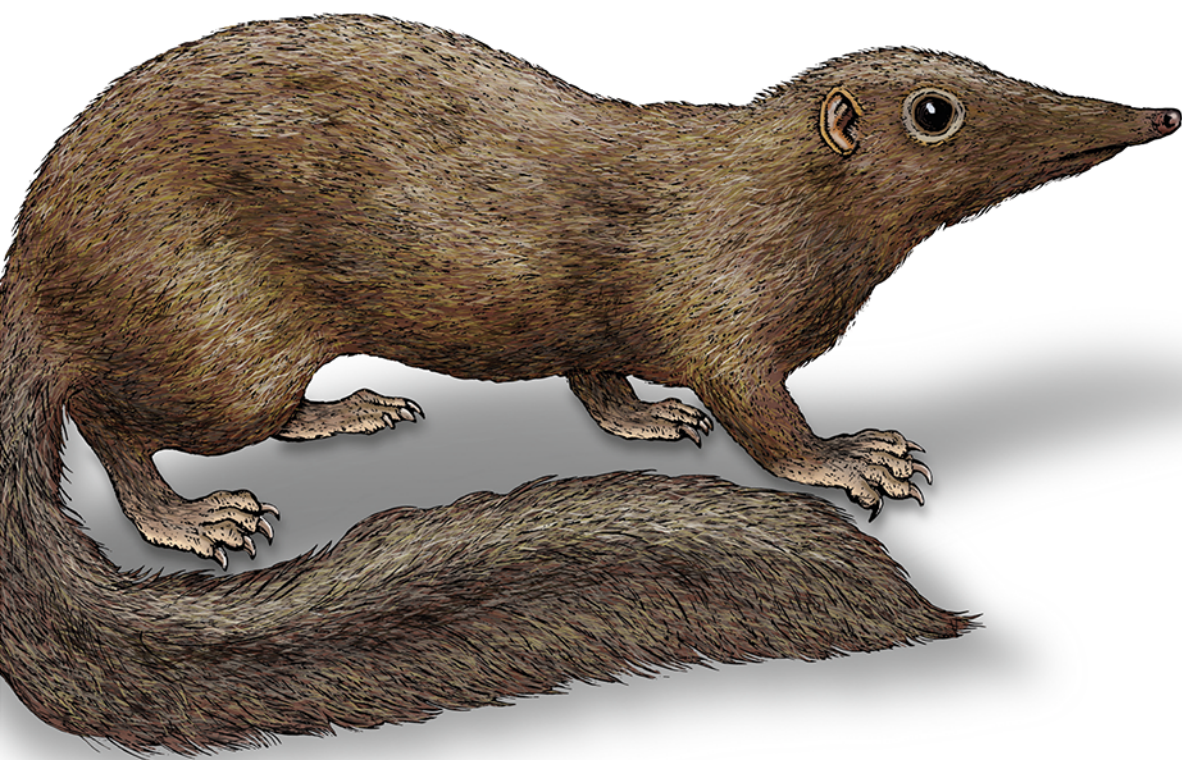


O'REILLY®

Wydanie III

# Python w analizie danych

Przetwarzanie danych za pomocą  
pakietów pandas i NumPy oraz środowiska Jupyter



Helion

Wes McKinney

Tytuł oryginału: Python for Data Analysis: Data Wrangling with pandas, NumPy,  
and Jupyter, 3<sup>rd</sup> Edition

Tłumaczenie: Andrzej Watrak

ISBN: 978-83-8322-323-0

© 2023 Helion S.A.

Authorized Polish translation of the English edition of *Python for Data Analysis, 3<sup>rd</sup> Edition*  
ISBN 9781098104030 © 2022 Wesley McKinney.

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

Polish edition copyright © 2023 by Helion S.A.

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 jakiegokolwiek 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 wydawca 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 wydawca nie ponoszą również  
żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<https://helion.pl/user/opinie/pyanda>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<https://ftp.helion.pl/przyklady/pyanda.zip>

Helion S.A.

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 230 98 63

e-mail: [helion@helion.pl](mailto:helion@helion.pl)

WWW: <https://helion.pl> (księgarnia internetowa, katalog książek)

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

---

# Spis treści

<b>Przedmowa .....</b>	<b>11</b>
<b>1. Wstęp .....</b>	<b>17</b>
1.1. O czym jest ta książka?	17
Jakie rodzaje danych?	17
1.2. Dlaczego warto korzystać z Pythona w celu przeprowadzenia analizy danych?	18
Python jako spoiwo	18
Rozwiązywanie problemu „dwujęzyczności”	19
Dlaczego nie Python?	19
1.3. Podstawowe biblioteki Pythona	20
NumPy	20
pandas	21
Matplotlib	22
IPython i Jupyter	22
SciPy	23
Scikit-learn	23
statsmodels	24
Inne pakiety	25
1.4. Instalacja i konfiguracja	25
Windows	25
GNU, Linux	26
macOS	26
Instalacja niezbędnych pakietów	27
Zintegrowane środowiska programistyczne i edytory tekstowe	28
1.5. Społeczność i konferencje	28
1.6. Nawigacja po książce	29
Przykłady kodu	30
Przykładowe dane	30
Konwencje importowania	31

<b>2. Podstawy Pythona oraz obsługi narzędzi IPython i Jupyter .....</b>	<b>32</b>
2.1. Interpreter Pythona	33
2.2. Podstawy interpretera IPython	34
Uruchamianie powłoki IPython	34
Uruchamianie notatnika Jupyter Notebook	35
Uzupełnianie poleceń	38
Introspekcja	39
2.3. Podstawy Pythona	40
Semantyka języka Python	40
Skalarne typy danych	48
Przepływ sterowania	55
2.4. Podsumowanie	58
<b>3. Wbudowane struktury danych, funkcje i pliki .....</b>	<b>59</b>
3.1. Struktury danych i sekwencje	59
Krotka	59
Lista	62
Słownik	66
Zbiór	70
Wbudowane funkcje obsługujące sekwencje	72
Lista, słownik i zbiór — składanie	74
3.2. Funkcje	76
Przestrzenie nazw, zakres i funkcje lokalne	77
Zwracanie wielu wartości	78
Funkcje są obiektami	79
Funkcje anonimowe (lambda)	80
Generatory	81
Błędy i obsługa wyjątków	83
3.3. Pliki i system operacyjny	86
Bajty i kodowanie Unicode w plikach	89
3.4. Podsumowanie	91
<b>4. Podstawy biblioteki NumPy: obsługa tablic i wektorów .....</b>	<b>92</b>
4.1. NumPy ndarray — wielowymiarowy obiekt tablicowy	94
Tworzenie tablic ndarray	95
Typ danych tablic ndarray	97
Działania matematyczne z tablicami NumPy	100
Podstawy indeksowania i przechwytywania części	101
Indeksowanie i wartości logiczne	106
Indeksowanie specjalne	108
Transponowanie tablic i zamiana osi	110

4.2.	Generowanie liczb pseudolosowych	111
4.3.	Funkcje uniwersalne — szybkie funkcje wykonywane na poszczególnych elementach tablicy	113
4.4.	Programowanie z użyciem tablic	115
	Logiczne operacje warunkowe jako operacje tablicowe	117
	Metody matematyczne i statystyczne	119
	Metody tablic logicznych	120
	Sortowanie	121
	Wartości unikalne i operacje logiczne	122
4.5.	Tablice i operacje na plikach	123
4.6.	Algebra liniowa	124
4.7.	Przykład: błędzenie losowe	125
	Jednoczesne symulowanie wielu błędzeń losowych	127
4.8.	Podsumowanie	128
<b>5.</b>	<b>Rozpoczynamy pracę z biblioteką pandas .....</b>	<b>129</b>
5.1.	Wprowadzenie do struktur danych biblioteki pandas	130
	Obiekt Series	130
	Obiekt DataFrame	134
	Obiekty index	141
5.2.	Podstawowe funkcjonalności	142
	Uaktualnianie indeksu	143
	Odrzucanie elementów osi	145
	Indeksowanie, wybieranie i filtrowanie	147
	Działania arytmetyczne i wyrównywanie danych	156
	Funkcje apply i map	161
	Sortowanie i tworzenie rankingów	163
	Indeksy osi ze zduplikowanymi etykietami	167
5.3.	Podsumowywanie i generowanie statystyk opisowych	168
	Współczynnik korelacji i kowariancja	171
	Unikalne wartości, ich liczba i przynależność	173
5.4.	Podsumowanie	176
<b>6.</b>	<b>Odczyt i zapis danych, formaty plików .....</b>	<b>177</b>
6.1.	Odczyt i zapis danych w formacie tekstowym	177
	Wczytywanie części pliku tekstowego	184
	Zapis danych w formacie tekstowym	185
	Praca z plikami danych rozgraniczonych	187
	Dane w formacie JSON	189
	XML i HTML — web scraping	190

6.2.	Formaty danych binarnych	194
	Wczytywanie plików programu Microsoft Excel	195
	Obsługa formatu HDF5	196
6.3.	Obsługa interfejsów sieciowych	198
6.4.	Obsługa baz danych	200
6.5.	Podsumowanie	202
<b>7.</b>	<b>Czyszczenie i przygotowywanie danych .....</b>	<b>203</b>
7.1.	Obsługa brakujących danych	203
	Filtrowanie brakujących danych	205
	Wypełnianie brakujących danych	207
7.2.	Przekształcanie danych	209
	Usuwanie duplikatów	209
	Przekształcanie danych przy użyciu funkcji lub mapowania	210
	Zastępowanie wartości	212
	Zmiana nazw indeksów osi	213
	Dyskretyzacja i podział na koszyki	214
	Wykrywanie i filtrowanie elementów odstających	216
	Permutacje i próbkowanie losowe	218
	Przetwarzanie wskaźników i zmiennych zastępczych	219
7.3.	Rozszerzone typy danych	222
7.4.	Operacje przeprowadzane na łańcuchach	225
	Metody obiektu typu string	225
	Wyrażenia regularne	227
	Funkcje tekstowe w pakiecie pandas	230
7.5.	Dane kategoriczne	233
	Kontekst i motywacja	233
	Rozszerzony typ Categorical w bibliotece pandas	235
	Obliczenia na obiektach typu Categorical	237
	Metody obiektu kategoriicznego	239
7.6.	Podsumowanie	242
<b>8.</b>	<b>Przetwarzanie danych — operacje łączenia, wiązania i przekształcania .....</b>	<b>243</b>
8.1.	Indeksowanie hierarchiczne	243
	Zmiana kolejności i sortowanie poziomów	246
	Parametry statystyki opisowej z uwzględnieniem poziomu	247
	Indeksowanie z kolumnami ramki danych	247
8.2.	Łączenie zbiorów danych	249
	Łączenie ramek danych w stylu łączenia elementów baz danych	249
	Łączenie przy użyciu indeksu	253
	Konkatenacja wzdłuż osi	258
	Łączenie częściowo nakładających się danych	263

8.3.	Zmiana kształtu i operacje osiowe	264
	Przekształcenia z indeksowaniem hierarchicznym	264
	Przekształcanie z formatu „długiego” na „szeroki”	267
	Przekształcanie z formatu „szerokiego” na „długi”	270
8.4.	Podsumowanie	272
<b>9.</b>	<b>Wykresy i wizualizacja danych .....</b>	<b>273</b>
9.1.	Podstawy obsługi interfejsu pakietu matplotlib	274
	Obiekty figure i wykresy składowe	275
	Kolory, oznaczenia i style linii	279
	Punkty, etykiety i legendy	281
	Adnotacje i rysunki	283
	Zapisywanie wykresów w postaci plików	286
	Konfiguracja pakietu matplotlib	287
9.2.	Generowanie wykresów za pomocą pakietów pandas i seaborn	287
	Wykresy liniowe	287
	Wykresy słupkowe	290
	Histogramy i wykresy gęstości	296
	Wykresy punktowe	298
	Wykresy panelowe i dane kategoryczne	300
9.3.	Inne narzędzia przeznaczone do wizualizacji danych w Pythonie	301
9.4.	Podsumowanie	303
<b>10.</b>	<b>Agregacja danych i operacje wykonywane na grupach .....</b>	<b>304</b>
10.1.	Mechanika interfejsu groupby	305
	Iteracja po grupach	308
	Wybieranie kolumny lub podzbioru kolumn	310
	Grupowanie przy użyciu słowników i serii	311
	Grupowanie przy użyciu funkcji	312
	Grupowanie przy użyciu poziomów indeksu	312
10.2.	Agregacja danych	313
	Przetwarzanie kolumna po kolumnie i stosowanie wielu funkcji	315
	Zwracanie zagregowanych danych bez indeksów wierszy	318
10.3.	Metoda apply — ogólne zastosowanie techniki dziel-zastosuj-połącz	319
	Usuwanie kluczy grup	321
	Kwantyle i analiza koszykowa	321
	Przykład: wypełnianie brakujących wartości przy użyciu wartości charakterystycznych dla grupy	323
	Przykład: losowe generowanie próbek i permutacja	325
	Przykład: średnie ważone grup i współczynnik korelacji	327
	Przykład: regresja liniowa grup	329
10.4.	Transformacje grup i „nieobudowane” operacje grupowania	330

