

O'REILLY®

Python i Excel

Nowoczesne środowisko
do automatyzacji i analizy danych



Helion 

Felix Zumstein

Tytuł oryginału: Python for Excel: A Modern Environment for Automation and Data Analysis

Tłumaczenie: Leszek Sagalara

ISBN: 978-83-283-8287-9

© 2022 Helion S.A.

Authorized Polish translation of the English edition Python for Excel

ISBN 9781492081005 © 2021 Zoomer Analytics LLC.

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.

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.

Helion S.A.

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/pytexc.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/pytexc>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Wprowadzenie	11
--------------------	----

CZĘŚĆ I. Wprowadzenie do Pythona **17**

Rozdział 1. Dlaczego Python w Excelu? **19**

Excel jest językiem programowania	20
Excel w wiadomościach	21
Najlepsze praktyki programistyczne	21
Nowoczesny Excel	26
Python dla Excela	27
Czytelność i łatwość utrzymania	28
Biblioteka standardowa i menedżer pakietów	29
Obliczenia naukowe	30
Nowoczesne cechy języka	31
Kompatybilność międzyplatformowa	32
Podsumowanie	32

Rozdział 2. Środowisko programistyczne **34**

Dystrybucja Anaconda Python	35
Instalacja	35
Anaconda Prompt	35
REPL: interaktywna sesja Pythona	38
Menedżery pakietów: Conda i pip	39
Środowiska Condy	41
Notatniki Jupyter	41
Uruchamianie notatników Jupyter	42
Komórki notatnika	43
Tryb edycji a tryb poleceń	44
Kolejność uruchamiania ma znaczenie	46
Zamykanie notatników Jupyter	46

Visual Studio Code	47
Instalacja i konfiguracja	49
Uruchamianie skryptu Pythona	51
Podsumowanie	54
Rozdział 3. Wprowadzenie do Pythona	55
Typy danych	55
Obiekty	56
Typy liczbowe	57
Logiczny typ danych	59
Łańcuchy znaków	60
Indeksowanie i wycinanie	61
Indeksowanie	61
Wycinanie	62
Struktury danych	63
Listy	63
Słowniki	65
Krotki	66
Zbiory	67
Przeływ sterowania	68
Blok kodu i instrukcja pass	68
Instrukcja if i wyrażenia warunkowe	68
Pętle for i while	69
Lista, słownik i zbiory składane	72
Organizacja kodu	73
Funkcje	73
Moduły i instrukcja import	75
Klasa datetime	76
PEP 8 — przewodnik stylu kodowania w Pythonie	78
PEP 8 i VS Code	80
Informacje o typie	80
Podsumowanie	81

CZĘŚĆ II. Wprowadzenie do biblioteki pandas **83**

Rozdział 4. Podstawy NumPy	85
Pierwsze kroki z NumPy	85
Tablica NumPy	85
Wektoryzacja i rozgłaszanie	87
Funkcje uniwersalne	88

Tworzenie tablic i operowanie nimi	89
Pobieranie i wybieranie elementów tablicy	89
Przydatne konstruktory tablicowe	90
Widok a kopia	90
Podsumowanie	91
Rozdział 5. Analiza danych z biblioteką pandas	92
DataFrame i Series	92
Indeks	94
Kolumny	96
Operowanie danymi	97
Wybieranie danych	98
Ustawianie danych	103
Brakujące dane	105
Zduplikowane dane	106
Operacje arytmetyczne	107
Praca z kolumnami tekstowymi	109
Stosowanie funkcji	109
Widok a kopia	110
Łączenie obiektów DataFrame	111
Konkatenacja	111
Operacje join i merge	112
Statystyka opisowa i agregacja danych	114
Statystyka opisowa	114
Grupowanie	115
Funkcje pivot_table i melt	116
Tworzenie wykresów	117
Matplotlib	117
Plotly	119
Importowanie i eksportowanie obiektów DataFrame	121
Eksportowanie plików CSV	122
Importowanie plików CSV	123
Podsumowanie	124
Rozdział 6. Analiza szeregów czasowych za pomocą pandas	126
DatetimeIndex	127
Tworzenie DatetimeIndex	127
Filtrowanie DatetimeIndex	129
Praca ze strefami czasowymi	130

Typowe operacje na szeregach czasowych	131
Przesunięcia i zmiany procentowe	131
Zmiana podstawy i korelacja	133
Resampling	136
Okna kroczące	137
Ograniczenia związane z pandas	138
Podsumowanie	138

CZĘŚĆ III. Odczytywanie i zapisywanie plików Excela bez Excela **139**

Rozdział 7. Operowanie plikami Excela za pomocą pandas	141
Studium przypadku: raportowanie w Excelu	141
Odczytywanie i zapisywanie plików Excela za pomocą pandas	144
Funkcja read_excel i klasa ExcelFile	145
Metoda to_excel i klasa ExcelWriter	149
Ograniczenia związane z używaniem pandas z plikami Excela	151
Podsumowanie	151

Rozdział 8. Manipulowanie plikami Excela za pomocą pakietów do odczytu i zapisu	152
Pakiety do odczytu i zapisu	152
Kiedy używać którego pakietu?	153
Moduł excel.py	154
openpyxl	155
XlsxWriter	159
pyxlsb	161
xlrd, xlwt i xlutils	161
Zaawansowane zagadnienia związane z odczytem i zapisem	164
Praca z dużymi plikami Excela	164
Formatowanie obiektów DataFrame w Excelu	168
Studium przypadku (nowe podejście): raportowanie w Excelu	172
Podsumowanie	174

CZĘŚĆ IV. Programowanie aplikacji Excel za pomocą xlwings **175**

Rozdział 9. Automatyzacja Excela	177
Pierwsze kroki z xlwings	178
Excel jako przeglądarka danych	178
Model obiektowy Excela	179
Uruchamianie kodu VBA	185

Konwertery, opcje i kolekcje	186
Praca z obiektami DataFrame	186
Konwertery i opcje	187
Wykresy, obrazy i zdefiniowane nazwy	189
Studium przypadku (nowe podejście): raportowanie w Excelu	192
Zaawansowane zagadnienia związane z xlwings	194
Podstawy xlwings	194
Poprawa wydajności	196
Jak obejść brakującą funkcjonalność?	197
Podsumowanie	198
Rozdział 10. Narzędzia Excela działające w oparciu o język Python	199
Wykorzystanie Excela jako frontendu za pomocą xlwings	199
Dodatek do Excela	200
Polecenie quickstart	201
Przycisk Run main	202
Funkcja RunPython	203
Wdrażanie	206
Zależność od Pythona	207
Autonomiczne skoroszyty: sposób na pozbycie się dodatku xlwings	208
Hierarchia konfiguracji	208
Ustawienia	209
Podsumowanie	211
Rozdział 11. Tropiciel pakietów Pythona	212
Co będziemy budować?	212
Podstawowa funkcjonalność	213
Web API	214
Bazy danych	217
Wyjątki	225
Struktura aplikacji	227
Interfejs	227
Zaplecze	231
Debugowanie	234
Podsumowanie	235
Rozdział 12. Funkcje definiowane przez użytkownika (UDF)	236
Pierwsze kroki z funkcjami UDF	237
UDF z poleceniem quickstart	237
Studium przypadku: Google Trends	241
Wprowadzenie do Google Trends	242

Praca z obiektami DataFrames i dynamicznymi tablicami	243
Pobieranie danych z Google Trends	247
Tworzenie wykresów za pomocą funkcji UDF	251
Debugowanie funkcji UDF	252
Zaawansowane tematy dotyczące funkcji UDF	254
Podstawowa optymalizacja wydajności	254
Buforowanie	256
Dekorator sub	258
Podsumowanie	259
A. Środowiska Condy	261
B. Zaawansowane funkcjonalności VS Code	264
C. Zaawansowane pojęcia związane z Pythonem	269

Analiza danych z biblioteką pandas

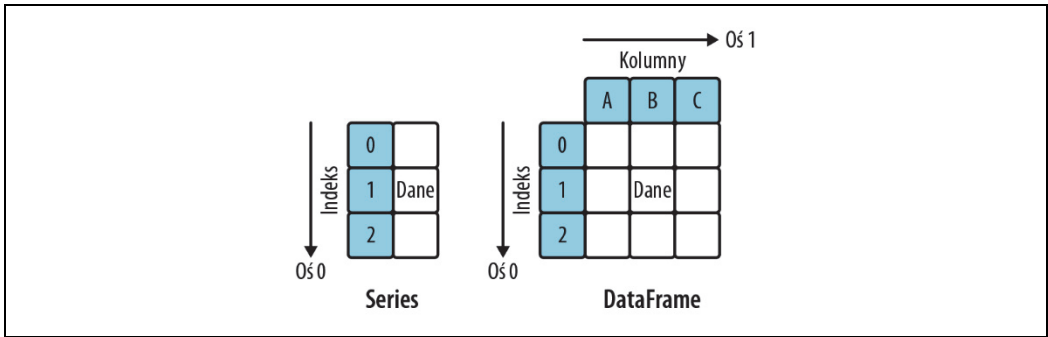
Ten rozdział jest wprowadzeniem do pandas (*Python Data Analysis Library* — biblioteka Pythona do analizy danych) lub — jak to lubię określać — opartego na Pythonie arkusza kalkulacyjnego z supermocami. Ma on tak potężne możliwości, że niektóre firmy, z którymi współpracowałem, zdołały całkowicie pozbyć się Excela, zastępując go połączeniem notatników Jupyter i pandas. Zakładam jednak, że jako czytelnik tej książki zachowasz Excela, a wtedy pandas posłuży Ci jako interfejs do wprowadzania i pobierania danych z arkuszy kalkulacyjnych. Biblioteka pandas sprawia, że zadania, które są szczególnie uciążliwe w Excelu, stają się łatwiejsze, zajmują mniej czasu i są bardziej odporne na błędy. Niektóre z tych zadań obejmują pobieranie dużych zbiorów danych z zewnętrznych źródeł oraz pracę z danymi statystycznymi, szeregami czasowymi i interaktywnymi wykresami. Najważniejsze atuty pandas to wektoryzacja i wyrównywanie danych. Jak już widzieliśmy w poprzednim rozdziale przy omawianiu tablic NumPy, wektoryzacja umożliwia pisanie zwięzłego kodu opartego na tablicach, podczas gdy wyrównywanie danych zapewnia, że nie dojdzie do niezgodności podczas pracy z wieloma zbiorami danych.

W tym rozdziale przedstawię całą procedurę analizy danych: zacznę od oczyszczania i przygotowywania danych, a następnie pokażę, jak wydobyć sens z większych zbiorów danych za pomocą agregacji, statystyki opisowej i wizualizacji. Na końcu rozdziału zobaczymy, jak można importować i eksportować dane za pomocą pandas. Ale na początek zapoznajmy się z głównymi strukturami danych w pandas: DataFrame i Series!

DataFrame i Series

DataFrame i Series są podstawowymi strukturami danych w pandas. Przedstawię je w tym podrozdziale, skupiając się na głównych komponentach DataFrame: indeksie, kolumnach i danych. **DataFrame** przypomina dwuwymiarową tablicę NumPy, ale ma etykiety kolumn i wierszy, a każda kolumna może zawierać różne typy danych. Wyodrębniając pojedynczą kolumnę lub wiersz z DataFrame, otrzymujemy jednowymiarowy obiekt **Series**, który przypomina jednowymiarową tablicę NumPy z etykietami. Gdy spojrzymy na strukturę DataFrame na rysunku 5.1, nie trzeba wiele wyobraźni, aby zauważyć, że obiekty DataFrame będą naszymi arkuszami kalkulacyjnymi opartymi na Pythonie.

Aby zobaczyć, jak łatwo można przejść z arkusza kalkulacyjnego na DataFrame, spojrzmy na tabelę Excela widoczną na rysunku 5.2, która przedstawia uczestników kursu online wraz z ich wynikami. Odpowiedni plik *course_participants.xlsx* znajdziesz w folderze *x1* w repozytorium towarzyszącym książce.



Rysunek 5.1. Series i DataFrame w pandas

	A	B	C	D	E	F
1	numer	imię	wiek	kraj	ocena	kontynent
2	1001	Mark		55 Włochy		4,5 Europa
3	1000	John		33 USA		6,7 Ameryka
4	1002	Tim		41 USA		3,9 Ameryka
5	1003	Jenny		12 Niemcy		9 Europa

Rysunek 5.2. course_participants.xlsx

Aby udostępnić tę tabelę Excela w Pythonie, zacznij od zaimportowania biblioteki pandas, a następnie użyj jej funkcji `read_excel`, która zwraca obiekt DataFrame:

```
In [1]: import pandas as pd
In [2]: pd.read_excel("x1/course_participants.xlsx")
Out[2]:
```

	numer	imię	wiek	kraj	ocena	kontynent
0	1001	Mark	55	Włochy	4.5	Europa
1	1000	John	33	USA	6.7	Ameryka
2	1002	Tim	41	USA	3.9	Ameryka
3	1003	Jenny	12	Niemcy	9.0	Europa



Funkcja `read_excel` w Pythonie 3.9

Jeśli uruchamiasz `pd.read_excel` z Pythonem w wersji 3.9 lub wyższej, upewnij się, że używasz pandas w wersji co najmniej 1.2, w przeciwnym razie podczas odczytu plików *xlsx* pojawi się błąd.

