OKIEM EKSPERTA

Pandas Receptury

Obliczenia naukowe, szeregi czasowe i eksploracyjna analiza danych w Pythonie

Wydanie III

William Ayd
Matthew Harrison

Helion

packt

Tytuł oryginału: Pandas Cookbook: Practical recipes for scientific computing, time series, and exploratory data analysis using Python, 3rd Edition

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-3173-2

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/panre3
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/panre3.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 5

Algorytmy i ich zastosowanie

W tej książce przyjrzeliśmy się już różnym sposobom tworzenia struktur danych w bibliotece pandas oraz wybierania i przypisywania danych w nich zawartych. Ponadto wyjaśniliśmy, jak przechowywać te struktury w popularnych formatach. Te funkcje same w sobie czynią te bibliotekę potężnym narzędziem w dziedzinie wymiany danych, ale to dopiero początek jej możliwości.

Kluczowym elementem analizy danych i ogólnie pojętych obliczeń jest stosowanie *algorytmów*, które opisują sekwencję kroków, jakie komputer powinien wykonać, aby przetworzyć dane. W najprostszej formie popularne algorytmy przetwarzania danych opierają się na podstawowych działaniach arytmetycznych (np. „zsumuj tę kolumnę"), ale mogą rozszerzyć się do dowolnej sekwencji kroków potrzebnych do wykonania własnych obliczeń.

Jak zobaczysz w tym rozdziale, pandas oferuje wiele popularnych algorytmów przetwarzania danych, a także zapewnia solidne ramy, dzięki którym możesz tworzyć i stosować własne algorytmy. Algorytmy wbudowane w bibliotekę są szybsze niż cokolwiek, co mógłbyś napisać ręcznie w Pythonie. W miarę postępów w pracy z danymi przekonasz się, że umiejętne wykorzystanie dostępnych algorytmów może zaspokoić ogromną część potrzeb z zakresu przetwarzania danych.

W tym rozdziale omówimy następujące zagadnienia:

- podstawowe operacje arytmetyczne na obiektach `pd.Series`;
- podstawowe operacje arytmetyczne na obiektach `pd.DataFrame`;
- agregacje;
- transformacje;
- mapowanie;
- stosowanie funkcji;
- podsumowujące dane statystyczne;
- algorytmy grupowania;
- kodowanie „1 z *n*" za pomocą funkcji `pd.get_dummies`;
- łączenie operacji za pomocą metody `.pipe`;
- wybieranie filmów o najniższym budżecie z listy stu najlepszych;
- obliczanie ceny dla kroczącego zlecenia stop;
- wyszukiwanie najlepszych baseballistów;
- ustalanie pozycji zdobywającej najwięcej punktów dla drużyny.

Podstawowe operacje arytmetyczne na obiektach pd.Series

Najłatwiejszym punktem wyjścia do eksploracji algorytmów biblioteki pandas jest obiekt `pd.Series`, ponieważ jest to najbardziej podstawowa struktura, jaką oferuje. Podstawowe operacje arytmetyczne obejmują dodawanie, odejmowanie, mnożenie i dzielenie. Jak zobaczysz w tym podrozdziale, pandas oferuje dwa sposoby ich wykonywania. Pierwsze podejście pozwala na pracę z operatorami `+`, `-`, `*` i `/` wbudowanymi w język Python, co jest intuicyjne dla nowych użytkowników biblioteki. Jednak aby obsłużyć funkcje ściśle związane z analizą danych, nieobsługiwane przez sam język Python, oraz umożliwić łączenie za pomocą metody `.pipe` (co omawiamy w dalszej części rozdziału), pandas oferuje również metody `pd.Series.add`, `pd.Series.sub`, `pd.Series.mul` i `pd.Series.div`.

Twórcy biblioteki pandas dołożyli starań, aby zachować spójność API we wszystkich strukturach danych. Dzięki temu wiedzę zdobytą w tym podrozdziale można łatwo wykorzystać w pracy ze strukturą `pd.DataFrame`. Jedyna różnica polega na tym, że obiekt `pd.Series` jest jednowymiarowy, a `pd.DataFrame` jest dwuwymiarowy.

Jak to zrobić?

Tworzymy prosty obiekt `pd.Series` z wykorzystaniem wywołania `range()` Pythona:

```
ser = pd.Series(range(3), dtype=pd.Int64Dtype())
ser
0    0
1    1
2    2
dtype: Int64
```

Aby ustalić terminologię, rozważmy wyrażenie takie jak `a + b`. W takim wyrażeniu używamy **operatora dwuargumentowego** (`+`). Określenie „dwuargumentowy" odnosi się do faktu, że musimy dodać do siebie dwa elementy, aby to wyrażenie miało sens — postać typu `a +` byłaby bez sensu. Te dwa *elementy* są nazywane *operandami*. Zatem w przypadku wyrażenia `a + b` mamy lewy operand `a` i prawy operand `b`.

