośrednie otoczenie, aż do 18 pozycji po obu stronach wyszukanego zapytania. Jeśli wyodrębnimy zapytanie wraz z otaczającym je tekstem, możemy zobaczyć kontekst tekstowy znalezionego łańcucha — podobnie jak Google prezentuje fragmenty tekstu po obu stronach wyszukiwanego słowa kluczowego. W listingu 2.5 szukamy łańcucha `'SQL'` w liście skierowanym przez firmę Amazon do akcjonariuszy — z bezpośrednim otoczeniem do 18 pozycji po obu stronach łańcucha `'SQL'`.

Kup ksi k Pole ksi k

Listing 2.5. Jednowierszowe rozwiązanie do wyszukiwania w tekście łańcuchów znaków oraz ich bezpośredniego otoczenia

```
## Dane
letters_amazon = '''Spędziliśmy kilka lat, budując własny silnik bazy danych,
↪Amazon Aurora. Jest to w pełni zarządzana usługa zgodna z MySQL i PostgreSQL,
↪która pod względem trwałości i dostępności dorównuje silnikom komercyjnym,
↪a nawet je prześciga - ale za jedną dziesiątą kosztów. Nie byliśmy zaskoczeni,
↪kiedy okazało się, że to działa.'''

## Jednowierszowiec
find = lambda x, q: x[x.find(q)-18:x.find(q)+18] if q in x else -1

## Wynik
print(find(letters_amazon, 'SQL'))
```

Spróbuj odgadnąć wyjście tego kodu.

Jak to działa?

Definiujesz funkcję lambda z dwoma argumentami: wartością łańcucha `x` oraz zapytaniem `q` do wyszukania w tekście. Funkcji lambda przypisujesz nazwę `find`. Funkcja `find(x, q)` odnajduje zapytanie o łańcuch znaków `q` w łańcuchu tekstowym `x`.

Jeżeli zapytanie `q` nie pojawia się w łańcuchu `x`, od razu zwrócony zostanie wynik `–1`. W przeciwnym razie na łańcuchu tekstowym zastosowana zostanie operacja wycinania, aby wyodrębnić pierwsze wystąpienie zapytania wraz z 18 znakami po lewej i po prawej stronie, aby uchwycić kontekst. Zauważ, że do wyszukania indeksu pierwszego wystąpienia `q` w łańcuchu `x` została użyta funkcja łańcuchowa `x.find(q)`. Jest ona wywoływana dwukrotnie, aby określić indeksy początkowy i końcowy wycinka, lecz oba wywołania funkcji zwracają tę samą wartość, ponieważ zarówno zapytanie `q`, jak i łańcuch `x` nie ulegają zmianie. Chociaż ten kod działa doskonale, nadmiarowe wywoływanie funkcji niesie ze sobą niepotrzebne obliczenia — wadę tę można łatwo naprawić, dodając zmienną pomocniczą w celu tymczasowego zapisania wyniku pierwszego wywołania funkcji, po czym wykorzystać go ponownie przez uzyskanie dostępu do wartości przechowywanej w zmiennej pomocniczej.

To zagadnienie podkreśla ważny kompromis: ograniczając się do jednego wiersza kodu, nie możesz zdefiniować i ponownie użyć zmiennej pomocniczej do przechowywania indeksu pierwszego wystąpienia zapytania. Zamiast tego musisz zastosować tę samą funkcję `find`, aby obliczyć indeks początkowy (i zmniejszyć wynik o 18 pozycji w indeksie) oraz indeks końcowy (i zwiększyć wynik o 18 pozycji w indeksie). W rozdziale 5. poznasz bardziej efektywny sposób wyszukiwania wzorców w łańcuchach (przy użyciu wyrażeń regularnych), który rozwiąże ten problem.

Wyszukując zapytanie `'SQL'` w liście firmy Amazon do akcjonariuszy, znajdziesz jego wystąpienie w tekście:

```
## Wynik
print(find(letters_amazon, 'SQL'))
> usługa zgodna z MySQL i PostgreSQL,
```

W rezultacie otrzymasz łańcuch znaków i kilka otaczających go słów, które dostarczą kontekst wyszukiwania. Wycinanie w Pythonie jest kluczową umiejętnością, którą musisz opanować. Aby pogłębić Twoją znajomość tego zagadnienia, przedstawię jeszcze jeden jednowierszowiec wykorzystujący wycinanie.

Połączenie listy składanej i wycinania

W tej sekcji przedstawię połączenie listy składanej i wycinania w celu próbkowania dwuwymiarowego zbioru danych. Naszym celem będzie utworzenie mniejszej, ale reprezentatywnej próbki danych z niewspółmiernie dużej próbki.