Jeśli uruchomisz ten kod w notatniku Jupyter, DataFrame będzie ładnie sformatowany jako tabela HTML, co jeszcze bardziej zbliży go do wyglądu tabeli w Excelu. Odczytywanie i zapisywanie plików Excela za pomocą pandas będzie tematem całego rozdziału 7., to był tylko wstępny przykład, aby pokazać, że arkusze kalkulacyjne i DataFrame są bardzo podobne. Utwórzmy teraz ponownie DataFrame od podstaw, bez odczytywania danych z pliku Excela: jednym ze sposobów utworzenia DataFrame jest dostarczenie danych w postaci listy zagnieżdżonej wraz z wartościami dla kolumn (`columns`) i indeksu (`index`):

```
In [3]: data=[["Mark", 55, "Italy", 4.5, "Europa"],
              ["John", 33, "USA", 6.7, "Ameryka"],
```

```

["Tim", 41, "USA", 3.9, "Ameryka"],
["Jenny", 12, " Niemcy", 9.0, "Europa"]]
df = pd.DataFrame(data=data,
                  columns=["imię", "wiek", "kraj",
                          "ocena", "kontynent"],
                  index=[1001, 1000, 1002, 1003])

df
Out[3]:
   imię  wiek  kraj  ocena  kontynent
1001  Mark   55  Włochy  4.5    Europa
1000  John   33   USA   6.7    Ameryka
1002  Tim   41   USA   3.9    Ameryka
1003  Jenny  12  Niemcy  9.0    Europa

```

Wywołując metodę `info`, uzyskasz kilka podstawowych informacji, przede wszystkim liczbę punktów danych oraz typy danych dla każdej kolumny:

```

In [4]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4 entries, 1001 to 1003
Data columns (total 5 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   imię        4 non-null      object
1   wiek        4 non-null      int64
2   kraj        4 non-null      object
3   ocena       4 non-null      float64
4   kontynent   4 non-null      object
dtypes: float64(1), int64(1), object(3)
memory usage: 192.0+ bytes

```

Jeśli interesuje Cię tylko typ danych w Twoich kolumnach, uruchom zamiast tego `df.dtypes`. Kolumny zawierające łańcuchy znaków lub mieszane typy danych będą miały typ danych `object`¹. Przyjrzyjmy się teraz bliżej indeksowi i kolumnom `DataFrame`.

Indeks

Etykiety wierszy `DataFrame` są nazywane **indeksem**. Jeśli nie masz sensownego indeksu, pomiń go podczas konstruowania `DataFrame`, a `pandas` automatycznie utworzy wtedy indeks oparty na liczbach całkowitych, zaczynając od zera. Widzieliśmy to w pierwszym przykładzie, kiedy odczytywaliśmy `DataFrame` z pliku Excela. Indeks umożliwia bibliotece `pandas` szybsze wyszukiwanie danych i jest niezbędny do wielu typowych operacji, np. łączenia dwóch obiektów `DataFrame`. Dostęp do indeksu uzyskujemy w następujący sposób:

```

In [5]: df.index
Out[5]: Int64Index([1001, 1000, 1002, 1003], dtype='int64')

```

Jeśli ma to znaczenie, nadaj indeksowi nazwę. Podążając za przykładem tabeli w Excelu, nadajmy mu nazwę `numer`:

```

In [6]: df.index.name = "numer"
df

```

¹ W `pandas` 1.0.0 wprowadzono wyspecjalizowany typ danych `string`, aby ułatwić niektóre operacje i uczynić je bardziej spójnymi z tekstem. Ponieważ jest on wciąż eksperymentalny, nie zamierzam go wykorzystywać w tej książce.

```
Out[6]:      imię  wiek   kraj  ocena kontynent
numer
1001   Mark   55  Włochy   4.5   Europa
1000   John   33   USA     6.7   Ameryka
1002   Tim    41   USA     3.9   Ameryka
1003   Jenny  12   Niemcy  9.0   Europa
```

W odróżnieniu od klucza głównego bazy danych indeks DataFrame może mieć duplikaty, ale wyszukiwanie wartości może być w takim przypadku wolniejsze. Aby przekształcić indeks w zwykłą kolumnę, użyj metody `reset_index`, a aby ustawić nowy indeks, użyj `set_index`. Jeśli nie chcesz stracić istniejącego indeksu podczas ustawiania nowego, upewnij się, że najpierw go zresetowałeś:

```
In [7]: # Metoda "reset_index" przekształca indeks w kolumnę, zastępując
# indeks indeksem domyślnym. Odpowiada to pierwszemu DataFrame,
# który wczytaliśmy z Excela.
```

```
df.reset_index()
```

```
Out[7]:      numer  imię  wiek   kraj  ocena kontynent
0     1001  Mark   55  Włochy   4.5   Europa
1     1000  John   33   USA     6.7   Ameryka
2     1002  Tim    41   USA     3.9   Ameryka
3     1003  Jenny  12   Niemcy  9.0   Europa
```

```
In [8]: # Metoda "reset_index" zmienia "numer" w zwykłą kolumnę,
# a "set_index" zmienia kolumnę "imię" w indeks.
```

```
df.reset_index().set_index("imię")
```

```
Out[8]:      numer  wiek   kraj  ocena kontynent
imię
Mark    1001   55  Włochy   4.5   Europa
John    1000   33   USA     6.7   Ameryka
Tim     1002   41   USA     3.9   Ameryka
Jenny   1003   12   Niemcy  9.0   Europa
```

Wykonując `df.reset_index().set_index("name")`, stosujesz **łańcuchowanie metod**: ponieważ `reset_index()` zwraca DataFrame, możesz bezpośrednio wywołać inną metodę DataFrame bez konieczności uprzedniego wypisywania wyniku pośredniego.



Metody DataFrame zwracają kopie

Za każdym razem, gdy na DataFrame wywołasz metodę w postaci `df.nazwa_metody()`, otrzymasz w wyniku kopię tego DataFrame z zastosowaną metodą, natomiast oryginalny obiekt DataFrame pozostanie nietknięty. Właśnie to zrobiliśmy poprzez wywołanie `df.reset_index()`. Gdybyś chciał zmienić oryginalny DataFrame, musiałbyś przypisać zwróconą wartość z powrotem do pierwotnej zmiennej, jak poniżej:

```
df = df.reset_index()
```

Ponieważ tego nie robimy, oznacza to, że nasza zmienna `df` nadal przechowuje swoje pierwotne dane. Następne przykłady również wywołują metody DataFrame, tzn. nie zmieniają oryginalnego obiektu DataFrame.

Aby zmienić indeks, użyj metody `reindex`:

```
In [9]: df.reindex([999, 1000, 1001, 1004])
Out[9]:      imię  wiek   kraj  ocena kontynent
numer
999     NaN   NaN   NaN   NaN     NaN
1000   John  33.0   USA     6.7   Ameryka
```

```
1001    Mark  55.0  Włochy    4.5  Europa
1004     NaN   NaN    NaN     NaN   NaN
```

Jest to pierwszy przykład wyrównywania danych: metoda `reindex` przejmie wszystkie wiersze, które pasują do nowego indeksu, i wprowadzi wiersze z brakującymi wartościami (NaN), dla których nie ma żadnych informacji. Pominięte elementy indeksu zostaną usunięte. Wartość NaN omówię nieco później w tym rozdziale. Na koniec, aby posortować indeks, użyj metody `sort_index`:

```
In [10]: df.sort_index()
Out[10]:
```

	imię	wiek	kraj	ocena	kontynent
numer					
1000	John	33	USA	6.7	Ameryka
1001	Mark	55	Włochy	4.5	Europa
1002	Tim	41	USA	3.9	Ameryka
1003	Jenny	12	Niemcy	9.0	Europa

Jeśli zamiast tego chcesz posortować wiersze według jednej kolumny (lub kilku), użyj `sort_values`:

```
In [11]: df.sort_values(["kontynent", "wiek"])
Out[11]:
```

	imię	wiek	kraj	ocena	kontynent
numer					
1000	John	33	USA	6.7	Ameryka
1002	Tim	41	USA	3.9	Ameryka
1003	Jenny	12	Niemcy	9.0	Europa
1001	Mark	55	Włochy	4.5	Europa

Przykład pokazuje, jak posortować najpierw według kolumny `kontynent`, a następnie według kolumny `wiek`. Jeśli chcesz sortować tylko według jednej kolumny, możesz również podać nazwę kolumny jako łańcuch znaków:

```
df.sort_values("kontynent")
```

W ten sposób poznaliśmy podstawy działania indeksów. Zwróćmy teraz naszą uwagę na ich poziomy odpowiednik, czyli kolumny `DataFrame`!

Kolumny

Aby uzyskać informacje o kolumnach `DataFrame`, uruchom następujący kod:

```
In [12]: df.columns
Out[12]: Index(['imię', 'wiek', 'kraj', 'ocena', 'kontynent'], dtype='object')
```

Jeśli przy konstruowaniu `DataFrame` nie podasz żadnych nazw kolumn, `pandas` ponumeruje je za pomocą liczb całkowitych, zaczynając od zera. W przypadku kolumn zwykle nie jest to dobry pomysł, ponieważ reprezentują one zmienne i dlatego są łatwe do nazwania. Nazwy do nagłówek kolumn przypisujemy w ten sam sposób, jak robiliśmy to z indeksem:

```
In [13]: df.columns.name = "cechy"
df
Out[13]:
```

cechy	imię	wiek	kraj	ocena	kontynent
numer					
1000	John	33	USA	6.7	Ameryka
1001	Mark	55	Włochy	4.5	Europa
1002	Tim	41	USA	3.9	Ameryka
1003	Jenny	12	Niemcy	9.0	Europa

Jeśli nie odpowiadają Ci nazwy kolumn, możesz je zmienić:

```
In [14]: df.rename(columns={"imię": "Imię", "wiek": "Wiek"})
Out[14]:
```

cechy	Imię	Wiek	kraj	ocena	kontynent
numer					
1000	John	33	USA	6.7	Ameryka
1001	Mark	55	Włochy	4.5	Europa
1002	Tim	41	USA	3.9	Ameryka
1003	Jenny	12	Niemcy	9.0	Europa

Jeśli chcesz usunąć kolumny, użyj następującej składni (przykład pokazuje, jak usunąć jednocześnie kolumny i indeksy):

```
In [15]: df.drop(columns=["imię", "kraj"],
                    index=[1000, 1003])
Out[15]:
```

cechy	wiek	ocena	kontynent
numer			
1001	55	4.5	Europa
1002	41	3.9	Ameryka

Kolumny i indeks DataFrame są reprezentowane przez obiekt `Index`, tak więc możesz zamienić kolumny na wiersze (i odwrotnie) poprzez transpozycję DataFrame:

```
In [16]: df.T # Skrót od df.transpose()
Out[16]:
```

numer	1001	1000	1002	1003
cechy				
imię	Mark	John	Tim	Jenny
wiek	55	33	41	12
kraj	Włochy	USA	USA	Niemcy
ocena	4.5	6.7	3.9	9
kontynent	Europa	Ameryka	Ameryka	Europa

Warto przy tym pamiętać, że nasz obiekt DataFrame `df` jest wciąż niezmienniony, ponieważ nigdy nie przypisywaliśmy DataFrame zwracanego z wywołania metody z powrotem do oryginalnej zmiennej `df`. Jeżeli chciałbyś zmienić kolejność kolumn DataFrame, mógłbyś użyć metody `reindex`, której użyliśmy z indeksem, ale wybieranie kolumn w porządkanej kolejności jest często bardziej intuicyjne:

```
In [17]: df.loc[:, ["kontynent", "kraj", "imię", "wiek", "ocena"]]
Out[14]:
```

cechy	kontynent	kraj	imię	wiek	ocena
numer					
1001	Europa	Włochy	Mark	55	4.5
1000	Ameryka	USA	John	33	6.7
1002	Ameryka	USA	Tim	41	3.9
1003	Europa	Niemcy	Jenny	12	9.0

Ten ostatni przykład wymaga kilku wyjaśnień; metodę `loc` i działanie selekcji danych omówię w następnym podrozdziale.

Operowanie danymi

Dane pochodzące ze świata rzeczywistego raczej nie są podawane na srebrnej tacy, więc zanim zaczniesz z nimi pracować, musisz je oczyścić i przekształcić w strawną formę. Zacniemy od tego, jak wybierać dane z DataFrame, jak je zmieniać oraz jak sobie radzić z brakującymi i zduplikowanymi danymi. Następnie wykonamy kilka obliczeń z wykorzystaniem obiektów DataFrame i zobaczymy, jak pracować z danymi tekstowymi. Na zakończenie tego podrozdziału dowiemy się, kiedy pandas zwraca widok, a kiedy kopię danych. Sporo koncepcji zawartych w tym podrozdziale jest powiązanych z tym, z czym mieliśmy już do czynienia przy okazji tablic NumPy w poprzednim rozdziale.

Wybieranie danych

Zacznijmy od dostępu do danych na podstawie etykiety i pozycji, a następnie przyjrzymy się innym metodom, w tym indeksowaniu logicznemu i wybieraniu danych poprzez MultiIndex.

Wybór na podstawie etykiety

Najczęstszym sposobem dostępu do danych DataFrame jest odwołanie się do etykiety. Aby określić, które wiersze i kolumny chcesz pobrać, użyj atrybutu `loc` (skrót od ang. *location* — położenie):

```
df.loc[wybór_wierszy, wybór_kolumn]
```

Atrybut `loc` obsługuje notację wycinków i dlatego akceptuje dwukropek, aby wybrać odpowiednio wszystkie wiersze lub wszystkie kolumny. Dodatkowo możesz podać listy z etykietami, jak również pojedynczą nazwę kolumny lub wiersza. Spójrz na tabelę 5.1, w której przedstawiono kilka przykładów wybierania różnych części z naszego przykładowego DataFrame `df`.

Tabela 5.1. Wybieranie danych na podstawie etykiet

Wybór	Typ zwracanych danych	Przykład
Pojedyncza wartość	Skalar	<code>df.loc[1000, "kraj"]</code>
Jedna kolumna (1 wymiar)	Series	<code>df.loc[:, "kraj"]</code>
Jedna kolumna (2 wymiary)	DataFrame	<code>df.loc[:, ["kraj"]]</code>
Wiele kolumn	DataFrame	<code>df.loc[:, ["kraj", "wiek"]]</code>
Zakres kolumn	DataFrame	<code>df.loc[:, "imię":"kraj"]</code>
Jeden wiersz (1 wymiar)	Series	<code>df.loc[1000, :]</code>
Jeden wiersz (2 wymiary)	DataFrame	<code>df.loc[[1000], :]</code>
Wiele wierszy	DataFrame	<code>df.loc[[1003, 1000], :]</code>
Zakres wierszy	DataFrame	<code>df.loc[1000:1002, :]</code>



Przy wycinaniu etykiet przedziały są zamknięte

Używanie notacji wycinków z wykorzystaniem etykiet jest niespójne z tym, jak działa wszystko inne w Pythonie i pandas: końcowa wartość jest *uwzględniana* w wycinku.