Gdy jeden z operandów jest obiektem `pd.Series`, najprostsze wyrażenie algorytmiczne w bibliotece pandas obejmowałoby sytuację, w której drugi operand jest **skalarem**, czyli pojedynczą wartością. W takim przypadku wartość skalarna jest *rozgłaszana* do każdego elementu `pd.Series` w celu zastosowania algorytmu.

Na przykład jeśli chcielibyśmy dodać liczbę `42` do każdego elementu obiektu `pd.Series`, moglibyśmy to wyrazić po prostu jako:

```
ser + 42
0    42
1    43
2    44
dtype: Int64
```

Pandas potrafi zastosować operację dodawania do obiektów `pd.Series` w sposób *zwektoryzowany* (tzn. liczba `42` jest dodawana do wszystkich wartości jednocześnie, bez konieczności używania pętli `for` w Pythonie).

Odejmowanie można wykonać w podobnie naturalny sposób, używając odpowiedniego operatora:

```
ser - 42
0    -42
1    -41
2    -40
dtype: Int64
```

Podobnie mnożenie można wykonać za pomocą operatora *:

```
ser * 2
0    0
1    2
2    4
dtype: Int64
```

Jak zapewne się już domyślasz, do dzielenia służy operator /:

```
ser / 2
0    0.0
1    0.5
2    1.0
dtype: Float64
```

Równie poprawne jest użycie dwóch argumentów typu `pd.Series`:

```
ser2 = pd.Series(range(10, 13), dtype=pd.Int64Dtype())
ser + ser2
0    10
1    12
2    14
dtype: Int64
```

Jak wspomnieliśmy na początku rozdziału, chociaż wbudowane operatory Pythona są powszechnie stosowane i sprawdzają się w większości przypadków, pandas oferuje metody przeznaczone dla operacji na seriach danych: `pd.Series.add` (dodawanie), `pd.Series.sub` (odejmowanie), `pd.Series.mul` (mnożenie) i `pd.Series.div` (dzielenie):

```
ser1 = pd.Series([1., 2., 3.], dtype=pd.Float64Dtype())
ser2 = pd.Series([4., pd.NA, 6.], dtype=pd.Float64Dtype())
ser1.add(ser2)
0     5.0
1    <NA>
2     9.0
dtype: Float64
```

Zaletą metody `pd.Series.add`, w porównaniu do wbudowanego operatora, jest możliwość użycia opcjonalnego argumentu `fill_value=`, który pozwala na obsługę brakujących danych:

```
ser1.add(ser2, fill_value=0.)
0    5.0
1    2.0
2    9.0
dtype: Float64
```

W dalszej części tego rozdziału poznasz również łączenie operacji za pomocą metody `.pipe`, która najlepiej współpracuje z metodami biblioteki pandas, a nie z wbudowanymi operatorami Pythona.

To jeszcze nie wszystko…

Gdy oba argumenty w wyrażeniu są obiektami `pd.Series`, należy pamiętać, że pandas dopasuje je według etykiet wierszy. To zachowanie jest uważane za funkcjonalność, ale może zaskoczyć nowych użytkowników.

Aby zrozumieć, dlaczego to jest istotne, zacznijmy od dwóch obiektów `pd.Series` z identycznym indeksem wierszy. Gdy próbujemy je dodać, otrzymujemy dość przewidywalny wynik:

```
ser1 = pd.Series(range(3), dtype=pd.Int64Dtype())
ser2 = pd.Series(range(3), dtype=pd.Int64Dtype())
ser1 + ser2
0    0
1    2
2    4
dtype: Int64
```

Co jednak się dzieje, gdy wartości indeksu wierszy nie są identyczne? Prosty przypadek może obejmować dodawanie dwóch obiektów `pd.Series`, gdzie indeks wierszy jednego z nich jest podzbiorem drugiego. Można to zaobserwować na przykładzie utworzonego tutaj obiektu `ser3`, który ma tylko dwie wartości i używa domyślnego egzemplarza `pd.RangeIndex` z wartościami `[0, 1]`. Po dodaniu do `ser1` nadal otrzymujemy `pd.Series` z trzema elementami, ale wartości są dodawane tylko tam, gdzie etykiety indeksu wierszy można dopasować z obu obiektów `pd.Series`:

```
ser3 = pd.Series([2, 4], dtype=pd.Int64Dtype())
ser1 + ser3
0       2
1       5
2    <NA>
dtype: Int64
```

Teraz przyjrzyjmy się, co się dzieje, gdy dodajemy dwa obiekty `pd.Series` o tej samej długości, ale o różnych wartościach indeksu wierszy:

```
ser4 = pd.Series([2, 4, 8], index=[1, 2, 3], dtype=pd.Int64Dtype())
ser1 + ser4
0    <NA>
1       3
2       6
3    <NA>
dtype: Int64
```

Rozważmy jeszcze bardziej skrajny przypadek, w którym jedna z serii `pd.Series` ma indeks wierszy zawierający powtarzające się wartości:

```
ser5 = pd.Series([2, 4, 8], index=[0, 1, 1], dtype=pd.Int64Dtype())
ser1 + ser5
0       2
1       5
1       9
2    <NA>
dtype: Int64
```