Podstawy

Powiedzmy, że pracujesz jako analityk finansowy dla dużego banku i szkolisz nowy model uczenia maszynowego do prognozowania cen akcji. Masz do dyspozycji treningowy zbiór danych na temat rzeczywistych cen akcji. Jednak zbiór ten jest ogromny, a szkolenie modelu na Twoim komputerze wydaje się trwać wieki. Dla przykładu w uczeniu maszynowym często sprawdza się dokładność prognozowania dla różnych zestawów parametrów modelu. Powiedzmy, że w naszej aplikacji trzeba czekać godzinami, aż program treningowy dobiegnie końca (szkolenie bardzo złożonych modeli na dużych zbiorach danych rzeczywiście zajmuje całe godziny). Aby przyspieszyć pracę, zmniejszasz zbiór danych o połowę, wykluczając co drugi punkt danych o cenach akcji. Oczekujesz, że ta modyfikacja nie spowoduje znacznego zmniejszenia dokładności modelu.

W tej sekcji wykorzystasz dwie funkcjonalności Pythona, które poznałeś wcześniej w tym rozdziale: listę składaną i wycinanie. Lista składana umożliwia iterację po każdym elemencie listy, a następnie jego modyfikację. Wycinanie pozwala na szybkie wybranie co drugiego elementu z danej listy, co w naturalny sposób umożliwia proste operacje filtrowania. Przyjrzyjmy się dokładnie, jak można wykorzystać te dwie funkcjonalności w połączeniu.

Kod

Naszym celem jest utworzenie nowej próbki treningowej z danych — listy list, z których każda zawiera sześć liczb zmiennoprzecinkowych obejmujących jedynie co drugą wartość zmiennoprzecinkową z oryginalnego zestawu danych. Spójrz na listing 2.6.

Jak zwykle, sprawdź, czy potrafisz odgadnąć wyjście.

Kup ksi k Pole ksi k

Listing 2.6. Jednowierszowe rozwiązanie dla próbkowania danych

```
## Dane (dzienne kursy akcji (w dolarach))
price = [[9.9, 9.8, 9.8, 9.4, 9.5, 9.7],
         [9.5, 9.4, 9.4, 9.3, 9.2, 9.1],
         [8.4, 7.9, 7.9, 8.1, 8.0, 8.0],
         [7.1, 5.9, 4.8, 4.8, 4.7, 3.9]]

## Jednowierszowiec
sample = [line[::2] for line in price]

## Wynik
print(sample)
```

Jak to działa?

Nasze rozwiązanie to podejście dwuetapowe. Najpierw używamy listy składanej do iteracji po wszystkich wierszach oryginalnej listy z cenami akcji. Następnie tworzymy nową listę liczb zmiennoprzecinkowych poprzez wycinanie każdego wiersza; używamy instrukcji `line[start:stop:krok]` z domyślnymi parametrami początku i końca oraz wielkością kroku `2`. Nowa lista zawiera tylko trzy (a nie sześć) liczby zmiennoprzecinkowe, co daje następującą tablicę:

```
## Wynik
print(sample)
> [[9.9, 9.8, 9.5], [9.5, 9.4, 9.2], [8.4, 7.9, 8.0], [7.1, 4.8, 4.7]]
```

Ten jednowierszowiec wykorzystujący wbudowaną funkcjonalność Pythona nie jest skomplikowany. Jednak w rozdziale 3. poznasz jeszcze krótszą wersję, która wykorzystuje bibliotekę NumPy do obliczeń z zakresu analizy danych.

ĆWICZENIE 2.2

Po zapoznaniu się z rozdziałem 3. wróć do tego jednowierszowca i zaproponuj bardziej zwięzłe jednowierszowe rozwiązanie z użyciem biblioteki NumPy. Podpowiedź: wykorzystaj bardziej zaawansowane możliwości NumPy w zakresie wycinania.

Przypisywanie do wycinków w celu skorygowania uszkodzonych list

W tej sekcji przedstawiam zaawansowaną technikę wycinania w Pythonie: **przypisywanie do wycinków** (ang. *slice assignments*). Przypisywanie do wycinków wykorzystuje notację wycinania *z lewej strony* operacji przypisywania, aby zmodyfikować sekwencję podrzędną oryginalnej sekwencji.

Kup ksi k

Pole ksi k

Podstawy

Wyobraź sobie, że pracujesz w niewielkiej firmie internetowej, która monitoruje przeglądarki swoich użytkowników (Chrome, Firefox, Safari). Dane te są przechowywane w bazie danych. Aby przeprowadzić analizę danych, wczytujesz dane zebrane z przeglądarek do dużej listy łańcuchów, ale z powodu błędu w Twoim algorytmie śledzenia co drugi łańcuch jest uszkodzony i trzeba go zastąpić właściwym.