Wykorzystując naszą wiedzę z tabeli 5.1, zastosujemy atrybut `loc` do wybrania skalarów, obiektów Series i DataFrame:

```
In [18]: # Użycie skalarów do wyboru zarówno wierszy, jak i kolumn, zwraca skalar
df.loc[1001, "imię"]
Out[18]: 'Mark'
In [19]: # Użycie skalara do wyboru wiersza lub kolumny zwraca obiekt Series
df.loc[[1001, 1002], "wiek"]
Out[19]: numer
1001    55
1002    41
Name: wiek, dtype: int64
In [20]: # Wybranie wielu wierszy i kolumn zwraca obiekt DataFrame
df.loc[:1002, ["imię", "kraj"]]
Out[20]: cechy  imię  kraj
numer
```

1001	Mark	Włochy
1000	John	USA
1002	Tim	USA

Ważne jest, abyś zrozumiał różnicę pomiędzy DataFrame z jedną lub wieloma kolumnami a Series: nawet z jedną kolumną, obiekty DataFrame są dwuwymiarowe, podczas gdy Series są jednowymiarowe. Zarówno DataFrame, jak i Series mają indeks, ale tylko DataFrame ma nagłówki kolumn. Kiedy wybierzesz kolumnę jako Series, nagłówek kolumny staje się nazwą obiektu Series. Wiele funkcji i metod będzie działać zarówno na obiektach Series, jak i na DataFrame, ale podczas wykonywania obliczeń arytmetycznych zachowanie będzie różne: w przypadku obiektów DataFrame pandas wyrównuje dane zgodnie z nagłówkami kolumn — więcej na ten temat w dalszej części tego rozdziału.



Skrót do wyboru kolumn

Ponieważ wybieranie kolumn jest bardzo powszechną operacją, pandas oferuje skrót. Zamiast:

```
df.loc[:, wybór_kolumn]
```

możesz napisać:

```
df[wybór_kolumn]
```

Przykładowo `df["kraj"]` zwraca Series z naszego przykładowego DataFrame, a `df[["imię", "kraj"]]` zwraca DataFrame z dwiema kolumnami.

Wybór na podstawie pozycji

Wybieranie podzbioru DataFrame na podstawie pozycji odpowiada temu, co robiliśmy na początku tego rozdziału z tablicami NumPy. W przypadku obiektów DataFrame musisz jednak użyć atrybutu `iloc` (skrót od ang. *integer location* — położenie całkowite):

```
df.iloc[wybór_wierszy, wybór_kolumn]
```

Używając wycinków, mamy do czynienia ze standardowymi przedziałami półotwartymi. Tabela 5.2 przedstawia te same przypadki, którym przyglądaliśmy się wcześniej w tabeli 5.1.

Tabela 5.2. Wybieranie danych na podstawie pozycji

Wybór	Typ zwracanych danych	Przykład
Pojedyncza wartość	Skalar	<code>df.iloc[1, 2]</code>
Jedna kolumna (1 wymiar)	Series	<code>df.iloc[:, 2]</code>
Jedna kolumna (2 wymiary)	DataFrame	<code>df.iloc[:, [2]]</code>
Wiele kolumn	DataFrame	<code>df.iloc[:, [2, 1]]</code>
Zakres kolumn	DataFrame	<code>df.iloc[:, :3]</code>
Jeden wiersz (1 wymiar)	Series	<code>df.iloc[1, :]</code>
Jeden wiersz (2 wymiary)	DataFrame	<code>df.iloc[[1], :]</code>
Wiele wierszy	DataFrame	<code>df.iloc[[3, 1], :]</code>
Zakres wierszy	DataFrame	<code>df.iloc[1:3, :]</code>

Oto przykłady użycia atrybutu `iloc` — ponownie z tymi samymi próbkami, których używaliśmy wcześniej z `loc`:


```

In [21]: df.iloc[0, 0] # Zwraca skalar
Out[21]: 'Mark'
In [22]: df.iloc[[0, 2], 1] # Zwraca Series
Out[22]: numer
          1001    55
          1002    41
          Name: wiek, dtype: int64
In [23]: df.iloc[:3, [0, 2]] # Zwraca DataFrame
Out[23]: cechy  imię  kraj
          numer
          1001  Mark  Włochy
          1000  John  USA
          1002  Tim   USA

```

Wybieranie danych na podstawie etykiety lub pozycji nie jest jedynym sposobem dostępu do podzbioru DataFrame. Innym ważnym rozwiązaniem jest użycie indeksowania logicznego; zobaczmy, jak to działa!

Wybieranie przy użyciu indeksowania logicznego

Indeksowanie logiczne (ang. *boolean indexing*) odnosi się do wyboru podzbiorów DataFrame za pomocą obiektów Series lub DataFrame, których dane składają się tylko z wartości True lub False. Obiekty Series typu logicznego są używane do wybierania określonych kolumn i wierszy DataFrame, podczas gdy obiekty DataFrame typu logicznego są używane do wybierania określonych wartości w całym DataFrame. Najczęściej indeksowanie logiczne jest używane do filtrowania wierszy DataFrame. Potraktuj to jak funkcję Autofiltru w Excelu. Oto przykład użycia tej funkcji do filtrowania obiektu DataFrame w taki sposób, aby widoczne były tylko osoby, które mieszkają w USA i mają więcej niż 40 lat:

```

In [24]: tf = (df["wiek"] > 40) & (df["kraj"] == "USA")
          tf # To jest obiekt Series zawierający wyłącznie wartości True lub False
Out[24]: numer
          1001 False
          1000 False
          1002 True
          1003 False
          dtype: bool
In [25]: df.loc[tf, :]
Out[25]: cechy  imię  wiek  kraj  ocena  kontynent
          numer
          1002  Tim   41   USA    3.9   Ameryka

```

Są dwie rzeczy, które muszę tu wyjaśnić. Po pierwsze, z powodu ograniczeń technicznych nie można używać operatorów logicznych Pythona z rozdziału 3. z obiektami DataFrame. Zamiast tego należy użyć symboli, jak pokazano w tabeli 5.3.

Tabela 5.3. Operatory logiczne

Podstawowe operatory logiczne Pythona	DataFrames i Series
and	&
or	
not	~

Po drugie, jeśli masz więcej niż jeden warunek, upewnij się, że umieściłeś każde wyrażenie logiczne w nawiasach, aby nie przeszkodziło Ci pierwszeństwo operatorów: np. & ma wyższy priorytet niż ==. Tak więc bez nawiasów wyrażenie z naszego przykładowego kodu zostałoby zinterpretowane jako:

```
df["wiek"] > (40 & df["kraj"]) == "USA"
```

Jeśli chcesz filtrować indeks, możesz się do niego odwołać poprzez `df.index`:

```
In [26]: df.loc[df.index > 1001, :]  
Out[26]: cechy    imię    wiek    kraj    ocena    kontynent  
         numer  
         1002    Tim     41     USA     3.9     Ameryka  
         1003    Jenny     12    Niemcy    9.0     Europa
```

W tych przypadkach, w których użyłbyś operatora `in` z podstawowymi strukturami danych Pythona, takimi jak listy, z Series użyj `isin`. W ten sposób można przefiltrować DataFrame, wybierając uczestników z Włoch i Niemiec:

```
In [27]: df.loc[df["kraj"].isin(["Włochy", "Niemcy"]), :]  
Out[27]: cechy    imię    wiek    kraj    ocena    kontynent  
         numer  
         1001    Mark     55    Włochy    4.5     Europa  
         1003    Jenny     12    Niemcy    9.0     Europa
```

Choć obiekty Series typu logicznego są przekazywane za pomocą atrybutu `loc`, obiekty DataFrame oferują specjalną składnię bez `loc`, umożliwiającą wybranie wartości z DataFrame zawierającego wartości logiczne:

```
df[logiczny_df]
```

Jest to szczególnie przydatne, jeżeli masz obiekty DataFrame, które zawierają tylko liczby. Przekazanie DataFrame z wartościami logicznymi zwraca DataFrame z wartościami NaN wszędzie tam, gdzie DataFrame typu logicznego ma wartość `False`. Wartość NaN omówię bardziej szczegółowo za chwilę. Zacznijmy od stworzenia nowego przykładowego DataFrame o nazwie `rainfall`, który składa się tylko z liczb:

```
In [28]: # Może to być roczna suma opadów w milimetrach  
rainfall = pd.DataFrame(data={"Miasto 1": [300.1, 100.2],  
                              "Miasto 2": [400.3, 300.4],  
                              "Miasto 3": [1000.5, 1100.6]})  
  
rainfall  
Out[28]:   Miasto 1  Miasto 2  Miasto 3  
0      300.1     400.3    1000.5  
1      100.2     300.4     1100.6  
  
In [29]: rainfall < 400  
Out[29]:   Miasto 1  Miasto 2  Miasto 3  
0      True     False     False  
1      True     True     False  
  
In [30]: rainfall[rainfall < 400]  
Out[30]:   Miasto 1  Miasto 2  Miasto 3  
0      300.1     NaN     NaN  
1      100.2     300.4     NaN
```

Zauważ, że w tym przykładzie użyłem słownika do skonstruowania nowego obiektu DataFrame — jest to często wygodne, jeśli dane istnieją już w tej formie. Taki sposób pracy z wartościami logicznymi jest najczęściej używany do odfiltrowania określonych wartości, takich jak elementy odstające.

Na zakończenie części poświęconej selekcji danych przedstawię specjalny typ indeksu określany mianem `MultiIndex`.

Wybieranie poprzez `MultiIndex`

`MultiIndex` to indeks, który ma więcej niż jeden poziom. Umożliwia on hierarchiczne grupowanie danych i zapewnia łatwy dostęp do podzbiorów. Dla przykładu, jeśli ustawisz indeks naszego przykładowego obiektu `DataFrame` `df` jako połączenie etykiet kontynent i kraj, możesz łatwo wybrać wszystkie wiersze z określonym kontynentem:

```
In [31]: # MultiIndex trzeba posortować
df_multi = df.reset_index().set_index(["kontynent", "kraj"])
df_multi = df_multi.sort_index()
df_multi

Out[31]:
```

	cechy	numer	imię	wiek	ocena
kontynent	kraj				
Ameryka	USA	1000	John	33	6.7
	USA	1002	Tim	41	3.9
Europa	Niemcy	1003	Jenny	12	9.0
	Włochy	1001	Mark	55	4.5

```
In [32]: df_multi.loc["Europa", :]
Out[32]:
```

cechy	numer	imię	wiek	ocena
kraj				
Niemcy	1003	Jenny	12	9.0
Włochy	1001	Mark	55	4.5

Zauważ, że `MultiIndex` jest nieco upiękaszony na wyjściu dzięki temu, że pandas nie powtarza najbardziej lewego poziomu indeksu (kontynenty) dla każdego wiersza. Zamiast tego wypisuje kontynent tylko wtedy, gdy ten się zmienia. Wybór pomiędzy wieloma poziomami indeksu odbywa się poprzez przekazanie krotki:

```
In [33]: df_multi.loc[("Europa", "Włochy"), :]
Out[33]:
```

cechy	numer	imię	wiek	ocena	
kontynent	kraj				
Europa	Włochy	1001	Mark	55	4.5

Jeśli chcesz wybiórczo zresetować część `MultiIndexu`, podaj poziom jako argument. Zero to pierwsza kolumna od lewej:

```
In [34]: df_multi.reset_index(level=0)
Out[34]:
```

cechy	kontynent	numer	imię	wiek	ocena
kraj					
USA	Ameryka	1000	John	33	6.7
USA	Ameryka	1002	Tim	41	3.9
Niemcy	Europa	1003	Jenny	12	9.0
Włochy	Europa	1001	Mark	55	4.5

Chociaż w tej książce nie będziemy ręcznie tworzyć `MultiIndexu`, istnieją pewne operacje, takie jak grupy, które spowodują, że pandas zwróci obiekt `DataFrame` z `MultiIndexem`, więc warto wiedzieć, co to jest. Z grupy spotkamy się w dalszej części tego rozdziału.

Teraz, gdy znasz już różne sposoby *wybijania* danych, nadszedł czas, aby dowiedzieć się, jak *zmieniać* dane.

Ustawianie danych

Najprostszym sposobem zmiany danych w DataFrame jest przypisanie wartości do określonych elementów za pomocą atrybutów `loc` lub `iloc`. To jest punkt wyjścia tej sekcji, zanim przejdziemy do innych sposobów operowania na istniejących obiektach DataFrame: zastępowania wartości i dodawania nowych kolumn.

