O'REILLY®

Wydanie II

Python
w analizie danych

PRZETWARZANIE DANYCH ZA POMOCĄ PAKIETÓW
PANDAS I NUMPY ORAZ ŚRODOWISKA IPYTHON

Helion

Wes McKinney

Tytuł oryginału: Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython, 2nd Edition

Tłumaczenie: Konrad Matuk

ISBN: 978-83-283-4081-7

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

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

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

Printed in Poland.

Spis treści

ROZDZIAŁ 5.

Rozpoczynamy pracę z biblioteką pandas

Większość dalszej części tej książki będzie dotyczyła biblioteki pandas. W jej skład wchodzą struktury danych i narzędzia przeznaczone do przetwarzania danych, które ułatwiają i przyśpieszają oczyszczanie danych i analizę w Pythonie. Biblioteka pandas jest często używana w połączeniu z innymi narzędziami przeznaczonymi do przetwarzania danych numerycznych, takimi jak NumPy i SciPy, bibliotekami analitycznymi, takimi jak statsmodels i scikit-learn, a także bibliotekami przeznaczonymi do wizualizacji danych, takimi jak matplotlib. Pakiet pandas przypomina pakiet NumPy — jest nastawiony na przetwarzanie tablic, oferuje wiele funkcji operujących na tablicach i umożliwia przetwarzanie danych bez pętli `for`.

W bibliotece pandas zastosowano wiele rozwiązań zaczerpniętych z NumPy, ale największą różnicą pomiędzy tymi bibliotekami jest to, że pandas została zaprojektowana z myślą o pracy z danymi w formie tabel lub danymi o charakterze heterogenicznym, a biblioteka NumPy jest zoptymalizowana pod kątem pracy z homogenicznymi tablicami danych liczbowych.

Praca nad otwartym projektem pandas rozpoczęła się w 2010 roku. Od tego czasu biblioteka ta rozwinęła się na tyle, że jest stosowana do rozwiązywania wielu rzeczywistych problemów. Obecnie utrzymuje ją społeczność ponad 800 programistów. Bardzo często są to osoby, które zdecydowały się na współudział w rozwoju tego projektu po tym, jak korzystały z niego podczas codziennej pracy z danymi.

W całej książce importuję bibliotekę pandas przy użyciu następującej konwencji:

```
In [1]: import pandas as pd
```

W związku z tym we wszystkich miejscach kodu, gdzie zobaczysz zapis `pd.`, miej na uwadze, że dany fragment odwołuje się do biblioteki pandas. W związku z tym, że bardzo często będziemy również korzystać z modułów Series i DataFrame, warto załadować je do swojej lokalnej przestrzeni nazw:

```
In [2]: from pandas import Series, DataFrame
```

5.1. Wprowadzenie do struktur danych biblioteki pandas

Aby móc korzystać z biblioteki pandas, musisz poznać jej dwie główne struktury danych: serie i ramki danych. Nie są to rozwiązania uniwersalne, ale struktury te tworzą solidne i proste w zastosowaniu podstawy wspomagające rozwiązywanie większości problemów.

Obiekt Series

Seria (ang. *Series*) to jednowymiarowy obiekt przypominający tablicę. Składa się on z sekwencji wartości (typy tych wartości są podobne do typów obsługiwanych przez pakiet NumPy) i związanej z danymi tablicy etykiet określanej mianem **indeksu**. Serię najprościej jest utworzyć na podstawie tablicy danych:

```
In [11]: obj = pd.Series([4, 7, -5, 3])

In [12]: obj
Out[12]:
0   4
1   7
2  -5
3   3
dtype: int64
```

W sesji interaktywnej serie są wyświetlane tak, że po lewej stronie znajduje się indeks, a po prawej wartości odpowiadające poszczególnym elementom indeksu. Nie określiliśmy indeksu danych, ale w takiej sytuacji generowany jest domyślny indeks w postaci liczb całkowitych od *0* do *N–1*, gdzie *N* jest długością utworzonych danych. W celu wyświetlenia wartości serii skorzystaj z metody `values`, a w celu wyświetlenia indeksu — z metody `index`:

```
In [13]: obj.values
Out[13]: array([ 4,  7, -5,  3])

In [14]: obj.index  # Przypomina działanie funkcji range(4).
Out[14]: RangeIndex(start=0, stop=4, step=1)
```

Często będziesz chciał utworzyć obiekt typu `Series` z indeksem identyfikującym każdy element serii za pomocą etykiety:

```
In [15]: obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [16]: obj2
Out[16]:
d  4
b  7
a  -5
c  3
dtype: int64

In [17]: obj2.index
Out[17]: Index(['d', 'b', 'a', 'c'], dtype='object')
```

W celu wybrania pojedynczej wartości lub zbioru wartości możesz — w przeciwieństwie do tablic NumPy — korzystać z etykiet umieszczonych w indeksie:

```
In [18]: obj2['a']
Out[18]: -5

In [19]: obj2['d'] = 6

In [20]: obj2[['c', 'a', 'd']]
Out[20]:
c  3
a  -5
d  6
dtype: int64
```

W zaprezentowanym przykładzie `['c', 'a', 'd']` stanowi listę indeksów — jak widzisz, indeksy nie muszą być wartościami liczbowymi, mogą być również łańcuchami.

Korzystanie z funkcji biblioteki NumPy, wykonywanie operacji takich jak filtrowanie za pomocą tablicy wartości, mnożenie macierzy lub stosowanie standardowych funkcji matematycznych nie rozerwie powiązania pomiędzy indeksem a wartością:

```
In [21]: obj2[obj2 > 0]
Out[21]:
d  6
b  7
c  3
dtype: int64

In [22]: obj2 * 2
Out[22]:
d   12
b   14
a  -10
c    6
dtype: int64

In [23]: np.exp(obj2)
Out[23]:
d   403.428793
b  1096.633158
a     0.006738
c    20.085537
dtype: float64
```

Obiekty typu `Series` można porównać do uporządkowanych słowników o określonej długości — w przypadku obu struktur mamy do czynienia z przypisaniem wartości indeksu do wartości danych. Obiekty `Series` można stosować w wielu kontekstach, w których używa się słowników:

```
In [24]: 'b' in obj2
Out[24]: True

In [25]: 'e' in obj2
Out[25]: False
```

Jeżeli dysponujesz danymi w formie słownika, to możesz przekształcić go na serię. Wystarczy przekazać go do funkcji `Series`:

```
In [26]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}

In [27]: obj3 = pd.Series(sdata)

In [28]: obj3
Out[28]:
Ohio    35000
Oregon  16000
Texas   71000
Utah     5000
dtype: int64
```

Jeżeli do funkcji `Series` przekazujesz tylko słownik, to w wygenerowanej serii posortowane klucze słownika będą pełniły funkcję indeksu. To domyślne zachowanie możesz obejść, przekazując klucze słownika w zaplanowanej przez siebie kolejności:

```
In [29]: states = ['California', 'Ohio', 'Oregon', 'Texas']
```

```
In [30]: obj4 = pd.Series(sdata, index=states)

In [31]: obj4
Out[31]:
California  NaN
Ohio    35000.0
Oregon  16000.0
Texas   71000.0
dtype: float64
```

W zaprezentowanym przykładzie trzy wartości znajdujące się w słowniku `sdata` zostały wyświetlone w określonej wcześniej kolejności. Nie znaleziono wartości klucza `'California'`, a więc przypisano mu wartość `NaN` (nie-liczba) — w bibliotece pandas brakujące wartości są oznaczane właśnie w ten sposób. Klucz `'Utah'` nie został uwzględniony w liście `states`, a więc nie umieszczono go w obiekcie wyjściowym.

Brakujące wartości (brakujące dane) będę czasem również określał mianem wartości `NA`. Funkcje biblioteki pandas `isnull` i `notnull` mogą być użyte w celu wykrycia brakujących danych:

```
In [32]: pd.isnull(obj4)
Out[32]:
California  True
Ohio       False
Oregon     False
Texas      False

dtype: bool

In [33]: pd.notnull(obj4)
Out[33]:
California  False
Ohio         True
Oregon       True
Texas        True
dtype: bool
```

Obiekty typu `Series` umożliwiają również korzystanie z tych funkcji jak z ich metod:

```
In [34]: obj4.isnull()
Out[34]:
California  True
Ohio       False
Oregon     False
Texas      False
dtype: bool
```

Zagadnienia związane z obsługą brakujących danych opiszę bardziej szczegółowo w rozdziale 7.

Przydatną cechą obiektów typu `Series` jest to, że automatycznie wyrównują one wartości na podstawie indeksu podczas wykonywania operacji arytmetycznych:

```
In [35]: obj3
Out[35]:
Ohio    35000
Oregon  16000
Texas   71000
Utah     5000
dtype: int64

In [36]: obj4
Out[36]:
```

```
California  NaN
Ohio    35000.0
Oregon  16000.0
Texas   71000.0
dtype: float64

In [37]: obj3 + obj4
Out[37]:
California  NaN
Ohio    70000.0
Oregon  32000.0
Texas  142000.0
Utah        NaN
dtype: float64
```

Równanie danych zostanie później opisane w sposób bardziej szczegółowy. Jeżeli masz doświadczenie w pracy z bazami danych, to możesz to traktować jak operację `join`.

Obiekty `Series` i ich indeksy mają atrybut `name`, który integruje je z innymi kluczowymi elementami biblioteki pandas:

```
In [38]: obj4.name = 'population'

In [39]: obj4.index.name = 'state'

In [40]: obj4
Out[40]:
state
California  NaN
Ohio    35000.0
Oregon  16000.0
Texas   71000.0
Name: population, dtype: float64
```

Indeks obiektu `Series` może być modyfikowany w miejscu za pomocą operacji przypisania:

```
In [41]: obj
Out[41]:
0   4
1   7
2  -5
3   3
dtype: int64

In [42]: obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']

In [43]: obj
Out[43]:
Bob    4
Steve  7
Jeff  -5
Ryan   3
dtype: int64
```

Obiekt DataFrame

Obiekt `DataFrame` (**ramka danych**) jest prostokątną tabelą danych. Zawiera ona uporządkowany zbiór kolumn, a w każdej kolumnie może znaleźć się wartość innego typu (wartość liczbowa, łańcuch znaków, wartość logiczna itd.). Ramki danych posiadają indeksy wierszy i kolumn. Można je postrzegać jako słownik obiektów typu `Series` współdzielących ten sam indeks. Wewnętrznie Python nie

przechowuje tego typu danych w formie listy, słownika ani zbioru jednowymiarowych tablic — dane te są przechowywane w formie dwuwymiarowych bloków. Wyjaśnianie wewnętrznej struktury obiektów typu `DataFrame` wykracza poza zakres tematyczny tej książki.