Załóżmy, że Twój serwer WWW zawsze przekierowuje pierwsze żądanie użytkownika na inny adres URL (jest to powszechna praktyka w projektowaniu stron WWW, znana jako kod HTML 301: trwale przeniesiony; ang. *moved permanently*). Dochodzisz do wniosku, że pierwsza wartość przeglądarki będzie w większości przypadków równa drugiej, ponieważ przeglądarka użytkownika nie zmienia się w czasie oczekiwania na przekierowanie. Oznacza to, że można łatwo odtworzyć oryginalne dane. Zasadniczo chcesz powielić co drugi łańcuch na liście: lista `['Firefox', 'uszkodzone', 'Chrome', 'uszkodzone']` zmieni się w `['Firefox', 'Firefox', 'Chrome', 'Chrome']`.

Jak można to osiągnąć w szybki, czytelny i efektywny sposób (najlepiej w jednym wierszu kodu)? Twoim pierwszym pomysłem jest utworzenie nowej listy, iterowanie po uszkodzonej liście i dwukrotne dodanie każdego nieuszkodzonego wpisu do nowej listy. Ale odrzucasz ten pomysł, ponieważ wtedy musiałbyś utrzymywać dwie listy w swoim kodzie — a każda z nich może mieć miliony wpisów. Poza tym takie rozwiązanie wymagałoby kilku wierszy kodu, co zaszkodziłoby zwięzłości i czytelności kodu źródłowego.

Na szczęście przeczytałeś o pięknej funkcjonalności Pythona: przypisywaniu do wycinków. Możesz to wykorzystać do wybrania i zastąpienia *sekwencji elementów* pomiędzy indeksami `i` i `j` z użyciem notacji wycinania `lst[i:j] = [0 0 ...0]`. Ponieważ wycinanie `lst[i:j]` ma miejsce *po lewej stronie* operacji przypisania (a nie jak poprzednio po prawej), funkcję tę określa się jako *przypisywanie* do wycinków.

Idea przypisywania do wycinków jest prosta: w oryginalnej sekwencji wszystkie elementy wskazane po lewej stronie należy zastąpić elementami znajdującymi się po prawej stronie.

Kod

Naszym celem jest zastąpienie co drugiego łańcucha znaków łańcuchem znajdującym się bezpośrednio przed nim (zobacz listing 2.7).

Listing 2.7. Jednowierszowe rozwiązanie do zastąpienia wszystkich uszkodzonych łańcuchów

```
## Dane
visitors = ['Firefox', 'uszkodzone', 'Chrome', 'uszkodzone',
            'Safari', 'uszkodzone', 'Safari', 'uszkodzone',
            'Chrome', 'uszkodzone', 'Firefox', 'uszkodzone']

## Jednowierszowiec
```


Kup ksi k Pole ksi k

```
visitors[1::2] = visitors[::2]

## Wynik
print(visitors)
```

Jak będzie wyglądać poprawiona sekwencja przeglądarek w tym kodzie?

Jak to działa?

Jednowierszowe rozwiązanie zastępuje łańcuchy `'uszkodzone'` identyfikatorami przeglądarek, które poprzedzają je na liście. Aby uzyskać dostęp do każdego uszkodzonego elementu na liście `visitors`, należy użyć notacji przypisywania do wycinków. W poniższym fragmencie kodu wyróżniłem wybierane elementy:

```
visitors = ['Firefox', 'uszkodzone', 'Chrome', 'uszkodzone',
            'Safari', 'uszkodzone', 'Safari', 'uszkodzone',
            'Chrome', 'uszkodzone', 'Firefox', 'uszkodzone']
```

Kod zastępuje te wybrane elementy wycinkiem znajdującym się po prawej stronie operacji przypisywania. Elementy te są wyróżnione w poniższym fragmencie kodu:

```
visitors = ['Firefox', 'uszkodzone', 'Chrome', 'uszkodzone',
            'Safari', 'uszkodzone', 'Safari', 'uszkodzone',
            'Chrome', 'uszkodzone', 'Firefox', 'uszkodzone']
```

Pierwsze elementy są zastępowane drugimi. W związku z tym wynikowa lista `visitors` jest następująca (zastąpione elementy zostały wyróżnione):

```
## Wynik
print(visitors)
> ['Firefox', 'Firefox', 'Chrome', 'Chrome', 'Safari', 'Safari', 'Safari',
↪'Safari', 'Chrome', 'Chrome', 'Firefox', 'Firefox']
```

Wynikiem jest oryginalna lista, w której każdy łańcuch `'uszkodzone'` został zastąpiony poprzedzającym go identyfikatorem przeglądarki. W ten sposób można naprawić uszkodzony zbiór danych.

Użycie przypisywania do wycinków jest najszybszym i najefektywniejszym sposobem wykonania tego prostego zadania. Zauważ, że oczyszczone dane charakteryzują się bezstronnymi statystykami użycia przeglądarek: przeglądarka z 70-procentowym udziałem w uszkodzonych danych utrzyma swój 70-procentowy udział w naprawionych danych. Naprawione dane mogą być następnie wykorzystane do dalszej analizy, na przykład w celu sprawdzenia, czy użytkownicy Safari są lepszymi klientami (w końcu zazwyczaj wydają więcej pieniędzy na sprzęt). Właśnie nauczyłeś się prostego i zwięzłego sposobu programowego modyfikowania istniejącej listy.