10.5. Tabele przestawne i krzyżowe	333
Tabele krzyżowe	336
10.6. Podsumowanie	337
<b>11. Szeregi czasowe .....</b>	<b>338</b>
11.1. Typy danych i narzędzia przeznaczone do obsługi daty i czasu	339
Konwersja pomiędzy obiektami string i datetime	340
11.2. Podstawy szeregów czasowych	342
Indeksowanie i wybieranie	343
Szeregi czasowe z duplikatami indeksów	346
11.3. Zakresy dat, częstotliwości i przesunięcia	347
Generowanie zakresów dat	347
Częstotliwości i przesunięcia daty	350
Przesuwanie daty	351
11.4. Obsługa strefy czasowej	354
Lokalizacja i konwersja stref czasowych	355
Operacje z udziałem obiektów Timestamp o wyznaczonej strefie czasowej	357
Operacje pomiędzy różnymi strefami czasowymi	358
11.5. Okresy i przeprowadzanie na nich operacji matematycznych	358
Konwersja częstotliwości łańcuchów	359
Kwartalne częstotliwości okresów	361
Konwersja znaczników czasu na okresy (i z powrotem)	362
Tworzenie obiektów PeriodIndex na podstawie tablic	364
11.6. Zmiana rozdzielczości i konwersja częstotliwości	365
Zmniejszanie częstotliwości	366
Zwiększanie rozdzielczości i interpolacja	369
Zmiana rozdzielczości z okresami	371
Grupowa zmiana częstotliwości	372
11.7. Funkcje ruchomego okna	374
Funkcje ważone wykładniczo	377
Binarne funkcje ruchomego okna	378
Funkcje ruchomego okna definiowane przez użytkownika	380
11.8. Podsumowanie	381
<b>12. Wprowadzenie do bibliotek modelujących .....</b>	<b>382</b>
12.1. Łączenie pandas z kodem modelu	382
12.2. Tworzenie opisów modeli przy użyciu biblioteki Patsy	385
Przekształcenia danych za pomocą formuł Patsy	387
Patsy i dane katagoryczne	389
12.3. Wprowadzenie do biblioteki statsmodels	391
Szacowanie modeli liniowych	392
Szacowanie procesów szeregów czasowych	394



12.4. Wprowadzenie do pakietu scikit-learn	395
12.5. Podsumowanie	399
<b>13. Przykłady analizy danych .....</b>	<b>400</b>
13.1. Dane USA.gov serwisu Bitly	400
Liczenie stref czasowych w czystym Pythonie	401
Liczenie stref czasowych przy użyciu pakietu pandas	403
13.2. Zbiór danych MovieLens 1M	409
Wyznaczenie rozbieżności ocen	413
13.3. Imiona nadawane dzieciom w USA w latach 1880 – 2010	416
Analiza trendów imion	421
13.4. Baza danych USDA Food	430
13.5. Baza danych 2012 Federal Election Commission	436
Statystyki datków z podziałem na wykonywany zawód i pracodawcę	438
Podział kwot datków na koszyki	441
Statystyki datków z podziałem na poszczególne stany	443
13.6. Podsumowanie	444
<b>A. Zaawansowane zagadnienia związane z biblioteką NumPy .....</b>	<b>445</b>
A.1. Szczegóły budowy obiektu ndarray	445
Hierarchia typów danych NumPy	446
A.2. Zaawansowane operacje tablicowe	447
Zmiana wymiarów tablic	448
Kolejności charakterystyczne dla języków C i Fortran	449
Łączenie i dzielenie tablic	450
Powtarzanie elementów — funkcje tile i repeat	452
Alternatywy indeksowania specjalnego — metody take i put	454
A.3. Rozgłaszanie	455
Rozgłaszanie wzdłuż innych osi	457
Przypisywanie wartości elementom tablicy poprzez rozgłaszanie	460
A.4. Zaawansowane zastosowania funkcji uniwersalnych	461
Metody instancji funkcji uniwersalnych	461
Pisanie nowych funkcji uniwersalnych w Pythonie	463
A.5. Tablice o złożonej strukturze	464
Zagnieżdżone typy danych i pola wielowymiarowe	464
Do czego przydają się tablice o złożonej strukturze?	465
A.6. Jeszcze coś o sortowaniu	466
Sortowanie pośrednie — metody argsort i lexsort	467
Alternatywne algorytmy sortowania	468
Częściowe sortowanie tablic	469
Wyszukiwanie elementów w posortowanej tablicy za pomocą metody numpy.searchsorted	470

A.7.	Pisanie szybkich funkcji NumPy za pomocą pakietu Numba	471
	Tworzenie obiektów <code>numpy.ufunc</code> za pomocą pakietu Numba	472
A.8.	Zaawansowane tablicowe operacje wejścia i wyjścia	473
	Pliki mapowane w pamięci	473
	HDF5 i inne możliwości zapisu tablic	474
A.9.	Jak zachować wysoką wydajność?	475
	Dlaczego warto korzystać z sąsiadujących ze sobą obszarów pamięci?	475
<b>B.</b>	<b>Dodatkowe informacje dotyczące systemu IPython .....</b>	<b>477</b>
B.1.	Skróty klawiaturowe	477
B.2.	Magiczne polecenia	478
	Polecenie <code>%run</code>	480
	Uruchamianie kodu zapisanego w schowku	481
B.3.	Korzystanie z historii poleceń	482
	Przeszukiwanie i korzystanie z historii poleceń	482
	Zmienne wejściowe i wyjściowe	483
B.4.	Interakcja z systemem operacyjnym	484
	Polecenia powłoki systemowej i aliasy	484
	System tworzenia skrótów do katalogów	485
B.5.	Narzędzia programistyczne	486
	Interaktywny debugger	486
	Pomiar czasu — funkcje <code>%time</code> i <code>%timeit</code>	490
	Podstawowe profilowanie — funkcje <code>%prun</code> i <code>%run-p</code>	492
	Profilowanie funkcji linia po linii	494
B.6.	Wskazówki dotyczące produktywnego tworzenia kodu w środowisku IPython	496
	Przeładowywanie modułów	496
	Wskazówki dotyczące projektowania kodu	497
B.7.	Zaawansowane funkcje środowiska IPython	498
	Profile i konfiguracja	498
B.8.	Podsumowanie	500

# Czyszczenie i przygotowywanie danych

Podczas analizowania i modelowania danych znaczną część czasu poświęca się na przygotowanie danych: ładowanie, czyszczenie, przekształcanie i przekładanie. Wykonywanie tego typu zadań zajmuje nawet 80% czasu pracy analityka. Czasami sposób przechowywania danych w pliku lub w bazie danych jest niewłaściwy z punktu widzenia zadania, które ma wykonać analityk. Wielu badaczy decyduje się na doraźne przetwarzanie danych z jednej formy na drugą za pomocą języka programowania ogólnego przeznaczenia, takiego jak Python, Perl, R, Java, lub narzędzi systemu Unix przeznaczonych do przetwarzania danych tekstowych, takich jak sed lub awk. Na szczęście pakiet pandas oraz elementy wbudowane w Pythona tworzą zestaw uniwersalnych i szybkich wysokopoziomowych narzędzi, które umożliwiają przekształcenie danych do właściwej formy.

Jeżeli potrafisz zdefiniować operację przekształcania danych, która nie została uwzględniona w tej książce lub nie jest obsługiwana w żaden sposób przez bibliotekę pandas, to podziel się nią, a także jej zastosowaniem na jednej z list mailingowych Pythona lub na stronie pakietu pandas znajdującej się w serwisie GitHub. Większość elementów pakietu pandas została zaprojektowana i zaimplementowana z myślą o potrzebie rozwiązywania realnych problemów.

W tym rozdziale zajmę się narzędziami przeznaczonymi do rozwiązywania problemów dotyczących brakujących czy zduplikowanych danych, a także narzędziami przetwarzającymi łańcuchy i wykonującymi inne przekształcenia zbioru danych. W kolejnym rozdziale skupię się na zagadnieniach dotyczących łączenia i modyfikowania zbiorów danych.

## 7.1. Obsługa brakujących danych

Z problemem brakujących danych spotykamy się w wielu zbiorach danych. Jednym z celów istnienia pakietu pandas jest maksymalne uproszczenie obsługi brakujących danych poprzez np. domyślne odrzucanie brakujących danych przez większość funkcji obliczających parametry statystyki opisowej.

Sposób reprezentacji brakujących danych zaimplementowany w pakiecie pandas można uznać za nieidealny, ale z punktu widzenia wielu użytkowników jest on praktyczny. W przypadku danych numerycznych pakiet pandas korzysta ze zmiennoprzecinkowej wartości NaN (nie-liczba) w celu oznaczenia brakujących danych. Jest to tzw. **wartość zastępcza**, której występowanie można łatwo wykryć:

```
In [14]: float_data = pd.Series([1.2, -3.5, np.nan, 0])
```

```
In [15]: float_data
```

```
Out[15]:
0    1.2
1   -3.5
2    NaN
3    0.0
dtype: float64
```

Metoda `isna` zwraca serię wartości logicznych, w której `True` oznacza brak wartości w oryginalnej serii:

```
In [16]: float_data.isna()
Out[16]:
0    False
1    False
2     True
3    False
dtype: bool
```

W pakiecie `pandas` przyjęliśmy konwencję zaczerpniętą z języka R — brakujące wartości określamy mianem wartości **NA** (wartości **niedostępnych**, z ang. *not available*). W zastosowaniach statystycznych dane niedostępne mogą być danymi, które nie istnieją, lub danymi, które istnieją, ale nie zostały zaobserwowane (np. z powodu problemu wynikającego ze sposobu zbierania danych). Podczas oczyszczania danych przed przeprowadzeniem ich analizy często warto przeprowadzić analizę samych brakujących wartości. Pozwoli to zidentyfikować problemy wynikające z techniki zbierania danych lub potencjalne tendencje spowodowane brakiem pewnych wartości.

W tablicach za wartość `NA` przyjmuje się wbudowaną w standardową bibliotekę Pythona wartość `None`:

```
In [17]: string_data = pd.Series(["aardvark", np.nan, None, "avocado"])
```

```
In [18]: string_data
Out[18]:
0    aardvark
1         NaN
2         None
3    avocado
dtype: object
```

```
In [19]: string_data.isna()
Out[19]:
0    False
1     True
2     True
3    False
dtype: bool
```

```
In [20]: float_data = pd.Series([1, 2, None], dtype='float64')
```

```
In [21]: float_data
Out[21]:
0    1.0
1    2.0
2    NaN
dtype: float64
```

```
In [22]: float_data.isna()
Out[22]:
0    False
```

```
1 False
2 True
dtype: bool
```

Twórcy projektu pandas starają się ujednolicić obsługę brakujących danych różnych typów. Funkcje takie jak `pandas.isna` ukrywają przed użytkownikiem wiele skomplikowanych szczegółów implementacyjnych. W tabeli 7.1 wymieniono wybrane funkcje przydatne podczas pracy z brakującymi danymi.

Tabela 7.1. Metody przydatne podczas obsługi brakujących danych

Argument	Opis
<code>dropna</code>	Filtruje etykiety osi na podstawie występowania brakujących danych; możliwe jest zdefiniowanie zmiennych wartości progowych określających liczbę tolerowanych brakujących danych.
<code>fillna</code>	Wypełnia brakujące dane jakimiś wartościami lub robi to za pomocą metody interpolacji, takiej jak np. <code>"ffill"</code> lub <code>"bfill"</code> .
<code>isna</code>	Zwraca wartości logiczne określające miejsce występowania brakujących wartości.
<code>notna</code>	Negacja <code>isna</code> . Zwraca <code>True</code> , jeżeli wartość istnieje, a <code>False</code> w przeciwnym wypadku.

## Filtrowanie brakujących danych

Istnieje kilka technik umożliwiających odfiltrowanie brakujących danych. Możesz to zrobić ręcznie za pomocą funkcji `pandas.isna` i indeksowania wartości logicznych, ale łatwiej jest skorzystać z funkcji `dropna`. W przypadku obiektu typu `Series` funkcja ta zwraca tylko dane o wartości innej niż `null` i indeksy odwołujące się jedynie do tych danych:

```
In [23]: data = pd.Series([1, np.nan, 3.5, np.nan, 7])

In [24]: data.dropna()
Out[24]:
0 1.0
2 3.5
4 7.0
dtype: float64
```

Jest to odpowiednik następującego kodu:

```
In [25]: data[data.notna()]
Out[25]:
0 1.0
2 3.5
4 7.0
dtype: float64
```

Sprawy nieco komplikują się podczas pracy z obiektami `DataFrame`. Możesz chcieć odrzucić wiersze lub kolumny, które zawierają tylko wartości NA, lub takie, które zawierają chociażby jedną wartość NA. Funkcja `dropna` domyślnie odrzuca wiersze zawierające przynajmniej jedną brakującą wartość:

```
In [26]: data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan],
.....:                        [np.nan, np.nan, np.nan], [np.nan, 6.5, 3.]])

In [27]: data
Out[27]:
0 1 2
```

```
0 1.0 6.5 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 6.5 3.0
```

```
In [28]: data.dropna()
Out[28]:
   0  1  2
0 1.0 6.5 3.0
```

Przekazanie parametru `how='all'` spowoduje odrzucenie tylko wierszy, które zawierają same wartości NA:

```
In [29]: data.dropna(how='all')
Out[29]:
   0  1  2
0 1.0 6.5 3.0
1 1.0 NaN NaN
3 NaN 6.5 3.0
```

Pamiętaj, że opisane funkcje domyślnie zwracają nowe obiekty, tj. nie modyfikują zawartości oryginalnych obiektów.