Co prawda obiekt `DataFrame` ma charakter dwuwymiarowy, ale pomimo tego może być używany do reprezentacji danych o większej liczbie wymiarów — służy do tego indeksowanie hierarchiczne (zagadnienie to opiszę w rozdziale 8.), a także zaawansowane funkcje obsługi danych oferowane przez bibliotekę pandas.

Obiekty typu `DataFrame` mogą być tworzone na wiele różnych sposobów, ale najczęściej generuje się je na podstawie słownika list o równej długości lub tablic NumPy:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
```

Otrzymany obiekt `DataFrame` będzie posiadał automatycznie przypisany indeks (mechanizm ten działa tak samo jak w przypadku obiektu `Series`), a kolumny będą posortowane:

```
In [45]: frame
Out[45]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
5  3.2  Nevada  2003
```

Jeżeli pracujesz w notatniku Jupyter Notebook, to obiekty `dataFrame` będą wyświetlane w formacie tabeli, która może być poprawnie wyświetlana przez przeglądarkę internetową.

Podczas pracy z dużymi ramkami danych warto korzystać z metody `head`, która wybiera tylko pięć pierwszych wierszy:

```
In [46]: frame.head()
Out[46]:
   pop   state  year
0  1.5    Ohio  2000
1  1.7    Ohio  2001
2  3.6    Ohio  2002
3  2.4  Nevada  2001
4  2.9  Nevada  2002
```

Możesz określić kolejność, w jakiej mają być ustawione kolumny obiektu `DataFrame`:

```
In [47]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
Out[47]:
   year   state  pop
0  2000    Ohio  1.5
1  2001    Ohio  1.7
2  2002    Ohio  3.6
3  2001  Nevada  2.4
4  2002  Nevada  2.9
5  2003  Nevada  3.2
```

Jeżeli podczas tej operacji przekażesz kolumnę, która nie znajduje się w słowniku, to zostanie ona dodana do obiektu `DataFrame`, ale zostanie wypełniona wartościami `NaN`:

```
In [48]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                       index=['one', 'two', 'three', 'four',
   ....:                              'five', 'six'])

In [49]: frame2
Out[49]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7   NaN
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4   NaN
five   2002  Nevada  2.9   NaN
six    2003  Nevada  3.2   NaN

In [50]: frame2.columns
Out[50]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Dostęp do kolumny obiektu `DataFrame` można uzyskać za pomocą notacji przypominającej notację słownikową lub za pomocą atrybutu (w obu przypadkach zwrócony zostanie obiekt `Series`):

```
In [51]: frame2['state']
Out[51]:
one      Ohio
two      Ohio
three    Ohio
four   Nevada
five   Nevada
six    Nevada
Name: state, dtype: object

In [52]: frame2.year
Out[52]:
one    2000
two    2001
three  2002
four   2001
five   2002
six    2003
Name: year, dtype: int64
```

Stosując w środowisku IPython technikę dostępu opartą na atrybucie (np. `frame2.year`), możesz korzystać z automatycznego uzupełniania nazw kolumn (klawisza *Tab*), co znacznie ułatwia pracę.

Składnia `frame2[`*`kolumna`*`]` działa z dowolną nazwą kolumny, a składnia `frame2.`*`kolumna`* działa tylko wtedy, gdy nazwa kolumny jest poprawną nazwą zmiennej Pythona.

Zauważ, że zwrócony obiekt `Series` ma ten sam indeks co obiekt `DataFrame`, a dodatkowo obu obiektom przypisywane są poprawne atrybuty `name`.

Dostęp do wierszy można uzyskać również za pomocą pozycji lub nazwy i specjalnego atrybutu `loc` (później przedstawię więcej informacji na ten temat):

```
In [53]: frame2.loc['three']
Out[53]:
year   2002
state  Ohio
pop     3.6
debt    NaN
Name: three, dtype: object
```

Kolumny mogą być modyfikowane za pomocą operacji przypisywania. Na przykład do pustej kolumny `'debt'` można przypisać wartość skalarną lub tablicę z wieloma wartościami:

```
In [54]: frame2['debt'] = 16.5

In [55]: frame2
Out[55]:
       year   state  pop  debt
one    2000    Ohio  1.5  16.5
two    2001    Ohio  1.7  16.5
three  2002    Ohio  3.6  16.5
four   2001  Nevada  2.4  16.5
five   2002  Nevada  2.9  16.5
six    2003  Nevada  3.2  16.5

In [56]: frame2['debt'] = np.arange(6.)

In [57]: frame2
Out[57]:
       year   state  pop  debt
one    2000    Ohio  1.5   0.0
two    2001    Ohio  1.7   1.0
three  2002    Ohio  3.6   2.0
four   2001  Nevada  2.4   3.0
five   2002  Nevada  2.9   4.0
six    2003  Nevada  3.2   5.0
```

Podczas przypisywania list lub tablic do kolumny długość przypisywanej zmiennej musi być równa długości obiektu `DataFrame`. W przypadku przypisywania obiektu typu `Series` etykiety tego obiektu zostaną wyrównane zgodnie z indeksem obiektu `DataFrame`, a we wszystkie dziury zostaną wstawione wartości `NaN`:

```
In [58]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [59]: frame2['debt'] = val

In [60]: frame2
Out[60]:
       year   state  pop  debt
one    2000    Ohio  1.5   NaN
two    2001    Ohio  1.7  -1.2
three  2002    Ohio  3.6   NaN
four   2001  Nevada  2.4  -1.5
five   2002  Nevada  2.9  -1.7
six    2003  Nevada  3.2   NaN
```

Przypisywanie nieistniejącej kolumny spowoduje utworzenie nowej kolumny. Słowo kluczowe `del`, podobnie jak w przypadku słowników, jest używane do kasowania kolumn.

W celu zaprezentowania działania słowa kluczowego `del` utworzę nową kolumnę wartości logicznych, w której umieszczę wartość `True` wszędzie tam, gdzie w kolumnie `state` znajduje się łańcuch `'Ohio'`:

```
In [61]: frame2['eastern'] = frame2.state == 'Ohio'

In [62]: frame2
Out[62]:
       year   state  pop  debt  eastern
one    2000    Ohio  1.5   NaN     True
two    2001    Ohio  1.7  -1.2     True
three  2002    Ohio  3.6   NaN     True
```

```
four   2001  Nevada  2.4  -1.5    False
five   2002  Nevada  2.9  -1.7    False
six    2003  Nevada  3.2   NaN    False
```

Nowe kolumny nie mogą być tworzone za pomocą składni `frame2.eastern`.

Utworzoną przed chwilą kolumnę można usunąć za pomocą metody `del`:

```
In [63]: del frame2['eastern']

In [64]: frame2.columns
Out[64]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

Kolumna zwrócona w wyniku indeksowania obiektu `DataFrame` jest widokiem oryginalnych danych, a nie ich kopią. W związku z tym wszelkie operacje modyfikujące zwrócony obiekt `Series` w miejscu będą modyfikowały również obiekt `DataFrame`. Kolumna może zostać skopiowana w sposób jawny za pomocą metody `copy` obiektu `Series`.

Inną popularną formą danych jest zagnieżdżony słownik słowników:

```
In [65]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

Jeżeli zagnieżdżony słownik zostanie przekazany do obiektu `DataFrame`, to biblioteka pandas potraktuje klucze zewnętrznego słownika jako kolumny, a klucze wewnętrznego słownika jako indeksy wierszy:

```
In [66]: frame3 = pd.DataFrame(pop)

In [67]: frame3
Out[67]:
     Nevada  Ohio
2000    NaN   1.5
2001    2.4   1.7
2002    2.9   3.6
```

Obiekt `DataFrame` może zostać transponowany (poddany operacji zamiany wierszy z kolumnami) za pomocą składni podobnej do tej, która była używana w przypadku transponowania tablic NumPy:

```
In [68]: frame3.T
Out[68]:
       2000  2001  2002
Nevada  NaN   2.4   2.9
Ohio    1.5   1.7   3.6
```

Klucze w wewnętrznych słownikach są łączone i sortowane w celu utworzenia indeksu, ale operacja ta nie jest przeprowadzana w przypadku jawnego określenia indeksu:

```
In [69]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[69]:
     Nevada  Ohio
2001    2.4   1.7
2002    2.9   3.6
2003    NaN   NaN
```

W podobny sposób traktowane są słowniki obiektów `Series`:

```
In [70]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [71]: pd.DataFrame(pdata)
Out[71]:
     Nevada  Ohio
2000    NaN   1.5
2001    2.4   1.7
```

W tabeli 5.1 znajdziesz listę wszystkich obiektów, które możesz przekazać do konstruktora obiektu DataFrame.

Tabela 5.1. *Typy danych obsługiwane przez konstruktor ramki danych*

Typ	Uwagi
Dwuwymiarowa tablica *ndarray*	Macierz danych; umożliwia przekazanie dodatkowych etykiet wierszy i kolumn.
Słownik tablic, list lub krotek	Każda sekwencja staje się kolumną ramki danych; wszystkie sekwencje muszą być tej samej długości.
Tablica z rekordami o strukturze zgodnej z NumPy	Dane traktowane tak samo jak w przypadku słownika tablic.
Słownik serii	Każda wartość staje się kolumną; w razie niezdefiniowania indeksu w sposób jawny klucze poszczególnych serii są łączone w unię w celu utworzenia indeksu wierszy ramki danych.
Słownik słowników	Każdy wewnętrzny słownik staje się kolumną; klucze są łączone w unię w celu utworzenia indeksu wierszy (tak samo jak w przypadku słownika serii).
Lista słowników lub serii	Każdy element staje się wierszem ramki danych; unia kluczy słownika lub indeksów serii staje się etykietami kolumn ramki danych.
Lista list lub krotek	Traktowana tak samo jak dwuwymiarowa tablica *ndarray*.
Inna ramka danych	W razie nieprzekazania indeksów w sposób jawny wczytywane są indeksy ramki danych.
Obiekt NumPy `MaskedArray`	Obiekt jest traktowany tak samo jak dwuwymiarowa tablica *ndarray*, ale maskowane wartości są traktowane jako brakujące dane.