Ustawianie danych na podstawie etykiety lub pozycji

Jak wspomniano wcześniej w tym rozdziale, jeśli wywołamy metodę DataFrame, taką jak `df.reset_index()`, zostanie ona zawsze zastosowana do kopii, pozostawiając oryginalny obiekt DataFrame nietknięty. Jednakże przypisywanie wartości za pomocą atrybutów `loc` i `iloc` zmienia oryginalny DataFrame. Ponieważ nie chcę zmieniać naszego DataFrame `df`, pracuję tutaj na kopii, którą nazwałem `df2`. Jeśli chcesz zmienić pojedynczą wartość, wykonaj poniższy kod:

```
In [35]: # Najpierw skopiuj DataFrame, aby oryginał pozostał nietknięty
df2 = df.copy()
In [36]: df2.loc[1000, "imię"] = "JOHN"
df2
Out[36]:
```

cechy numer	imię	wiek	kraj	ocena	kontynent
1001	Mark	55	Włochy	4.5	Europa
1000	JOHN	33	USA	6.7	Ameryka
1002	Tim	41	USA	3.9	Ameryka
1003	Jenny	12	Niemcy	9.0	Europa

Możesz także zmieniać wiele wartości jednocześnie. Jednym ze sposobów na zmianę oceny użytkowników o numerach identyfikacyjnych 1000 i 1001 jest użycie listy:

```
In [37]: df2.loc[[1000, 1001], "ocena"] = [3, 4]
df2
Out[37]:
```

cechy numer	imię	wiek	kraj	ocena	kontynent
1001	Mark	55	Włochy	4.0	Europa
1000	JOHN	33	USA	3.0	Ameryka
1002	Tim	41	USA	3.9	Ameryka
1003	Jenny	12	Niemcy	9.0	Europa

Zmiana danych na podstawie pozycji poprzez `iloc` działa w ten sam sposób. Zobaczmy teraz, jak można zmienić dane za pomocą indeksowania logicznego.

Ustawianie danych przy użyciu indeksowania logicznego

Indeksowanie logiczne, którego używaliśmy do filtrowania wierszy, może być również użyte do przypisywania wartości w DataFrame. Wyobraź sobie, że potrzebujesz zanonimizować wszystkie dane osobowe uczestników, którzy mają mniej niż 20 lat lub pochodzą z USA:

```
In [38]: tf = (df2["wiek"] < 20) | (df2["kraj"] == "USA")
df2.loc[tf, "imię"] = "xxx"
df2
Out[38]:
```

cechy numer	imię	wiek	kraj	ocena	kontynent
1001	Mark	55	Włochy	4.0	Europa
1000	xxx	33	USA	3.0	Ameryka

```
1002 xxx 41 USA 3.9 Ameryka
1003 xxx 12 Niemcy 9.0 Europa
```

Czasem masz zbiór danych, w którym musisz zastąpić określone wartości dla całego zbioru, a nie tylko dla określonych kolumn. W takim przypadku ponownie użyj specjalnej składni i przekaz cały obiekt DataFrame z wartościami logicznymi w poniższy sposób (w przykładzie ponownie wykorzystano DataFrame `rainfall`):

```
In [39]: # Najpierw skopiuj DataFrame, aby oryginał pozostał nietknięty
rainfall2 = rainfall.copy()
rainfall2
Out[39]:   Miasto 1  Miasto 2  Miasto 3
0      300.1    400.3    1000.5
1      100.2    300.4    1100.6
In [40]: # Ustaw 0 wszędzie tam, gdzie wartości są poniżej 400
rainfall2[rainfall2 < 400] = 0
rainfall2
Out[39]:   Miasto 1  Miasto 2  Miasto 3
0         0.0    400.3    1000.5
1         0.0         0.0    1100.6
```

Jeśli chcesz po prostu zastąpić jakąś wartość inną, jest na to prostszy sposób, który pokażę Ci za chwilę.

Ustawianie danych poprzez zamianę wartości

Jeżeli chcesz zastąpić określoną wartość w całym DataFrame lub w wybranych kolumnach, użyj metody `replace`:

```
In [41]: df2.replace("USA", "Stany Zjednoczone")
Out[41]:   cechy  imię  wiek      kraj  ocena  kontynent
numer
1001  Mark    55      Włochy    4.0    Europa
1000  xxx     33  Stany Zjednoczone  3.0    Ameryka
1002  xxx     41  Stany Zjednoczone  3.9    Ameryka
1003  xxx     12      Niemcy    9.0    Europa
```

Jeśli zamiast tego chciałbyś działać tylko na kolumnie `kraj`, mógłbyś użyć następującej składni:

```
df2.replace({"kraj": {"USA": "Stany Zjednoczone"}})
```

W tym przypadku, ponieważ wartość `USA` pojawia się tylko w kolumnie `kraj`, daje to taki sam wynik jak w poprzednim przykładzie. Na koniec tej sekcji zobaczymy, jak można dodać dodatkowe kolumny do DataFrame.

Ustawianie danych poprzez dodanie nowej kolumny

Aby dodać nową kolumnę do DataFrame, należy przypisać wartości do nazwy nowej kolumny. Możesz na przykład dodać nową kolumnę do DataFrame za pomocą skalarą lub listy:

```
In [42]: df2.loc[:, "rabat"] = 0
df2.loc[:, "cena"] = [49.9, 49.9, 99.9, 99.9]
df2
Out[42]:   cechy  imię  wiek  kraj  ocena  kontynent  rabat  cena
numer
1001  Mark    55  Włochy    4.0    Europa        0  49.9
```

1000	xxx	33	USA	3.0	Ameryka	0	49.9
1002	xxx	41	USA	3.9	Ameryka	0	99.9
1003	xxx	12	Niemcy	9.0	Europa	0	99.9

Dodanie nowej kolumny często wymaga obliczeń wektorowych:

```
In [43]: df2 = df.copy() # zaczniemy od nowej kopii
df2.loc[:, "rok urodzenia"] = 2021 - df2["wiek"]
df2
Out[43]:
```

cechy numer	imię	wiek	kraj	ocena	kontynent	rok urodzenia
1001	Mark	55	Włochy	4.5	Europa	1966
1000	John	33	USA	6.7	Ameryka	1988
1002	Tim	41	USA	3.9	Ameryka	1980
1003	Jenny	12	Niemcy	9.0	Europa	2009

Za chwilę przedstawię więcej szczegółów na temat obliczeń z wykorzystaniem obiektów DataFrame, ale zanim do tego dojdziemy — czy pamiętasz, że już kilka razy użyłem wartości NaN? W następnej części dowiesz się więcej na temat brakujących danych.

Brakujące dane

Brakujące dane mogą stanowić problem, ponieważ mogą potencjalnie wpłynąć na wyniki analizy danych, co sprawi, że wnioski będą mniej wiarygodne. Niemniej jednak bardzo często zdarzają się braki w zbiorach danych, z czym będziesz musiał sobie poradzić. W Excelu zwykle masz do czynienia z pustymi komórkami lub błędami #N/D, ale pandas dla brakujących danych używa stałej NumPy np.nan, wyświetlanej jako NaN. NaN jest standardową wartością typu zmiennoprzecinkowego oznaczającą *Not-a-Number* (nie-liczba). Dla znaczników czasu używane jest pd.NaT, a dla tekstu pandas używa None. Używając None lub np.nan, można wprowadzić brakujące wartości:

```
In [44]: df2 = df.copy() # zaczniemy od nowej kopii
df2.loc[1000, "ocena"] = None
df2.loc[1003, :] = None
df2
Out[44]:
```

cechy numer	imię	wiek	kraj	ocena	kontynent
1001	Mark	55.0	Włochy	4.5	Europa
1000	John	33.0	USA	NaN	Ameryka
1002	Tim	41.0	USA	3.9	Ameryka
1003	None	NaN	None	NaN	None

Aby oczyścić DataFrame, często chcemy usunąć wiersze z brakującymi danymi. Jest to bardzo proste:

```
In [45]: df2.dropna()
Out[45]:
```

cechy numer	imię	wiek	kraj	ocena	kontynent
1001	Mark	55.0	Włochy	4.5	Europa
1002	Tim	41.0	USA	3.9	Ameryka

Jeśli jednak chcesz usunąć tylko te wiersze, w których brakuje *wszystkich* wartości, użyj parametru how:

```
In [46]: df2.dropna(how="all")
Out[46]:
```

cechy numer	imię	wiek	kraj	ocena	kontynent
1001	Mark	55.0	Włochy	4.5	Europa
1000	John	33.0	USA	NaN	Ameryka
1002	Tim	41.0	USA	3.9	Ameryka

Aby uzyskać DataFrame lub Series z wartościami logicznymi w zależności od tego, czy występuje wartość NaN, czy nie, użyj `isna`:

```
In [47]: df2.isna()
Out[47]:
```

	cechy	imię	wiek	kraj	ocena	kontynent
numer						
1001	False	False	False	False	False	False
1000	False	False	False	False	True	False
1002	False	False	False	False	False	False
1003	True	True	True	True	True	True

Aby uzupełnić brakujące wartości, użyj `fillna`. Przykładowo, aby zastąpić wartości NaN w kolumnie `ocena` ich średnią (za chwilę wprowadzę statystyki opisowe takie jak `mean`), wpisz:

```
In [48]: df2.fillna({"ocena": df2["ocena"].mean()})
Out[48]:
```

	cechy	imię	wiek	kraj	ocena	kontynent
numer						
1001	Mark	55.0	Włochy	4.5	Europa	
1000	John	33.0	USA	NaN	Ameryka	
1002	Tim	41.0	USA	3.9	Ameryka	
1003	None	NaN	None	4.2	None	

Brakujące dane nie są jedynym przypadkiem, który wymaga od nas oczyszczenia zbioru danych. To samo dotyczy zduplikowanych danych, zobaczmy więc, jakie mamy możliwości!

Zduplikowane dane

Podobnie jak brakujące dane, duplikaty negatywnie wpływają na wiarygodność analizy. Aby się pozbyć zduplikowanych wierszy, użyj metody `drop_duplicates`. Opcjonalnie jako argument możesz podać podzbiór kolumn:

```
In [49]: df.drop_duplicates(["kraj", "kontynent"])
Out[49]:
```

	cechy	imię	wiek	kraj	ocena	kontynent
numer						
1001	Mark	55	Włochy	4.0	Europa	
1000	John	33	USA	3.0	Ameryka	
1003	Jenny	12	Niemcy	9.0	Europa	

Domyslnie pozostawia to pierwsze wystąpienie. Aby sprawdzić, czy dana kolumna zawiera duplikaty lub aby uzyskać jej unikatowe wartości, użyj następujących dwóch poleceń (użyj `df.index` zamiast `df["kraj"]`), jeśli chcesz uruchomić to na indeksie:

```
In [50]: df["kraj"].is_unique
Out[50]: False
In [51]: df["kraj"].unique()
Out[51]: array(['Włochy', 'USA', 'Niemcy'], dtype=object)
```

I w końcu, aby dowiedzieć się, które wiersze są duplikatami, użyj metody `duplicated`, która zwraca obiekt Series z wartościami logicznymi: domyslnie używa ona parametru `keep="first"`, który zachowuje pierwsze wystąpienie i oznacza tylko duplikaty jako `True`. Przy ustawieniu parametru `keep=False` metoda zwróci wartość `True` dla wszystkich wierszy z duplikatami, łącznie z pierwszym wystąpieniem, dzięki czemu łatwo jest uzyskać DataFrame ze wszystkimi zduplikowanymi wierszami. W poniższym przykładzie szukamy duplikatów w kolumnie `kraj`, ale w rzeczywistości często sprawdza się indeks lub całe wiersze. W takim przypadku musiałbyś użyć `df.index.duplicated()` lub `df.duplicated()`:

```

In [52]: # Domyślnie jako True oznaczane są tylko duplikaty
# (bez pierwszego wystąpienia)
df["country"].duplicated()
Out[52]: numer
1001 False
1000 False
1002 True
1003 False
Name: kraj, dtype: bool
In [53]: # Aby uzyskać wszystkie wiersze, w których "kraj" jest zduplikowany,
# użyj keep=False
df.loc[df["kraj"].duplicated(keep=False), :]
Out[53]: cechy imię wiek kraj ocena kontynent
numer
1000 John 33 USA 6.7 Ameryka
1002 Tim 41 USA 3.9 Ameryka

```

Po oczyszczeniu DataFrame poprzez usunięcie brakujących i zduplikowanych danych, być może będziesz chciał wykonać pewne operacje arytmetyczne — następna sekcja zawiera wprowadzenie do tego tematu.

Operacje arytmetyczne

Podobnie jak tablice NumPy, obiekty DataFrame i Series wykorzystują wektoryzację. Na przykład, aby dodać liczbę do każdej wartości DataFrame `rainfall`, wystarczy wprowadzić następujący kod:

```

In [54]: rainfall
Out[54]: Miasto 1 Miasto 2 Miasto 3
0 300.1 400.3 1000.5
1 100.2 300.4 1100.6
In [55]: rainfall + 100
Out[55]: Miasto 1 Miasto 2 Miasto 3
0 400.1 500.3 1100.5
1 200.2 400.4 1200.6

```

Jednak prawdziwa siła pandas tkwi w mechanizmie automatycznego **wyrównywania danych**: gdy stosujesz operatory arytmetyczne z więcej niż jednym obiektem DataFrame, pandas automatycznie wyrównuje je na podstawie ich indeksów kolumn i wierszy. Utwórzmy drugi DataFrame z tymi samymi etykietami wierszy i kolumn. Następnie utworzymy sumę:

```

In [56]: more_rainfall = pd.DataFrame(data=[[100, 200], [300, 400]],
index=[1, 2],
columns=["Miasto 1", "Miasto 4"])
more_rainfall
Out[56]: Miasto 1 Miasto 4
1 100 200
2 300 400
In [57]: rainfall + more_rainfall
Out[57]: Miasto 1 Miasto 2 Miasto 3 Miasto 4
0 NaN NaN NaN NaN
1 200.2 NaN NaN NaN
2 NaN NaN NaN NaN

```

Indeks i kolumny wynikowego DataFrame są połączeniem indeksów i kolumn dwóch składowych obiektów DataFrame: pola, które w obu DataFrame zawierają wartość, pokazują sumę, podczas gdy w pozostałej części DataFrame pojawia się wartość NaN. To coś, do czego musisz się przyzwyczaić,

jeśli pracowałeś wcześniej w Excelu, gdzie puste komórki są automatycznie zamieniane na zera, gdy używasz ich w operacjach arytmetycznych. Chcąc uzyskać takie samo zachowanie jak w Excelu, użyj metody `add` z parametrem `fill_value`, aby zastąpić wartości NaN zerami:

```
In [58]: rainfall.add(more_rainfall, fill_value=0)
Out[58]: Miasto 1  Miasto 2  Miasto 3  Miasto 4
         0    300.1    400.3    1000.5    NaN
         1    200.2    300.4    1100.6    200.0
         2    300.0      NaN      NaN    400.0
```

Analogicznie działa to również dla innych operatorów arytmetycznych, jak pokazano w tabeli 5.4.