Jeśli masz doświadczenie w pracy z SQL, zachowanie biblioteki pandas w tym przypadku przypomina zapytanie `FULL OUTER JOIN` w bazie danych. Każda etykieta z indeksu wiersza zostaje uwzględniona w wyniku, a pandas dopasowuje etykiety, które występują w obu obiektach typu `pd.Series`. Można to bezpośrednio odwzorować w bazie danych, takiej jak PostgreSQL:

```
WITH ser1 AS (
  SELECT * FROM (
    VALUES
      (0, 0),
      (1, 1),
      (2, 2)
   ) AS t(index, val1)
),

ser5 AS (
  SELECT * FROM (
    VALUES
      (0, 2),
      (1, 4),
      (1, 8)
   ) AS t(index, val2)
)

SELECT * FROM ser1 FULL OUTER JOIN ser5 USING(index);
```

Po uruchomieniu tego fragmentu kodu w PostgreSQL otrzymasz następujący wynik:

```
index | val1 | val2
------+------+------
    0 |    0 |    2
    1 |    1 |    8
    1 |    1 |    4
    2 |    2 |
(4 rows)
```

Pomijając różnice w kolejności, można zauważyć, że baza danych zwraca nam wszystkie unikalne wartości indeksów z wariantów `[0, 1, 2]` i `[0, 1, 1]`, wraz z powiązanymi wartościami `val1` i `val2`. Mimo że `ser1` miał tylko jedną wartość indeksu równą `1`, ta sama wartość pojawiła się dwukrotnie w kolumnie indeksu w `ser5`. W efekcie zapytanie `FULL OUTER JOIN` pokazuje obie wartości `val2` z `ser5` (4 i 8), jednocześnie powielając wartość `val1` pochodzącą z `ser1` (1).

Gdybyśmy następnie zsumowali `val1` i `val2` w bazie danych, otrzymalibyśmy wynik odpowiadający sumie `ser1 + ser5`, z tą różnicą, że baza danych może wybrać inną kolejność dla wygenerowanych danych wyjściowych:

```
WITH ser1 AS (
  SELECT * FROM (
    VALUES
      (0, 0),
      (1, 1),
      (2, 2)
   ) AS t(index, val1)
),

ser5 AS (
  SELECT * FROM (
    VALUES
      (0, 2),
      (1, 4),
      (1, 8)
   ) AS t(index, val2)
)

SELECT index, val1 + val2 AS value FROM ser1 FULL OUTER JOIN ser5 USING(
↪index);
index | value
------+-------
    0 |     2
    1 |     9
    1 |     5
    2 |
(4 rows)
```

Podstawowe operacje arytmetyczne na obiektach pd.DataFrame

Po omówieniu podstawowych operacji arytmetycznych na obiektach `pd.Series` wiesz już, że analogiczne operacje na obiektach `pd.DataFrame` są praktycznie identyczne. Jedyną różnicą jest to, że nasze algorytmy działają teraz w dwóch wymiarach zamiast w jednym. Dzięki temu interfejs pandas ułatwia interpretację danych niezależnie od ich kształtu, bez konieczności tworzenia pętli przeznaczonych do pracy z danymi. Znacząco redukuje to nakład pracy programisty i pomaga tworzyć szybszy kod — co jest korzystne dla wszystkich programistów.

Jak to zrobić?

Tworzymy małą ramkę danych (obiekt `pd.DataFrame`) o wymiarach 3 na 3 z wykorzystaniem losowo wybranych liczb:

```
np.random.seed(42)
df = pd.DataFrame(
    np.random.randn(3, 3),
    columns=["col1", "col2", "col3"],
    index=["row1", "row2", "row3"],
).convert_dtypes(dtype_backend="numpy_nullable")
df
```

	col1	col2	col3
row1	0.496714	-0.138264	0.647689
row2	1.52303	-0.234153	-0.234137
row3	1.579213	0.767435	-0.469474

Obiekt `pd.DataFrame`, podobnie jak `pd.Series`, obsługuje wbudowane operatory dwuargumentowe z argumentem skalarnym. Oto prosty przykład operacji dodawania:

```
df + 1
```

	col1	col2	col3
row1	1.496714	0.861736	1.647689
row2	2.52303	0.765847	0.765863
row3	2.579213	1.767435	0.530526

A to prosty przykład operacji mnożenia:

```
df * 2
```

	col1	col2	col3
row1	0.993428	-0.276529	1.295377
row2	3.04606	-0.468307	-0.468274
row3	3.158426	1.534869	-0.938949

Możesz również wykonywać operacje arytmetyczne na obiektach `pd.Series`. Domyślnie każda etykieta wiersza obiektu `pd.Series` jest wyszukiwana i dopasowywana do kolumn obiektu `pd.DataFrame`. Aby to zilustrować, tworzymy mały obiekt `pd.Series`, którego etykiety indeksu odpowiadają etykietom kolumn `df`:

```
ser = pd.Series(
    [20, 10, 0],
    index=["col1", "col2", "col3"],
    dtype=pd.Int64Dtype(),
)
ser
```

```
col1    20
col2    10
col3     0
dtype: Int64
```