Aby odrzucić w ten sam sposób kolumny, należy skorzystać z parametru `axis="columns"`:

```
In [30]: data[4] = np.nan
```

```
In [31]: data
Out[31]:
   0  1  2  4
0 1.0 6.5 3.0 NaN
1 1.0 NaN NaN NaN
2 NaN NaN NaN NaN
3 NaN 6.5 3.0 NaN
```

```
In [32]: data.dropna(axis="columns", how="all")
Out[32]:
   0  1  2
0 1.0 6.5 3.0
1 1.0 NaN NaN
2 NaN NaN NaN
3 NaN 6.5 3.0
```

Załóżmy, że chcesz zachować tylko wiersze zawierające określoną liczbę obserwacji. Możesz to zrobić za pomocą argumentu `thresh`:

```
In [33]: df = pd.DataFrame(np.random.randn(7, 3))
```

```
In [34]: df.iloc[:4, 1] = np.nan
```

```
In [35]: df.iloc[:2, 2] = np.nan
```

```
In [36]: df
Out[36]:
   0  1  2
0 -0.204708 NaN NaN
1 -0.555730 NaN NaN
2 0.092908 NaN 0.769023
```

```
3 1.246435      NaN -1.296221
4 0.274992  0.228913  1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

```
In [37]: df.dropna()
```

```
Out[37]:
```

	0	1	2
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

```
In [38]: df.dropna(thresh=2)
```

```
Out[38]:
```

	0	1	2
2	0.092908	NaN	0.769023
3	1.246435	NaN	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

## Wypełnianie brakujących danych

Zamiast filtrować brakujące dane (i narażać się na jednoczesne usunięcie również innych danych), czasami lepiej jest w jakiś sposób te „dziury” wypełnić. Najczęściej robi się to za pomocą funkcji `fillna`. Wywołanie funkcji `fillna` wraz ze stałą spowoduje wstawienie zadeklarowanej wartości w miejsce brakujących danych:

```
In [39]: df.fillna(0)
```

```
Out[39]:
```

	0	1	2
0	-0.204708	0.000000	0.000000
1	-0.555730	0.000000	0.000000
2	0.092908	0.000000	0.769023
3	1.246435	0.000000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

Wywołując `fillna` wraz ze słownikiem, możesz wypełnić brakujące elementy poszczególnych kolumn różnymi wartościami:

```
In [40]: df.fillna({1: 0.5, 2: 0})
```

```
Out[40]:
```

	0	1	2
0	-0.204708	0.500000	0.000000
1	-0.555730	0.500000	0.000000
2	0.092908	0.500000	0.769023
3	1.246435	0.500000	-1.296221
4	0.274992	0.228913	1.352917
5	0.886429	-2.001637	-0.371843
6	1.669025	-0.438570	-0.539741

Metody interpolacji dostępne podczas wykonywania operacji uaktualniania indeksu (patrz tabela 5.3) mogą być również użyte wraz z funkcją `fillna`:

```
In [41]: df = pd.DataFrame(np.random.standard_normal((6, 3)))
```

```
In [42]: df.iloc[2:, 1] = np.nan
```

```
In [43]: df.iloc[4:, 2] = np.nan
```

```
In [44]: df
```

```
Out[44]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	NaN	1.343810
3	-0.713544	NaN	-2.370232
4	-1.860761	NaN	NaN
5	-1.265934	NaN	NaN

```
In [45]: df.fillna(method='ffill')
```

```
Out[45]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.124121	1.343810
3	-0.713544	0.124121	-2.370232
4	-1.860761	0.124121	-2.370232
5	-1.265934	0.124121	-2.370232

```
In [46]: df.fillna(method='ffill', limit=2)
```

```
Out[46]:
```

	0	1	2
0	0.476985	3.248944	-1.021228
1	-0.577087	0.124121	0.302614
2	0.523772	0.124121	1.343810
3	-0.713544	0.124121	-2.370232
4	-1.860761	NaN	-2.370232
5	-1.265934	NaN	-2.370232

Funkcja `fillna` wykonuje również inne operacje, na przykład interpoluje dane przy użyciu mediany lub średniej:

```
In [47]: data = pd.Series([1., np.nan, 3.5, np.nan, 7])
```

```
In [48]: data.fillna(data.mean())
```

```
Out[48]:
```

0	1.000000
1	3.833333
2	3.500000
3	3.833333
4	7.000000

dtype: float64

W tabeli 7.2 wymieniono argumenty funkcji `fillna`.



Tabela 7.2. Argumenty funkcji `fillna`

Argument	Opis
<code>value</code>	Wartość skalarna lub słownik — element używany do wypełnienia miejsc występowania brakujących danych.
<code>method</code>	Interpolacja: <code>"bfill"</code> (wypełnianie wstecz) lub <code>"ffill"</code> (w przód); domyślnie <code>None</code> .
<code>axis</code>	Wypełniana oś: <code>"index"</code> (domyślnie) lub <code>"columns"</code> .
<code>limit</code>	W przypadku wypełniania w przód i w tył określa maksymalną liczbę kolejnych wypełnianych okresów.

## 7.2. Przekształcanie danych

Dotychczas zajmowaliśmy się tylko tematem obsługi brakujących danych. Filtrowanie, czyszczenie i inne przekształcenia tworzą kolejną klasę ważnych operacji.

### Usuwanie duplikatów

Istnieje wiele powodów pojawiania się zduplikowanych wierszy w ramkach danych. Oto przykład takiej sytuacji:

```
In [49]: data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
....:                        'k2': [1, 1, 2, 3, 3, 4, 4]})
```

```
In [50]: data
```

```
Out[50]:
   k1 k2
0  one  1
1  two  1
2  one  2
3  two  3
4  one  3
5  two  4
6  two  4
```

Metoda `duplicated` ramki danych zwraca serię wartości logicznych określającą, czy poszczególne wiersze ramki danych są duplikatami, tj. czy wartości w kolejnych kolumnach są identyczne z zawartymi w poprzednim wierszu:

```
In [51]: data.duplicated()
```

```
Out[51]:
0  False
1  False
2  False
3  False
4  False
5  False
6   True
dtype: bool
```

Metoda `drop_duplicates` zwraca ramkę danych, w przypadku której metoda `duplicated` zwraca same wartości `False`:

```
In [52]: data.drop_duplicates()
```

```
Out[52]:
   k1 k2
```

```
0 one 1
1 two 1
2 one 2
3 two 3
4 one 3
5 two 4
```

Obie z zaprezentowanych metod biorą pod uwagę wszystkie kolumny, ale możesz wybrać podzbiór kolumn, które mają zostać wzięte pod uwagę podczas wykrywania duplikatów. Załóżmy, że mamy dodatkową kolumnę wartości i chcemy dokonać operacji filtrowania duplikatów tylko na podstawie kolumny k1:

```
In [53]: data['v1'] = range(7)
```

```
In [54]: data
```

```
Out[54]:
   k1 k2 v1
0 one 1 0
1 two 1 1
2 one 2 2
3 two 3 3
4 one 3 4
5 two 4 5
6 two 4 6
```

```
In [55]: data.drop_duplicates(subset=["k1"])
```

```
Out[55]:
   k1 k2 v1
0 one 1 0
1 two 1 1
```

Metody `drop_duplicates` i `drop_duplicates` domyślnie zachowują pierwszą znaną kombinację wartości. Parametr `keep='last'` spowoduje zwrócenie ostatniej takiej kombinacji:

```
In [56]: data.drop_duplicates(['k1', 'k2'], keep='last')
```

```
Out[56]:
   k1 k2 v1
0 one 1 0
1 two 1 1
2 one 2 2
3 two 3 3
4 one 3 4
6 two 4 6
```

## Przekształcanie danych przy użyciu funkcji lub mapowania

W przypadku wielu zbiorów danych może zachodzić konieczność wykonania pewnych przekształceń na podstawie wartości umieszczonych w tablicy, serii lub kolumnie ramki danych. Przyjrzyj się poniższemu hipotetycznemu zbiorowi danych opisującemu różne rodzaje mięsa:

```
In [57]: data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
....:                                'Pastrami', 'corned beef', 'Bacon',
....:                                'pastrami', 'honey ham', 'nova lox'],
....:                        'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
```

```
In [58]: data
```

```
Out[58]:
      food  ounces
0      bacon    4.0
1  pulled pork    3.0
2      bacon   12.0
3    Pastrami    6.0
4  corned beef    7.5
5      Bacon    8.0
6    pastrami    3.0
7  honey ham     5.0
8    nova lox    6.0
```

Załóżmy, że chciałbyś dodać kolumnę określającą zwierzę, z którego zrobiony jest każdy z produktów spożywczych. Utwórzmy mapowanie każdego typu mięsa do zwierzęcia:

```
meat_to_animal = {
    'bacon': 'pig',
    'pulled pork': 'pig',
    'pastrami': 'cow',
    'corned beef': 'cow',
    'honey ham': 'pig',
    'nova lox': 'salmon'
}
```

Metoda `map` używana na obiektach `Series` (opisana w rozdziale 5., w podrozdziale „Funkcje apply i map”) przyjmuje funkcję lub słownik — element definiujący mapowanie:

```
In [60]: data['animal'] = lowercased.map(meat_to_animal)
```

```
In [61]: data
Out[61]:
      food  ounces  animal
0      bacon    4.0    pig
1  pulled pork    3.0    pig
2      bacon   12.0    pig
3    Pastrami    6.0    cow
4  corned beef    7.5    cow
5      Bacon    8.0    pig
6    pastrami    3.0    cow
7  honey ham     5.0    pig
8    nova lox    6.0  salmon
```

Istnieje również możliwość przekazania funkcji wykonującej wszystkie operacje:

```
In [62]: def get_animal(x):
....:     return meat_to_animal[x]

In [63]: data["food"].map(get_animal)
Out[63]:
0    pig
1    pig
2    pig
3    cow
4    cow
5    pig
6    cow
7    pig
8  salmon
Name: food, dtype: object
```

Korzystanie ze słowa kluczowego `map` to wygodny sposób na wykonywanie przekształceń elementów z elementu oraz innych operacji związanych z czyszczeniem danych.

## Zastępowanie wartości

Wypełnianie brakujących danych za pomocą metody `fillna` to specjalny przypadek zastosowania bardziej uniwersalnego mechanizmu zastępowania wartości. W poprzedniej sekcji dowiedziałeś się, że słowo kluczowe `map` może zostać użyte w celu zmodyfikowania podzbioru wartości obiektu, ale funkcja `replace` pozwala na wykonanie tego w bardziej elastyczny i prostszy sposób. Oto przykładowy obiekt typu `Series`:

```
In [64]: data = pd.Series([1., -999., 2., -999., -1000., 3.])

In [65]: data
Out[65]:
0    1.0
1  -999.0
2    2.0
3  -999.0
4 -1000.0
5    3.0
dtype: float64
```

Wartości `-999` mogą zastępować brakujące dane. W celu zastąpienia ich wartością `NaN` możemy skorzystać z funkcji `replace`, która wygeneruje nowy obiekt typu `Series`:

```
In [66]: data.replace(-999, np.nan)
Out[66]:
0    1.0
1    NaN
2    2.0
3    NaN
4 -1000.0
5    3.0
dtype: float64
```

Jeżeli chcesz za jednym zamachem zastąpić wiele wartości, możesz przekazać do tej funkcji listę wartości, które mają zostać zastąpione:

```
In [67]: data.replace([-999, -1000], np.nan)
Out[67]:
0  1.0
1  NaN
2  2.0
3  NaN
4  NaN
5  3.0
dtype: float64
```

W celu zastąpienia każdej z wartości inną należy przekazać listę elementów zastępujących:

```
In [68]: data.replace([-999, -1000], [np.nan, 0])
Out[68]:
0  1.0
1  NaN
```

```
2 2.0
3 NaN
4 0.0
5 3.0
dtype: float64
```

Przekazany argument może być również słownikiem:

```
In [69]: data.replace({-999: np.nan, -1000: 0})
Out[69]:
0 1.0
1 NaN
2 2.0
3 NaN
4 0.0
5 3.0
dtype: float64
```



Metoda `data.replace` działa inaczej niż metoda `data.str.replace` wykonująca podmianianie łańcuchów element po elemencie. Użycie metod łańcucha na obiekcie typu `Series` zostanie opisane w dalszej części tego rozdziału.

## Zmiana nazw indeksów osi

Etykiety osi mogą być przekształcone podobnie jak wartości obiektu `Series`. W celu uzyskania nowych obiektów o innych etykietach można skorzystać z funkcji lub mapowania. Osie mogą być również modyfikowane w miejscu — bez tworzenia nowych struktur danych. Oto prosty przykład:

```
In [70]: data = pd.DataFrame(np.arange(12).reshape((3, 4)),
.....:                       index=['Ohio', 'Colorado', 'New York'],
.....:                       columns=['one', 'two', 'three', 'four'])
```

Indeksy osi, podobnie jak obiekty `Series`, obsługują metodę `map`:

```
In [71]: def transform(x):
.....:     return x[:4].upper()

In [72]: data.index.map(transform)
Out[72]: Index(['OHIO', 'COLO', 'NEW'], dtype='object')
```

Przypisywanie do obiektu `index` może modyfikować ramkę danych w miejscu:

```
In [73]: data.index = data.index.map(transform)

In [74]: data
Out[74]:
   one  two  three  four
OHIO  0    1     2     3
COLO  4    5     6     7
NEW   8    9    10    11
```

Jeżeli chcesz utworzyć przekształconą wersję zbioru danych bez modyfikowania oryginalnego zbioru, to warto skorzystać z metody `rename`:

```
In [75]: data.rename(index=str.title, columns=str.upper)
Out[75]:
   ONE  TWO  THREE  FOUR
```

```
Ohio 0 1 2 3
Colo 4 5 6 7
New 8 9 10 11
```

Metoda `rename` może być używana w połączeniu z obiektem będącym słownikiem zawierającym nowe wartości podzbioru etykiet osi:

```
In [76]: data.rename(index={'OHIO': 'INDIANA'},
....:                columns={'three': 'peekaboo'})
Out[76]:
```

	one	two	peekaboo	four
INDIANA	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

Korzystanie z metody `rename` zwalnia z obowiązku ręcznego kopiowania ramki danych i przypisywania jej atrybutów `index` i `columns`.

## Dyskretyzacja i podział na koszyki

Ciągłe dane są często poddawane dyskretyzacji lub dzielone w inny sposób na „koszyki” umożliwiające ich dalszą analizę. Załóżmy, że dysponujesz danymi dotyczącymi osób wchodzących w skład grupy badawczej i chcesz dokonać ich podziału na koszyki w zależności od ich wieku:

```
In [77]: ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

Dokonajmy podziału na koszyki 18 – 25, 26 – 36, 36 – 60 i 61+. W tym celu możemy skorzystać z funkcji `pandas.cut`:

```
In [78]: bins = [18, 25, 35, 60, 100]
```

```
In [79]: age_categories = pd.cut(ages, bins)
```

```
In [80]: age_categories
```

```
Out[80]:
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 100], (35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64]): [(18, 25] < (25, 35] < (35, 60] < (60, 100]]
```

Obiekt zwrócony przez `pandas` jest specjalnym obiektem kategorycznym (typu `Categorical`). Zaprezentowane dane wyjściowe opisują koszyki wygenerowane za pomocą funkcji `pandas.cut`. Każdy koszyk jest identyfikowany przez specjalny, charakterystyczny dla biblioteki `pandas` interwał, opisujący dolną i górną granicę:

```
In [81]: age_categories.codes
```

```
Out[81]: array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

```
In [82]: age_categories.categories
```

```
Out[82]: IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]], dtype='interval[int64, right]')
```

```
In [83]: age_categories.categories[0]
```

```
Out[83]: Interval(18, 25, closed='right')
```

```
In [84]: pd.value_counts(age_categories)
```

```

Out[84]:
(18, 25]    5
(25, 35]    3
(35, 60]    3
(60, 100]   1
dtype: int64

```

Zauważ, że polecenie `pd.value_counts(categories)` zwraca liczby elementów znajdujących się w poszczególnych koszykach będących wynikiem funkcji `pandas.cut`.

Zgodnie z matematycznym zapisem interwałów nawias okrągły oznacza **otwarcie** zbioru, a nawias kwadratowy jego **domknięcie** (wartość graniczna zalicza się do zbioru). Zmiany otwartej strony zbioru możesz dokonać za pomocą argumentu `right=False`:

```

In [85]: pd.cut(ages, bins, right=False)
Out[85]:
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 100), [36, 61), [36, 61),
[26, 36)]
Length: 12
Categories (4, interval[int64]): [[18, 26) < [26, 36) < [36, 61) < [61, 100)]

```

Możesz zdefiniować własne nazwy koszyków. Wystarczy przekazać nazwę listy lub tablicę nazw za pomocą opcji `labels`:

```

In [86]: group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

In [87]: pd.cut(ages, bins, labels=group_names)
Out[87]:
[Youth, Youth, Youth, YoungAdult, Youth, ..., YoungAdult, Senior, MiddleAged, MiddleAged,
YoungAdult]
Length: 12
Categories (4, object): [Youth < YoungAdult < MiddleAged < Senior]

```

Jeżeli funkcji `pandas.cut` przekażesz liczbę całkowitą koszyków zamiast jawnych definicji krawędzi koszyków, to polecenie to zwróci koszyki o równej długości na podstawie minimalnej i maksymalnej wartości znajdującej się w dzielonym zbiorze danych. Oto przykład danych o równym rozkładzie podzielonych na cztery koszyki:

```

In [88]: data = np.random.rand(20)

In [89]: pd.cut(data, 4, precision=2)
Out[89]:
[(0.34, 0.55], (0.34, 0.55], (0.76, 0.97], (0.76, 0.97], (0.34, 0.55], ..., (0.34, 0.55],
(0.34, 0.55], (0.55, 0.76], (0.34, 0.55], (0.12, 0.34)]
Length: 20
Categories (4, interval[float64]): [(0.12, 0.34] < (0.34, 0.55] < (0.55, 0.76] < (0.76, 0.97)]

```

Opcja `precision=2` ogranicza precyzję wartości zmiennoprzecinkowych do dwóch miejsc po przecinku.

Funkcja `pandas.qcut` działa podobnie, ale dzieli dane na kwartyli. Zwykle funkcja `pandas.cut` nie dzieli zbioru na koszyki o takiej samej liczbie elementów (zależy to od rozkładu danych). Funkcja `pandas.qcut` korzysta z kwartyli, a więc wygenerowane przez nią koszyki będą charakteryzowały się zbliżoną liczbą elementów:

```

In [90]: data = np.random.standard_normal(1000)

In [91]: quartiles = pd.qcut(data, 4, precision=2)

```

```
In [92]: quartiles
Out[92]:
[(-0.026, 0.62], (0.62, 3.93], (-0.68, -0.026], (0.62, 3.93], (-0.026, 0.62], ...
, (-0.68, -0.026], (-0.68, -0.026], (-2.96, -0.68], (0.62, 3.93], (-0.68, -0.026]
]
Length: 1000
Categories (4, interval[float64, right]): [(-2.96, -0.68] < (-0.68, -0.026] < (-0.026, 0.62] < (0.62, 3.93]]
```

```
In [93]: pd.value_counts(quartiles)
Out[93]:
(-2.96, -0.68]      250
(-0.68, -0.026]    250
(-0.026, 0.62]     250
(0.62, 3.93]       250
dtype: int64
```

Do funkcji `pandas.cut` możesz przekazać definicje własnych kwartyli (liczby z zakresu od 0 do 1 — przedział obustronnie domknięty):

```
In [94]: pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
Out[94]:
(-2.9499999999999997, -1.187]      100
(-1.187, -0.0265]                  400
(-0.0265, 1.286]                   400
(1.286, 3.928]                      100
dtype: int64
```

Do funkcji `pandas.cut` i `pandas.qcut` wrócimy w dalszej części rozdziału przy okazji agregacji i operacji przeprowadzanych na grupach. Wymienione funkcje dyskretyzacji są szczególnie przydatne podczas analizy kwartyli i grup.

## Wykrywanie i filtrowanie elementów odstających

Filtrowanie i przekształcanie elementów odstających polega głównie na przeprowadzaniu operacji tablicowych. Oto przykładowa ramka danych z wartościami charakteryzującymi się rozkładem normalnym:

```
In [95]: data = pd.DataFrame(np.random.standard_normal((1000, 4)))
```

```
In [96]: data.describe()
Out[96]:
```

	0	1	2	3
count	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.049091	0.026112	-0.002544	-0.051827
std	0.996947	1.007458	0.995232	0.998311
min	-3.645860	-3.184377	-3.745356	-3.428254
25%	-0.599807	-0.612162	-0.687373	-0.747478
50%	0.047101	-0.013609	-0.022158	-0.088274
75%	0.756646	0.695298	0.699046	0.623331
max	2.653656	3.525865	2.735527	3.366626



Załóżmy, że chcesz znaleźć elementy, których wartość bezwzględna jest większa od 3:

```
In [97]: col = data[2]
```

```
In [98]: col[np.abs(col) > 3]
```

```
Out[98]:
```

```
41 -3.399312
```

```
136 -3.745356
```

```
Name: 2, dtype: float64
```

W celu wybrania wszystkich wierszy z wartościami przekraczającymi 3 lub  $-3$  należy skorzystać z metody `any` obiektu `DataFrame`:

```
In [99]: data[(data.abs() > 3).any(axis="columns")]
```

```
Out[99]:
```

```
      0      1      2      3
41  0.457246 -0.025907 -3.399312 -0.974657
60  1.951312  3.260383  0.963301  1.201206
136  0.508391 -0.196713 -3.745356 -1.520113
235 -0.242459 -3.056990  1.918403 -0.578828
258  0.682841  0.326045  0.425384 -3.428254
322  1.179227 -3.184377  1.369891 -1.074833
544 -3.548824  1.553205 -2.186301  1.277104
635 -0.578093  0.193299  1.397822  3.366626
782 -0.207434  3.525865  0.283070  0.544635
803 -3.645860  0.255475 -0.549574 -1.907459
```

Nawiasy obejmujące wyrażenie `data.abs() > 3` są niezbędne, aby wynik porównania można było przetworzyć za pomocą metody `any`.

Wartości mogą być określane na podstawie tych kryteriów. Oto kod wyszukujący wartości poza zakresem od  $-3$  do  $3$ :

```
In [100]: data[np.abs(data) > 3] = np.sign(data) * 3
```

```
In [101]: data.describe()
```

```
Out[101]:
```

```
      0      1      2      3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean    0.050286    0.025567   -0.001399   -0.051765
std     0.992920    1.004214    0.991414    0.995761
min    -3.000000   -3.000000   -3.000000   -3.000000
25%    -0.599807   -0.612162   -0.687373   -0.747478
50%     0.047101   -0.013609   -0.022158   -0.088274
75%     0.756646    0.695298    0.699046    0.623331
max     2.653656    3.000000    2.735527    3.000000
```

Instrukcja `np.sign(data)` generuje wartości  $1$  i  $-1$  w zależności od tego, czy wartości obiektu `data` są dodatnie, czy ujemne:

```
In [102]: np.sign(data).head()
```

```
Out[102]:
```

```
      0      1      2      3
0 -1.0  1.0 -1.0  1.0
1  1.0 -1.0  1.0 -1.0
2  1.0  1.0  1.0 -1.0
3 -1.0 -1.0  1.0 -1.0
4 -1.0  1.0 -1.0 -1.0
```

## Permutacje i próbkowanie losowe

Permutację (czyli losową zmianę kolejności) obiektu typu `Series` lub wiersza ramki danych można przeprowadzić z łatwością za pomocą funkcji `numpy.random.permutation`. Wywołanie funkcji `permutation` i przekazanie jej w roli argumentu długości osi, która ma zostać poddana permutacji, powoduje wygenerowanie tablicy wartości całkowitoliczbowych określających nową kolejność:

```
In [103]: df = pd.DataFrame(np.arange(5 * 7).reshape((5, 7)))
```

```
In [104]: df
```

```
Out[104]:
```

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34

```
In [105]: sampler = np.random.permutation(5)
```

```
In [106]: sampler
```

```
Out[106]: array([3, 1, 4, 2, 0])
```

Tablica ta może być następnie użyta w indeksowaniu opartym na funkcji `iloc` lub podobnej funkcji take:

```
In [107]: df.take(sampler)
```

```
Out[107]:
```

	0	1	2	3	4	5	6
3	21	22	23	24	25	26	27
1	7	8	9	10	11	12	13
4	28	29	30	31	32	33	34
2	14	15	16	17	18	19	20
0	0	1	2	3	4	5	6

```
In [108]: df.iloc[sampler]
```

```
Out[108]:
```

	0	1	2	3	4	5	6
3	21	22	23	24	25	26	27
1	7	8	9	10	11	12	13
4	28	29	30	31	32	33	34
2	14	15	16	17	18	19	20
0	0	1	2	3	4	5	6

Wywołując metodę `take` z argumentem `axis="columns"`, można uzyskać wynik permutacji kolumn:

```
In [109]: column_sampler = np.random.permutation(7)
```

```
In [110]: column_sampler
```

```
Out[110]: array([4, 6, 3, 2, 1, 0, 5])
```

```
In [111]: df.take(column_sampler, axis="columns")
```

```
Out[111]:
```

	4	6	3	2	1	0	5
0	4	6	3	2	1	0	5
1	11	13	10	9	8	7	12

```
2 18 20 17 16 15 14 19
3 25 27 24 23 22 21 26
4 32 34 31 30 29 28 33
```

W celu wybrania losowego podzbioru bez zastępowania (tj. tak, aby żaden wiersz nie pojawił się dwukrotnie) możesz skorzystać z metody `sample` obiektu `Series` lub `DataFrame`:

```
In [112]: df.sample(n=3)
Out[112]:
   0  1  2  3  4  5  6
2  14 15 16 17 18 19 20
4  28 29 30 31 32 33 34
0   0  0  1  2  3  4  5  6
```

W celu wygenerowania próbki z zastępowaniem (elementy mogą być wybierane więcej niż raz) skorzystaj z funkcji `sample` z argumentem `replace=True`:

```
In [113]: choices = pd.Series([5, 7, -1, 6, 4])

In [114]: choices.sample(n=10, replace=True)
Out[114]:
2  -1
0   5
3   6
1   7
4   4
0   5
4   4
0   5
4   4
4   4
dtype: int64
```

## Przetwarzanie wskaźników i zmiennych zastępczych

Innym rodzajem transformacji przydatnej podczas modelowania statystycznego lub uczenia maszynowego jest konwersja zmiennej kategorycznej na macierz „elementów zastępczych” lub „wskaźników”. Jeżeli w kolumnie ramki danych znajduje się  $k$  odmiennych wartości, to można utworzyć na jej podstawie macierz pochodną zawierającą  $k$  kolumn wypełnionych jedynekami i zerami. Służy do tego funkcja `pandas.get_dummies`, ale funkcję taką można z łatwością napisać samemu. Wróćmy do zaprezentowanego wcześniej przykładu ramki danych:

```
In [115]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
.....:                       'data1': range(6)})

In [116]: df
Out[116]:
   key  data1
0    b      0
1    b      1
2    a      2
3    c      3
4    a      4
5    b      5

In [117]: pd.get_dummies(df['key'])
```

```
Out[117]:
  a b c
0 0 1 0
1 0 1 0
2 1 0 0
3 0 0 1
4 1 0 0
5 0 1 0
```

Czasami zachodzi potrzeba dodania prefiksu do nazw kolumn wygenerowanej ramki danych, które można następnie połączyć z innymi danymi. Do tego właśnie służy argument `prefix` funkcji `pandas.get_dummies`:

```
In [118]: dummies = pd.get_dummies(df['key'], prefix='key')
```

```
In [119]: df_with_dummy = df[['data1']].join(dummies)
```

```
In [120]: df_with_dummy
```

```
Out[120]:
  data1 key_a key_b key_c
0     0     0     1     0
1     1     0     1     0
2     2     1     0     0
3     3     0     0     1
4     4     1     0     0
5     5     0     1     0
```

Metoda `DataFrame.join` zostanie dokładniej opisana w następnym rozdziale.

Jeżeli wiersz ramki danych należy do wielu kategorii, należy zmienne zastępcze utworzyć w inny sposób. Przyjrzyjmy się zbiorowi danych MovieLens 1M, który zostanie przeanalizowany w sposób bardziej szczegółowy w rozdziale 13.:

```
In [121]: mnames = ['movie_id', 'title', 'genres']
```

```
In [122]: movies = pd.read_table('dane/movielens/movies.dat', sep='::',
.....:                          header=None, names=mnames)
```

```
In [123]: movies[:10]
```

```
Out[123]:
  movie_id title genres
0         1 Toy Story (1995) Animation|Children's|Comedy
1         2 Jumanji (1995) Adventure|Children's|Fantasy
2         3 Grumpier Old Men (1995) Comedy|Romance
3         4 Waiting to Exhale (1995) Comedy|Drama
4         5 Father of the Bride Part II (1995) Comedy
5         6 Heat (1995) Action|Crime|Thriller
6         7 Sabrina (1995) Comedy|Romance
7         8 Tom and Huck (1995) Adventure|Children's
8         9 Sudden Death (1995) Action
9        10 GoldenEye (1995) Action|Adventure|Thriller
```

Obiekt `Series` posiada specjalną metodę `str.get_dummies` (metody o nazwach rozpoczynających się od `str` będą dokładniej opisane w podrozdziale „Operacje przeprowadzane na łańcuchach”), przystosowaną do sytuacji, w których przynależność do wielu grup jest zakodowana za pomocą ciągu ich nazw oddzielonych separatorem:

```
In [124]: dummies = movies["genres"].str.get_dummies("|")
```

```
In [125]: dummies.iloc[:10, :6]
```

```
Out[125]:
```

	Action	Adventure	Animation	Children's	Comedy	Crime
0	0	0	1	1	1	0
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	0
4	0	0	0	0	1	0
5	1	0	0	0	0	1
6	0	0	0	0	1	0
7	0	1	0	1	0	0
8	1	0	0	0	0	0
9	1	1	0	0	0	0

Teraz tak jak wcześniej możemy połączyć ten wynik z obiektem `movies` podczas dodawania za pomocą metody `add_prefix` prefiksu "Genre\_" do nazw kolumn zawartych w obiekcie `DataFrame`:

```
In [126]: movies_windic = movies.join(dummies.add_prefix('Genre_'))
```

```
In [127]: movies_windic.iloc[0]
```

```
Out[127]:
```

movie_id	1
title	Toy Story (1995)
genres	Animation Children's Comedy
Genre_Action	0
Genre_Adventure	0
Genre_Animation	1
Genre_Children's	1
Genre_Comedy	1
Genre_Crime	0
Genre_Documentary	0
Genre_Drama	0
Genre_Fantasy	0
Genre_Film-Noir	0
Genre_Horror	0
Genre_Musical	0
Genre_Mystery	0
Genre_Romance	0
Genre_Sci-Fi	0
Genre_Thriller	0
Genre_War	0
Genre_Western	0

Name: 0, Length: 21, dtype: object



Przedstawiona metoda tworzenia zmiennych wskazujących działa dość wolno przy dużych zbiorach danych. W takich przypadkach lepiej jest napisać niskopoziomą funkcję zapisującą dane bezpośrednio w tablicy NumPy, a następnie obudować wynik jej pracy obiektem `DataFrame`.

W zastosowaniach statystycznych warto połączyć `pandas.get_dummies` z funkcją dyskretyzacji `pandas.cut`:

```
In [128]: np.random.seed(12345) # Aby przykład był powtarzalny
```

```
In [129]: values = np.random.uniform(size=10)
```

```
In [130]: values
Out[130]:
array([ 0.9296,  0.3164,  0.1839,  0.2046,  0.5677,  0.5955,  0.9645,  0.6532,  0.7489,
        0.6536])
```

```
In [131]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]
```

```
In [132]: pd.get_dummies(pd.cut(values, bins))
Out[132]:
```

	(0.0, 0.2]	(0.2, 0.4]	(0.4, 0.6]	(0.6, 0.8]	(0.8, 1.0]
0	0	0	0	0	1
1	0	1	0	0	0
2	1	0	0	0	0
3	0	1	0	0	0
4	0	0	1	0	0
5	0	0	1	0	0
6	0	0	0	0	1
7	0	0	0	1	0
8	0	0	0	1	0
9	0	0	0	1	0

Do funkcji `pandas.get_dummies` wrócimy jeszcze w podrozdziale „Tworzenie fikcyjnych zmiennych używanych podczas modelowania”.

## 7.3. Rozszerzone typy danych



Temat tego rozdziału jest nowy i bardziej zaawansowany, więc wielu użytkownikom biblioteki `pandas` może nie być dobrze znany. Przedstawiam go tutaj dla kompletności opisu, jak również dlatego, że w wielu miejscach w następujących rozdziałach będę odwoływał się do rozszerzonych typów danych.

Biblioteka `pandas` została pierwotnie zbudowana na bazie funkcjonalności obecnie dostępnych w `NumPy` — macierzowej bibliotece obliczeniowej, używanej głównie do pracy z danymi liczbowymi. Wiele z tych funkcjonalności, na przykład obsługa brakujących danych, zostało zaimplementowanych z myślą o maksymalnej kompatybilności z bibliotekami, które wykorzystują jednocześnie `NumPy` i `pandas`.

Bazowanie na bibliotece `NumPy` jest przyczyną wielu mankamentów biblioteki `pandas`, takich jak na przykład:

- Niepełna obsługa brakujących danych całkowitoliczbowych i logicznych. W efekcie biblioteka `pandas` używa do reprezentowania brakujących danych obiektu np. `nan`, a istniejące wartości przekształca w liczby zmiennoprzecinkowe, co w wielu algorytmach skutkuje subtelnymi błędami.
- Operacje wykonywane na dużych zbiorach danych tekstowych są kosztowne obliczeniowo i zajmują dużo pamięci.
- Niektóre typy danych, m.in. szeregi czasowe, interwały i znaczniki z identyfikatorami stref nie mogą być wydajnie przetwarzane bez użycia kosztownych obliczeniowo tablic obiektów Pythona.

Ostatnio biblioteka pandas została rozbudowana o *rozszerzone typy* danych, dzięki którym można przetwarzać nowe typy, nieobsługiwane natywnie przez bibliotekę NumPy. Wraz z tablicami można je traktować jako typy pierwszoklasowe.

Przyjrzyjmy się przykładowi utworzenia serii liczb całkowitych z brakującą wartością:

```
In [133]: s = pd.Series([1, 2, 3, None])
```

```
In [134]: s
```

```
Out[134]:
```

```
0    1.0
1    2.0
2    3.0
3    NaN
dtype: float64
```

```
In [135]: s.dtype
```

```
Out[135]: dtype('float64')
```

Głównie ze względu na kompatybilność wsteczną typ Series wykorzystuje standardowy typ float64 i obiekt np.nan oznaczający brakujące wartości. Tego rodzaju serię można również utworzyć, używając metody pandas.Int64Dtype:

```
In [136]: s = pd.Series([1, 2, 3, None], dtype=pd.Int64Dtype())
```

```
In [137]: s
```

```
Out[137]:
```

```
0     1
1     2
2     3
3  <NA>
dtype: Int64
```

```
In [138]: s.isna()
```

```
Out[138]:
```

```
0    False
1    False
2    False
3     True
dtype: bool
```

```
In [139]: s.dtype
```

```
Out[139]: Int64Dtype()
```

Ciąg <NA> oznacza brak wartości w tablicy zawierającej dane rozszerzonego typu. Jest to specjalny obiekt zastępczy pandas.NA:

```
In [140]: s[3]
```

```
Out[140]: <NA>
```

```
In [141]: s[3] is pd.NA
```

```
Out[141]: True
```

Zamiast metody `pd.Int64Dtype` można użyć krótkiego oznaczenia rozszerzonego typu `"Int64"`. Ważna jest wielkość liter. Jeżeli zostanie użyta inna, typ zostanie potraktowany jako nierozszerzony, właściwy dla biblioteki NumPy:

```
In [142]: s = pd.Series([1, 2, 3, None], dtype="Int64")
```

Biblioteka pandas zawiera również rozszerzony typ danych tekstowych, który nie wykorzystuje tablic obiektów NumPy (wymaga natomiast biblioteki pyarrow, którą trzeba zainstalować osobno):

```
In [143]: s = pd.Series(['one', 'two', None, 'three'], dtype=pd.StringDtype())
```

```
In [144]: s
Out[144]:
0      one
1      two
2      <NA>
3     three
dtype: string
```

Duże tablice danych tego typu zazwyczaj zajmują znacznie mniej pamięci, a operacje na nich są bardziej wydajne.

Innym ważnym typem rozszerzonym jest `Categorical`, który będzie dokładniej opisany w podrozdziale „Dane kategoryczne”. Tabela 7.3 przedstawia pełną listę rozszerzonych typów dostępnych w chwili pisania tego tekstu.

Tabela 7.3. Rozszerzone typy danych w bibliotece pandas

Typ wyrażenia	Opis
<code>BooleanDtype</code>	Wartość logiczna, dopuszczone wartości puste, oznaczenie tekstowe <code>"boolean"</code> .
<code>CategoricalDtype</code>	Wartość kategoryczna, oznaczenie tekstowe <code>"category"</code> .
<code>DatetimeTZDtype</code>	Data i czas z oznaczeniem strefy.
<code>Float32Dtype</code>	32-bitowa liczba zmiennoprzecinkowa, dopuszczone wartości puste, oznaczenie tekstowe <code>"Float32"</code> .
<code>Float64Dtype</code>	32-bitowa liczba zmiennoprzecinkowa, dopuszczone wartości puste, oznaczenie tekstowe <code>"Float64"</code> .
<code>Int8Dtype</code>	8-bitowa liczba całkowita ze znakiem, dopuszczone wartości puste, oznaczenie tekstowe <code>"Int8"</code> .
<code>Int16Dtype</code>	16-bitowa liczba całkowita ze znakiem, dopuszczone wartości puste, oznaczenie tekstowe <code>"Int16"</code> .
<code>Int32Dtype</code>	32-bitowa liczba całkowita ze znakiem, dopuszczone wartości puste, oznaczenie tekstowe <code>"Int32"</code> .
<code>Int64Dtype</code>	64-bitowa liczba całkowita ze znakiem, dopuszczone wartości puste, oznaczenie tekstowe <code>"Int64"</code> .
<code>UInt8Dtype</code>	8-bitowa liczba całkowita bez znaku, dopuszczone wartości puste, oznaczenie tekstowe <code>"UInt8"</code> .
<code>UInt16Dtype</code>	16-bitowa liczba całkowita bez znaku, dopuszczone wartości puste, oznaczenie tekstowe <code>"UInt16"</code> .
<code>UInt32Dtype</code>	32-bitowa liczba całkowita bez znaku, dopuszczone wartości puste, oznaczenie tekstowe <code>"UInt32"</code> .
<code>UInt64Dtype</code>	64-bitowa liczba całkowita bez znaku, dopuszczone wartości puste, oznaczenie tekstowe <code>"UInt64"</code> .

Oznaczenia rozszerzonych typów można umieszczać w argumencie metody `astype` obiektu `Series`, co ułatwia przekształcanie danych w procesie ich oczyszczania:

```
In [145]: df = pd.DataFrame({"A": [1, 2, None, 4],
.....:                      "B": ["one", "two", "three", None],
.....:                      "C": [False, None, False, True]})
```



```

In [146]: df
Out[146]:
   A      B      C
0  1.0  one  False
1  2.0  two  None
2  NaN three  False
3  4.0  None   True

In [147]: df["A"] = df["A"].astype("Int64")

In [148]: df["B"] = df["B"].astype("string")

In [149]: df["C"] = df["C"].astype("boolean")

In [150]: df
Out[150]:
   A      B      C
0   1  one  False
1   2  two  <NA>
2 <NA> three  False
3   4  <NA>   True

```

## 7.4. Operacje przeprowadzane na łańcuchach

Popularność Pythona jako języka przeznaczonego do manipulowania nieprzetwarzanymi wcześniej danymi wynika między innymi z łatwości wykonywania operacji na łańcuchach i przetwarzania tekstu. Większość operacji tekstowych można wykonać prosto za pomocą wbudowanych metod obiektu typu string. W celu bardziej złożonych operacji dobierania wzorców i obróbki tekstu może zachodzić konieczność korzystania z wyrażeń regularnych. Pakiet pandas umożliwia stosowanie operacji łańcuchowych i wyrażeń regularnych na całych tablicach danych i dodatkowo pozwala rozwiązać problem występowania brakujących danych.

### Metody obiektu typu string

Wbudowane metody łańcuchów wystarczają do przeprowadzania wielu operacji przetwarzania łańcuchów i tworzenia skryptów. Np. metoda `split` dzieli na fragmenty łańcuch, w którym rolę separatora pełni przecinek:

```

In [151]: val = 'a,b, guido'

In [152]: val.split(',')
Out[152]: ['a', 'b', ' guido']

```

Metoda `split` jest często używana w połączeniu z metodą `strip`, która usuwa białe znaki (w tym również znaki końca wiersza):

```

In [153]: pieces = [x.strip() for x in val.split(',')]

In [154]: pieces
Out[154]: ['a', 'b', 'guido']

```

Podłańcuchy można połączyć za pomocą podwójnego dwukropka i operacji dodawania:

```
In [155]: first, second, third = pieces

In [156]: first + '::' + second + '::' + third
Out[156]: 'a::b::guido'
```

Nie jest to jednak praktyczne rozwiązanie. Można to zrobić szybciej, przekazując listę lub krotkę do metody `join` łańcucha `'::'`:

```
In [157]: '::'.join(pieces)
Out[157]: 'a::b::guido'
```

Pozostałe metody są przeznaczone do szukania podłańcuchów. Podłańcuchy najlepiej jest wyszukiwać za pomocą słowa kluczowego `in`, ale można korzystać również z metod `index` i `find`:

```
In [158]: 'guido' in val
Out[158]: True

In [159]: val.index(',')
Out[159]: 1

In [160]: val.find(':')
Out[160]: -1
```

Zapamiętaj, że różnicą pomiędzy metodą `find` i `index` jest to, że metoda `index` generuje wyjątek w razie nieznaledzenia łańcucha (metoda `find` zwraca wtedy wartość `-1`):

```
In [161]: val.index(':')
-----
ValueError Traceback (most recent call last)
<ipython-input-144-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

Metoda `count` zwraca liczbę wystąpień określonego podłańcucha:

```
In [162]: val.count(',')
Out[162]: 2
```

Metoda `replace` zastępuje miejsca występowania jednego ciągu znaków innym ciągiem znaków. Jest ona często używana do kasowania podłańcuchów (wystarczy przekazać do niej pusty łańcuch):

```
In [163]: val.replace(',', '::')
Out[163]: 'a::b:: guido'

In [164]: val.replace(',', '')
Out[164]: 'ab guido'
```

W tabeli 7.4 znajdziesz listę wybranych metod obiektów typu `string`.