Tabela 5.4. Operatory arytmetyczne

Operator	Metoda
*	mul
+	add
-	sub
/	div
**	pow

Jeśli w swoich obliczeniach używasz `DataFrame` i `Series`, domyślnie obiekt `Series` jest rozgłaszany wzdłuż indeksu:

```
In [59]: # Obiekt Series pobrany z wiersza
rainfall.loc[1, :]
Out[59]: Miasto 1    100.2
         Miasto 2    300.4
         Miasto 3   1100.6
         Name: 1, dtype: float64
In [60]: rainfall + rainfall.loc[1, :]
Out[60]: Miasto 1  Miasto 2  Miasto 3
         0    400.3    700.7    2101.1
         1    200.4    600.8    2201.2
```

Dlatego, aby dodać obiekt `Series` z uwzględnieniem położenia kolumn, musisz użyć metody `add` z jawnym argumentem osi (`axis`):

```
In [61]: # Obiekt Series pobrany z kolumny
rainfall.loc[:, "Miasto 2"]
Out[61]: 0    400.3
         1    300.4
         Name: Miasto 2, dtype: float64
In [62]: rainfall.add(rainfall.loc[:, "Miasto 2"], axis=0)
Out[62]: Miasto 1  Miasto 2  Miasto 3
         0    700.4    800.6    1400.8
         1    400.6    600.8    1401.0
```

Omówiliśmy już obiekty `DataFrame` z liczbami i ich zachowanie w operacjach arytmetycznych, natomiast w następnej części poznamy możliwości związane z przetwarzaniem tekstu w `DataFrame`.

Praca z kolumnami tekstowymi

Jak widzieliśmy na początku tego rozdziału, kolumny z tekstem lub mieszanymi typami danych mają typ danych `object`. Aby wykonywać operacje na kolumnach z łańcuchami znaków, należy użyć atrybutu `str`, który daje dostęp do metod łańcuchów znaków w Pythonie. Kilka z tych metod poznaliśmy już w rozdziale 3., ale nie zaszkodzi zajrzeć do dostępnych metod w dokumentacji Pythona (<https://oreil.ly/-e7SC>). Dla przykładu, aby usunąć wiodący i końcowy znak niedrukowalny, należy użyć metody `strip`; aby zmienić wszystkie pierwsze litery na wielkie, należy użyć metody `capitalize`. Połączenie tych metod pozwoli oczyścić kolumny z niechlujnego tekstu, który często jest wynikiem ręcznego wprowadzania danych:

```
In [63]: # Utwórzmy nowy obiekt DataFrame
users = pd.DataFrame(data=[" mArk ", "JOHN ", "Tim", " jenny"],
                    columns=["imię"])
users
Out[63]:   imię
0  mArk
1  JOHN
2   Tim
3  jenny

In [64]: users_cleaned = users.loc[:, "imię"].str.strip().str.capitalize()
users_cleaned
Out[64]: 0   Mark
1   John
2   Tim
3   Jenny
Name: imię, dtype: object
```

Możemy też wyszukać wszystkie imiona, które zaczynają się na literę „J”:

```
In [65]: users_cleaned.str.startswith("J")
Out[65]: 0   False
1    True
2   False
3    True
Name: imię, dtype: bool
```

Metody łańcuchów znaków są łatwe w użyciu, ale czasami może zajść konieczność przetworzenia `DataFrame` w sposób, który nie jest wbudowany. W takim przypadku możesz stworzyć własną funkcję i zastosować ją do `DataFrame`, co zostanie pokazane w następnej sekcji.

Stosowanie funkcji

`DataFrames` oferują metodę `applymap`, która zastosuje funkcję do każdego pojedynczego elementu, co jest przydatne, jeśli nie ma dostępnych funkcji uniwersalnych `NumPy`. Przykładowo nie ma funkcji uniwersalnych dla formatowania łańcuchów znaków, więc możemy sformatować każdy element `DataFrame`, jak poniżej:

```
In [66]: rainfall
Out[66]:   Miasto 1  Miasto 2  Miasto 3
0     300.1    400.3    1000.5
1     100.2    300.4    1100.6

In [67]: def format_string(x):
return f"{x:,.2f}"
```

```
In [68]: # Zauważ, że przekazujemy funkcję bez jej wywołania,
        # czyli format_string, a nie format_string()!
        rainfall.applymap(format_string)
Out[68]:   Miasto 1  Miasto 2  Miasto 3
         0    300.10    400.30  1,000.50
         1    100.20    300.40  1,100.60
```

Rozkładając to na czynniki pierwsze: sformatowany literał łańcucha (*f-string*) `f"{x}"` zwraca `x` jako łańcuch znaków. Aby dodać formatowanie, dołącz do zmiennej dwukropek, a następnie łańcuch formatujący, `.2f`. Przecinek jest tu separatorem tysięcy, a `.2f` oznacza zapis stałoprzecinkowy z dwiema cyframi po separatorze dziesiętnym. Aby uzyskać więcej szczegółów na temat formatowania łańcuchów, zapoznaj się z sekcją *Format Specification Mini-Language* (<https://oreil.ly/NgsG8>), która jest częścią dokumentacji Pythona.

Wyrażenia lambda

Python pozwala na zdefiniowanie funkcji w pojedynczym wierszu za pomocą **wyrażeń lambda**. Wyrażenia lambda są funkcjami anonimowymi, co oznacza, że nie mają nazwy. Rozważmy taką funkcję:

```
def nazwa_funkcji(arg1, arg2, ...):
    return zwracana_wartość
```

Ta funkcja może zostać przepisana jako wyrażenie lambda w następujący sposób:

```
lambda arg1, arg2, ...: zwracana_wartość
```

Krótko mówiąc, zastępujesz `def` przez `lambda`, pomijasz słowo kluczowe `return` i nazwę funkcji i umieszczasz wszystko w jednym wierszu. W tym przypadku, jak widzieliśmy na przykładzie metody `applymap`, może to być naprawdę wygodne, ponieważ nie musimy definiować funkcji dla czegoś, co jest używane tylko raz.

W tego typu przypadkach często używa się *wyrażeń lambda* (patrz ramka), ponieważ umożliwiają one zapisanie tego samego w pojedynczym wierszu, bez konieczności definiowania osobnej funkcji. Dzięki wyrażeniom lambda możemy przepisać poprzedni przykład w następujący sposób:

```
In [69]: rainfall.applymap(lambda x: f"{x:,.2f}")
Out[69]:   Miasto 1  Miasto 2  Miasto 3
         0    300.10    400.30  1,000.50
         1    100.20    300.40  1,100.60
```

Wymieniłem już wszystkie najważniejsze metody operowania na danych, ale zanim przejdziemy dalej, ważne jest, aby zrozumieć, kiedy pandas używa widoku `DataFrame`, a kiedy kopii.

Widok a kopia

Być może pamiętasz z poprzedniego rozdziału, że wycinanie tablic `NumPy` zwraca widok. W przypadku obiektów `DataFrame` jest to niestety bardziej skomplikowane: nie zawsze można łatwo przewidzieć, czy atrybuty `loc` i `iloc` zwracają widok, czy kopię, co sprawia, że jest to jeden z bardziej zagmatwanych tematów. Ponieważ istnieje duża różnica, czy zmieniasz widok, czy kopię `DataFrame`, pandas regularnie zgłasza następujące ostrzeżenie, gdy myśli, że ustawiasz dane w niezamierzony sposób: `SettingWithCopyWarning`. Oto kilka rad, jak obejść to dość zagadkowe ostrzeżenie:

Ustawiaj wartości w oryginalnym DataFrame, a nie w takim, które zostało wycięte z innego DataFrame.

Jeśli po operacji wycinania chcesz mieć niezależny obiekt DataFrame, utwórz jawnie jego kopię:

```
selection = df.loc[:, ["kraj", "kontynent"]].copy()
```

Choć z `loc` i `iloc` sprawa jest skomplikowana, warto pamiętać, że wszystkie metody DataFrame, takie jak `df.dropna()` lub `df.sort_values("nazwa_kolumny")`, zawsze zwracają kopię.

Do tej pory pracowaliśmy głównie z jednym obiektem DataFrame naraz. W następnym podrozdziale przedstawione zostaną różne sposoby łączenia wielu obiektów DataFrame w jeden, co jest bardzo częstym zadaniem, do którego pandas oferuje wydajne narzędzia.

Łączenie obiektów DataFrame

Łączenie różnych zbiorów danych w Excelu może być uciążliwym zadaniem i zazwyczaj wiąże się z dużą ilością formuł WYSZUKAJ.PIONOWO. Na szczęście łączenie obiektów DataFrame to jedna z najbardziej atrakcyjnych funkcji pandas, która znacznie ułatwia życie i zmniejsza możliwość popełniania błędów. Obiekty DataFrame można łączyć i scalać na różne sposoby; w tym podrozdziale przyjrzymy się tylko najczęstszym przypadkom, w których stosuje się funkcje `concat`, `join` i `merge`. Chociaż funkcje te pokrywają się, każda z nich bardzo upraszcza określone zadanie. Zacznę od funkcji `concat`, następnie wyjaśnię różne opcje funkcji `join`, a na koniec przedstawię `merge`, najbardziej ogólną funkcję z tych trzech.

Konkatenacja

Jeśli chcesz po prostu skleić ze sobą wiele obiektów DataFrame, najlepszym rozwiązaniem jest funkcja `concat`. Jak można wywnioskować z jej nazwy, przeprowadza ona proces zwany technicznie **konkatenacją**. Domyślnie funkcja `concat` skleja obiekty DataFrame wzdłuż wierszy i automatycznie wyrównuje kolumny. W poniższym przykładzie tworzę kolejny DataFrame o nazwie `more_users` i dołączam go do dolnej części naszego przykładowego DataFrame `df`:

```
In [70]: data=[[15, "Francja", 4.1, "Becky"],
              [44, "Kanada", 6.1, "Leanne"]]
more_users = pd.DataFrame(data=data,
                           columns=["wiek", "kraj", "ocena", "imię"],
                           index=[1000, 1011])
```

```
more_users
Out[70]:   wiek  kraj  ocena  imię
1000   15  Francja   4.1  Becky
1011   44  Kanada   6.1  Leanne
```

```
In [71]: pd.concat([df, more_users], axis=0)
Out[71]:   imię  wiek  kraj  ocena  kontynent
1001  Mark   55  Włochy   4.5  Europa
1000  John   33   USA   6.7  Ameryka
1002  Tim    41   USA   3.9  Ameryka
1003  Jenny  12  Niemcy   9.0  Europa
1000  Becky  15  Francja   4.1   NaN
1011  Leanne 44  Kanada   6.1   NaN
```

Zauważ, że masz teraz zduplikowane elementy indeksu, ponieważ `concat` skleja dane na wskazanej osi (wiersze) i tylko wyrównuje dane na drugiej osi (kolumny), automatycznie dopasowując nazwy

kolumn — nawet jeśli nie występują one w tej samej kolejności w dwóch DataFrame! Jeśli chcesz skleić dwa obiekty DataFrame wzdłuż kolumn, ustaw `axis=1`:

```
In [72]: data=[[3, 4],
              [5, 6]]
more_categories = pd.DataFrame(data=data,
                               columns=["testy", "logowania"],
                               index=[1000, 2000])

more_categories
Out[72]:   testy logowania
1000      3          4
2000      5          6

In [73]: pd.concat([df, more_categories], axis=1)
Out[73]:   imię  wiek  kraj  ocena  kontynent  testy  logowania
1000  John  33.0  USA    6.7  Ameryka    3.0    4.0
1001  Mark  55.0  Włochy  4.5  Europa    NaN    NaN
1002  Tim   41.0  USA    3.9  Ameryka    NaN    NaN
1003  Jenny 12.0  Niemcy  9.0  Europa    NaN    NaN
2000   NaN  NaN   NaN    NaN    NaN    5.0    6.0
```

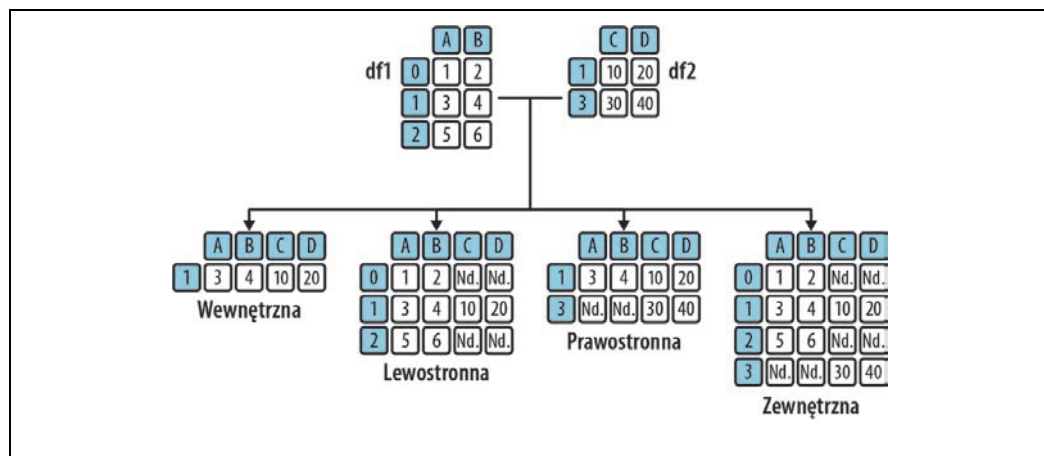
Szczególną i bardzo użyteczną cechą funkcji `concat` jest to, że akceptuje ona więcej niż dwa obiekty DataFrame. Skorzystamy z tego w następnym rozdziale, aby utworzyć pojedynczy DataFrame z wielu plików CSV:

```
pd.concat([df1, df2, df3, ...])
```

Z kolei funkcje `join` i `merge`, jak zobaczymy za chwilę, działają tylko z dwoma obiektami DataFrame.