Jeżeli dodasz te dane do naszej ramki danych (obiekt `pd.DataFrame`), to wartość z `col1` w `pd.Series` zostanie dodana do każdego elementu w `col1` ramki danych, powtarzając się dla każdego indeksu:

```
df + ser
        col1         col2        col3
row1    20.496714    9.861736    0.647689
row2    21.52303     9.765847   -0.234137
row3    21.579213    10.767435  -0.469474
```

W przypadkach gdy etykiety wierszy serii nie odpowiadają etykietom kolumn ramki danych, może się okazać, że brakuje pewnych danych:

```
ser = pd.Series(
    [20, 10, 0, 42],
    index=["col1", "col2", "col3", "new_column"],
    dtype=pd.Int64Dtype(),
)
ser + df
        col1         col2        col3        new_column
row1    20.496714    9.861736    0.647689    NaN
row2    21.52303     9.765847    -0.234137   NaN
row3    21.579213    10.767435   -0.469474   NaN
```

Jeśli chcesz kontrolować sposób wyrównywania obiektów `pd.Series` i `pd.DataFrame`, możesz użyć parametru `axis=` w metodach takich jak `pd.DataFrame.add`, `pd.DataFrame.sub`, `pd.DataFrame.mul` i `pd.DataFrame.div`.

Zobaczmy to w działaniu. Utwórz nowy obiekt `pd.Series` z etykietami wierszy, które lepiej pasują do etykiet wierszy naszego obiektu `pd.DataFrame`:

```
ser = pd.Series(
    [20, 10, 0, 42],
    index=["row1", "row2", "row3", "row4"],
    dtype=pd.Int64Dtype(),
)
ser
row1    20
row2    10
row3     0
row4    42
dtype: Int64
```

Wywołanie `df.add(ser, axis=0)` spowoduje dopasowanie etykiet wierszy zarówno dla obiektów `pd.Series`, jak i `pd.DataFrame`.

```
df.add(ser, axis=0)
        col1         col2        col3
row1    20.496714    19.861736   20.647689
row2    11.52303     9.765847    9.765863
row3    1.579213     0.767435   -0.469474
row4    <NA>         <NA>        <NA>
```

Możesz również używać dwóch argumentów typu `pd.DataFrame` jako operandów dodawania, odejmowania, mnożenia i dzielenia. Oto jak przeprowadzić operację mnożenia dwóch obiektów `pd.DataFrame`:

```
df * df
        col1       col2       col3
row1    0.246725   0.019117   0.4195
row2    2.31962    0.054828   0.05482
row3    2.493913   0.588956   0.220406
```

Oczywiście, wykonując te operacje, musisz pamiętać o zasadach wyrównywania indeksów — elementy są zawsze wyrównywane według etykiet, a nie według pozycji!

Tworzymy nowy obiekt `pd.DataFrame` o wymiarach 3 na 3 z innymi etykietami wierszy i kolumn, aby zademonstrować tę koncepcję:

```
np.random.seed(42)
df2 = pd.DataFrame(np.random.randn(3, 3))
df2 = df2.convert_dtypes(dtype_backend="numpy_nullable")
df2
     0           1           2
0    0.496714    -0.138264    0.647689
1    1.52303     -0.234153   -0.234137
2    1.579213     0.767435   -0.469474
```

Próba dodania tego do poprzedniego obiektu `pd.DataFrame` spowoduje utworzenie indeksu wierszy z etykietami `["row1", "row2", "row3", 0, 1, 2]` oraz indeksu kolumn z etykietami `["col1", "col2", "col3", 0, 1, 2]`. Ponieważ żadne etykiety nie mogły zostać wyrównane, wszystkie wartości zostaną oznaczone jako brakujące:

```
df + df2
        col1    col2    col3    0       1       2
row1    <NA>    <NA>    <NA>    <NA>    <NA>    <NA>
row2    <NA>    <NA>    <NA>    <NA>    <NA>    <NA>
row3    <NA>    <NA>    <NA>    <NA>    <NA>    <NA>
0       <NA>    <NA>    <NA>    <NA>    <NA>    <NA>
1       <NA>    <NA>    <NA>    <NA>    <NA>    <NA>
2       <NA>    <NA>    <NA>    <NA>    <NA>    <NA>
```

Agregacje

Agregacje (nazywane także *redukcjami*) pomagają zmniejszyć wiele wartości z serii do pojedynczej wartości. Nawet jeśli to pojęcie techniczne jest dla Ciebie nowe, z pewnością spotkałeś się już z wieloma agregacjami w swojej pracy z danymi. Takie rzeczy jak *liczba* rekordów, *suma* sprzedaży czy *średnia* cena to bardzo powszechne przykłady agregacji.