Jeżeli elementy `index` (indeks) i `columns` (kolumny) mają zdefiniowane atrybuty `name` (nazwa), to wartości przypisane do tych atrybutów zostaną również wyświetlone wraz z zawartością ramki danych:

```
In [72]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [73]: frame3
Out[73]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

Atrybut `values` obiektu `DataFrame` zwraca dane w postaci dwuwymiarowej tablicy *ndarray*:

```
In [74]: frame3.values
Out[74]:
```

```
array([[ nan,  1.5],
       [ 2.4,  1.7],
       [ 2.9,  3.6]])
```

Jeżeli kolumny ramki danych są danymi różnego typu, to wszystkie dane wszystkich kolumn zostaną umieszczone w tablicy:

```
In [75]: frame2.values
Out[75]:
array([[2000,   'Ohio', 1.5,  nan],
       [2001,   'Ohio', 1.7, -1.2],
       [2002,   'Ohio', 3.6,  nan],
       [2001, 'Nevada', 2.4, -1.5],
       [2002, 'Nevada', 2.9, -1.7],
       [2003, 'Nevada', 3.2,  nan]], dtype=object)
```

Obiekty index

Indeksy (obiekty `index`) są używane do przechowywania etykiet osi lub innych metadanych, takich jak np. nazwy osi. Tablica lub inna sekwencja etykiet może zostać użyta podczas tworzenia serii lub ramki danych w celu jawnego zdefiniowania indeksu:

```
In [76]: obj = pd.Series(range(3), index=['a', 'b', 'c'])

In [77]: index = obj.index

In [78]: index
Out[78]: Index(['a', 'b', 'c'], dtype='object')

In [79]: index[1:]
Out[79]: Index(['b', 'c'], dtype='object')
```

Indeksy są obiektami niemodyfikowalnymi, a więc użytkownik nie może ich zmieniać:

```
index[1] = 'd'  # Błąd typu (TypeError)
```

Niemodyfikowalność sprawia, że współdzielenie obiektów typu `index` pomiędzy strukturami danych jest bezpieczniejsze:

```
In [80]: labels = pd.Index(np.arange(3))

In [81]: labels
Out[81]: Int64Index([0, 1, 2], dtype='int64')

In [82]: obj2 = pd.Series([1.5, -2.5, 0], index=labels)

In [83]: obj2
Out[83]:
0   1.5
1  -2.5
2   0.0
dtype: float64

In [84]: obj2.index is labels
Out[84]: True
```

Większość problemów analitycznych można rozwiązać bez korzystania z indeksów, ale mechanikę działania indeksów należy zrozumieć, ponieważ niektóre operacje generują dane wyjściowe zawierające indeksy.

Indeksy zachowują się jak tablice, ale również jak zbiór danych o stałym rozmiarze:

```
In [85]: frame3
Out[85]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6

In [86]: frame3.columns
Out[86]: Index(['Nevada', 'Ohio'], dtype='object', name='state')

In [87]: 'Ohio' in frame3.columns
Out[87]: True

In [88]: 2003 in frame3.index
Out[88]: False
```

W przeciwieństwie do zbiorów Pythona indeksy pandas mogą zawierać zduplikowane etykiety:

```
In [89]: dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])

In [90]: dup_labels
Out[90]: Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Przeprowadzenie operacji wyboru ze zdublowanymi etykietami spowoduje wybranie wszystkich wystąpień danej etykiety.

Każdy indeks obsługuje wiele metod i własności, które mogą przydać się podczas analizy umieszczonych w nim danych. W tabeli 5.2 przedstawiono najczęściej używane metody i własności obiektów typu `index`.

Tabela 5.2. Wybrane metody i własności indeksów

Metoda	Opis
`append`	Łączy obiekty typu indeks w celu utworzenia nowego indeksu.
`difference`	Zwraca różnicę zbiorów w postaci indeksu.
`insertion`	Zwraca efekt operacji wstawiania.
`union`	Zwraca efekt operacji sumowania.
`isin`	Generuje tablicę wartości logicznych informujących o tym, czy każda z wartości znajduje się w przekazanym ciągu.
`delete`	Tworzy nowy indeks po usunięciu elementu znajdującego się pod indeksem `i`.
`drop`	Tworzy nowy indeks po usunięciu przekazanych wartości.
`insert`	Tworzy nowy indeks po wstawieniu elementu pod indeksem `i`.
`is_monotonic`	Zwraca `True`, jeżeli każdy element jest większy od poprzedniego elementu (lub jest mu równy).
`is_unique`	Zwraca `True`, jeżeli indeks nie zawiera zduplikowanych wartości.
`unique`	Tworzy tablicę unikalnych wartości indeksu.

5.2. Podstawowe funkcjonalności

W tym podrozdziale znajdziesz informacje na temat podstawowych mechanizmów obsługi danych umieszczonych w obiektach typu `Series` i `DataFrame`. W kolejnych rozdziałach zgłębię zagadnienia związane z analizą danych i ich przekształcaniem za pomocą biblioteki pandas. Książka, którą trzymasz w ręku, nie jest dokładną dokumentacją biblioteki pandas. Pisząc ją, chciałem się skupić na najważniejszych możliwościach oferowanych przez ten pakiet. Rzadziej używane funkcje biblioteki pandas możesz zgłębić samodzielnie podczas pracy z danymi.

Uaktualnianie indeksu

Jedną z najważniejszych metod obiektów pandas jest metoda `reindex` (uaktualnianie indeksu). Umożliwia ona utworzenie nowego obiektu z danymi *dopasowanymi* do nowego indeksu. Przyjrzyj się następującemu przykładowi:

```
In [91]: obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

In [92]: obj
Out[92]:
d   4.5
b   7.2
a  -5.3
c   3.6
dtype: float64
```

Wywołanie metody `reindex` na serii powoduje zmianę kolejności danych — przystosowanie jej do nowego indeksu i wprowadzenie brakujących wartości w miejsca nowych indeksów:

```
In [93]: obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])

In [94]: obj2
Out[94]:
a  -5.3
b   7.2
c   3.6
d   4.5
e   NaN
dtype: float64
```

Podczas zmiany indeksów danych o charakterze szeregu (np. szeregów czasowych) konieczne może okazać się przeprowadzenie operacji takich jak interpolacja lub filtrowanie. W celu wykonania takiej operacji należy skorzystać z opcji `method` i np. metody `ffill`, która wypełnia wartości do przodu:

```
In [95]: obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

In [96]: obj3
Out[96]:
0    blue
2  purple
4  yellow
dtype: object

In [97]: obj3.reindex(range(6), method='ffill')
Out[97]:
0    blue
1    blue
```

```
2  purple
3  purple
4  yellow
5  yellow
dtype: object
```

Metoda `reindex` w przypadku obiektu `DataFrame` może zmieniać kolejność (wierszy) indeksu, kolumn lub obu tych elementów. W przypadku przekazania tylko sekwencji wykonywana jest jedynie operacja uaktualniania indeksów wierszy:

```
In [98]: frame = pd.DataFrame(np.arange(9).reshape((3, 3)),
   ....:                      index=['a', 'c', 'd'],
   ....:                      columns=['Ohio', 'Texas', 'California'])

In [99]: frame
Out[99]:
  Ohio  Texas  California
a    0      1           2
c    3      4           5
d    6      7           8

In [100]: frame2 = frame.reindex(['a', 'b', 'c', 'd'])
In [101]: frame2
Out[101]:
  Ohio  Texas  California
a  0.0    1.0         2.0
b  NaN    NaN         NaN
c  3.0    4.0         5.0
d  6.0    7.0         8.0
```

W celu uaktualnienia indeksu kolumn należy skorzystać ze słowa kluczowego `columns`:

```
In [102]: states = ['Texas', 'Utah', 'California']

In [103]: frame.reindex(columns=states)
Out[103]:
  Texas  Utah  California
a     1   NaN           2
c     4   NaN           5
d     7   NaN           8
```

W tabeli 5.3 wymieniono więcej argumentów metody `reindex`.

Tabela 5.3. Argumenty uaktualniania indeksu

Argument	Opis
`index`	W charakterze indeksu używana jest nowa sekwencja. Może być to egzemplarz obiektu `index` lub dowolny inny sekwencyjny obiekt Pythona. Indeks zostanie użyty bezpośrednio, bez kopiowania.
`method`	Metoda interpolacji (wypełniania) — `ffill` wypełnia w przód, a `bfill` wypełnia wstecz.
`fill_value`	Wartość, która wypełnia brakujące wartości przypisywane nowym indeksom.
`limit`	Maksymalny rozmiar przerwy (wyrażony za pomocą liczby elementów) do wypełnienia podczas wypełniania w przód lub w tył.
`tolerance`	Maksymalny rozmiar przerwy (wyrażony za pomocą odległości bezwzględnej) do wypełnienia w przypadku niedokładnych dopasowań podczas wypełniania w przód lub w tył.

Tabela 5.3. Argumenty uaktualniania indeksu — ciąg dalszy

Argument	Opis
`level`	Dopasowuje prosty indeks na poziomie wieloindeksu; w przeciwnym wypadku wybiera jego podzbiór.
`copy`	W przypadku wartości `True` zawsze kopiuje dane (dotyczy to również sytuacji, w której nowy indeks jest taki sam jak stary); w przypadku wartości `False` nie kopiuje danych, gdy indeksy są identyczne.