Operacje `join` i `merge`

Wykonując operację `join`, łączysz kolumny każdego z dwóch obiektów DataFrame w nowy DataFrame, decydując jednocześnie o tym, co stanie się z wierszami, w oparciu o teorię zbiorów. Jeśli pracowałeś wcześniej z relacyjnymi bazami danych, jest to ta sama koncepcja, która występuje przy klauzuli `JOIN` w zapytaniach SQL. Rysunek 5.3 przedstawia działanie czterech typów operacji łączenia (wewnętrznej, lewej, prawej i zewnętrznej) na przykładzie dwóch przykładowych obiektów DataFrame, `df1` i `df2`.



Rysunek 5.3. Operacje łączenia

Przy stosowaniu funkcji `join` pandas wykorzystuje indeksy obu obiektów `DataFrame` do wyrównania wierszy. **Wewnętrzna operacja łączenia** (ang. *inner join*) zwraca obiekt `DataFrame` zawierający tylko te wiersze, których indeksy się pokrywają. **Lewostronna operacja łączenia** (ang. *left join*) pobiera wszystkie wiersze z lewego `DataFrame` (`df1`) i dopasowuje wiersze z prawego `DataFrame` (`df2`) zgodnie z indeksem. Tam, gdzie `df2` nie ma pasującego wiersza, pandas wypełni pola wartościami `NaN`. Lewostronna operacja łączenia odpowiada formule `WYSZUKAJ.PIONOWO` w Excelu. **Prawostronna operacja łączenia** (ang. *right join*) pobiera wszystkie wiersze z prawej tabeli `df2` i dopasowuje je do wierszy z `df1` zgodnie z indeksem. I wreszcie **zewnętrzna operacja łączenia** (ang. *outer join*, pełna nazwa to *full outer join*) tworzy sumę indeksów z obu obiektów `DataFrame` i dopasowuje wartości tam, gdzie jest to możliwe. Tabela 5.5 jest odpowiednikiem rysunku 5.3 w formie tekstowej.

Tabela 5.5. Operacje łączenia

Typ	Opis
inner	Tylko wiersze, których indeks występuje w obu obiektach <code>DataFrame</code>
left	Wszystkie wiersze z lewego <code>DataFrame</code> , pasujące wiersze z prawego <code>DataFrame</code>
right	Wszystkie wiersze z prawego <code>DataFrame</code> , pasujące wiersze z lewego <code>DataFrame</code>
outer	Suma indeksów wierszy z obu obiektów <code>DataFrame</code>

Zobaczmy, jak to działa w praktyce, wprowadzając w życie przykłady z rysunku 5.3:

```
In [74]: df1 = pd.DataFrame(data=[[1, 2], [3, 4], [5, 6]],
                           columns=["A", "B"])

Out[74]:   A  B
0  1  2
1  3  4
2  5  6

In [75]: df2 = pd.DataFrame(data=[[10, 20], [30, 40]],
                           columns=["C", "D"], index=[1, 3])

Out[75]:   C  D
1  10 20
3  30 40

In [76]: df1.join(df2, how="inner")
Out[76]:   A  B  C  D
1  3  4 10 20

In [77]: df1.join(df2, how="left")
Out[77]:   A  B  C  D
0  1  2  NaN NaN
1  3  4 10.0 20.0
2  5  6  NaN NaN

In [78]: df1.join(df2, how="right")
Out[78]:   A  B  C  D
1  3.0 4.0 10 20
3  NaN NaN 30 40

In [79]: df1.join(df2, how="outer")
Out[79]:   A  B  C  D
0  1.0 2.0  NaN NaN
1  3.0 4.0 10.0 20.0
2  5.0 6.0  NaN NaN
3  NaN NaN 30.0 40.0
```

Jeśli chcesz dołączyć co najmniej jedną kolumnę DataFrame, zamiast polegać na indeksie, użyj merge zamiast join. Funkcja merge przyjmuje argument on, aby podać co najmniej jedną kolumnę jako **warunek łączenia** (ang. *join condition*): kolumny te, które muszą występować w obu obiektach DataFrame, zostaną wykorzystane do dopasowania wierszy:

```
In [80]: # Dodaj kolumnę o nazwie "kategoria" do obu obiektów DataFrame
df1["kategoria"] = ["a", "b", "c"]
df2["kategoria"] = ["c", "b"]

In [81]: df1
Out[81]:   A  B  kategoria
0  1  2          a
1  3  4          b
2  5  6          c

In [82]: df2
Out[82]:   C  D  kategoria
1  10 20          c
3  30 40          b

In [83]: df1.merge(df2, how="inner", on=["kategoria"])
Out[83]:   A  B  kategoria  C  D
0  3  4          b  30 40
1  5  6          c  10 20

In [84]: df1.merge(df2, how="left", on=["kategoria"])
Out[84]:   A  B  kategoria  C  D
0  1  2          a  NaN  NaN
1  3  4          b  30.0 40.0
2  5  6          c  10.0 20.0
```

Ponieważ funkcje join i merge akceptują sporo opcjonalnych argumentów, aby uwzględnić bardziej złożone scenariusze, zapraszam do zapoznania się z oficjalną dokumentacją (<https://oreil.ly/OZ4WV>), z której dowiesz się na ten temat więcej.

Wiesz już, jak operować na jednym lub kilku obiektach DataFrame, co prowadzi nas do kolejnego zagadnienia w naszej podróży po analizie danych: jak sprawić, aby dane były zrozumiałe?

Statystyka opisowa i agregacja danych

Jednym ze sposobów ułatwiających zrozumienie dużych zbiorów danych jest obliczanie statystyk opisowych, takich jak suma bądź średnia, dla całego zbioru danych lub dla jego znaczących podzbiorów. W tym podrozdziale przyjrzymy się, jak to działa w pandas, a następnie zaprezentuję dwa sposoby agregacji danych w podzbiory: metodę groupby i funkcję pivot_table.

Statystyka opisowa

Statystyka opisowa pozwala na podsumowanie zbiorów danych za pomocą miar ilościowych. Dla przykładu prostą statystyką opisową jest liczba punktów danych. Innymi popularnymi przykładami są średnie, mediany lub dominanty. Obiekty DataFrame i Series zapewniają wygodny dostęp do statystyk opisowych za pomocą metod takich jak sum, mean i count, by wymienić tylko kilka. Wiele z nich spotkasz w tej książce, a pełna lista jest dostępna w dokumentacji pandas (<https://oreil.ly/t2q9Q>). Domyślnie zwracają one obiekt Series wzdłuż osi 0 (axi s=0), co oznacza, że otrzymujemy statystyki dotyczące kolumn:

```
In [85]: rainfall
Out[85]:   Miasto 1  Miasto 2  Miasto 3
          0    300.1    400.3    1000.5
          1    100.2    300.4    1100.6

In [86]: rainfall.mean()
Out[86]:   Miasto 1    200.15
          Miasto 2    350.35
          Miasto 3   1050.55
          dtype: float64
```

Jeśli chcesz uzyskać statystyki dla każdego wiersza, podaj argument `axis`:

```
In [87]: rainfall.mean(axis=1)
Out[87]:   0    566.966667
          1    500.400000
          dtype: float64
```

Brakujące wartości nie są domyślnie uwzględniane w statystykach opisowych, takich jak suma (sum) czy średnia (mean). Jest to zgodne z tym, jak Excel traktuje puste komórki, więc użycie formuły ŚREDNIA w Excelu na zakresie z pustymi komórkami da taki sam wynik, jak metoda `mean` zastosowana na obiekcie `Series` zawierającym te same liczby i wartości `NaN` zamiast pustych komórek.

Czasami nie wystarczą statystyki dla wszystkich wierszy `DataFrame` i potrzebne są bardziej szczegółowe informacje — na przykład średnia dla kategorii. Zobaczmy, jak to się robi!

Grupowanie

Używając ponownie naszego przykładowego `DataFrame` `df`, znajdziemy średnią ocen dla każdego kontynentu! Aby to zrobić, najpierw pogrupuj wiersze według kontynentów, a następnie zastosuj metodę `mean`, która obliczy średnią dla grupy. Wszystkie kolumny nieliczbowe są automatycznie wykluczane:

```
In [88]: df.groupby(["kontynent"]).mean()
Out[88]:   cechy  wiek  ocena
kontynent
Ameryka   37.0   5.30
Europa   33.5   6.75
```

Jeśli dołączasz więcej niż jedną kolumnę, wynikowy `DataFrame` będzie miał indeks hierarchiczny — `MultiIndex`, który poznaliśmy wcześniej:

```
In [89]: df.groupby(["kontynent", "kraj"]).mean()
Out[89]:   cechy  wiek  ocena
kontynent kraj
Ameryka   USA    37    5.3
Europa   Niemcy  12    9.0
         Włochy  55    4.5
```

Zamiast `mean` możesz zastosować większość statystyk opisowych, które oferuje `pandas`, a jeśli chcesz użyć własnej funkcji, użyj metody `agg`. Dla przykładu oto sposób uzyskania różnicy między maksymalną i minimalną wartością na grupę:

```
In [90]: df.groupby(["kontynent"]).agg(lambda x: x.max() - x.min())
Out[90]:   cechy  wiek  ocena
kontynent
Ameryka    8    2.8
Europa    43    4.5
```


Popularnym sposobem na uzyskanie statystyk dla poszczególnych grup w Excelu jest użycie tabel przestawnych. Wprowadzają one drugi wymiar i pozwalają spojrzeć na dane z różnych perspektyw; pandas również dysponuje funkcjonalnością tabel przestawnych, o czym przekonamy się za chwilę.

Funkcje `pivot_table` i `melt`

Jeśli korzystasz z tabel przestawnych w Excelu, nie będziesz miał problemów z zastosowaniem funkcji pandas `pivot_table`, ponieważ działa ona niemal w ten sam sposób. Dane w poniższym DataFrame są zorganizowane w podobny sposób jak rekordy przechowywane w bazie danych; każdy wiersz przedstawia transakcję sprzedaży określonych owoców w danym regionie:

```
In [91]: data = [{"Pomarańcze", "Północ", 12.30},
                ["Jabłka", "Południe", 10.55],
                ["Pomarańcze", "Południe", 22.00],
                ["Banany", "Południe", 5.90],
                ["Banany", "Północ", 31.30],
                ["Pomarańcze", "Północ", 13.10]]

sales = pd.DataFrame(data=data,
                    columns=["Owoce", "Region", "Przychód"])
```

```
Out[91]:
```

	Owoce	Region	Przychód
0	Pomarańcze	Północ	12.30
1	Jabłka	Południe	10.55
2	Pomarańcze	Południe	22.00
3	Banany	Południe	5.90
4	Banany	Północ	31.30
5	Pomarańcze	Północ	13.10

Aby utworzyć tabelę przestawną, przekazujesz DataFrame jako pierwszy argument funkcji `pivot_table`. Argumenty `index` i `columns` definiują, które kolumny DataFrame staną się odpowiednio etykietami wierszy i kolumn tabeli przestawnej. Wartości z kolumny wskazywanej przez argument `values` zostaną zagregowane w części danych wynikowego DataFrame za pomocą `aggfunc`, funkcji, która może być przekazana jako łańcuch znaków lub funkcja uniwersalna NumPy. I wreszcie argument `margins` odpowiada opcji *Sumy końcowe* w Excelu, tzn. jeśli pominiemy `margins` i `margins_name`, kolumna i wiersz Ogółem nie będą wyświetlane:

```
In [92]: pivot = pd.pivot_table(sales,
                                index="Owoce", columns="Region",
                                values="Przychód", aggfunc="sum",
                                margins=True, margins_name="Ogółem")
```

```
Out[92]:
```

	Region	Południe	Północ	Ogółem
Owoce				
Banany		5.90	31.3	37.20
Jabłka		10.55	NaN	10.55
Pomarańcze		22.00	25.4	47.40
Ogółem		38.45	56.7	95.15

Podsumowując, przestawienie danych oznacza pobranie unikatowych wartości z kolumny (w naszym przypadku Region) i przekształcenie ich w nagłówki kolumn tabeli przestawnej, a tym samym zagregowanie wartości z innej kolumny. W ten sposób można łatwo odczytać informacje summaryczne dotyczące interesujących nas wymiarów. W naszej tabeli przestawnej od razu widać, że w regionie

północnym nie sprzedano żadnych jabłek, a w regionie południowym większość przychodów pochodzi z pomarańczy. Jeśli chcesz zrobić to na odwrót i zamienić nagłówki kolumn w wartości pojedynczej kolumny, użyj funkcji `melt`. W tym znaczeniu funkcja `melt` jest przeciwieństwem funkcji `pivot_table`:

```
In [93]: pd.melt(pivot.iloc[:-1,:-1].reset_index(),
               id_vars="Owoce",
               value_vars=["Północ", "Południe"], value_name="Przychód")
Out[93]:
```

	Owoce	Region	Przychód
0	Banany	Północ	31.30
1	Jabłka	Północ	NaN
2	Pomarańcze	Północ	25.40
3	Banany	Południe	5.90
4	Jabłka	Południe	10.55
5	Pomarańcze	Południe	22.00

Tutaj rolę danych wejściowych pełni nasza tabela przestawna, ale używam metody `ilco`, aby pozbyć się wiersza i kolumny z sumami łącznymi. Resetuję również indeks, aby wszystkie informacje były dostępne jako zwykłe kolumny. Następnie przekazuję argumenty `id_vars`, aby wskazać identyfikatory, oraz `value_vars`, aby zdefiniować kolumny, dla których chcę „odwrócić przestawienie”. Funkcja `melt` może być przydatna, jeśli chcesz przygotować dane w taki sposób, aby można je było zapisać z powrotem do bazy danych, która oczekuje takiego formatu.