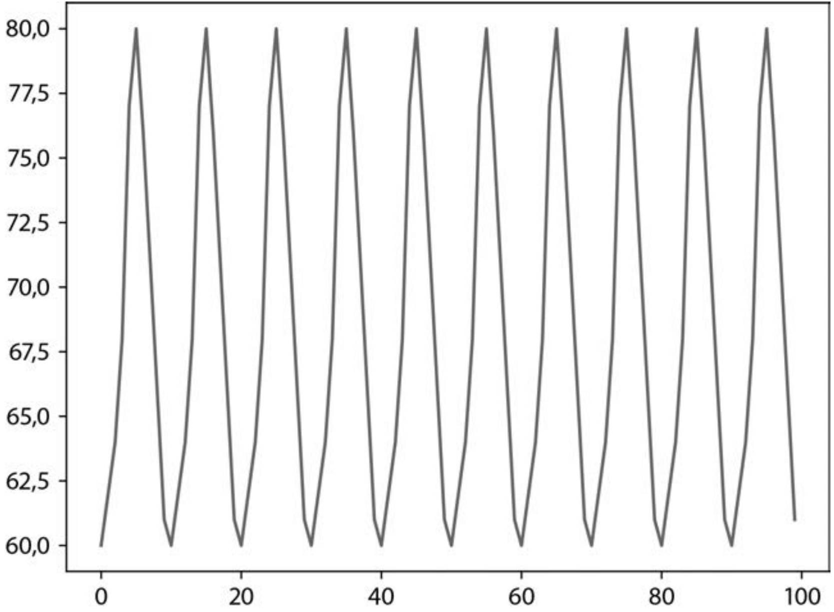
Kup książkę Poleć książkę

Analiza danych dotyczących pracy serca za pomocą konkatenacji list

Z tej sekcji dowiesz się, jak używać konkatenacji list, aby wielokrotnie kopiować mniejsze listy i łączyć je w większą listę w celu wygenerowania danych cyklicznych.

Podstawy

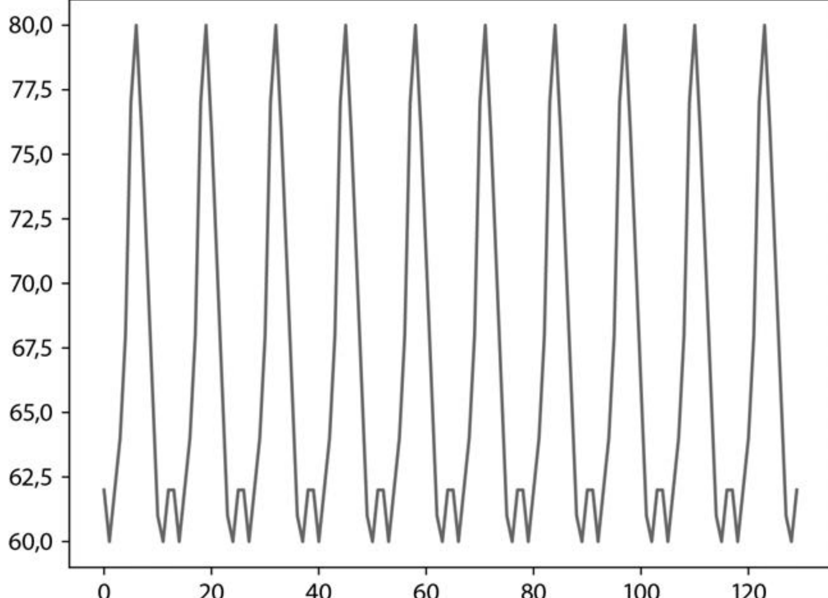
Tym razem pracujesz nad małym projektem informatycznym dla szpitala. Twoim celem jest monitorowanie i wizualizacja statystyk dotyczących zdrowia pacjentów poprzez śledzenie ich cykli pracy serca. Wykreślając oczekiwane dane dotyczące cyklu pracy serca, umożliwisz pacjentom i lekarzom monitorowanie wszelkich odchyleń od tego cyklu. Na przykład mając serię pomiarów przechowywanych na liście `[62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]` dla jednego cyklu pracy serca, chcesz uzyskać wizualizację przedstawioną na rysunku 2.2.



Rysunek 2.2. Wizualizacja oczekiwanych cykli pracy serca poprzez skopiowanie wybranych wartości z danych pomiarowych

Problem polega na tym, że pierwsza i ostatnie dwie wartości danych na liście są nadmiarowe: `[62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]`. Mogło to być przydatne przy wykreślaniu tylko jednego cyklu pracy serca, aby wskazać, że zwizualizowany został jeden pełny cykl. Musimy jednak pozbyć się tych nadmiarowych danych, aby przy kopiowaniu tego samego cyklu nasze oczekiwane cykle pracy serca nie wyglądały tak, jak na rysunku 2.3.