W dalszej części książki przedstawię więcej informacji na temat bardziej zwięzłego uaktualniania indeksów poprzez indeksowanie etykiet metodą `loc`. Wielu użytkowników Pythona korzysta tylko z tej techniki:

```
In [104]: frame.loc[['a', 'b', 'c', 'd'], states]
Out[104]:
  Texas  Utah  California
a   1.0   NaN         2.0
b   NaN   NaN         NaN
c   4.0   NaN         5.0
d   7.0   NaN         8.0
```

Odrzucanie elementów osi

Odrzucanie jednego elementu lub kilku elementów osi jest proste do przeprowadzenia, jeżeli dysponujesz tablicą indeksu lub listą bez tych elementów. Metoda `drop` zwraca nowy obiekt, z którego osi usunięto wybrane wartości (może to się przydać podczas czyszczenia danych i wykonywania operacji logicznych):

```
In [105]: obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

In [106]: obj
Out[106]:
a  0.0
b  1.0
c  2.0
d  3.0
e  4.0
dtype: float64

In [107]: new_obj = obj.drop('c')
In [108]: new_obj
Out[108]:
a  0.0
b  1.0
d  3.0
e  4.0
dtype: float64

In [109]: obj.drop(['d', 'c'])
Out[109]:
a  0.0
b  1.0
e  4.0
dtype: float64
```

W przypadku ramki danych wartości indeksu mogą być usuwane z dowolnej osi. Aby zademonstrować tę możliwość, najpierw utwórzmy przykładowy obiekt typu `DataFrame`:

```
In [110]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:  index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:  columns=['one', 'two', 'three', 'four'])

In [111]: data
Out[111]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Wywołanie metody `drop` z sekwencją etykiet spowoduje odrzucenie wartości z wierszy oznaczonych tymi etykietami (z osi 0):

```
In [112]: data.drop(['Colorado', 'Ohio'])
Out[112]:

          one  two  three  four
Utah        8    9     10    11
New York   12   13     14    15
```

Wartości znajdujące się w wybranych kolumnach można odrzucać za pomocą argumentu `axis=1` lub `axis='columns'`:

```
In [113]: data.drop('two', axis=1)
Out[113]:

          one  three  four
Ohio        0      2     3
Colorado    4      6     7
Utah        8     10    11
New York   12     14    15

In [114]: data.drop(['two', 'four'], axis='columns')
Out[114]:
          one  three
Ohio        0      2
Colorado    4      6
Utah        8     10
New York   12     14
```

Wiele funkcji modyfikujących rozmiar lub kształt obiektów typu `Series` lub `DataFrame` (przykładem takiej funkcji jest `drop`) może działać *w miejscu* bez zwracania nowego obiektu. Wystarczy skorzystać z argumentu `inplace`:

```
In [115]: obj.drop('c', inplace=True)

In [116]: obj
Out[116]:
a  0.0
b  1.0
d  3.0
e  4.0
dtype: float64
```

Zachowaj ostrożność, korzystając z argumentu `inplace`, ponieważ jego użycie powoduje nieodwracalne kasowanie odrzucanych elementów.

Indeksowanie, wybieranie i filtrowanie

Indeksowanie serii (`obiekt[…]`) działa analogicznie do indeksowania tablic NumPy, ale zamiast z wartości całkowitoliczbowych musisz korzystać z wartości przypisanych do indeksu serii. Oto ilustrujące to przykłady:

```
In [117]: obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

In [118]: obj
Out[118]:
a  0.0
b  1.0
c  2.0
d  3.0
dtype: float64

In [119]: obj['b']
Out[119]: 1.0

In [120]: obj[1]
Out[120]: 1.0

In [121]: obj[2:4]
Out[121]:
c  2.0
d  3.0
dtype: float64

In [122]: obj[['b', 'a', 'd']]
Out[122]:
b  1.0
a  0.0
d  3.0
dtype: float64

In [123]: obj[[1, 3]]
Out[123]:
b  1.0
d  3.0
dtype: float64

In [124]: obj[obj < 2]
Out[124]:
a  0.0
b  1.0
dtype: float64
```

Wycinki z etykietami działają nieco inaczej niż w normalnym Pythonie, bo do wycinanego zbioru zaliczany jest również element końcowy:

```
In [125]: obj['b':'c']
Out[125]:
b   1.0
c   2.0
dtype: float64
```

Przypisywanie wartości za pomocą tych metod modyfikuje wybraną sekcję serii:

```
In [126]: obj['b':'c'] = 5

In [127]: obj
Out[127]:
```

```
a   0.0
b   5.0
c   5.0
d   3.0
dtype: float64
```

Indeks ramki danych umożliwia uzyskanie dostępu do wybranej kolumny lub wybranego ciągu kolumn (należy skorzystać z pojedynczej wartości lub z sekwencji):

```
In [128]: data = pd.DataFrame(np.arange(16).reshape((4, 4)),
   .....:                     index=['Ohio', 'Colorado', 'Utah', 'New York'],
   .....:                     columns=['one', 'two', 'three', 'four'])

In [129]: data
Out[129]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7
Utah        8    9     10    11
New York   12   13     14    15

In [130]: data['two']
Out[130]:
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64

In [131]: data[['three', 'one']]
Out[131]:
          three  one
Ohio          2    0
Colorado      6    4
Utah         10    8
New York     14   12
```

Istnieje kilka specjalnych przypadków takiego indeksowania. Pierwszym z nich jest operacja wycinania lub wybierania danych przy użyciu tablicy wartości logicznych:

```
In [132]: data[:2]
Out[132]:
          one  two  three  four
Ohio        0    1      2     3
Colorado    4    5      6     7

In [133]: data[data['three'] > 5]
Out[133]:
           one  two  three  four
Colorado     4    5      6     7
Utah         8    9     10    11
New York    12   13     14    15
```

Dla uproszczenia pracy skorzystaliśmy ze składni wybierania danych `data[:2]`. Przekazanie do operatora pojedynczego elementu lub listy `[]` powoduje wybranie kolumn.

Kolejnym ze wspomnianych wcześniej przypadków jest zastosowanie operacji porównywania skalarów:

```
In [134]: data < 5
Out[134]:
              one    two  three   four
```

```
Ohio       True   True   True   True
Colorado   True  False  False  False
Utah      False  False  False  False
New York  False  False  False  False

In [135]: data[data < 5] = 0

In [136]: data
Out[136]:
          one  two  three  four
Ohio        0    0      0     0
Colorado    0    5      6     7
Utah        8    9     10    11
New York   12   13     14    15
```

Jeśli chodzi o składnię, to w tym przypadku ramka danych zachowuje się jak dwuwymiarowa tablica NumPy.

Wybieranie za pomocą operatorów loc i iloc

Podczas obsługi indeksu w postaci etykiet wierszy ramki danych często przydają się operatory indeksowania `loc` i `iloc`. Pozwalają one na wybranie podzbioru wierszy i kolumn z ramki danych za pomocą notacji podobnej do notacji NymPy przy użyciu etykiet osi (`loc`) lub wartości całkowitoliczbowych (`iloc`).

W pierwszym przykładzie wybieram za pomocą etykiet pierwszy rząd i dwie kolumny:

```
In [137]: data.loc['Colorado', ['two', 'three']]
Out[137]:
two    5
three  6
Name: Colorado, dtype: int64
```

Teraz wykonam podobne operacje wyboru za pomocą liczb całkowitych i operatora `iloc`:

```
In [138]: data.iloc[2, [3, 0, 1]]
Out[138]:
four  11
one    8
two    9
Name: Utah, dtype: int64

In [139]: data.iloc[2]
Out[139]:
one     8
two     9
three  10
four   11
Name: Utah, dtype: int64

In [140]: data.iloc[[1, 2], [3, 0, 1]]
Out[140]:
          four  one  two
Colorado     7    0    5
Utah        11    8    9
```

Obie funkcje indeksowania działają poza pojedynczymi etykietami i listami etykiet również z wycinkami:

```
In [141]: data.loc[:'Utah', 'two']
Out[141]:
Ohio      0
```

```
Colorado  5
Utah      9
Name: two, dtype: int64

In [142]: data.iloc[:, :3][data.three > 5]
Out[142]:
          one  two  three
Colorado    0    5      6
Utah        8    9     10
New York   12   13     14
```

Jak widzisz, istnieje wiele sposobów wybierania i przegrupowywania danych umieszczonych w obiektach pakietu pandas. W tabeli 5.4 umieszczono krótkie podsumowanie takich operacji w kontekście ramki danych. W dalszej części tej książki dowiesz się, że istnieje wiele dodatkowych opcji obsługi indeksów hierarchicznych.

Tabela 5.4. *Opcje indeksowania obiektu DataFrame*

Typ	Uwagi
`df[`*`wartość`*`]`	Wybiera pojedynczą kolumnę lub sekwencję kolumn z ramki danych; specjalne przypadki użycia: tablica wartości binarnych (filtruje wiersze), wycinek (wycina wiersze), ramka danych z wartościami logicznymi (definiuje wartości na podstawie określonego kryterium).
`df.loc[`*`wartość`*`]`	Wybiera pojedynczy wiersz lub podzbiór wierszy z ramki danych na podstawie etykiet.
`df.loc[:, `*`wartość`*`]`	Wybiera pojedynczą kolumnę lub podzbiór kolumn na podstawie etykiet.
`df.loc[`*`wartość1`*`, `*`wartość2`*`]`	Wybiera wiersze oraz kolumny na podstawie etykiet.
`df.iloc[`*`miejsce`*`]`	Wybiera pojedynczy wiersz lub podzbiór wierszy z ramki danych na podstawie pozycji określonej za pomocą wartości całkowitoliczbowej.
`df.iloc[:, `*`miejsce`*`]`	Wybiera pojedynczą kolumnę lub podzbiór kolumn na podstawie pozycji określonej za pomocą wartości całkowitoliczbowej.
`df.iloc[`*`miejsce_i`*`, `*`miejsce_j`*`]`	Wybiera wiersze i kolumny na podstawie pozycji określonej za pomocą wartości całkowitoliczbowej.
`df.at[`*`etykieta_i`*`, `*`etykieta_j`*`]`	Wybiera pojedynczą wartość skalarną na podstawie etykiety wiersza i kolumny.
`df.iat[`*`i`*`, `*`j`*`]`	Wybiera pojedynczą wartość skalarną z wiersza i kolumny zdefiniowanych za pomocą pozycji (wartości całkowitoliczbowych).
metoda `reindex`	Wybiera wiersze lub kolumny na podstawie etykiet.
metody `get_value` i `set_value`	Wybór pojedynczej wartości na podstawie etykiet wiersza i kolumny.

Na początku procesu projektowania biblioteki pandas uważałem, że wybieranie kolumny za pomocą notacji `ramkadanych[:, kolumna]` jest zbyt kłopotliwe i naraża kod na potencjalne błędy, ponieważ wybieranie kolumn jest jedną z najczęściej wykonywanych operacji. Poszedłem na kompromis i upchnąłem techniki obsługi indeksów (etykiet, a także wartości całkowitoliczbowych) do operatora `ix`. W praktyce doprowadziło to do wielu problematycznych sytuacji z danymi, w których etykiety osi oznaczono etykietami w formie liczb całkowitych, dlatego zespół programistów pakietu pandas zdecydował się na utworzenie operatorów `loc` i `iloc`, które miały obsługiwać tylko odpowiednie indeksy w formie etykiet (`loc`) i liczb całkowitych (`iloc`).

Operator indeksowania `ix` jest wciąż dostępny, ale jego używanie jest niezalecane. Odradzam Ci korzystanie z niego.

Indeksy w postaci liczb całkowitych

Praca z obiektami pandas oznaczonymi indeksami w postaci liczb całkowitych jest czymś, co sprawia trudność wielu nowym użytkownikom. Wynika to z różnic składni indeksowania pomiędzy strukturami pandas a wbudowanymi strukturami Pythona — listami i krotkami. Prawdopodobnie nie uważasz, że poniższy kod może wygenerować błąd:

```
ser = pd.Series(np.arange(3.))
ser
ser[-1]
```

Pakiet pandas może rozpocząć indeksowanie od końca, ale ogólnie rzecz biorąc, w wielu sytuacjach może to doprowadzić do powstania błędów. W zaprezentowanym przykładzie indeks składa się z liczb 0, 1 i 2, ale pandas ma problem z określeniem tego, czego chce użytkownik (biblioteka nie wie, czy zastosowano indeksowanie za pomocą etykiet, czy za pomocą numeru pozycji):

```
In [144]: ser
Out[144]:
0   0.0
1   1.0
2   2.0
dtype: float64
```

W przypadku indeksów niemających formy liczb całkowitych nie ma problemu z dwuznacznością:

```
In [145]: ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])

In [146]: ser2[-1]
Out[146]: 2.0
```

W przypadku etykiet osi zawierających wartości liczbowe wybór danych będzie zawsze oparty na etykietach — dla ujednolicenia. W celu zwiększenia precyzji można skorzystać z atrybutu `loc` (dla etykiet) lub `iloc` (dla liczb całkowitych):

```
In [147]: ser[:1]
Out[147]:
0   0.0
dtype: float64

In [148]: ser.loc[:1]
Out[148]:
```

```
0   0.0
1   1.0
dtype: float64

In [149]: ser.iloc[:1]
Out[149]:
0   0.0
dtype: float64
```

Działania arytmetyczne i wyrównywanie danych

Ważną cechą pakietu pandas z punktu widzenia niektórych zastosowań jest sposób przeprowadzenia operacji arytmetycznych na obiektach z różnymi indeksami. Jeżeli podczas dodawania któraś z par indeksów nie jest taka sama, to wartości oznaczone takimi indeksami zostaną zaprezentowane w wyniku w postaci unii. Informacja dla osób mających doświadczenie w pracy z bazami danych: działanie to przypomina automatyczne wykonywanie zewnętrznej operacji łączenia (*outer join*) na etykietach indeksu. Przyjrzyj się następującemu przykładowi:

```
In [150]: s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

In [151]: s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],
   .....:                index=['a', 'c', 'e', 'f', 'g'])

In [152]: s1
Out[152]:
a   7.3
c  -2.5
d   3.4
e   1.5
dtype: float64

In [153]: s2
Out[153]:
a  -2.1
c   3.6
e  -1.5
f   4.0
g   3.1
dtype: float64
```

Po dodaniu tych obiektów otrzymamy:

```
In [154]: s1 + s2
Out[154]:
a  5.2
c  1.1
d  NaN
e  0.0
f  NaN
g  NaN
dtype: float64
```

Wewnętrzny mechanizm wyrównywania danych wstawia brakujące wartości w miejsca, w których etykiety się nie nakładają. Brakujące wartości przechodzą dalej do kolejnych operacji matematycznych.

W przypadku obiektu `DataFrame` wyrównaniu podlegają wiersze i kolumny:

```
In [155]: df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),
   .....:                    index=['Ohio', 'Texas', 'Colorado'])
```

```
In [156]: df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),
   .....:                    index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [157]: df1
Out[157]:
            b    c    d
Ohio      0.0  1.0  2.0
Texas     3.0  4.0  5.0
Colorado  6.0  7.0  8.0

In [158]: df2
Out[158]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0
```

W wyniku dodania do siebie tych dwóch obiektów powstanie ramka danych, której indeksy i kolumny są uniami kolumn i indeksów dodawanych obiektów:

```
In [159]: df1 + df2
Out[159]:
            b    c     d    e
Colorado  NaN  NaN   NaN  NaN
Ohio      3.0  NaN   6.0  NaN
Oregon    NaN  NaN   NaN  NaN
Texas     9.0  NaN  12.0  NaN
Utah      NaN  NaN   NaN  NaN
```

Kolumny `c` i `e` nie występują w obu dodawanych obiektach `DataFrame`, a więc w obiekcie wyjściowym są one wypełnione brakującymi wartościami. Takie samo zjawisko występuje również wtedy, gdy jakieś etykiety wierszy nie występują w obu obiektach.

Jeżeli dodasz do siebie dwa obiekty `DataFrame`, w których nie występują żadne wspólne etykiety kolumn i wierszy, to w wyniku znajdą się same wartości `null`:

```
In [160]: df1 = pd.DataFrame({'A': [1, 2]})

In [161]: df2 = pd.DataFrame({'B': [3, 4]})

In [162]: df1
Out[162]:
   A
0  1
1  2

In [163]: df2
Out[163]:
   B
0  3
1  4

In [164]: df1 - df2
Out[164]:
    A   B
0 NaN NaN
1 NaN NaN
```

Metody arytmetyczne i wypełnianie wartościami

Podczas wykonywania operacji matematycznych pomiędzy obiektami o różnych indeksach w sytuacji, gdy oś o danej etykiecie znajduje się tylko w jednym obiekcie, możesz wstawić określoną wartość, taką jak np. 0:

```
In [165]: df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),
   .....:                    columns=list('abcd'))

In [166]: df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),
   .....:                    columns=list('abcde'))

In [167]: df2.loc[1, 'b'] = np.nan

In [168]: df1
Out[168]:
     a    b     c     d
0  0.0  1.0   2.0   3.0
1  4.0  5.0   6.0   7.0
2  8.0  9.0  10.0  11.0

In [169]: df2
Out[169]:
      a     b     c     d     e
0   0.0   1.0   2.0   3.0   4.0
1   5.0   NaN   7.0   8.0   9.0
2  10.0  11.0  12.0  13.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

Dodanie do siebie tych obiektów spowoduje umieszczenie wartości *NA* w miejscach, w których etykiety obu obiektów nie pokrywają się:

```
In [170]: df1 + df2
Out[170]:
      a     b     c     d    e
0   0.0   2.0   4.0   6.0  NaN
1   9.0   NaN  13.0  15.0  NaN
2  18.0  20.0  22.0  24.0  NaN
3   NaN   NaN   NaN   NaN  NaN
```

Używając metody `add` na obiekcie `df1`, mogę przekazać obiekt `df2` oraz argument definiujący wartość używaną do wypełnienia (`fill_value`):

```
In [171]: df1.add(df2, fill_value=0)
Out[171]:
      a     b     c     d     e
0   0.0   2.0   4.0   6.0   4.0
1   9.0   5.0  13.0  15.0   9.0
2  18.0  20.0  22.0  24.0  14.0
3  15.0  16.0  17.0  18.0  19.0
```

W tabeli 5.5 wymieniono metody operacji arytmetycznych wykonywanych na obiektach `Series` i `DataFrame`. Każda z metod ma swój odpowiednik rozpoczynający się od litery `r`, który charakteryzuje się odwróconymi argumentami. W związku z tym dwie poniższe metody dają taki sam efekt:

```
In [172]: 1 / df1
Out[172]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

```
In [173]: df1.rdiv(1)
Out[173]:
          a         b         c         d
0       inf  1.000000  0.500000  0.333333
1  0.250000  0.200000  0.166667  0.142857
2  0.125000  0.111111  0.100000  0.090909
```

Tabela 5.5. Elastyczne metody arytmetyczne

Metoda	Opis
`add, radd`	Metody służące do dodawania (+).
`sub, rsub`	Metody służące do odejmowania (–).
`div, rdiv`	Metody służące do dzielenia (/).
`floordiv, rfloordiv`	Metody służące do dzielenia całkowitego (//).
`mul, rmul`	Metody służące do mnożenia (*).
`pow, rpow`	Metody służące do potęgowania (**).

Podczas uaktualniania indeksu obiektu typu `Series` lub `DataFrame` możesz również określić inną wartość używaną do wypełniania brakujących wartości:

```
In [174]: df1.reindex(columns=df2.columns, fill_value=0)
Out[174]:
     a    b     c     d  e
0  0.0  1.0   2.0   3.0  0
1  4.0  5.0   6.0   7.0  0
2  8.0  9.0  10.0  11.0  0
```

Operacje pomiędzy obiektami DataFrame i Series

Operacje arytmetyczne pomiędzy obiektami `DataFrame` i `Series` są zdefiniowane podobnie jak w przypadku tablic NumPy o różnej liczbie wymiarów. Przyjrzyj się poniższemu przykładowi — różnicy pomiędzy dwuwymiarową tablicą i jednym z jej wierszy:

```
In [175]: arr = np.arange(12.).reshape((3, 4))

In [176]: arr
Out[176]:
array([[  0.,  1.,  2.,  3.],
       [  4.,  5.,  6.,  7.],
       [  8.,  9.,  10.,  11.]])

In [177]: arr[0]
Out[177]: array([ 0.,  1.,  2.,  3.])