Agregowanie danych statystycznych ułatwia ich zrozumienie, ale nikt nie lubi czytać stron wypełnionych liczbami. Jeśli chcesz, aby informacje były łatwo zrozumiałe, nie ma nic lepszego niż utworzenie wizualizacji, co jest naszym następnym tematem.

Tworzenie wykresów

Tworzenie wykresów pozwala na wizualizację wyników analizy danych i może być najważniejszym krokiem w całym procesie. Do tworzenia wykresów użyjemy dwóch bibliotek: na początku przyjrzymy się `Matplotlib`, domyślnej bibliotece `pandas` do tworzenia wykresów, a następnie skupimy się na `Plotly`, nowoczesnej bibliotece, która pełni tę samą rolę, ale zapewnia większą interaktywność w notatnikach Jupyter.

Matplotlib

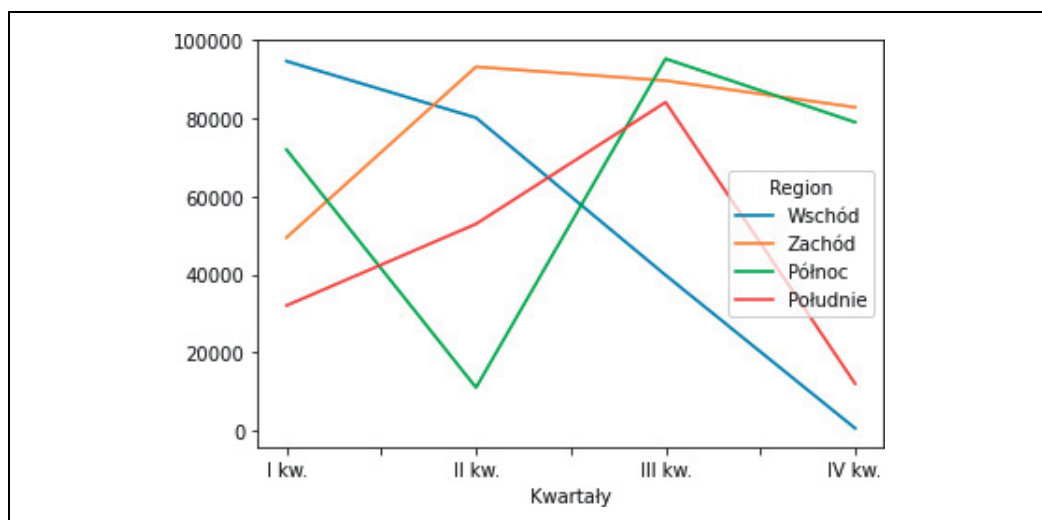
`Matplotlib` jest pakietem do tworzenia wykresów, który istnieje od dłuższego czasu i jest zawarty w dystrybucji `Anaconda`. Dzięki niemu można generować wykresy w różnych formatach, w tym jako grafikę wektorową do wydruków o wysokiej jakości. Gdy wywołasz metodę `plot` na `DataFrame`, `pandas` domyślnie wygeneruje wykres `Matplotlib`.

Aby używać `Matplotlib` w notatniku Jupyter, musisz najpierw uruchomić jedno z dwóch magicznych poleceń (zobacz ramkę „Magiczne polecenia”): `%matplotlib inline` lub `%matplotlib notebook`. Konfigurują one notatnik w taki sposób, że wykresy mogą być wyświetlane w samym notatniku. To drugie polecenie dodaje nieco więcej interaktywności, umożliwiając zmianę rozmiaru lub współczynnika powiększenia wykresu. Zacznijmy od utworzenia pierwszego wykresu przy użyciu `pandas` i `Matplotlib` (patrz rysunek 5.4):

```

In [94]: import numpy as np
         %matplotlib inline
         # lub %matplotlib notebook
In [95]: data = pd.DataFrame(data=np.random.rand(4, 4) * 100000,
                             index=["I kw.", "II kw.", "III kw.", "IV kw."],
                             columns=["Wschód", "Zachód", "Północ", "Południe"])
         data.index.name = "Kwartały"
         data.columns.name = "Region"
Out[95]:
   Region      Wschód      Zachód      Północ      Południe
Kwartały
I kw.    94650.974948  49456.118583  71981.667429  32033.826642
II kw.   80155.219825  93212.096993  11024.224588  52949.961643
III kw.  39825.387582  89669.923274  95266.653859  84131.561450
IV kw.   584.966373   82841.321963  79006.649491  12015.042632
In [96]: data.plot() # Skróć od data.plot.line()
Out[96]: <AxesSubplot: xlabel='Kwartały'>

```



Rysunek 5.4. Wykres Matplotlib

Magiczne polecenia

Polecenie `%matplotlib inline`, którego użyliśmy, aby Matplotlib działał poprawnie z notatnikami Jupyter, to tzw. **magiczne polecenie**. Magiczne polecenia to zestaw prostych poleceń, które powodują, że komórka notatnika Jupyter zachowuje się w określony sposób, lub sprawiają, że uciążliwe zadania stają się tak proste, iż wydaje się to niemal magią. Magiczne polecenia zapisuje się w komórkach podobnie jak kod Pythona, ale zaczynają się one albo od znaków `%%`, albo `%`. Polecenia, które mają wpływ na całą komórkę, zaczynają się od `%%`, a te, które mają wpływ tylko na pojedynczy wiersz w komórce, zaczynają się od znaku `%`.

W następnych rozdziałach zobaczymy więcej tego rodzaju poleceń, ale jeśli chcesz wyświetlić listę wszystkich aktualnie dostępnych magicznych poleceń, uruchom `%lsmagic`, a aby uzyskać szczegółowy opis, uruchom `%magic`.

Zauważ, że w tym przykładzie do skonstruowania obiektu DataFrame biblioteki pandas użyłem tablicy NumPy. Przekazanie tablic NumPy pozwala na wykorzystanie konstruktorów NumPy, które poznaliśmy w ostatnim rozdziale; tutaj używamy NumPy do wygenerowania obiektu DataFrame biblioteki pandas w oparciu o liczby pseudolosowe. Dlatego też uruchamiając ten przykład u siebie, otrzymasz inne wartości.

Nawet jeśli użyjesz magicznego polecenia `%matplotlib notebook`, prawdopodobnie zauważysz, że Matplotlib został pierwotnie zaprojektowany do wykresów statycznych, a nie do wprowadzania interaktywności na stronie internetowej. Dlatego właśnie zamierzamy użyć Plotly, biblioteki do tworzenia wykresów zaprojektowanej z myślą o sieci WWW.

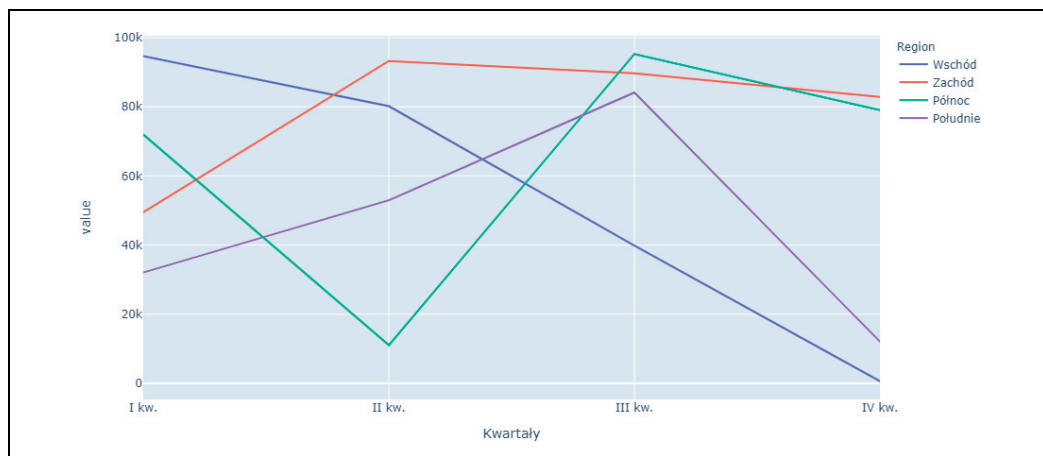
Plotly

Plotly jest pakietem opartym na JavaScript i od wersji 4.8.0 może być używany jako silnik tworzenia wykresów biblioteki pandas, oferując przy tym dużą interaktywność: można łatwo powiększać, klikać na legendę, aby wybrać lub odznaczyć kategorię, a także wyświetlać podpowiedzi z dodatkowymi informacjami na temat punktu danych, na który najedziemy kursorem. Plotly nie jest dołączony do Anacondy, więc jeśli jeszcze go nie zainstalowałeś, zrób to teraz, wykonując poniższe polecenie:

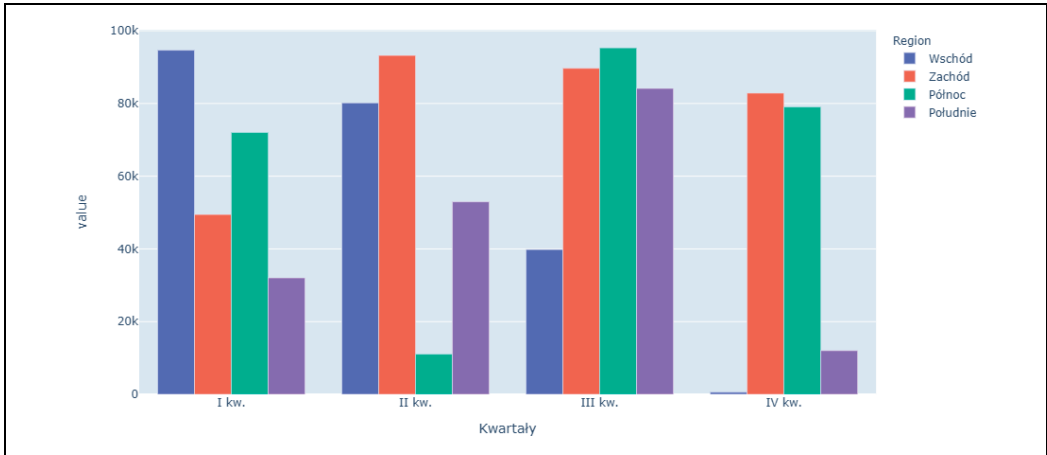
```
(base)> conda install plotly
```

Po uruchomieniu poniższej komórki silnik tworzenia wykresów w całym notatniku zostanie ustawiony na Plotly i jeśli ponownie uruchomisz poprzednią komórkę, ona również zostanie wyświetlona jako wykres Plotly. W przypadku Plotly zamiast uruchamiania magicznego polecenia, wystarczy wprowadzić poniższe ustawienie, aby móc wykreślić wykresy widoczne na rysunkach 5.5 i 5.6:

```
In [97]: # Ustaw silnik tworzenia wykresów na Plotly
pd.options.plotting.backend = "plotly"
In [98]: data.plot()
In [99]: # Wyświetl te same dane jako wykres słupkowy
data.plot.bar(barmode="group")
```



Rysunek 5.5. Wykres liniowy Plotly



Rysunek 5.6. Wykres słupkowy Plotly



Różnice w silnikach tworzenia wykresów

Jeśli używasz Plotly jako silnika tworzenia wykresów, będziesz musiał sprawdzić akceptowane argumenty metod dotyczących wykresów bezpośrednio w dokumentacji Plotly. Dla przykładu, aby uzyskać więcej informacji na temat argumentu `barmode=group`, możesz zajrzeć do dokumentacji wykresów słupkowych Plotly (<https://oreil.ly/Ekurd>).

Pakiet `pandas` i powiązane z nim biblioteki do tworzenia wykresów oferują mnóstwo typów wykresów i opcji umożliwiających ich formatowanie w niemal dowolny sposób. Możliwe jest również ułożenie wielu wykresów w serię podwykresów. Tabela 5.6 zawiera przegląd dostępnych typów wykresów.

Tabela 5.6. Typy wykresów `pandas`

Typ	Opis
<code>line</code>	Wykres liniowy, domyślny po uruchomieniu <code>df.plot()</code>
<code>bar</code>	Pionowy wykres słupkowy
<code>barh</code>	Poziomy wykres słupkowy
<code>hist</code>	Histogram
<code>box</code>	Wykres pudełkowy
<code>kde</code>	Wykres gęstości, można go również użyć poprzez <code>density</code>
<code>area</code>	Wykres warstwowy
<code>scatter</code>	Wykres punktowy
<code>hexbin</code>	Wykresy przedziałów heksagonalnych
<code>pie</code>	Diagram kołowy

Oprócz tego `pandas` oferuje pewne zaawansowane narzędzia i techniki tworzenia wykresów, które składają się z wielu pojedynczych komponentów. Aby uzyskać szczegółowe informacje, zapoznaj się z dokumentacją dotyczącą wizualizacji w `pandas` (<https://oreil.ly/FxYg9>).

Inne biblioteki do tworzenia wykresów

Środowisko wizualizacji naukowych w Pythonie jest bardzo aktywne; oprócz Matplotlib i Plotly do wyboru mamy wiele innych wysokiej jakości rozwiązań, które w niektórych przypadkach mogą się okazać lepsze:

Seaborn

Seaborn (<https://oreil.ly/a3U1t>) jest zbudowany na bazie biblioteki Matplotlib. Poprawia domyślny styl i wprowadza dodatkowe wykresy, takie jak mapy cieplne, które często ułatwiają pracę: możesz tworzyć zaawansowane wykresy statystyczne za pomocą zaledwie kilku wierszy kodu.

Bokeh

Bokeh (<https://docs.bokeh.org>) jest podobny do Plotly pod względem technologii i funkcjonalności: jest oparty na JavaScript i dlatego świetnie nadaje się również do tworzenia interaktywnych wykresów w notatnikach Jupyter. Bokeh jest dołączony do Anacondy.