Kup ksi k Pole ksi k



Rysunek 2.3. Wizualizacja oczekiwanych cykli pracy serca poprzez skopiowanie wszystkich wartości z danych pomiarowych (bez odfiltrowania danych nadmiarowych)

Oczywiście musisz *oczyścić* oryginalną listę, usuwając pierwszą i ostatnie dwie nadmiarowe wartości danych: lista `[62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]` zmieni się w `[60, 62, 64, 68, 77, 80, 76, 71, 66, 61]`.

Połączysz wycinanie z nową funkcjonalnością Pythona, czyli **konkatenacją list**, która tworzy nową listę poprzez *konkatenację* (czyli *łączenie*) istniejących list. Przykładowo operacja `[1, 2, 3] + [4, 5]` wygeneruje nową listę `[1, 2, 3, 4, 5]`, nie zastępując jednak oryginalnych list. Można tu użyć operatora *, aby przeprowadzić wielokrotną konkatenację *tej samej listy* w celu utworzenia większej listy: np. operacja `[1, 2, 3] * 3` wygeneruje nową listę `[1, 2, 3, 1, 2, 3, 1, 2, 3]`.

Ponadto do wygenerowania danych dotyczących pracy serca użyjesz modułu `matplotlib.pyplot`. Funkcja `plot(dane)` biblioteki matplotlib oczekuje argumentów w postaci iterowalnych danych (obiekt **iterowalny** to taki, na którym można wykonać iterację, np. lista) i wykorzystuje je jako wartości y dla kolejnych punktów danych na wykresie dwuwymiarowym. Przeanalizujmy to na przykładzie.

Kod

Mając do dyspozycji listę liczb całkowitych, które odzwierciedlają pomiar cyklu pracy serca, chcesz najpierw oczyścić dane, usuwając z listy pierwszą i ostatnie dwie wartości. Następnie tworzysz nową listę z oczekiwanym przyszłym rytmem serca, kopiując cykl pracy serca do przyszłych wartości czasowych. Kod widoczny jest na listingu 2.8.

Kup ksi k Pole ksi k

Listing 2.8. Jednowierszowe rozwiązanie do przewidywania rytmu serca na przestrzeni czasu

```
## Zależności
import matplotlib.pyplot as plt

## Dane
cardiac_cycle = [62, 60, 62, 64, 68, 77, 80, 76, 71, 66, 61, 60, 62]

## Jednowierszowiec
expected_cycles = cardiac_cycle[1:-2] * 10

## Wynik
plt.plot(expected_cycles)
plt.show()
```

Wyjaśnię teraz, jaki jest wynik działania tego fragmentu kodu.

Jak to działa?

Ten jednowierszowiec składa się z dwóch kroków. Po pierwsze, oczyszczasz dane za pomocą wycinania, używając ujemnego argumentu końca `-2`, aby wyciąć wszystko aż do prawej strony, ale z pominięciem dwóch ostatnich nadmiarowych wartości. Po drugie, dokonujesz dziesięciokrotnej konkatenacji wynikowych wartości danych za pomocą operatora replikacji `*`. Wynikiem jest lista $10 \cdot 10 = 100$ liczb całkowitych składających się na połączone dane cyklu pracy serca. Po wykreśleniu wyniku otrzymujemy pożądane wyjście pokazane wcześniej na rysunku 2.2.

Użycie wyrażeń generatora do wyszukania firm, które płacą poniżej płacy minimalnej

Ta sekcja zawiera niektóre z podstaw Pythona, które już znasz, i wprowadza użyteczną funkcję `any()`.

Podstawy

Pracujesz w Państwowej Inspekcji Pracy, szukając firm, które płacą stawki niższe od minimalnych, aby móc przeprowadzić w nich kontrolę. Inspektorzy pracy, jak wygłodniałe psy za ciężarówką z mięsem, czekają już na listę firm, które naruszyły przepisy o płacy minimalnej. Możesz im ją dać?

Oto twoja broń: funkcja Pythona `any()`, która przyjmuje obiekt iterowalny, taki jak lista, i zwraca wartość `True`, jeśli przynajmniej jeden z elementów obiektu iterowalnego zostanie oceniony jako `True`. Na przykład wyrażenie `any([True, False, False, False])` zostanie ocenione jako `True`, podczas gdy wyrażenie `any([2<1, 3+2>5+5, 3-2<0, 0])` zostanie ocenione jako `False`.