W kontekście wielu tych operacji można również stosować wyrażenia regularne (wkrótce przedstawię przykłady takich rozwiązań).

Tabela 7.4. Wbudowane metody obiektu string

Argument	Opis
count	Zwraca liczbę nienakładających się wystąpień danego ciągu znaków w łańcuchu.
endswith	Zwraca True, jeżeli łańcuch kończy się sufiksem.
startswith	Zwraca True, jeżeli łańcuch rozpoczyna się prefiksem.
join	Używa łańcucha w roli elementu rozdzielającego inne łączone ze sobą łańcuchy.
index	Zwraca pozycję pierwszego znaku ciągu znaków znalezione w łańcuchu; w razie nieznaalezienia ciągu generowany jest błąd ValueError.
find	Zwraca pozycję pierwszego znaku <i>pierwszego</i> wystąpienia ciągu znaków w łańcuchu; działa jak metoda index, ale w razie nieznaalezienia ciągu zwraca wartość -1.
rfind	Zwraca pozycję pierwszego znaku <i>ostatniego</i> wystąpienia ciągu znaków w łańcuchu; w razie nieznaalezienia ciągu zwraca wartość -1.
replace	Zastępuje ciąg znaków innym ciągiem znaków.
strip,rstrip, lstrip	Usuwa białe znaki, w tym znaki końca wiersza odpowiednio z obu końców, z lewej i prawej strony ciągu.
split	Dzieli łańcuch na podłańcuchy przy użyciu przekazanego znaku rozdzielającego.
lower	Konwertuje znaki na małe litery.
upper	Konwertuje znaki na wielkie litery.
casefold	Konwertuje wielkie litery na małe, a także konwertuje wszelkie kombinacje znaków specjalnych charakterystycznych dla niektórych języków na standardowe znaki, które mogą być porównywane.
ljust, rjust	Justowanie do lewej lub prawej; dodaje spacje lub inne znaki wypełnienia po przeciwnej stronie łańcucha w celu zwrócenia łańcucha o minimalnej szerokości.

## Wyrażenia regularne

Stosowanie **wyrażeń regularnych** to uniwersalny sposób wyszukiwania lub dobierania (często złożonych) ciągów znaków w tekście. Pojedyncze wyrażenie regularne określa się mianem wyrażenia **regex**. Jest to łańcuch utworzony zgodnie ze składnią języka wyrażeń regularnych. Wbudowany moduł Pythona re jest używany do obsługi wyrażeń regularnych w kontekście łańcuchów. W dalszej części tej sekcji przedstawię dużo przykładów zastosowania takich wyrażeń.



Na temat tworzenia wyrażeń regularnych można napisać oddzielny rozdział, ale zagadnienie to wykracza nieco poza zakres tematyczny tej książki. W innych książkach, a także w internecie można znaleźć wiele dobrych poradników dotyczących obsługi wyrażeń regularnych.

Funkcje modułu re można podzielić na trzy kategorie: dopasowywanie ciągu, zastępowanie i dzielenie. Oczywiście wszystkie te operacje są ze sobą związane. Wyrażenie regularne opisuje wzorzec, który ma zostać znaleziony w tekście, a więc można je stosować w celu wykonania różnych operacji. Przyjrzyj się następującemu przykładowi: założmy, że chcemy podzielić łańcuch zawierający zmienną liczbę białych znaków (spacji, tabulacji, znaków nowego wiersza). Wyrażenie regularne \s+ jest symbolem zastępczym przynajmniej jednego białego znaku:

```
In [165]: import re
In [166]: text = "foo bar\tbaz \tqux"
In [167]: re.split('\s+', text)
Out[167]: ['foo', 'bar', 'baz', 'qux']
```

Po uruchomieniu funkcji `re.split('\s+', text)` wyrażenie regularne jest najpierw *skompilowane*, a następnie na przekazanym tekście wywoływana jest metoda `split`. Możesz samodzielnie skompilować wyrażenie regularne za pomocą funkcji `re.compile`, co pozwala na uzyskanie obiektu wyrażenia regularnego, który może być użyty wielokrotnie:

```
In [168]: regex = re.compile('\s+')
In [169]: regex.split(text)
Out[169]: ['foo', 'bar', 'baz', 'qux']
```

Gdybyś natomiast chciał uzyskać listę wszystkich wzorców pasujących do wyrażenia regularnego, możesz skorzystać z metody `findall`:

```
In [170]: regex.findall(text)
Out[170]: [' ', '\t', ' \t']
```



Aby uniknąć korzystania w wyrażeniach regularnych z sekwencji specjalnej rozpoczynającej się od znaku `\`, korzystaj z surowych literałów łańcuchowych, takich jak `r'C:\x'`, zamiast z ich odpowiedników w postaci `'C:\\x'`.

W przypadku stosowania tego samego wyrażenia do wielu łańcuchów zaleca się tworzenie obiektu wyrażenia regularnego za pomocą funkcji `re.compile`. Dzięki temu zmniejszona zostanie liczba użytych cykli procesora.

Metody `match` i `search` są związane z metodą `findall`. Metoda `findall` zwraca wszystkie przypadki dopasowania znalezione w łańcuchu, a metoda `search` zwraca *tylko* pierwsze dopasowanie, a dokładnie rzecz biorąc, zwraca ona początek wyszukiwanego ciągu znaków. Przyjrzyjmy się przykładowi wyrażenia regularnego mogącego zidentyfikować większość adresów poczty elektronicznej umieszczonych w bloku tekstu:

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

# Flaga re.IGNORECASE sprawia, że wyrażenie regularne nie zwraca uwagi na wielkość liter.
regex = re.compile(pattern, flags=re.IGNORECASE)
```

Użycie metody `findall` na tekście zwraca listę adresów:

```
In [172]: regex.findall(text)
Out[172]:
['dave@google.com',
'steve@gmail.com',
'rob@gmail.com',
'ryan@yahoo.com']
```

Metoda `search` zwraca specjalny obiekt dopasowania dla pierwszego adresu znalezionej w tekście. W przypadku zaprezentowanego wcześniej wyrażenia regularnego obiekt dopasowania może tylko poinformować nas o początku i końcu wzorca znalezionej w łańcuchu:

```
In [173]: m = regex.search(text)

In [174]: m
Out[174]: <_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>

In [175]: text[m.start():m.end()]
Out[175]: 'dave@google.com'
```

Metoda `regex.match` zwraca `None`, ponieważ jest ona w stanie znaleźć wzorzec tylko wtedy, gdy znajduje się on na początku łańcucha:

```
In [176]: print(regex.match(text))
None
```

Metoda `sub` zwróci nowy łańcuch, w którym miejsca wystąpień wzorca zostaną zastąpione nowym łańcuchem:

```
In [177]: print(regex.sub('REDACTED', text))
Dave REDACTED
Steve REDACTED
Rob REDACTED
Ryan REDACTED
```

Żałujemy, że chcesz znaleźć adres e-mail i jednocześnie dokonać segmentacji każdego adresu na trzy komponenty: nazwę użytkownika, nazwę domeny i sufiks domeny. W tym celu musisz umieścić nawiasy okrągłe wokół części wzorca, które mają być poddane segmentacji:

```
In [178]: pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

In [179]: regex = re.compile(pattern, flags=re.IGNORECASE)
```

Obiekt dopasowania wygenerowany przez to zmodyfikowane wyrażenie regularne zwraca krotkę z komponentami wzorca (w celu uzyskania do niej dostępu należy skorzystać z metody `groups`):

```
In [180]: m = regex.match('wesm@bright.net')

In [181]: m.groups()
Out[181]: ('wesm', 'bright', 'net')
```

Gdy wzorzec zawiera grupy, metoda `findall` zwraca listę krotek:

```
In [182]: regex.findall(text)
Out[182]:
[('dave', 'google', 'com'),
 ('steve', 'gmail', 'com'),
 ('rob', 'gmail', 'com'),
 ('ryan', 'yahoo', 'com')]
```

Metoda `sub` umożliwia uzyskanie dostępu do grup każdego dopasowania za pomocą symboli w rodzaju `\1` i `\2`. Symbol `\1` odwołuje się do pierwszej dopasowanej grupy, symbol `\2` odwołuje się do drugiej dopasowanej grupy itd.:

```
In [183]: print(regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text))
Dave Username: dave, Domain: google, Suffix: com
Steve Username: steve, Domain: gmail, Suffix: com
Rob Username: rob, Domain: gmail, Suffix: com
Ryan Username: ryan, Domain: yahoo, Suffix: com
```

Python obsługuje o wiele więcej wyrażeń regularnych, ale korzystanie z większości z nich wykracza poza zakres tematyczny tej książki. W tabeli 7.5 podsumowano wybrane metody wyrażeń regularnych.

Tabela 7.5. Metody wyrażeń regularnych

Metoda	Opis
<code>findall</code>	Zwraca wszystkie nienakładające się dopasowania w formie listy.
<code>finditer</code>	Działa jak metoda <code>findall</code> , ale zwraca iterator.
<code>match</code>	Dopasowuje wzorzec na początku łańcucha i opcjonalnie dzieli komponenty wzorca na grupy; w przypadku dopasowania wzorca zwraca obiekt dopasowania, a w przeciwnym wypadku zwraca <code>None</code> .
<code>search</code>	Skanuje łańcuch w poszukiwaniu wzorca; w przypadku znalezienia go zwraca obiekt dopasowania; w przeciwieństwie do metody <code>match</code> wzorzec może być znaleziony nie tylko na początku przeszukiwanego łańcucha.
<code>split</code>	Dzieli łańcuch na kawałki w miejscu wystąpienia określonego wzorca.
<code>sub, subn</code>	Zastępuje wszystkie ( <code>sub</code> ) lub <code>n</code> wystąpień ( <code>subn</code> ) wzorca w łańcuchu wyrażeniem zastępczym; w celu odwołania się do grup elementów w łańcuchu zastępczym korzystaj z symboli <code>\1, \2...</code>

## Funkcje tekstowe w pakiecie pandas

Oczyszczanie zbioru danych przed analizą często wiąże się z koniecznością przekształcania łańcuchów. Sprawę komplikują dodatkowo brakujące dane kolumny z łańcuchami:

```
In [184]: data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
.....:           'Rob': 'rob@gmail.com', 'Wes': np.nan}
```

```
In [185]: data = pd.Series(data)
```

```
In [186]: data
Out[186]:
Dave    dave@google.com
Rob     rob@gmail.com
Steve   steve@gmail.com
Wes     NaN
dtype: object
```

```
In [187]: data.isna ()
Out[187]:
Dave    False
Rob     False
Steve   False
Wes     True
dtype: bool
```

W celu zastosowania metod łańcucha lub wyrażenia regularnego na każdej wartości (przekazując funkcję `lambda` lub inną funkcję), można skorzystać z notacji `data.map`, ale w przypadku wartości `NA` (`null`) wywoła to komunikat błędu. W związku z tym obiekt `Series` dysponuje metodami tablicowymi

przeznaczonymi do przetwarzania łańcuchów, które pomijają brakujące wartości. Dostęp do tych metod uzyskuje się za pomocą atrybutu `str` obiektu typu `Series`. Za pomocą metody `str.contains` możemy np. sprawdzić, czy każdy adres e-mail zawiera ciąg znaków `'gmail'`:

```
In [188]: data.str.contains('gmail')
Out[188]:
Dave    False
Rob     True
Steve   True
Wes     NaN
dtype: object
```

Zwróć uwagę, że wynikiem tej operacji jest obiekt. Biblioteka `pandas` implementuje *rozszerzone typy* danych, umożliwiające wykonywanie na ciągach znaków, liczbach całkowitych i wartościach logicznych specjalnych operacji, z którymi do niedawna były pewne problemy, jeżeli brakowało wartości:

```
In [189]: data_as_string_ext = data.astype('string')
```

```
In [190]: data_as_string_ext
Out[190]:
Dave    dave@google.com
Steve   steve@gmail.com
Rob     rob@gmail.com
Wes     <NA>
dtype: string
```

```
In [191]: data_as_string_ext.str.contains("gmail")
Out[191]:
Dave    False
Steve   True
Rob     True
Wes     <NA>
dtype: boolean
```

Wspomniane typy zostały opisane w podrozdziale „Rozszerzone typy danych”.

Możliwe jest również zastosowanie wyrażeń regularnych oraz innych opcji modułu `re`, takich jak `IGNORECASE`:

```
In [192]: pattern
Out[192]: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+\.[A-Z]{2,4})'

In [193]: data.str.findall(pattern, flags=re.IGNORECASE)
Out[193]:
Dave    [(dave, google, com)]
Rob     [(rob, gmail, com)]
Steve   [(steve, gmail, com)]
Wes     NaN
dtype: object
```

Istnieje kilka sposobów przeprowadzenia wektorowego wyszukiwania elementów. Można to zrobić za pomocą funkcji `str.get` lub poprzez przekazanie indeksu do funkcji `str`:

```
In [194]: matches = data.str.findall(pattern, flags=re.IGNORECASE).str[0]

In [195]: matches
```

```

Out[195]:
Dave      (dave, google, com)
Steve     (steve, gmail, com)
Rob       (rob, gmail, com)
Wes              NaN
dtype: object

```

```

In [196]: matches.str.get(1)
Out[196]:
Dave      google
Steve     gmail
Rob       gmail
Wes              NaN
dtype: object

```

Łańcuch możesz również pociąć za pomocą następującej składni:

```

In [197]: data.str[:5]
Out[197]:
Dave  dave@
Rob   rob@g
Steve steve
Wes   NaN
dtype: object

```

Metoda `str.extract` zwraca obiekty `DataFrame` zawierające grupy wyodrębnione za pomocą wyrażenia regularnego:

```

In [198]: data.str.extract(pattern, flags=re.IGNORECASE)
Out[198]:
      0      1      2
Dave  dave  google  com
Steve steve  gmail  com
Rob   rob   gmail  com
Wes   NaN   NaN   NaN

```

Więcej metod obiektów typu `string` obsługiwanych przez bibliotekę `pandas` znajdziesz w tabeli 7.6.

Tabela 7.6. Wybrane metody obiektu `Series`

Metoda	Opis
<code>Cat</code>	Łączy łańcuchy element po elemencie, stosując opcjonalny łańcuch rozdzielający.
<code>Contains</code>	Zwraca tablicę wartości logicznych informujących o tym, czy każdy z łańcuchów zawiera określony wzorzec lub wyrażenie regularne.
<code>Count</code>	Określa liczbę wystąpień wzorca.
<code>Extract</code>	Korzysta z wyrażenia regularnego z grupami w celu wyciągnięcia jednego lub więcej łańcuchów z obiektu <code>Series</code> zawierającego łańcuchy; wynik będzie miał postać ramki danych, której poszczególne kolumny będą zawierały po jednej grupie.
<code>Endswith</code>	Odpowiednik notacji <code>x.endswith(pattern)</code> dla każdego elementu.
<code>Startswith</code>	Odpowiednik notacji <code>x.startswith(pattern)</code> dla każdego elementu.
<code>Findall</code>	Zwraca listę wszystkich wystąpień wzorca lub wyrażenia regularnego w każdym łańcuchu.
<code>get</code>	Indeksuje każdy element (pobiera <i>i</i> -ty element).
<code>isalnum</code>	Odpowiednik wbudowanej metody <code>str.isalnum</code> .



Tabela 7.6. Wybrane metody obiektu Series (ciąg dalszy)

Metoda	Opis
<code>isalpha</code>	Odpowiednik wbudowanej metody <code>str.isalpha</code> .
<code>isdecimal</code>	Odpowiednik wbudowanej metody <code>str.isdecimal</code> .
<code>isdigit</code>	Odpowiednik wbudowanej metody <code>str.isdigit</code> .
<code>islower</code>	Odpowiednik wbudowanej metody <code>str.islower</code> .
<code>isnumeric</code>	Odpowiednik wbudowanej metody <code>str.isnumeric</code> .
<code>isupper</code>	Odpowiednik wbudowanej metody <code>str.isupper</code> .
<code>join</code>	Łączy łańcuchy w każdym elemencie obiektu Series przy użyciu przekazanego separatora.
<code>len</code>	Określa długość każdego łańcucha.
<code>lower, upper</code>	Zmiana wielkości liter; odpowiedniki funkcji <code>x.lower()</code> i <code>x.upper()</code> przetwarzających wszystkie elementy.
<code>match</code>	Korzysta z modułu <code>re.match</code> , przekazując za jego pomocą wyrażenia regularne w celu przetworzenia każdego elementu; w zależności od wyniku dopasowania zwraca wartość <code>True</code> lub <code>False</code> .
<code>pad</code>	Dodaje białe znaki po lewej lub prawej stronie łańcuchów; znaki mogą być również dodawane po obu stronach łańcuchów.
<code>center</code>	Odpowiednik <code>pad(side='both')</code> .
<code>repeat</code>	Duplikuje wartości (np. notacja <code>s.str.repeat(3)</code> jest odpowiednikiem operacji <code>x * 3</code> wykonanej dla każdego łańcucha).
<code>replace</code>	Zastępuje wystąpienia wzorca lub wyrażenia regularnego innym łańcuchem.
<code>slice</code>	Tnie łańcuchy wchodzące w skład serii.
<code>split</code>	Rozdziela łańcuchy na określonym znaku lub wyrażeniu regularnym.
<code>strip</code>	Ucina białe znaki po obu stronach łańcucha (dotyczy to również znaków nowego wiersza).
<code>rstrip</code>	Ucina białe znaki po prawej stronie.
<code>lstrip</code>	Ucina białe znaki po lewej stronie.

## 7.5. Dane kategoryczne

W tej sekcji wprowadzę typ `Categorical` obsługiwany przez pakiet `pandas`. Dowiesz się, jak można z niego korzystać w celu uzyskania lepszej wydajności i zmniejszenia zużycia pamięci podczas wykonywania niektórych operacji. Ponadto przedstawię wybrane narzędzia przeznaczone do używania danych kategorycznych w zastosowaniach związanych ze statystyką i uczeniem maszynowym.

### Kontekst i motywacja

Często kolumna tabeli zawiera powtarzające się instancje mniejszego zbioru różnych wartości. Podczas lektury poprzednich rozdziałów poznałeś już funkcje `unique` i `value_counts`, które wyciągają odmienne wartości z tablicy i określają częstotliwości ich występowania:

```
In [199]: values = pd.Series(['apple', 'orange', 'apple',
.....:                       'apple'] * 2)
```

```
In [200]: values
Out[200]:
0 apple
```

```
1 orange
2 apple
3 apple
4 apple
5 orange
6 apple
7 apple
dtype: object
```

```
In [201]: pd.unique(values)
Out[201]: array(['apple', 'orange'], dtype=object)
```

```
In [202]: pd.value_counts(values)
Out[202]:
apple    6
orange   2
dtype: int64
```

Wiele systemów obsługujących dane (składających je, przetwarzających w celach statystycznych itd.) korzysta ze specjalnych technik przedstawiania danych zawierających powtarzające się wartości, co pozwala na zmniejszenie objętości danych i przyspieszanie ich przetwarzania. Najlepszą praktyką stosowaną w przypadku hurtowni danych jest korzystanie z *tablic wymiaru* zawierających unikalne wartości. W takim przypadku pierwotne obserwacje są zapisywane w postaci kluczy numerycznych odwołujących się do tabeli wymiaru:

```
In [203]: values = pd.Series([0, 1, 0, 0] * 2)
```

```
In [204]: dim = pd.Series(['apple', 'orange'])
```

```
In [205]: values
Out[205]:
0 0
1 1
2 0
3 0
4 0
5 1
6 0
7 0
dtype: int64
```

```
In [206]: dim
Out[206]:
0 apple
1 orange
dtype: object
```

W celu przywrócenia początkowej struktury serii łańcuchów możemy skorzystać z metody take:

```
In [207]: dim.take(values)
Out[207]:
0 apple
1 orange
0 apple
0 apple
0 apple
```

```
1 orange
0 apple
0 apple
dtype: object
```

Przedstawienie danych za pomocą wartości liczbowych określamy mianem reprezentacji **kategorycznej** lub **kodowanej słownikowo**. Tablicę unikalnych wartości określamy mianem **kategorii, słownika** lub **poziomów** danych. W tej książce będę posługiwał się terminami *reprezentacja kategoryczna* i *kategorie*. Wartości liczbowe odwołujące się do kategorii określamy mianem **kodów kategorii** lub po prostu **kodów**.

Reprezentacja kategoryczna pozwala na znaczne zwiększenie wydajności przeprowadzania procesów analitycznych. Ponadto umożliwia przeprowadzanie transformacji kategorii bez modyfikowania kodów. Transformacje, które można przeprowadzić względnie niskim kosztem, to:

- zmiana nazw kategorii;
- dodanie nowej kategorii bez zmieniania kolejności i położenia utworzonych wcześniej kategorii.

## Rozszerzony typ Categorical w bibliotece pandas

Biblioteka pandas oferuje specjalny, rozszerzony typ danych Categorical przeznaczony do przechowywania danych, w których kategorie są przedstawiane (*zakodowane*) przy użyciu wartości liczbowych. Jest to popularna technika kompresji danych, w których wielokrotnie występują podobne wartości. Zapewnia znacznie wyższą wydajność przy mniejszym zużyciu pamięci, zwłaszcza w przypadku danych łańcuchowych. Przyjrzyjmy się jeszcze raz użytemu wcześniej obiektowi typu Series:

```
In [208]: fruits = ['apple', 'orange', 'apple', 'apple'] * 2
In [209]: N = len(fruits)
In [210]: rng = np.random.default_rng(seed=12345)
In [211]: df = pd.DataFrame({'fruit': fruits,
.....:                      'basket_id': np.arange(N),
.....:                      'count': np.random.randint(3, 15, size=N),
.....:                      'weight': np.random.uniform(0, 4, size=N)},
.....:                      columns=['basket_id', 'fruit', 'count', 'weight'])
In [212]: df
Out[212]:
```

	basket_id	fruit	count	weight
0	0	apple	11	1.564438
1	1	orange	5	1.331256
2	2	apple	12	2.393235
3	3	apple	6	0.746937
4	4	apple	5	2.691024
5	5	orange	12	3.767211
6	6	apple	10	0.992983
7	7	apple	11	3.795525

W zaprezentowanym przykładzie `df['fruit']` jest tablicą obiektów typu string (łańcuchów). Możemy dokonać jej konwersji na typ kategoryczny za pomocą następującej funkcji:

```
In [213]: fruit_cat = df['fruit'].astype('category')
```

```
In [214]: fruit_cat
```

```
Out[214]:
```

```
0  apple
1  orange
2  apple
3  apple
4  apple
5  orange
6  apple
7  apple
```

```
Name: fruit, dtype: category
```

```
Categories (2, object): [apple, orange]
```

Teraz wartościami parametru `fruit_cat` są instancje typu `pandas.Categorical`, do których można się odwoływać za pomocą atrybutu `array`:

```
In [215]: c = fruit_cat.array
```

```
In [216]: type(c)
```

```
Out[216]: pandas.core.categorical.Categorical
```

Obiekt `Categorical` ma atrybuty `categories` (kategorie) i `codes` (kody):

```
In [217]: c.categories
```

```
Out[217]: Index(['apple', 'orange'], dtype='object')
```

```
In [218]: c.codes
```

```
Out[218]: array([0, 1, 0, 0, 0, 1, 0, 0], dtype=int8)
```

Do tych danych można się łatwiej odwoływać za pomocą atrybutu dostępowego `cat`, który będzie wkrótce opisany w podrozdziale „Metody obiektu kategoriycznego”.

Powiązania pomiędzy kodami a kategoriami można uzyskać, stosując następującą przydatną sztuczkę:

```
In [219]: dict(enumerate(c.categories))
```

```
Out[219]: {0: 'apple', 1: 'orange'}
```

Możliwe jest wykonanie konwersji kolumny obiektu `DataFrame` na obiekt typu `Categorical` poprzez operację przypisania:

```
In [220]: df['fruit'] = df['fruit'].astype('category')
```

```
In [221]: df['fruit']
```

```
Out[221]:
```

```
0  apple
1  orange
2  apple
3  apple
4  apple
5  orange
6  apple
7  apple
```

```
Name: fruit, dtype: category
```

```
Categories (2, object): [apple, orange]
```

Instancje obiektów `pandas.Categorical` mogą być tworzone bezpośrednio na podstawie innych typów sekwencji Pythona:

```
In [222]: my_categories = pd.Categorical(['foo', 'bar', 'baz', 'foo', 'bar'])

In [223]: my_categories
Out[223]:
[foo, bar, baz, foo, bar]
Categories (3, object): [bar, baz, foo]
```

W przypadku uzyskania danych zakodowanych kategorycznie z innego źródła możesz skorzystać z alternatywnego konstruktora `from_codes`:

```
In [224]: categories = ['foo', 'bar', 'baz']

In [225]: codes = [0, 1, 2, 0, 0, 1]

In [226]: my_cats_2 = pd.Categorical.from_codes(codes, categories)

In [227]: my_cats_2
Out[227]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo, bar, baz]
```

Jeśli kolejność kategorii nie zostanie określona w sposób jawny, to podczas konwersji nie przyjmuje się żadnego konkretnego sposobu sortowania kategorii. W związku z tym kolejność elementów tablicy `categories` zależy od kolejności danych wejściowych. Podczas korzystania z `from_codes` lub innego konstruktora możesz określić konieczność ustawienia kategorii w sposób uporządkowany:

```
In [228]: ordered_cat = pd.Categorical.from_codes(codes, categories,
....:                                             ordered=True)

In [229]: ordered_cat
Out[229]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

Wygenerowany komunikat `[foo < bar < baz]` informuje o tym, że element `'foo'` znajduje się przed elementem `'bar'` itd. Kolejność instancji obiektu typu `Categorical` można określić później za pomocą metody `as_ordered`:

```
In [230]: my_cats_2.as_ordered()
Out[230]:
[foo, bar, baz, foo, foo, bar]
Categories (3, object): [foo < bar < baz]
```

Dane kategoryczne nie muszą być łańcuchami (mimo że we wszystkich przytoczonych przeze mnie przykładach są one właśnie łańcuchami). Tablica kategoryczna może zawierać dowolne wartości typu niemodyfikowalnego.

## Obliczenia na obiektach typu `Categorical`

Ogólnie rzecz biorąc, obiekty typu `Categorical` i ich odpowiedniki pozbawione kodowania kategorii działają tak samo. Niektóre funkcje pakietu `pandas` działają sprawniej z obiektami kategorycznymi (dotyczy do np. funkcji `groupby`). Niektóre funkcje mogą również korzystać z flagi `ordered`.