Altair

Altair (<https://oreil.ly/t06t7>) jest biblioteką do wizualizacji statystycznych opartą na projekcji Vega (<https://oreil.ly/RN6A7>). Altair jest również oparty na JavaScript i oferuje pewną interaktywność, np. powiększanie.

HoloViews

HoloViews (<https://holoviews.org>) to kolejny pakiet oparty na JavaScript, który koncentruje się na ułatwieniu analizy i wizualizacji danych. Za pomocą kilku wierszy kodu możesz uzyskać złożone wykresy statystyczne.

W następnym rozdziale utworzymy więcej wykresów do analizy szeregów czasowych, ale zanim to zrobimy, zakończymy ten rozdział nauką importowania i eksportowania danych za pomocą pandas!

Importowanie i eksportowanie obiektów DataFrame

Do tej pory konstruowaliśmy obiekty DataFrame od podstaw, używając zagnieżdżonych list, słowników lub tablic NumPy. Są to ważne techniki, które warto znać, ale zazwyczaj dane są już dostępne i trzeba je po prostu przekształcić w DataFrame. W tym celu pandas oferuje różne funkcje wczytujące. Ale nawet jeśli potrzebujesz dostępu do zastrzeżonego systemu, dla którego pandas nie oferuje wbudowanego czytnika, często dysponujesz pakietem Pythona, aby połączyć się z tym systemem, a kiedy masz już dane, wystarczy zamienić je w DataFrame. W Excelu import danych jest rodzajem operacji, które zazwyczaj wykonuje się za pomocą Power Query.

Po przeanalizowaniu i zmodyfikowaniu zbioru danych możesz chcieć przenieść wyniki z powrotem do bazy danych, wyeksportować je do pliku CSV lub — mając na uwadze tytuł tej książki — przedstawić je w skoroszycie Excela swojemu przełożonemu. Aby wyeksportować obiekty DataFrame biblioteki pandas, należy użyć jednej z metod eksportu, które oferują DataFrame. Tabela 5.7 przedstawia przegląd najczęściej stosowanych metod importu i eksportu.

Tabela 5.7. Importowanie i eksportowanie obiektów DataFrame

Format (system) danych	Import: funkcja pandas (pd)	Eksport: metoda DataFrame (df)
Pliki CSV	pd.read_csv	df.to_csv
JSON	pd.read_json	df.to_json
HTML	pd.read_html	df.to_html
Schówek	pd.read_clipboard	df.to_clipboard
Pliki Excela	pd.read_excel	df.to_excel
Bazy danych SQL	pd.read_sql	df.to_sql

Z funkcjami `pd.read_sql` i `pd.to_sql` spotkamy się w rozdziale 11., gdzie wykorzystamy je jako część studium przypadku. A ponieważ cały rozdział 7. zamierzam poświęcić tematowi odczytu i zapisu plików Excela za pomocą pandas, w tym podrozdziale skupię się na imporcie i eksporcie plików CSV. Zacznijmy od wyeksportowania istniejącego obiektu DataFrame!

Eksportowanie plików CSV

Jeśli musisz przekazać obiekt DataFrame koledze, który może nie używać Pythona lub pandas, zazwyczaj dobrym pomysłem jest przekazanie go w formie pliku CSV: prawie każdy program wie, jak importować takie pliki. Aby wyeksportować nasz przykładowy DataFrame `df` do pliku CSV, użyj metody `to_csv`:

```
In [100]: df.to_csv("course_participants.csv")
```

Jeśli plik ma być przechowywany w innym katalogu, należy podać pełną ścieżkę dostępu jako surowy łańcuch znaków, np. `r"C:\ścieżka\do\żądanej\lokalizacji\msft.csv"`.



Używanie surowych łańcuchów znaków dla ścieżek dostępu do plików w systemie Windows

W łańcuchach znaków lewy ukośnik (ang. *backslash*) służy do modyfikacji określonych znaków. Dlatego przy zapisie ścieżek dostępu do plików w systemie Windows trzeba albo użyć podwójnych lewych ukośników (`C:\ścieżka\do\pliku.csv`), albo poprzedzić ciąg znaków literą `r`, aby przekształcić go w surowy ciąg znaków (ang. *raw string*), gdzie znaki są interpretowane dosłownie. Nie stanowi to problemu w systemach macOS i Linux, ponieważ tam w ścieżkach dostępu stosowane są prawe ukośniki.

Jeśli podasz tylko nazwę pliku, tak jak ja to zrobiłem, program utworzy w tym samym katalogu co notatnik plik `course_participants.csv` z następującą zawartością:

```
numer,imię,wiek,kraj,ocena,kontynent
1001,Mark,55,Włochy,4.5,Europa
1000,John,33,USA,6.7,Ameryka
1002,Tim,41,USA,3.9,Ameryka
1003,Jenny,12,Niemcy,9.0,Europa
```

Skoro już wiesz, jak używać metody `df.to_csv`, zobaczmy, jak zaimportować plik CSV!

Importowanie plików CSV

Importowanie lokalnego pliku CSV jest niezwykle proste — wystarczy przekazać jego ścieżkę dostępu do funkcji `read_csv`. *MSFT.csv* to plik CSV, który pobrałem z portalu Yahoo! Finance, zawierający dzienne historyczne ceny akcji firmy Microsoft — znajdziesz go w repozytorium towarzyszącym książce, w folderze *csv*:

```
In [101]: msft = pd.read_csv("csv/MSFT.csv")
```

Często oprócz nazwy pliku będziesz musiał przekazać do funkcji `read_csv` jeszcze kilka innych parametrów. Przykładowo parametr `sep` pozwala poinformować `pandas`, jakiego separatora lub delimitera używa plik CSV w przypadku, gdy nie jest to domyślny przecinek. W następnym rozdziale użyjemy jeszcze kilku innych parametrów, ale aby zapoznać się z ich pełną listą, zajrzyj do dokumentacji `pandas` (<https://oreil.ly/2GMhW>).

Teraz, gdy mamy do czynienia z dużymi obiektami `DataFrame` zawierającymi wiele tysięcy wierszy, zazwyczaj pierwszą rzeczą jest uruchomienie metody `info`, aby uzyskać podsumowanie `DataFrame`. Następnie możesz chcieć spojrzeć na kilka pierwszych i ostatnich wierszy `DataFrame`, używając metod `head` i `tail`. Metody te zwracają domyślnie pięć wierszy, ale można to zmienić, podając żądaną liczbę wierszy jako argument. Możesz również uruchomić metodę `describe`, aby uzyskać kilka podstawowych statystyk:

```
In [102]: msft.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8622 entries, 0 to 8621
Data columns (total 7 columns):
#   Column      Non-Null Count  Dtype
---  -
0   Date        8622 non-null   object
1   Open        8622 non-null   float64
2   High        8622 non-null   float64
3   Low         8622 non-null   float64
4   Close       8622 non-null   float64
5   Adj Close   8622 non-null   float64
6   Volume      8622 non-null   int64
dtypes: float64(5), int64(1), object(1)
memory usage: 471.6+ KB
In [103]: # Z powodu braku miejsca wybieram tylko kilka kolumn
# Możesz też po prostu uruchomić: msft.head()
msft.loc[:, ["Date", "Adj Close", "Volume"]].head()
Out[103]:
   Date      Adj Close  Volume
0  1986-03-13  0.062205  1031788800
1  1986-03-14  0.064427  308160000
2  1986-03-17  0.065537  133171200
3  1986-03-18  0.063871  67766400
4  1986-03-19  0.062760  47894400
In [104]: msft.loc[:, ["Date", "Adj Close", "Volume"]].tail(2)
Out[104]:
   Date      Adj Close  Volume
8620  2020-05-26  181.570007  36073600
8621  2020-05-27  181.809998  39492600
In [105]: msft.loc[:, ["Adj Close", "Volume"]].describe()
Out[105]:
   Adj Close  Volume
count  8622.000000  8.622000e+03
mean    24.921952  6.030722e+07
std     31.838096  3.877805e+07
```


min	0.057762	2.304000e+06
25%	2.247503	3.651632e+07
50%	18.454313	5.350380e+07
75%	25.699224	7.397560e+07
max	187.663330	1.031789e+09

Adj Close oznacza **skorygowaną cenę zamknięcia** (ang. *adjusted close price*) i koryguje cenę akcji o działania korporacyjne, takie jak podziały akcji. Wolumen to liczba akcji, które były przedmiotem obrotu. Podsumowanie różnych metod eksploracji DataFrame, które widzieliśmy w tym rozdziale, przedstawiłem w tabeli 5.8.

Tabela 5.8. Metody i atrybuty eksploracji DataFrame

Metoda lub atrybut DataFrame (df)	Opis
df.info()	Podaje liczbę punktów danych, typ indeksu, typ danych oraz użycie pamięci.
df.describe()	Udostępnia podstawowe statystyki, w tym liczebność, średnią, odchylenie standardowe, minimum, maksimum i percentyle.
df.head(n=5)	Zwraca pierwsze <i>n</i> wierszy DataFrame.
df.tail(n=5)	Zwraca ostatnie <i>n</i> wierszy DataFrame.
df.dtypes	Zwraca typ danych każdej kolumny.

Funkcja `read_csv` akceptuje również adres URL zamiast lokalnego pliku CSV. Oto sposób wczytania pliku CSV bezpośrednio z repozytorium towarzyszącego oryginalnemu wydaniu książki:

```
In [106]: # Adres URL został podzielony na wiersze, aby zmieścił się na stronie.
url = ("https://raw.githubusercontent.com/fzumstein/"
      "python-for-excel/1st-edition/csv/MSFT.csv")
msft = pd.read_csv(url)
In [107]: msft.loc[:, ["Date", "Adj Close", "Volume"]].head(2)
Out[107]:
```

	Date	Adj Close	Volume
0	1986-03-13	0.062205	1031788800
1	1986-03-14	0.064427	308160000

Pracę z tym zbiorem danych i funkcją `read_csv` będziemy kontynuować w następnym rozdziale o szereguach czasowych, gdzie przekształcimy kolumnę `Date` w `DatetimeIndex`.

Podsumowanie

Ten rozdział obfitował w nowe koncepcje i narzędzia do analizy zbiorów danych w pandas. Dowiedzieliśmy się, jak wczytywać pliki CSV, jak radzić sobie z brakującymi lub zduplikowanymi danymi oraz jak korzystać ze statystyk opisowych. Zobaczyliśmy również, jak łatwo można przekształcić obiekty DataFrame w interaktywne wykresy. Choć może trochę potrwać, zanim to wszystko przetrawisz, prawdopodobnie nie minie wiele czasu, nim zrozumiesz, ile zyskujesz, dodając pandas do swoich narzędzi. Po drodze porównywaliśmy pandas z następującymi funkcjonalnościami Excela:

Funkcja Autofiltru

Patrz punkt „Wybieranie przy użyciu indeksowania logicznego”.

Formuła WYSZUKAJ.PIONOWO

Patrz punkt „Operacje join i merge”.

Tabela przestawna

Patrz punkt „Funkcje pivot_table i melt”.

Power Query

Jest to połączenie podrozdziałów „Importowanie i eksportowanie obiektów DataFrame”, „Operowanie danymi” i „Łączenie obiektów DataFrame”.

Kolejny rozdział poświęcony jest analizie szeregów czasowych, czyli funkcjonalności, która doprowadziła do szerokiego zaadoptowania pandas przez branżę finansową. Zobaczmy, dlaczego ten element pandas ma taką przewagę nad Excelem!

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 

Użyj Pythona, a pokochasz Excela!

Bez Excela trudno sobie wyobrazić wykonywanie różnych złożonych zadań — to ulubione narzędzie naukowców, finansistów, analityków danych, a także profesjonalistów z innych branż. Każda z tych dziedzin ma swoje stale rosnące wymagania wobec Excela. Firma Microsoft wciąż rozwija ten kultowy arkusz kalkulacyjny, jednak język VBA nie nadąża za potrzebami wielu użytkowników. Osoby te często w codziennej pracy korzystają z Pythona do automatyzacji zadań, stąd integracja Excela i Pythona wydaje się naturalnym i wyjątkowo obiecującym rozwiązaniem.

Nie musisz dłużej czekać na włączenie Pythona jako języka skryptowego Excela — ta książka wyjaśnia, jak je połączyć i wyciągnąć z tej integracji maksimum korzyści. To wydanie przeznaczone dla zaawansowanych użytkowników Excela, którzy nie mają głębokiej wiedzy o Pythonie. Pokazuje, w jaki sposób manipulować danymi zawartymi w plikach Excela bez Excela, a także jak znakomicie zwiększać możliwości tego programu poprzez budowanie interaktywnych narzędzi do analizy danych. Niezależnie od tego, czy interesuje Cię praca z samymi arkuszami Excela, czy też chcesz stworzyć aplikacje Excela, znajdziesz tu mnóstwo wyczerpujących, jasnych i praktycznych wskazówek, popartych zrozumiałymi przykładami przydatnego kodu.

W książce między innymi:

- gruntowne podstawy Pythona i korzystania z notatników Jupyter i Visual Studio Code
- stosowanie biblioteki pandas do zastępowania typowych obliczeń w Excelu
- automatyzacja konsolidacji skoroszytów Excela i tworzenia raportów w Excelu
- tworzenie interaktywnych narzędzi Excela za pomocą xlwings
- współpraca Excela z bazą danych i plikami CSV
- stosowanie Pythona do zastąpienia VBA, Power Query i Power Pivot

Felix Zumstein jest ekspertem w dziedzinie zastosowania Excela w biznesie i w rozwiązywaniu problemów z tym programem w różnych branżach. Napisał i rozwija xlwings, popularny pakiet open source służący do automatyzacji pracy w Excelu za pomocą kodu Pythona.

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

Sprawdź nasze szkolenia!
SZKOLENIA
AKADEMIA IT & BUSINESS
HELIONSZKOLENIA.PL

KOD KORZYŚCI
Sięgnij po więcej! ▶
ISBN 978-83-283-8287-9
9 788328 382879