Kup ksi k Pole ksi k

WSKAZÓWKA *Twórca Pythona, Guido van Rossum, był wielkim fanem wbudowanej funkcji* `any()` *i zaproponował nawet włączenie jej jako wbudowanej funkcji do Pythona 3. Więcej szczegółów można znaleźć na jego blogu we wpisie z 2005 roku,* The Fate of reduce() in Python 3000, *pod adresem* https://www.artima.com/weblogs/viewpost.jsp?thread=98196.

Ciekawym rozszerzeniem Pythona jest uogólnienie listy składanej: wyrażenia generatora. **Wyrażenia generatora** (ang. *generator expressions*) działają dokładnie tak samo jak listy składane — ale bez tworzenia rzeczywistej listy w pamięci. Liczby są tworzone w locie, bez przechowywania ich bezpośrednio na liście. Przykładowo zamiast używać listy składanej do obliczania sumy kwadratów pierwszych 20 liczb, `sum([x*x for x in range(20)])`, możesz użyć wyrażenia generatora: `sum(x*x for x in range(20))`.

Kod

Nasze dane to słownik słowników zawierających stawki godzinowe pracowników firm. Chcesz wyodrębnić listę firm płacących poniżej minimalnej stawki godzinowej (< 17 zł) przynajmniej jednemu pracownikowi (zobacz listing 2.9).

Listing 2.9. Jednowierszowe rozwiązanie umożliwiające znalezienie firm, które płacą poniżej stawki minimalnej

```
## Dane
companies = {
    'FajnaFirma' : {'Alicja' : 62, 'Robert' : 53, 'Franciszek' : 55},
    'SkąpaFirma' : {'Anna' : 8, 'Leszek' : 17, 'Krystyna' : 13},
    'TakaSobieFirma' : {'Elżbieta' : 72, 'Karol' : 15, 'Paweł' : 34}}

## Jednowierszowiec
illegal = [x for x in companies if any(y<17 for y in companies[x].values())]

## Wynik
print(illegal)
```

W których firmach należy przeprowadzić kontrolę?

Jak to działa?

W tym jednowierszowcu użyte zostały dwa wyrażenia generatora.

Pierwsze wyrażenie generatora, `y<17 for y in companies[x].values()`, generuje wejście dla funkcji `any()`. Sprawdza ono, czy pracownicy poszczególnych firm są opłacani poniżej płacy minimalnej, `y<17`. Wynik jest obiektem iterowalnym zawierającym wartości typu logicznego. Została tu użyta funkcja słownika `values()`, aby zwrócić kolekcję wartości przechowywanych w słowniku. Na przykład wyrażenie `companies['FajnaFirma'].values()` zwraca kolekcję stawek godzinowych `dict_values([62, 53, 55])`. Jeśli przynajmniej jedna będzie niższa od stawki minimalnej, funkcja `any()` zwróci wartość `True`, a nazwa firmy `x` będzie przechowywana jako łańcuch znaków w wynikowej liście `illegal`, jak opisałem poniżej.

Kup ksi k Pole ksi k

Drugie wyrażenie generatora jest listą składaną `[x for x in companies if any(...)]` i tworzy listę nazw firm, dla których poprzednie wywołanie funkcji `any()` zwraca wartość `True`. Są to firmy, które płacą poniżej stawki minimalnej. Zauważ, że wyrażenie `for x in companies` sprawdza wszystkie klucze słownika — nazwy firm `'FajnaFirma'`, `'SkąpaFirma'` i `'TakaSobieFirma'`.

Wynik jest zatem następujący:

```
## Wynik
print(illegal)
> ['SkąpaFirma', 'TakaSobieFirma']
```

Dwie z trzech firm muszą zostać skontrolowane, ponieważ płacą zbyt mało przynajmniej jednemu pracownikowi. Twoi inspektorzy mogą zacząć rozmawiać z Anną, Krystyną i Karolem!

Formatowanie baz danych za pomocą funkcji zip()

Z tej sekcji dowiesz się, jak zastosować nazwy kolumn bazy danych do listy wierszy za pomocą funkcji `zip()`.

Podstawy

Funkcja `zip()` pobiera iterowalne obiekty `iter_1`, `iter_2`, ..., `iter_n` i łączy je w pojedynczy obiekt iterowalny, zestawiając odpowiednie *i*-te wartości w pojedynczą krotkę. Wynikiem jest *iterowalny obiekt* składający się z krotek. Na przykład spójrz na te dwie listy:

```
[1,2,3]
[4,5,6]
```

Jeśli zestawisz je ze sobą — po prostym skonwertowaniu danych, co zobaczysz za chwilę — otrzymasz nową listę:

```
[(1,4), (2,5), (3,6)]
```

Przywrócenie jej do pierwotnego stanu wymaga dwóch kroków. Najpierw należy usunąć zewnętrzne nawiasy kwadratowe listy wynikowej, aby uzyskać następujące trzy krotki:

```
(1,4)
(2,5)
(3,6)
```