W tym rozdziale przyjrzymy się wielu agregacjom wbudowanym w bibliotekę pandas i jednocześnie postaramy się wyjaśnić, jak są one stosowane. Większość analiz, które będziesz przeprowadzać w swojej pracy z danymi, polega na przetwarzaniu dużych zbiorów i agregowaniu zawartych w nich wartości do wyników, które Twoi odbiorcy mogą łatwo przyswoić. Kierownictwo większości firm nie jest zainteresowane otrzymywaniem surowych danych transakcyjnych — chce po prostu znać sumy, wartości minimalne i maksymalne, średnie itp. tych transakcji. Dlatego skuteczne wykorzystanie

i zastosowanie agregacji jest kluczowym elementem przekształcania złożonych procesów przetwarzania danych w proste wyniki, które inni mogą wykorzystać i na ich podstawie podejmować decyzje.

Jak to zrobić?

Wiele podstawowych operacji agregujących jest zaimplementowanych bezpośrednio jako metody obiektu `pd.Series`, co sprawia, że obliczanie często potrzebnych wartości, takich jak liczba elementów, suma czy maksimum, jest niezwykle proste.

Aby rozpocząć ten przykład, ponownie utworzymy obiekt `pd.Series` zawierający losowo wybrane liczby:

```
np.random.seed(42)
ser = pd.Series(np.random.rand(10_000), dtype=pd.Float64Dtype())
```

Biblioteka pandas udostępnia metody dla wielu powszechnie stosowanych agregacji, takich jak `pd.Series.count` (zliczanie), `pd.Series.mean` (średnia), `pd.Series.std` (odchylenie standardowe), `pd.Series.min` (minimum), `pd.Series.max` (maksimum) oraz `pd.Series.sum` (suma):

```
print(f"Count is: {ser.count()}")
print(f"Mean value is: {ser.mean()}")
print(f"Standard deviation is: {ser.std()}")
print(f"Minimum value is: {ser.min()}")
print(f"Maximum value is: {ser.max()}")
print(f"Summation is: {ser.sum()}")
Count is: 10000
Mean value is: 0.49415955768429964
Standard deviation is: 0.2876301265269928
Minimum value is: 1.1634755366141114e-05
Maximum value is: 0.9997176732861306
Summation is: 4941.595576842997
```

Zamiast bezpośredniego wywoływania tych metod bardziej uniwersalnym sposobem wykonywania agregacji jest użycie `pd.Series.agg`. Wystarczy podać nazwę żądanej agregacji jako ciąg tekstowy:

```
print(f"Count is: {ser.agg('count')}")
print(f"Mean value is: {ser.agg('mean')}")
print(f"Standard deviation is: {ser.agg('std')}")
print(f"Minimum value is: {ser.agg('min')}")
print(f"Maximum value is: {ser.agg('max')}")
print(f"Summation is: {ser.agg('sum')}")
Count is: 10000
Mean value is: 0.49415955768429964
Standard deviation is: 0.2876301265269928
Minimum value is: 1.1634755366141114e-05
Maximum value is: 0.9997176732861306
Summation is: 4941.595576842997
```

Zaletą korzystania z `pd.Series.agg` jest możliwość wykonywania wielu agregacji jednocześnie. Na przykład jeśli chcesz obliczyć wartość minimalną i maksymalną dla danego pola w jednym kroku, możesz to zrobić, przekazując listę do `pd.Series.agg`:

```
ser.agg(["min", "max"])
min    0.000012
max    0.999718
dtype: float64
```

Agregacja obiektu `pd.Series` jest prosta, ponieważ istnieje tylko jeden wymiar do zagregowania. W przypadku `pd.DataFrame` mamy dwa możliwe wymiary do agregacji, co daje użytkownikowi biblioteki więcej opcji do rozważenia.

Aby to zobrazować, tworzymy obiekt `pd.DataFrame` z losowo wybranymi liczbami:

```
np.random.seed(42)
df = pd.DataFrame(
    np.random.randn(10_000, 6),
    columns=list("abcdef"),
).convert_dtypes(dtype_backend="numpy_nullable")
df
```

	a	b	c	d	e	f
0	0.496714	-0.138264	0.647689	1.523030	-0.234153	-0.234137
1	1.579213	0.767435	-0.469474	0.542560	-0.463418	-0.465730
2	0.241962	-1.913280	-1.724918	-0.562288	-1.012831	0.314247
3	-0.908024	-1.412304	1.465649	-0.225776	0.067528	-1.424748
4	-0.544383	0.110923	-1.150994	0.375698	-0.600639	-0.291694
…	…	…	…	…	…	…
9995	1.951254	0.324704	1.937021	-0.125083	0.589664	0.869128
9996	0.624062	-0.317340	-1.636983	2.390878	-0.597118	2.670553
9997	-0.470192	1.511932	0.718306	0.764051	-0.495094	-0.273401
9998	-0.259206	0.274769	-0.084735	-0.406717	-0.815527	-0.716988
9999	0.533743	-0.701856	-1.099044	0.141010	-2.181973	-0.006398