Przyjrzyjmy się przykładowi losowych danych numerycznych przetwarzanych za pomocą funkcji tworzącej koszyki danych — `pandas.qcut`. W wyniku zaprezentowanej operacji zwrócony zostanie obiekt `pandas.Categorical`. W zaprezentowanych wcześniej fragmentach kodu korzystałem z funkcji `pandas.cut`, ale nie opisywałem szczegółów działania kategorii:

```
In [231]: rng = np.random.default_rng(seed=12345)
In [232]: draws = rng.standard_normal(1000)
In [233]: draws[:5]
Out[233]: array([-1.4238, 1.2637, -0.8707, -0.2592, -0.0753])
```

Podzielmy dane na koszyki określające kwartyle i obliczmy kilka parametrów statystycznych:

```
In [234]: bins = pd.qcut(draws, 4)
In [235]: bins
Out[235]:
[(-3.121, -0.675], (0.687, 3.211], (-3.121, -0.675], (-0.675, 0.0134], (-0.675, 0.0134],
...,
(0.0134, 0.687], (0.0134, 0.687], (-0.675, 0.0134], (0.0134, 0.687], (-0.675, 0.0134]]
Length: 1000
Categories (4, interval[float64, right]): [(-3.121, -0.675] < (-0.675, 0.0134] <
(0.0134, 0.687] < (0.687, 3.211]]
```

Dokładne granice kwartyli są przydatne, ale w raportach lepiej jest używać ich nazw. Można w tym celu użyć argumentu `labels` metody `qcut`:

```
In [236]: bins = pd.qcut(draws, 4, labels=['Q1', 'Q2', 'Q3', 'Q4'])
In [237]: bins
Out[237]:
['Q1', 'Q4', 'Q1', 'Q2', 'Q2', ..., 'Q3', 'Q3', 'Q2', 'Q3', 'Q2']
Length: 1000
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
In [238]: bins.codes[:10]
Out[238]: array([0, 3, 0, 1, 1, 0, 0, 2, 2, 0], dtype=int8)
```

Oznaczone etykietami kategorie `bins` nie zawierają informacji o krawędziach koszyków danych, więc w celu obliczenia parametrów statystycznych skorzystamy z funkcji `groupby`:

```
In [239]: bins = pd.Series(bins, name='quartile')
In [240]: results = (pd.Series(draws)
.....:                  .groupby(bins)
.....:                  .agg(['count', 'min', 'max'])
.....:                  .reset_index())
In [241]: results
Out[241]:
  quartile  count      min      max
0        Q1    250 -3.119609 -0.678494
1        Q2    250 -0.673305  0.008009
2        Q3    250  0.018753  0.686183
3        Q4    250  0.688282  3.211418
```

Kolumna `'quartile'` obiektu wyjściowego zawiera informacje o kategoriach koszyków, w tym między innymi określa ich kolejność:

```
In [242]: results['quartile']
Out[242]:
0 Q1
1 Q2
2 Q3
3 Q4
Name: quartile, dtype: category
Categories (4, object): ['Q1' < 'Q2' < 'Q3' < 'Q4']
```

## Obiekty kategoriyczne a zwiększenie wydajności

Na początku tego podrozdziału wspomniałem, że typy kategoriyczne poprawiają wydajność operacji i zajmują mniej pamięci. Przeanalizujmy więc kilka przykładów. Przyjrzyjmy się obiektowi Series zawierającemu 10 milionów elementów należących do małej liczby unikalnych kategorii:

```
In [243]: N = 10_000_000
```

```
In [244]: labels = pd.Series(['foo', 'bar', 'baz', 'qux'] * (N // 4))
```

Teraz dokonam konwersji etykiet (labels) na dane kategoriyczne:

```
In [245]: categories = labels.astype('category')
```

Obiekt labels zajmuje o wiele więcej miejsca w pamięci od obiektu categories:

```
In [246]: labels.memory_usage(deep=True)
Out[246]: 600000128
```

```
In [247]: categories.memory_usage(deep=True)
Out[247]: 10000540
```

Konwersja na dane kategoriyczne wymaga również pewnych kosztów, ale są one ponoszone jednorazowo:

```
In [248]: %timeit _ = labels.astype('category')
CPU times: user 469 ms, sys: 106 ms, total: 574 ms
Wall time: 577 ms
```

Operacje grupowania mogą zostać znacznie przyspieszone w wyniku zastosowania danych kategoriycznych, ponieważ algorytm tego typu operacji korzysta z tablic zawierających kody w postaci liczb zastępujących łańcuchy znaków. Porównajmy wydajność metody value\_counts, która wewnętrznie korzysta z algorytmu funkcji groupby:

```
In [249]: %timeit labels.value_counts()
840 ms +- 10.9 ms per loop (mean +- std. dev. of 7 runs, 1 loop each)
```

```
In [250]: %timeit categories.value_counts()
30.1 ms +- 549 us per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

## Metody obiektu kategoriycznego

Obiekt typu Series zawierający dane kategoriyczne dysponuje kilkoma specjalnymi metodami podobnymi do specjalnych metod Series.str przeznaczonych do obsługi łańcuchów znaków. Metody te umożliwiają również wygodne uzyskanie dostępu do kategorii i kodów. Przyjrzyj się następującemu przykładowi obiektu typu Series:

```
In [251]: s = pd.Series(['a', 'b', 'c', 'd'] * 2)
```

```
In [252]: cat_s = s.astype('category')
```

```
In [253]: cat_s
```

```
Out[253]:
```

```
0 a
```

```
1 b
```

```
2 c
```

```
3 d
```

```
4 a
```

```
5 b
```

```
6 c
```

```
7 d
```

```
dtype: category
```

```
Categories (4, object): [a, b, c, d]
```

Specjalny *atrybut* `cat` umożliwia uzyskanie dostępu do metod kategoriycznych:

```
In [254]: cat_s.cat.codes
```

```
Out[254]:
```

```
0 0
```

```
1 1
```

```
2 2
```

```
3 3
```

```
4 0
```

```
5 1
```

```
6 2
```

```
7 3
```

```
dtype: int8
```

```
In [255]: cat_s.cat.categories
```

```
Out[255]: Index(['a', 'b', 'c', 'd'], dtype='object')
```

Żałujemy, że wiemy o tym, że rzeczywisty zbiór kategorii tych danych wykracza poza cztery zaobserwowane wartości. W celu zmodyfikowania zbioru kategorii możemy skorzystać z metody `set_categories`:

```
In [256]: actual_categories = ['a', 'b', 'c', 'd', 'e']
```

```
In [257]: cat_s2 = cat_s.cat.set_categories(actual_categories)
```

```
In [258]: cat_s2
```

```
Out[258]:
```

```
0 a
```

```
1 b
```

```
2 c
```

```
3 d
```

```
4 a
```

```
5 b
```

```
6 c
```

```
7 d
```

```
dtype: category
```

```
Categories (5, object): [a, b, c, d, e]
```

Dane wydają się niezmodyfikowane, ale nowe kategorie zostaną odzwierciedlone w korzystających z nich operacjach. Funkcja `value_counts` jest przykładem funkcji biorącej pod uwagę obecne kategorie:

```
In [259]: cat_s.value_counts()
```

```
Out[259]:
```

```
d 2
```

```
c 2
```



```

b 2
a 2
dtype: int64

In [260]: cat_s2.value_counts()
Out[260]:
d 2
c 2
b 2
a 2
e 0
dtype: int64

```

W przypadku dużych zbiorów danych konwersja na dane katagoryczne może być wygodnym narzędziem pozwalającym na zmniejszenie zapotrzebowania na pamięć i przyspieszenie przetwarzania danych. Po przefiltrowaniu dużego obiektu typu DataFrame lub Series wiele kategorii może nie wystąpić w danych. W celu rozwiązania tego problemu poprzez usunięcie kategorii, do których nie należy żadna obserwacja, możemy skorzystać z metody `remove_unused_categories`:

```

In [261]: cat_s3 = cat_s[cat_s.isin(['a', 'b'])]

In [262]: cat_s3
Out[262]:
0 a
1 b
4 a
5 b
dtype: category
Categories (4, object): ['a', 'b', 'c', 'd']

In [263]: cat_s3.cat.remove_unused_categories()
Out[263]:
0 a
1 b
4 a
5 b
dtype: category
Categories (2, object): ['a', 'b']

```

Listę obsługiwanych metod katagorycznych znajdziesz w tabeli 7.7.

Tabela 7.7. Metody katagoryczne obiektu Series obsługiwane przez pakiet pandas

Metoda	Opis
<code>add_categories</code>	Dodaje nowe (nieużywane) kategorie do końca listy istniejących kategorii.
<code>as_ordered</code>	Sprawia, że kategorie są uporządkowane.
<code>as_unordered</code>	Sprawia, że kategorie są nieuporządkowane.
<code>remove_categories</code>	Usuwa kategorie, przypisując wszelkim usuniętym elementom wartość <code>NaN</code> .
<code>remove_unused_categories</code>	Usuwa wszelkie wartości kategorii, które nie pojawiają się w danych.
<code>rename_categories</code>	Zastępuje kategorie wskazanym zestawem nowych nazw kategorii; zmiana numerów kategorii nie jest możliwa.
<code>reorder_categories</code>	Działa jak metoda <code>rename_categories</code> , ale może dodatkowo zmodyfikować obiekt wyjściowy w celu uporządkowania kategorii.
<code>set_categories</code>	Zastępuje kategorie wskazanym zestawem nowych kategorii; umożliwia dodawanie i usuwanie kategorii.

## Tworzenie fikcyjnych zmiennych używanych podczas modelowania

Podczas korzystania z narzędzi statystycznych i przeznaczonych do uczenia maszynowego bardzo często przeprowadza się konwersję danych kategorycznych na **zmienne fikcyjne** (ang. *dummy variable*) — jest to tzw. kodowanie *one hot*. Wymaga ono utworzenia ramki danych zawierającej kolumny dla wszystkich unikalnych kategorii. Kolumny te są wypełniane wartością 1 w przypadku wystąpienia danej kategorii, a wartością 0, gdy dana kategoria nie występuje.

Przyjrzyjmy się jeszcze raz temu przykładowi:

```
In [264]: cat_s = pd.Series(['a', 'b', 'c', 'd'] * 2, dtype='category')
```

Jak wspomniałem wcześniej w tym rozdziale, funkcja `pandas.get_dummies` dokonuje konwersji jednowymiarowych danych kategorycznych na obiekt `DataFrame` zawierający zmienną fikcyjną:


```
In [265]: pd.get_dummies(cat_s)
Out[265]:
   a  b  c  d
0  1  0  0  0
1  0  1  0  0
2  0  0  1  0
3  0  0  0  1
4  1  0  0  0
5  0  1  0  0
6  0  0  1  0
7  0  0  0  1
```

## 7.6. Podsumowanie

Skuteczne przygotowanie danych może znacznie poprawić produktywność, pozwalając na poświęcenie większej ilości czasu na analizę danych poprzez skrócenie czasu spędzonego na przygotowaniu danych do analizy. Podczas lektury tego rozdziału poznałeś różne narzędzia, ale pamiętaj o tym, że nie są to wszystkie dostępne narzędzia przeznaczone do wstępnej obróbki danych. W kolejnym rozdziale opiszę zagadnienia związane z funkcjami pakietu `pandas` przeznaczonymi do łączenia i grupowania.

# 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>

GRUPA  
**Helion** 

*Wes McKinney zaktualizował swoją książkę, aby była podstawowym źródłem informacji o wszystkich zagadnieniach związanych z analizą danych przy użyciu języka Python i biblioteki pandas. Gorąco polecam tę pozycję!*

Paul Barry, wykładowca i autor książek

Wprawny analityk danych potrafi z nich uzyskać wiedzę ułatwiającą podejmowanie trafnych decyzji. Od kilku lat można do tego używać nowoczesnych narzędzi Pythona, które zbudowano specjalnie do tego celu. Praca z nimi nie wymaga głębokiej znajomości statystyki czy algebry. Aby cieszyć się uzyskanymi rezultatami, wystarczy się wprawić w stosowaniu kilku pakietów i środowisk Pythona.

Ta książka jest trzecim, starannie zaktualizowanym wydaniem wyczerpującego przewodnika po narzędziach analitycznych Pythona. Uwzględnia Pythona 3.0 i bibliotekę pandas 1.4. Została napisana w przystępny sposób, a poszczególne zagadnienia bogato zilustrowano przykładami, studiami rzeczywistych przypadków i fragmentami kodu. W trakcie lektury nauczysz się korzystać z możliwości oferowanych przez pakiety pandas i NumPy, a także środowiska IPython i Jupyter. Nie zabrakło wskazówek dotyczących używania uniwersalnych narzędzi przeznaczonych do ładowania, czyszczenia, przekształcania i łączenia zbiorów danych. Pozycję docenią analitycy zamierzający zacząć pracę w Pythonie, jak również programiści Pythona, którzy chcą się zająć analizą danych i obliczeniami naukowymi.

### Dzięki książce nauczysz się:

- eksplorować dane za pomocą powłoki IPython i środowiska Jupyter
- korzystać z funkcji pakietów NumPy i pandas
- używać pakietu matplotlib do tworzenia czytelnych wizualizacji
- analizować i przetwarzać dane regularnych i nieregularnych szeregów czasowych
- rozwiązywać rzeczywiste problemy analityczne

### Wes McKinney

twórca oprogramowania open source, autor projektu pandas i współtwórca Apache Arrow. Członek The Apache Software Foundation, a także PMC Apache Parquet. Obecnie pełni funkcję dyrektora technicznego Voltron Data, gdzie zajmuje się przyspieszonymi technologiami obliczeniowymi opartymi na Apache Arrow.

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

KOD KORZYŚCI  
Sięgnij po więcej! ▶



ISBN 978-83-8322-323-0



Cena: 119,00 zł