Kup ksi k Pole ksi k

Następnie, gdy zestawisz je ponownie, otrzymasz nową listę:

```
[(1,2,3), (4,5,6)]
```

Masz więc znowu swoje dwie oryginalne listy! Poniższy fragment kodu pokazuje ten proces w całości:

```
lst_1 = [1, 2, 3]
lst_2 = [4, 5, 6]

# Połączenie dwóch list
zipped = list(zip(lst_1, lst_2))
print(zipped)
> [(1, 4), (2, 5), (3, 6)]

# Przywrócenie poprzednich list
lst_1_new, lst_2_new = zip(❶*zipped)
print(list(lst_1_new))
print(list(lst_2_new))
```

Aby rozpakować ❶ wszystkie elementy listy, należy użyć operatora `*`. Usuwa on zewnętrzne nawiasy kwadratowe listy `zipped` tak, że wejście do funkcji `zip()` składa się z trzech iterowalnych obiektów (krotek `(1, 4)`, `(2, 5)`, `(3, 6)`). Jeśli zestawisz je ze sobą, otrzymasz dwie nowe krotki, z których jedna będzie zawierać trzy pierwsze wartości poprzednich krotek (`1`, `2` i `3`), a kolejna trzy drugie wartości krotek (`4`, `5` i `6`). W efekcie otrzymasz iterowalne obiekty `(1, 2, 3)` i `(4, 5, 6)`, które są oryginalnymi (przywróconymi) danymi.

Teraz wyobraź sobie, że pracujesz w komórce IT działu kontrolingu swojej firmy. Prowadzisz bazę danych wszystkich pracowników z nazwami kolumn: `'imię'`, `'wynagrodzenie'` i `'stanowisko'`. Twoje dane mają jednak nieodpowiednią formę — jest to zbiór wierszy w postaci (`'Robert'`, `99000`, `'menedżer średniego szczebla'`). Chcesz powiązać swoje nazwy kolumn z każdym wpisem danych, aby nadać im czytelną postać `{'imię': 'Robert', 'wynagrodzenie': 99000, 'stanowisko': 'menedżer średniego szczebla'}`. Jak to osiągnąć?

Kod

Twoje dane składają się z nazw kolumn i danych pracowników zorganizowanych w postaci listy krotek (wierszy). Przypisz nazwy kolumn do wierszy, tworząc w ten sposób listę słowników. Każdy słownik przyporządkowuje nazwy kolumn do odpowiednich wartości danych (zobacz listing 2.10).

Listing 2.10. Jednowierszowe rozwiązanie umożliwiające zastosowanie formatu bazy danych do listy krotek

```
## Dane
column_names = ['imię', 'wynagrodzenie', 'stanowisko']
db_rows = [('Alicja', 180000, 'analityk danych'),
```


Kup ksi k Pole ksi k

```
            ('Robert', 99000, 'menedżer średniego szczebla'),
            ('Franciszek', 87000, 'dyrektor generalny')]

## Jednowierszowiec
db = [dict(zip(column_names, row)) for row in db_rows]

## Wynik
print(db)
```

Jaki będzie wyjściowy format bazy danych `db`?

Jak to działa?

Tworzysz listę przy użyciu listy składanej (więcej informacji na temat wyrażenia i kontekstu znajduje się w sekcji „Użycie listy składanej do wyszukiwania osób o najwyższych dochodach”). Kontekst składa się z krotki każdego wiersza w zmiennej `db_rows`. Wyrażenie `zip(column_names, row)` zestawia ze sobą schemat nazw i każdy wiersz. Na przykład pierwszym elementem utworzonym przez listę składaną będzie `zip(['imię', 'wynagrodzenie', 'stanowisko'], ('Alicja', 180000, 'analityk danych'))`, co po zestawieniu da obiekt, który po konwersji na listę przyjmie postać `[('imię', 'Alicja'), ('wynagrodzenie', 180000), ('stanowisko', 'analityk danych')]`. Elementy mają postać (*`klucz`*, *`wartość`*), więc możesz przekonwertować je do słownika za pomocą funkcji konwertera `dict()`, aby uzyskać wymagany format bazy danych.

WSKAZÓWKA *Dla funkcji `zip()` nie ma znaczenia, czy wejście jest listą czy krotką. Wymaga ona tylko, aby wejście było obiektem iterowalnym (a takim jest zarówno lista, jak i krotka).*

Oto wyjście tego fragmentu kodu:

```
## Wynik
print(db)
> [{'imię': 'Alicja', 'wynagrodzenie': 180000, 'stanowisko': 'analityk danych'},
↪{'imię': 'Robert', 'wynagrodzenie': 99000, 'stanowisko': 'menedżer średniego
↪szczebla'}, {'imię': 'Franciszek', 'wynagrodzenie': 87000, 'stanowisko':
↪'dyrektor generalny'}]
```

Podsumowanie

W tym rozdziale opanowałeś listy składane, wczytywanie plików, funkcje `lambda`, `map()`, i `zip()`, kwantyfikator `all()`, wycinanie i podstawowe operacje na listach. Nauczyłeś się również posługiwać strukturami danych i operować na nich w celu rozwiązywania różnych praktycznych problemów.