10000 rows × 6 columns

Domyślnie wywołanie agregacji za pomocą wbudowanej metody, takiej jak `pd.Data`
`↪Frame.sum`, zostanie *zastosowane wzdłuż kolumn*, co oznacza, że każda kolumna jest agregowana indywidualnie. Następnie pandas wyświetli wynik agregacji każdej kolumny jako wpis w obiekcie `pd.Series`:

```
df.sum()
a    -21.365908
b     -7.963987
c    152.032992
d   -180.727498
e     29.399311
f     25.042078
dtype: Float64
```

Jeśli chcesz agregować dane w poszczególnych wierszach, możesz użyć argumentu `axis=1`. Jednak należy pamiętać, że pandas jest znacznie bardziej zoptymalizowana dla operacji w przypadku użycia argumentu `axis=0`. Oznacza to, że agregacja wierszy może być *znacznie wolniejsza* niż agregacja kolumn. Mimo to jest to dość unikalna funkcja, która może być przydatna, gdy wydajność nie jest głównym priorytetem.

```
df.sum(axis=1)
0       2.060878
1       1.490586
2      -4.657107
3      -2.437675
4      -2.101088
         ...
9995    5.54669
9996    3.134053
9997    1.755601
9998   -2.008404
9999   -3.314518
Length: 10000, dtype: Float64
```

Podobnie jak `pd.Series` obiekt `pd.DataFrame` ma metodę `.agg`, która umożliwia zastosowanie wielu agregacji jednocześnie:

```
df.agg(["min", "max"])
          a         b         c         d         e         f
min -4.295391 -3.436062 -3.922400 -4.465604 -3.836656 -4.157734
max  3.602415  3.745379  3.727833  4.479084  3.691625  3.942331
```

To jeszcze nie wszystko…

W przykładach omówionych w poprzednim punkcie przekazywaliśmy funkcje jako ciągi tekstowe, takie jak `min` i `max`, do metody `.agg`. Jest to dobre rozwiązanie dla prostych funkcji, ale w bardziej złożonych przypadkach możesz również przekazywać argumenty wywoływalne. Każdy taki argument powinien przyjmować pojedynczy obiekt `pd.Series` i redukować go do wartości skalarnej.

```
def mean_and_add_42(ser: pd.Series):
    return ser.mean() + 42

def mean_and_sub_42(ser: pd.Series):
    return ser.mean() 42

np.random.seed(42)
ser = pd.Series(np.random.rand(10_000), dtype=pd.Float64Dtype())
ser.agg([mean_and_add_42, mean_and_sub_42])
mean_and_add_42    42.49416
mean_and_sub_42   -41.50584
dtype: float64
```

Transformacje

W przeciwieństwie do agregacji transformacje nie redukują zbioru wartości do pojedynczej wartości, lecz zachowują strukturę obiektu, na którym są wykonywane. Ten konkretny przepis może wydawać się dość prozaiczny po omówieniu agregacji w poprzednim

podrozdziale, jednak transformacje i agregacje okażą się bardzo komplementarnymi narzędziami przy obliczaniu takich wartości jak procent sumy w grupie, co przedstawiamy w dalszej części książki.

Jak to zrobić?

Tworzymy mały obiekt typu `pd.Series`:

```
ser = pd.Series([-1, 0, 1], dtype=pd.Int64Dtype())
```

Metoda `pd.Series.transform`, podobnie jak `pd.Series.agg`, może przyjmować listę funkcji do zastosowania. Jednak podczas gdy `pd.Series.agg` oczekiwała, że te funkcje zwrócą pojedynczą wartość, `pd.Series.transform` wymaga, aby funkcje zwracały obiekt `pd.Series` o takim samym indeksie i kształcie.

```
def adds_one(ser: pd.Series) -> pd.Series:
    return ser + 1

ser.transform(["abs", adds_one])
```

	abs	adds_one
0	1	0
1	0	1
2	1	2

Analogicznie do tego, jak metoda `pd.DataFrame.agg` domyślnie *agregowała* każdą kolumnę, `pd.DataFrame.transform` domyślnie *transformuje* poszczególne kolumny. Aby zobaczyć to w działaniu, utworzymy mały obiekt `pd.DataFrame`:

```
df = pd.DataFrame(
    np.arange(-5, 4, 1).reshape(3, -1)
).convert_dtypes(dtype_backend="numpy_nullable")
df
```

	0	1	2
0	-5	-4	-3
1	-2	-1	0
2	1	2	3

Pomijając szczegóły implementacyjne, wywołanie w rodzaju `df.transform("abs")` spowoduje zastosowanie funkcji wartości bezwzględnej oddzielnie dla każdej kolumny, a następnie złożenie wyniku z powrotem na postać obiektu `pd.DataFrame`:

```
df.transform("abs")
```