In [178]: arr - arr[0]
Out[178]:
array([[ 0.,  0.,  0.,  0.],
       [ 4.,  4.,  4.,  4.],
       [ 8.,  8.,  8.,  8.]])
```

Jak widzisz, podczas odejmowania `arr[0]` od `arr` operacja odejmowania jest wykonywana raz dla każdego wiersza. Zjawisko to określa się mianem **rozgłaszania**. Więcej informacji na jego temat znajdziesz w „Dodatku A”. W podobny sposób działają operacje pomiędzy obiektami `DataFrame` i `Series`:

```
In [179]: frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),
   .....:                      columns=list('bde'),
```

```
   .....:                     index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [180]: series = frame.iloc[0]

In [181]: frame
Out[181]:
          b     d     e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [182]: series
Out[182]:
b  0.0
d  1.0
e  2.0
Name: Utah, dtype: float6
```

Domyślnie operacje arytmetyczne pomiędzy obiektami `DataFrame` i `Series` dobierają indeks obiektu `Series` do kolumn obiektu `DataFrame` i — wykonując operację rozgłaszania — przechodzą w dół wierszy:

```
In [183]: frame - series
Out[183]:
          b    d    e
Utah    0.0  0.0  0.0
Ohio    3.0  3.0  3.0
Texas   6.0  6.0  6.0
Oregon  9.0  9.0  9.0
```

Jeżeli wartość indeksu nie zostanie znaleziona w kolumnie obiektu `DataFrame` lub w indeksie obiektu `Series`, to obiekty zostaną przeindeksowane w celu utworzenia unii:

```
In [184]: series2 = pd.Series(range(3), index=['b', 'e', 'f'])

In [185]: frame + series2
Out[185]:
          b    d     e    f
Utah    0.0  NaN   3.0  NaN
Ohio    3.0  NaN   6.0  NaN
Texas   6.0  NaN   9.0  NaN
Oregon  9.0  NaN  12.0  NaN
```

Jeżeli zamiast tego chcesz przeprowadzić operację rozgłaszania po kolumnach, dopasowując wiersze, musisz skorzystać z jednej z metod arytmetycznych, takich jak:

```
In [186]: series3 = frame['d']

In [187]: frame
Out[187]:
b  d  e
Utah    0.0   1.0   2.0
Ohio    3.0   4.0   5.0
Texas   6.0   7.0   8.0
Oregon  9.0  10.0  11.0

In [188]: series3
Out[188]:
Utah     1.0
Ohio     4.0
Texas    7.0
```

```
Oregon  10.0
Name: d, dtype: float64

In [189]: frame.sub(series3, axis='index')
Out[189]:
          b    d    e
Utah   -1.0  0.0  1.0
Ohio   -1.0  0.0  1.0
Texas  -1.0  0.0  1.0
Oregon -1.0  0.0  1.0
```

Przekazany numer osi określa *oś, po której ma zostać przeprowadzone dobieranie*. W zaprezentowanym przypadku chciałem przeprowadzić dobieranie po indeksie wiersza obiektu `Dataframe` (`axis='index'` lub `axis=0`), a następnie wykonać rozgłaszanie w poprzek.

Funkcje apply i map

Funkcje uniwersalne pakietu NumPy (metody przetwarzające tablice element po elemencie) działają również z obiektami pandas:

```
In [190]: frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),
   .....:                      index=['Utah', 'Ohio', 'Texas', 'Oregon'])

In [191]: frame
Out[191]:
                b         d         e
Utah    -0.204708  0.478943 -0.519439
Ohio    -0.555730  1.965781  1.393406
Texas    0.092908  0.281746  0.769023
Oregon   1.246435  1.007189 -1.296221

In [192]: np.abs(frame)
Out[192]:
                b         d         e
Utah     0.204708  0.478943  0.519439
Ohio     0.555730  1.965781  1.393406
Texas    0.092908  0.281746  0.769023
Oregon   1.246435  1.007189  1.296221
```

Kolejną często wykonywaną operacją jest przetwarzanie każdego wiersza lub kolumny tablicy jednowymiarowej za pomocą funkcji. Można to zrobić za pomocą metody `apply` obiektu `DataFrame`:

```
In [193]: f = lambda x: x.max() - x.min()

In [194]: frame.apply(f)
Out[194]:
b  1.802165
d  1.684034
e  2.689627
dtype: float64
```

Funkcja `f` oblicza różnicę pomiędzy maksymalną i minimalną wartością obiektu `Series`. W zaprezentowanym przykładzie została wywołana dla każdej kolumny obiektu `frame`. Wynikiem jest obiekt `Series`, w którym indeksem są etykiety kolumn obiektu `frame`.

Jeżeli do funkcji `apply` przekazany zostanie argument `axis='columns'`, to funkcja będzie tym razem wywoływana jednokrotnie dla poszczególnych wierszy:

```
In [195]: frame.apply(f, axis='columns')
Out[195]:
Utah    0.998382
Ohio    2.521511
Texas   0.676115
Oregon  2.542656
dtype: float64
```

Większość standardowych parametrów statystycznych macierzy, takich jak suma (`sum`) i średnia (`mean`), można wyznaczyć za pomocą metod obiektu `DataFrame`, a więc korzystanie z funkcji `apply` nie jest w ich przypadku konieczne.

Funkcja przekazana do funkcji `apply` nie musi zwracać wartości skalarnej. Może również zwrócić serię zawierającą wiele wartości:

```
In [196]: def f(x):
   .....:     return pd.Series([x.min(), x.max()], index=['min', 'max'])

In [197]: frame.apply(f)
Out[197]:
            b          d          e
min  -0.555730   0.281746  -1.296221
max   1.246435   1.965781   1.393406
```

Istnieje również możliwość użycia funkcji Pythona przetwarzających element po elemencie. Załóżmy, że chcesz uzyskać sformatowany łańcuch na podstawie poszczególnych wartości zmiennoprzecinkowych znajdujących się w obiekcie `frame`. Możesz to zrobić za pomocą metody `applymap`:

```
In [198]: format = lambda x: '%.2f' % x

In [199]: frame.applymap(format)
Out[199]:
           b     d      e
Utah   -0.20  0.48  -0.52
Ohio   -0.56  1.97   1.39
Texas   0.09  0.28   0.77
Oregon  1.25  1.01  -1.30
```

Nazwa `applymap` (zastosuj mapowanie) wynika z tego, że obiekt `Series` dysponuje metodą `map` przeznaczoną do wykonywania funkcji przetwarzających dane element po elemencie:

```
In [200]: frame['e'].map(format)
Out[200]:
Utah    -0.52
Ohio     1.39
Texas    0.77
Oregon  -1.30
Name: e, dtype: object
```

Sortowanie i tworzenie rankingów

Sortowanie zbioru danych według określonych kryteriów to kolejny ważny element pakietu pandas. W celu wykonania sortowania leksykograficznego po wierszach lub kolumnach należy skorzystać z metody `sort_index`, która zwraca nowy, posortowany obiekt:

```
In [201]: obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])

In [202]: obj.sort_index()
Out[202]:
a  1
```

```
b  2
c  3
d  0
dtype: int64
```

Ramki danych mogą być sortowane po indeksie dowolnej osi:

```
In [203]: frame = pd.DataFrame(np.arange(8).reshape((2, 4)),
   .....:                      index=['three', 'one'],
   .....:                      columns=['d', 'a', 'b', 'c'])

In [204]: frame.sort_index()
Out[204]:
       d  a  b  c
one    4  5  6  7
three  0  1  2  3

In [205]: frame.sort_index(axis=1)
Out[205]:
       a  b  c  d
three  1  2  3  0
one    5  6  7  4
```

Dane są domyślnie sortowane w kolejności wzrastającej, ale mogą być również posortowane w kolejności malejącej:

```
In [206]: frame.sort_index(axis=1, ascending=False)
Out[206]:
       d  c  b  a
three  0  3  2  1
one    4  7  6  5
```

W celu posortowania serii według umieszczonych w niej wartości należy skorzystać z metody `sort_values`:

```
In [207]: obj = pd.Series([4, 7, -3, 2])

In [208]: obj.sort_values()
Out[208]:
2  -3
3   2
0   4
1   7
dtype: int64
```

Podczas sortowania serii domyślnie wszelkie brakujące wartości są umieszczane na końcu:

```
In [209]: obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])

In [210]: obj.sort_values()
Out[210]:
4  -3.0
5   2.0
0   4.0
2   7.0
1   NaN
3   NaN
dtype: float64
```

Podczas sortowania ramki danych możesz użyć danych z jednej lub z kilku kolumn w roli kluczy sortowania. W tym celu należy przekazać nazwę kolumny (lub nazwy kolumn) do opcji `by` metody `sort_values`:

```
In [211]: frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})

In [212]: frame
Out[212]:
   a  b
0  0  4
1  1  7
2  0 -3
3  1  2

In [213]: frame.sort_values(by='b')
Out[213]:
   a  b
2  0 -3
3  1  2
0  0  4
1  1  7
```

W celu wykonania sortowania po wielu kolumnach należy przekazać listę ich nazw:

```
In [214]: frame.sort_values(by=['a', 'b'])
Out[214]:
   a  b
2  0 -3
0  0  4
3  1  2
1  1  7
```

Tworzenie rankingu polega na przypisaniu rangi w postaci wartości od jednego do wartości równej liczbie poprawnych punktów danych tablicy. W celu utworzenia rankingu serii lub ramki danych należy skorzystać z metody `ran`. Metoda ta domyślnie zrywa powiązania, przypisując każdej grupie średnią wartość rangi:

```
In [215]: obj = pd.Series([7, -5, 7, 4, 2, 0, 4])

In [216]: obj.rank()
Out[216]:
0  6.5
1  1.0
2  6.5
3  4.5
4  3.0
5  2.0
6  4.5
dtype: float64
```

Rangi mogą być również przypisywane według kolejności występowania w danych:

```
In [217]: obj.rank(method='first')
Out[217]:
0  6.0
1  1.0
2  7.0
3  4.0
4  3.0
5  2.0
6  5.0
dtype: float64
```

W zaprezentowanym przykładzie zamiast korzystać ze średniej rangi równiej 6,5 dla elementów `0` i `2`, skorzystano z wartości `6` i `7`. Wynika to z tego, że etykieta `0` znajduje się przed etykietą `2`.

Możliwe jest również tworzenie rankingów w kolejności malejącej:

```
# Wartościom przypisz maksymalną rangę w grupie.
In [218]: obj.rank(ascending=False, method='max')
Out[218]:
0  2.0
1  7.0
2  2.0
3  4.0
4  5.0
5  6.0
6  4.0
dtype: float64
```

W tabeli 5.6 znajdziesz listę dostępnych metod przełamujących powiązania.

Tabela 5.6. Metody przerywające powiązania z rangami

Metoda	Opis
`'average'`	Metoda domyślna: przypisuje średnią rangę każdemu elementowi w równej grupie.
`'min'`	Używa minimalnej rangi dla całej grupy.
`'max'`	Używa maksymalnej rangi dla całej grupy.
`'first'`	Przypisuje rangi w kolejności, w jakiej wartości występują w danych.
`'dense'`	Działa podobnie jak `method='min'`, ale rangi pomiędzy grupami zawsze wzrastają o 1, a nie o liczbę równą liczbie elementów w grupie.

W przypadku obiektu `DataFrame` możliwe jest określenie rang w wierszach lub kolumnach:

```
In [219]: frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],
   .....:                       'c': [-2, 5, 8, -2.5]})

In [220]: frame
Out[220]:
   a    b    c
0  0  4.3 -2.0
1  1  7.0  5.0
2  0 -3.0  8.0
3  1  2.0 -2.5

In [221]: frame.rank(axis='columns')
Out[221]:
     a    b    c
0  2.0  3.0  1.0
1  1.0  3.0  2.0
2  2.0  1.0  3.0
3  2.0  3.0  1.0
```

Indeksy osi ze zduplikowanymi etykietami

Dotychczas analizowaliśmy tylko przykłady, w których etykiety osi (wartości indeksu) były unikalne. Wiele funkcji pakietu pandas (np. funkcja `reindex`) wymaga tego, aby etykiety były unikalne, ale nie jest to obowiązkowe. Oto przykład małej serii ze zduplikowanymi wartościami indeksu:

```
In [222]: obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])

In [223]: obj
Out[223]:
```

```
a  0
a  1
b  2
b  3
c  4
dtype: int64
```

Unikalność etykiet można określić za pomocą własności `is_unique`:

```
In [224]: obj.index.is_unique
Out[224]: False
```

Wybór danych jest jedną z operacji, które działają inaczej z duplikatami. Indeksowanie etykiety przypisanej do wielu elementów zwraca serię. W przypadku pojedynczego elementu przypisanego do etykiety zwracana jest wartość skalarna:

```
In [225]: obj['a']
Out[225]:
a  0
a  1
dtype: int64

In [226]: obj['c']
Out[226]: 4
```

Może to skomplikować Twój kod, ponieważ typ danych uzyskanych w wyniku indeksowania zależy od tego, czy dana etykieta jest unikalna, czy się powtarza.

Ta sama logika dotyczy również indeksowania wierszy ramki danych:

```
In [227]: df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])

In [228]: df
Out[228]:
          0         1         2
a  0.274992  0.228913  1.352917
a  0.886429 -2.001637 -0.371843
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228

In [229]: df.loc['b']
Out[229]:
          0         1         2
b  1.669025 -0.438570 -0.539741
b  0.476985  3.248944 -1.021228
```

5.3. Podsumowywanie i generowanie statystyk opisowych

Obiekty pakietu pandas obsługują zestaw praktycznych metod matematycznych i statystycznych. Większość z nich dostarcza **statystycznych danych podsumowujących** lub służy do **redukcji**. Metody tego typu zwracają pojedynczą wartość (np. sumę lub średnią) wygenerowaną na podstawie serii lub serii wartości z wierszy lub kolumn ramki danych. W porównaniu z podobnymi metodami tablic NumPy charakteryzują się one wbudowanymi procedurami obsługi brakujących danych. Przyjrzyj się następującemu przykładowi małej ramki danych:

```
In [230]: df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],
   .....:                    [np.nan, np.nan], [0.75, -1.3]],
   .....:                   index=['a', 'b', 'c', 'd'],
   .....:                   columns=['one', 'two'])
```

```
In [231]: df
Out[231]:
    one   two
a  1.40   NaN
b  7.10  -4.5
c   NaN   NaN
d  0.75  -1.3
```

Wywołanie metody `sum` tej ramki danych zwróci serię zawierającą sumy poszczególnych kolumn:

```
In [232]: df.sum()
Out[232]:
one   9.25
two  -5.80
dtype: float64
```

Przekazanie argumentu `axis='columns'` lub `axis=1` spowoduje zwrócenie sum elementów umieszczonych w poszczególnych kolumnach:

```
In [233]: df.sum(axis='columns')
Out[233]:
a   1.40
b   2.60
c    NaN
d  -0.55
dtype: float64
```

Brakujące wartości nie są brane pod uwagę, dopóki cały wycinek (wiersz lub kolumna) nie zawiera samych wartości typu NA. Funkcję tę można wyłączyć za pomocą opcji `skipna`:

```
In [234]: df.mean(axis='columns', skipna=False)
Out[234]:
a     NaN
b   1.300
c     NaN
d  -0.275
dtype: float64
```

W tabeli 5.7 wymieniono najczęściej używane opcje metod redukujących.

Tabela 5.7. *Opcje metod redukujących*

Opcja	Opis
`axis`	Wybór osi, po której ma zostać przeprowadzona redukcja; w przypadku ramki danych `0` powoduje przeprowadzenie operacji na wierszach, a `1` — na kolumnach.
`skipna`	Wykluczanie brakujących wartość; domyślnie włączone (`True`).
`level`	Redukcja grupy o poziom w przypadku hierarchicznego indeksowania osi (wieloindeks).

Niektóre metody, takie jak `idxmin` i `odxmax`, zwracają pośrednie statystyki — takie jak indeksy, pod którymi znajdują się wartości minimalne i maksymalne:

```
In [235]: df.idxmax()
Out[235]:
one  b
two  d
dtype: object
```

Inne metody zwracają **akumulacje**:

```
In [236]: df.cumsum()
Out[236]:
    one   two
a  1.40   NaN
b  8.50  -4.5
c   NaN   NaN
d  9.25  -5.8
```

Istnieją również metody, które nie są reduktorami ani nie generują akumulacji. Przykładem takiej metody jest `describe`. Metoda ta zwraca za jednym zamachem wiele statystyk podsumowujących obiekt:

```
In [237]: df.describe()
Out[237]:
             one       two
count  3.000000  2.000000
mean   3.083333 -2.900000
std    3.493685  2.262742
min    0.750000 -4.500000
25%    1.075000 -3.700000
50%    1.400000 -2.900000
75%    4.250000 -2.100000
max    7.100000 -1.300000
```

W przypadku danych nienumerycznych metoda `describe` zwraca inne parametry statystyczne:

```
In [238]: obj = pd.Series(['a', 'a', 'b', 'c'] * 4)

In [239]: obj.describe()
Out[239]:
count  16
unique  3
top     a
freq    8
dtype: object
```

W tabeli 5.8 znajdziesz pełną listę statystyk podsumowujących i związanych z nimi metod.

Tabela 5.8. Statystyki opisowe i podsumowujące

Metoda	Opis
`count`	Liczba wartości innych niż *NA*.
`describe`	Zwraca zestaw statystyk podsumowujących serię lub każdą kolumnę ramki danych.
`min, max`	Określa wartości minimalną i maksymalną.
`argmin, argmax`	Zwraca indeks (liczbę), pod którą znajduje się wartość minimalna lub maksymalna.
`idxmin, idxmax`	Zwraca etykietę indeksu, pod którą znajduje się wartość minimalna lub maksymalna.
`quantile`	Zwraca przykładowy kwantyl w zakresie od 0 do 1.
`sum`	Oblicza sumę wartości.
`mean`	Oblicza średnią wartości.
`median`	Określa medianę wartości (kwantyl 50%).
`mad`	Oblicza średnią odchylenia bezwzględnego od wartości średniej.
`prod`	Zwraca iloczyn wszystkich wartości.

Tabela 5.8. Statystyki opisowe i podsumowujące — ciąg dalszy

Metoda	Opis
`var`	Określa wariancję wartości próbki.
`std`	Oblicza odchylenie standardowe wartości próbki.
`skew`	Określa asymetrię (skośność) wartości próbki (trzeci moment).
`kurt`	Określa asymetrię (skośność) wartości próbki (czwarty moment).
`cumsum`	Oblicza sumę kumulacyjną wartości.
`cummin`, `cummax`	Oblicza kumulacyjne minimum i maksimum.
`cumprod`	Oblicza kumulacyjny iloczyn wartości.
`diff`	Oblicza pierwszą różnicę (metoda przydatna podczas pracy z szeregami czasowymi).
`pct_change`	Oblicza zmianę procentową.

Współczynnik korelacji i kowariancja

Niektóre statystyki podsumowujące, takie jak współczynnik korelacji i kowariancja, są obliczane na podstawie pary argumentów. Skorzystajmy z przykładów ramek danych zawierających dane na temat cen i nakładów aukcji pobranych z serwisu Yahoo! Finance za pomocą dodatkowego pakietu `pandas-datareader`. Jeżeli nie zainstalowałeś go wcześniej, to możesz to zrobić za pomocą polecenia `conda` lub `pip`.