Łatwe konwertowanie struktur danych i przywracanie ich do poprzedniej postaci to umiejętność, która ma ogromny wpływ na wydajność kodowania. Możesz mieć pewność, że Twoja efektywność programowania wzrośnie, gdy zwiększysz umiejęt-

Kup ksi k Pole ksi k

ności szybkiego operowania na danych. Małe zadania związane z przetwarzaniem danych, takie jak te, które widziałeś w tym rozdziale, znacząco przyczyniają się do powszechnej „kary tysiąca cięć": przytłaczającej szkody, jaką wykonywanie wielu małych zadań ma dla Twojej ogólnej wydajności. Dzięki sztuczkom, funkcjom i właściwościom Pythona, które wprowadziłem w tym rozdziale, uzyskałeś skuteczną ochronę przed tymi tysiącami cięć. Mówiąc metaforycznie, nowo nabyte narzędzia pomogą Ci znacznie szybciej odzyskać siły po każdym cięciu.

W następnym rozdziale jeszcze bardziej rozwiniesz swoje umiejętności z zakresu analizy danych, poznając nowy zestaw narzędzi dostarczany przez bibliotekę NumPy do obliczeń numerycznych w Pythonie.

ROZWIĄZANIE ĆWICZENIA 2.1

Oto sposób, w jaki można wykorzystać listę składaną zamiast funkcji `map()`, aby rozwiązać ten sam problem odfiltrowania wszystkich wierszy, które zawierają łańcuch znaków `'anonimowe'`. W tym przypadku proponuję zastosować szybszą i bardziej przejrzystą funkcję listy składanej.

```
mark = [(True, s) if 'anonimowe' in s else (False, s) for s in txt]
```


Kup ksi k

Pole ksi k

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

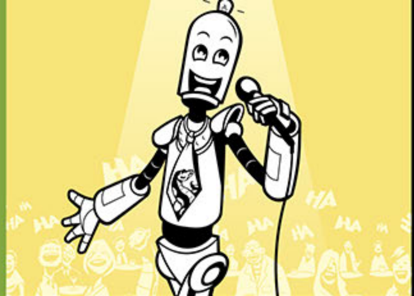
GRUPY HELION

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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl



ZEN PYTHONA: LICZY SIĘ CZYTELNOŚĆ!

W Pythonie najlepsza jest wyrazistość, zwięzłość i interaktywność kodu. To są właśnie cechy kodu idealnego: każdy bit powinien być dziełem sztuki. Python pozwala na wyrażenie jednej myśli na wiele sposobów. Mistrz programowania zna je wszystkie i w konkretnej sytuacji wybiera ten najbardziej trafny i zwięzły. Rozwiązywanie złożonych problemów programistycznych za pomocą jednego wiersza kodu daje przecież dużą satysfakcję. Osiągnięcie tak wyrafinowanych umiejętności wymaga jednak pokonania wielu drobnych trudności, jakie napotyka osoba ucząca się Pythona.

Ta książka ułatwi Ci naukę czytania i pisania zwięzłych, użytecznych instrukcji zajmujących jeden wiersz kodu. Nauczysz się także systematycznie rozkładać dowolny blok kodu na części pierwsze. Stopniowo Twój kod Pythona będzie się stawał piękny, zwięzły i prosty. Znajdziesz tu liczne wskazówki, zapoznasz się też z takimi zagadnieniami jak uczenie maszynowe, podstawy nauki o danych i użyteczne algorytmy. Niejako przy okazji poznasz i zrozumiesz kluczowe pojęcia z dziedziny informatyki i zaczniesz używać zaawansowanych funkcji Pythona, takich jak listy składane, wycinanie, funkcje lambda, wyrażenia regularne, funkcje map i reduce oraz przypisywanie do wycinków.

W książce między innymi:

- stosowanie struktur danych do rozwiązywania rzeczywistych problemów
- podstawowe funkcjonalności biblioteki NumPy
- statystyki wielowymiarowych tablic danych
- algorytm *k*-średnich w uczeniu nienadzorowanym
- zaawansowane wyrażenia regularne
- anagramy, palindromy, zbiory potęgowe, permutacje, silnie, liczby pierwsze, ciąg Fibonacciego, zatajanie tekstu, wyszukiwanie i sortowanie algorytmiczne

Dr Christian Mayer jest pasjonatem Pythona. Założył i prowadzi popularną stronę poświęconą Pythonowi (*https://blog.finxter.com/*). Jego wybitne umiejętności przekazywania wiedzy są doceniane przez tysiące adeptów tego języka programowania. Jest autorem książek z serii „Coffee Break Python”.