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

Jeśli metodzie `pd.DataFrame.transform` przekażesz wiele funkcji transformujących, otrzymasz w wyniku obiekt `pd.MultiIndex`:

```
def add_42(ser: pd.Series):
    return ser + 42
```

```
df.transform(["abs", add_42])
    0       1       2
    abs  add_42  abs  add_42  abs  add_42
0   5      37    4      38    3      39
1   2      40    1      41    0      42
2   1      43    2      44    3      45
```

To jeszcze nie wszystko…

Jak wspomnieliśmy na początku receptury, transformacje i agregacje mogą naturalnie współpracować z koncepcją `GroupBy`, którą omówimy dokładnie w rozdziale 8. W szczególności receptura dotycząca *podstaw grupowania* będzie pomocna w porównaniu agregacji z transformacjami. Pokaże ona również, jak transformacje można wykorzystać na potrzeby zwięzłego i ekspresyjnego obliczania wartości procentowych w grupach.

Mapowanie

Metody `.agg` i `.transform`, które dotychczas omówiliśmy, działają jednocześnie na całej *sekwencji* wartości. Zasadniczo w bibliotece pandas jest to korzystne, ponieważ umożliwia wykonywanie zoptymalizowanych operacji *wektorowych*, które są szybkie i wydajne obliczeniowo.

Niemniej jednak czasami jako użytkownik końcowy możesz zdecydować, że chcesz poświęcić wydajność na rzecz większej elastyczności lub dokładniejszej kontroli. W takich sytuacjach przydatna może okazać się metoda `.map`. Pozwala ona na zastosowanie funkcji indywidualnie do każdego elementu obiektu biblioteki pandas.

Jak to zrobić?

Załóżmy, że mamy obiekt `pd.Series` zawierający zarówno liczby, jak i listy liczb:

```
ser = pd.Series([123.45, [100, 113], 142.0, [110, 113, 119]])
ser
0             123.45
1         [100, 113]
2              142.0
3    [110, 113, 119]
dtype: object
```

Metody `.agg` i `.transform` nie są tu odpowiednie, ponieważ nie mamy do czynienia z jednolitym typem danych — musimy dokładnie przeanalizować poszczególne elementy, aby zdecydować, jak je obsłużyć.

Na potrzeby analizy załóżmy, że gdy natrafimy na liczbę, chcemy po prostu zwrócić jej wartość. Jeśli napotkamy listę wartości, chcemy obliczyć średnią wszystkich wartości z tej listy i ją zwrócić. Funkcja implementująca tę logikę wyglądałaby następująco:

```
def custom_average(value):
    if isinstance(value, list):
        return sum(value) / len(value)

    return value
```

Następnie możemy zastosować to do każdego elementu naszej serii `pd.Series`, używając metody `pd.Series.map`:

```
ser.map(custom_average)
0    123.45
1    106.50
2    142.00
3    114.00
dtype: float64
```

Gdybyśmy mieli ramkę danych (obiekt `pd.DataFrame`) zawierającą tego typu informacje, moglibyśmy z powodzeniem zastosować tę funkcję za pomocą metody `.map`:

```
df = pd.DataFrame([
    [2., [1, 2], 3.],
    [[4, 5], 5, 7.],
    [1, 4, [1, 1, 5.5]],
])
df
          0         1              2
0       2.0    [1, 2]            3.0
1    [4, 5]         5            7.0
2         1         4    [1, 1, 5.5]
df.map(custom_average)
       0     1     2
0    2.0   1.5   3.0
1    4.5   5.0   7.0
2    1.0   4.0   2.5
```

To jeszcze nie wszystko...

W powyższym przykładzie zamiast używać wywołania `pd.Series.map`, można było również zastosować `pd.Series.transform`:

```
ser.transform(custom_average)
0    123.45
1    106.50
2    142.00
3    114.00
dtype: float64
```

Jednakże *nie* uzyskasz takich samych wyników, używając metody `pd.DataFrame.transform`:

```
df.transform(custom_average)
          0         1              2
0       2.0    [1, 2]            3.0
1    [4, 5]         5            7.0
2         1         4    [1, 1, 5.5]
```

Dlaczego tak się dzieje? Pamiętaj, że metoda `.map` jawnie stosuje funkcję do każdego elementu, niezależnie od tego, czy pracujemy z obiektem `pd.Series`, czy `pd.DataFrame`. Metoda `pd.Series.transform` również chętnie aplikuje funkcję do każdego elementu serii, natomiast `pd.DataFrame.transform` w zasadzie iteruje po każdej kolumnie i przekazuje tę kolumnę jako argument do wywoływanych funkcji.