```
conda install pandas-datareader
```

Korzystam z modułu `pandas-datareader` w celu pobrania danych na temat kilku podmiotów giełdowych:

```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker)
            for ticker in ['AAPL', 'IBM', 'MSFT', 'GOOG']}
price = pd.DataFrame({ticker: data['Adj Close']
                      for ticker, data in all_data.items()})
volume = pd.DataFrame({ticker: data['Volume']
                       for ticker, data in all_data.items()})
```

Możliwe, że w chwili, gdy czytasz te słowa, serwis Yahoo! Finance już nie istnieje (spółka Yahoo! została przejęta w 2017 roku przez korporację Verizon). Opis aktualnych możliwości pakietu pandas-datareader znajdziesz w jego dokumentacji dostępnej w internecie.

Teraz obliczę zmiany procentowe cen. Jest to operacja wykonywana na danych szeregu czasowego, którą opiszę dokładniej w rozdziale 11.:

```
n [242]: returns = price.pct_change()

In [243]: returns.tail()
Out[243]:

                AAPL      GOOG       IBM      MSFT
Date
2016-10-17 -0.000680  0.001837  0.002072 -0.003483
2016-10-18 -0.000681  0.019616 -0.026168  0.007690
2016-10-19 -0.002979  0.007846  0.003583 -0.002255
2016-10-20 -0.000512 -0.005652  0.001719 -0.004867
2016-10-21 -0.003930  0.003011 -0.012474  0.042096
```

Metoda corr serii określa współczynnik korelacji nakładających się, wyrównanych za pomocą indeksu wartości umieszczonych w dwóch obiektach typu Series (dotyczy to wartości typu innego niż NA). Metoda cov w podobny sposób oblicza kowariancję:

```
n [244]: returns['MSFT'].corr(returns['IBM'])
Out[244]: 0.49976361144151144

In [245]: returns['MSFT'].cov(returns['IBM'])
Out[245]: 8.8706554797035462e-05
```

MSFT jest poprawnym atrybutem Pythona, a więc możemy dokonać wyboru kolumn za pomocą bardziej zwięzłej składni:

```
In [246]: returns.MSFT.corr(returns.IBM)
Out[246]: 0.49976361144151144
```

Metody corr i conv ramek danych zwracają pełną macierz współczynników korelacji i kowariancji w formie ramki danych:

```
In [247]: returns.corr()
Out[247]:
          AAPL      GOOG       IBM      MSFT
AAPL  1.000000  0.407919  0.386817  0.389695
GOOG  0.407919  1.000000  0.405099  0.465919
IBM   0.386817  0.405099  1.000000  0.499764
MSFT  0.389695  0.465919  0.499764  1.000000

In [248]: returns.cov()
Out[248]:
          AAPL      GOOG       IBM      MSFT
AAPL  0.000277  0.000107  0.000078  0.000095
GOOG  0.000107  0.000251  0.000078  0.000108
IBM   0.000078  0.000078  0.000146  0.000089
MSFT  0.000095  0.000108  0.000089  0.000215
```

Metoda corrwith obiektu DataFrame umożliwia obliczenie korelacji pomiędzy parami elementów kolumn lub wierszy ramki danych i innym obiektem typu Series lub DataFrame. Przekazanie serii powoduje zwrócenie serii zawierającej współczynniki korelacji obliczone dla każdej kolumny:

```
In [249]: returns.corrwith(returns.IBM)
Out[249]:
AAPL  0.386817
GOOG  0.405099
IBM   1.000000
MSFT  0.499764
dtype: float64
```

Przekazanie ramki danych spowoduje zwrócenie współczynników korelacji pomiędzy kolumnami o takich samych nazwach. W poniższym przykładzie obliczam korelacje pomiędzy procentowymi zmianami ilości:

```
In [250]: returns.corrwith(volume)
Out[250]:
AAPL  -0.075565
GOOG  -0.007067
IBM   -0.204849
MSFT  -0.092950
dtype: float64
```

Przekazanie argumentu axis='columns' spowoduje wykonanie operacji wiersz po wierszu. Niezależnie od tego, czy korelacja jest obliczana dla wierszy, czy dla kolumn, przed jej wyznaczeniem punkty danych są wyrównywane na podstawie etykiet.

Unikalne wartości, ich liczba i przynależność

Kolejna grupa metod, którą chciałbym opisać, służy do uzyskiwania informacji dotyczących wartości znajdujących się w jednowymiarowych seriach. Działanie tych metod przedstawię na przykładzie następującej serii:

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

Na początek przyjrzyjmy się funkcji `unique`, która zwraca tablicę unikalnych wartości znajdujących się w obiekcie `Series`:

```
In [252]: uniques = obj.unique()
In [253]: uniques

Out[253]: array(['c', 'a', 'd', 'b'], dtype=object)
```

Unikalne wartości nie zawsze muszą być zwracane w posortowanej kolejności, ale w razie konieczności można je posortować później (`uniques.sort()`). Funkcja `value_counts` zwraca natomiast obiekt `Series` zawierający dane na temat częstotliwości występowania poszczególnych wartości:

```
In [254]: obj.value_counts()
Out[254]:
c  3
a  3
b  2
d  1
dtype: int64
```

Dla ułatwienia wartości znajdujące się w obiekcie `Series` są sortowane w kolejności malejącej. Metoda `value_counts` jest jednocześnie wysokopoziomową metodą biblioteki pandas, która może być użyta w kontekście dowolnej tablicy lub sekwencji:

```
In [255]: pd.value_counts(obj.values, sort=False)
Out[255]:
a  3
b  2
c  3
d  1
dtype: int64
```

Metoda `isin` wykonuje wektorową operację sprawdzania przynależności do zbioru. Można z niej korzystać podczas filtrowania zbioru danych w celu uzyskania podzbioru wartości w formie obiektu `Series` lub kolumny obiektu `DataFrame`:

```
In [256]: obj
Out[256]:
0  c
1  a
2  d
3  a
4  a
5  b
6  b
7  c
8  c
dtype: object

In [257]: mask = obj.isin(['b', 'c'])

In [258]: mask
```

```
Out[258]:
0   True
1  False
2  False
3  False
4  False
5   True
6   True
7   True
8   True
dtype: bool

In [259]: obj[mask]
Out[259]:
0  c
5  b
6  b
7  c
8  c
dtype: object
```

Metoda `Index.get_indexer` jest związana z metodą `isin`. Zwraca ona tablicę indeksów wygenerowaną na podstawie tablicy, w której mogą znajdować się identyczne wartości, umożliwiając utworzenie kolejnej tablicy z unikalnymi wartościami:

```
In [260]: to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])

In [261]: unique_vals = pd.Series(['c', 'b', 'a'])

In [262]: pd.Index(unique_vals).get_indexer(to_match)
Out[262]: array([0, 2, 1, 1, 0, 2])
```

Informacje na temat tych metod podsumowano w tabeli 5.9.

Tabela 5.9. Metody umożliwiające określenie unikalnych wartości i liczby powtórzeń wartości oraz przynależności wartości do zbioru

Metoda	Opis
`isin`	Zwraca tablicę wartości logicznych określającą, czy każda z wartości serii znajduje się w przekazanej sekwencji wartości.
`match`	Określa wartości indeksów dla każdej wartości znajdującej się w tablicy. Dotyczy to indeksów, pod którymi znajdują się dane wartości w innej tablicy zawierającej unikalne wartości; przydaje się podczas wyrównywania danych i operacji łączenia.
`unique`	Zwraca tablicę unikalnych wartości wchodzących w skład serii; wartości są zwracane w kolejności, w jakiej zostały zaobserwowane.
`value_counts`	Zwraca serię, w której indeksem są unikalne wartości, a wartościami — liczby wystąpień poszczególnych wartości; dane są posortowane w kolejności malejącej.

Czasami zachodzi potrzeba obliczenia histogramu wielu spokrewnionych kolumn ramki danych. Oto przykład takiej operacji:

```
In [263]: data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],
   .....:                      'Qu2': [2, 3, 1, 2, 3],
   .....:                      'Qu3': [1, 5, 2, 4, 4]})
In [264]: data
Out[264]:
   Qu1  Qu2  Qu3
```

```
0    1    2    1
1    3    3    5
2    4    1    2
3    3    2    4
4    4    3    4
```

Przekazanie `pandas.value_counts` do metody `apply` tej ramki danych da następujący efekt:

```
In [265]: result = data.apply(pd.value_counts).fillna(0)

In [266]: result
Out[266]:
   Qu1  Qu2  Qu3
1  1.0  1.0  1.0
2  0.0  2.0  1.0
3  2.0  2.0  0.0
4  2.0  0.0  2.0
5  0.0  0.0  1.0
```

W zaprezentowanym przykładzie etykiety wierszy wygenerowanego obiektu są unikalnymi wartościami występującymi we wszystkich kolumnach. W poszczególnych kolumnach zwróconego obiektu znajdują się wartości określające liczbę wystąpień danej wartości.

5.4. Podsumowanie

W kolejnym rozdziale opiszę narzędzia biblioteki pandas służące do wczytywania (*ładowania*) zbiorów danych, a także zapisywania ich na dysku. W dalszej kolejności zajmę się narzędziami przeznaczonymi do oczyszczania, przetwarzania, analizy i graficznej interpretacji danych.

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Y

Z

Ż

O'REILLY®

Python: poznaj idealne narzędzie do analizy danych!

Analiza danych stała się samodzielną dyscypliną wiedzy, interesującą specjalistów z wielu branż: analityków biznesowych, statystyków, architektów oprogramowania czy też osoby zajmujące się sztuczną inteligencją. Wydobywanie informacji ze zbiorów danych pozwala na uzyskanie wiedzy niedostępnej w inny sposób. W tym celu dane trzeba odpowiednio przygotować, oczyścić, przetworzyć i oczywiście poddać analizie. Warto również zadbać o ich wizualizację. Do tych wszystkich zadań najlepiej wykorzystać specjalne narzędzia opracowane w języku Python.

To drugie, zaktualizowane i uzupełnione wydanie klasycznego podręcznika napisanego z myślą o analitykach, którzy dotychczas nie pracowali w Pythonie, oraz o programistach Pythona, którzy nie zajmowali się dotąd analizą danych ani obliczeniami naukowymi. Przedstawiono tu możliwości oferowane przez Pythona 3.6 oraz najnowsze funkcje pakietów Pandas i NumPy, a także środowisk IPython i Jupyter. Przy opisie poszczególnych narzędzi analitycznych wyjaśniono ich działanie i zaprezentowano przykłady ich efektywnego i kreatywnego wykorzystania. Ta książka powinna się znaleźć w podręcznej bibliotece każdego analityka danych!

Najważniejsze zagadnienia:

- eksploracja danych za pomocą powłoki IPython i środowiska Jupyter
- korzystanie z pakietów NumPy i Pandas
- tworzenie wizualizacji danych za pomocą pakietu Matplotlib
- praca z danymi regularnych i nieregularnych szeregów czasowych
- rozwiązywanie rzeczywistych problemów analitycznych

Wes McKinney – jest świetnie znany jako twórca pakietu Pandas, popularnej otwartej biblioteki Pythona przeznaczonej do analizy danych. Zajmuje się językami Python i C++. Jest związany ze środowiskiem analityków pracujących w Pythonie i z Apache Software Foundation, z którą rozwija wiele ciekawych projektów. Obecnie pracuje w Nowym Jorku jako architekt oprogramowania. Często występuje w roli prelegenta na konferencjach. Uwielbia podróże, interesuje się lingwistyką i językami obcymi.

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 978-83-283-4081-7
9 788328 340817

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 89,00 zł