Wynika to z tego, że nasza funkcja jest zaimplementowana następująco:

```
def custom_average(value):
    if isinstance(value, list):
        return sum(value) / len(value)

    return value
```

Wywołanie `isinstance(value, list)` nie działa poprawnie dla obiektu `pd.Series`, w efekcie zwraca po prostu ten sam obiekt `pd.Series`. Jeśli nieco zmodyfikujemy funkcję:

```
def custom_average(value):
    if isinstance(value, (pd.Series, pd.DataFrame)):
        raise TypeError("Received a pandas object expected a single value!")
    if isinstance(value, list):
        return sum(value) / len(value)

    return value
```

to sposób działania metody `transform` obiektu `pd.DataFrame` staje się bardziej zrozumiały:

```
df.transform(custom_average)
TypeError: Received a pandas object expected a single value!
```

Chociaż może występować pewne podobieństwo pod względem koncepcji, ogólnie rzecz biorąc, w kodzie należy traktować metodę `.map` jako działającą na pojedynczych elementach, podczas gdy metody `.agg` i `.transform` w miarę możliwości pracują na większych sekwencjach danych jednocześnie.

Stosowanie funkcji

Metoda `.apply` jest powszechnie stosowana, do tego stopnia, że można twierdzić, iż jest *nadużywana*. Metody `.agg`, `.transform` i `.map`, które omówiliśmy wcześniej, mają stosunkowo jasne zastosowania (`.agg` redukuje dane, `.transform` zachowuje kształt, `.map` stosuje funkcje do poszczególnych elementów), ale gdy sięgasz po `.apply`, możesz odtworzyć działanie każdej z nich. Ta elastyczność może wydawać się atrakcyjna na początku, ale ponieważ `.apply` pozostawia bibliotece pandas decyzję o tym, *co zrobić*, zazwyczaj lepiej jest wybrać bardziej jednoznaczne metody, aby uniknąć niespodzianek.

Mimo to często spotkasz się z kodem wykorzystującym `.apply`, szczególnie utworzonym przez osoby, które nie przeczytały tej książki. Dlatego zrozumienie, jak działa ta metoda i jakie są jej ograniczenia, może okazać się niezwykle cenne.

Jak to zrobić?

Wywołanie `pd.Series.apply` spowoduje, że metoda `.apply` będzie działać podobnie jak `.map` (tzn. funkcja zostanie zastosowana do każdego elementu obiektu `pd.Series`).

Przyjrzyjmy się nieco sztucznemu przykładowi funkcji, która wyświetla poszczególne elementy:

```
def debug_apply(value):
    print(f"Apply was called with value:\n{value}")
```

Przekazanie tego przez metodę `.apply`:

```
ser = pd.Series(range(3), dtype=pd.Int64Dtype())
ser.apply(debug_apply)
Apply was called with value:
0
Apply was called with value:
1
Apply was called with value:
2
0    None
1    None
2    None
dtype: object
```

daje dokładnie taki sam sposób działania jak w przypadku `pd.Series.map`:

```
ser.map(debug_apply)
Apply was called with value:
0
Apply was called with value:
1
Apply was called with value:
2
0    None
1    None
2    None
dtype: object
```

Metoda `pd.Series.apply` działa podobnie jak pętla w Pythonie i wywołuje funkcję dla każdego elementu. Ponieważ nasza funkcja nic nie zwraca, wynikowy obiekt `pd.Series` to tablica wartości `None` z zachowanymi indeksami.

Podczas gdy `pd.Series.apply` działa na poziomie elementów, `pd.DataFrame.apply` działa na każdej kolumnie jak `pd.Series`. Zobaczmy to w działaniu na przykładzie obiektu `pd.DataFrame` o kształcie `(3, 2)`:

```
df = pd.DataFrame(
    np.arange(6).reshape(3, -1),
    columns=list("ab"),
).convert_dtypes(dtype_backend="numpy_nullable")
df
```

```
      a     b
0     0     1
1     2     3
2     4     5
df.apply(debug_apply)
Apply was called with value:
0    0
1    2
2    4
Name: a, dtype: Int64
Apply was called with value:
0    1
1    3
2    5
Name: b, dtype: Int64
a    None
b    None
dtype: object
```

Jak widać w wygenerowanych danych wyjściowych, funkcja została wywołana tylko dwa razy, co odpowiada dwóm kolumnom danych. Natomiast w przypadku obiektu `pd.Series`, który miał trzy wiersze, została zastosowana trzy razy.

Poza liczbą faktycznych wywołań funkcji przez `pd.DataFrame.apply` kształt zwracanej wartości będzie zależny również od użycia metod `.agg` i `.transform`. Omawiany tutaj przykład jest bliższy wynikowi wywołania `.agg`, ponieważ zwraca pojedynczą wartość `None`, ale gdybyśmy zwrócili wyświetlany element, otrzymalibyśmy zachowanie bardziej podobne do działania metody `.transform`:

```
def debug_apply_and_return(value):
    print(value)
    return value
df.apply(debug_apply_and_return)
0    0
1    2
2    4
Name: a, dtype: Int64
0    1
1    3
2    5
Name: b, dtype: Int64
      a    b
0     0    1
1     2    3
2     4    5
```

Jeśli uważasz to za mylące, nie jesteś w tym odosobniony. Poleganie na tym, że pandas *podejmie właściwe działanie* przy użyciu metody `.apply`, może być ryzykowne. Zdecydowanie zalecamy, aby przed sięgnięciem po `.apply` wyczerpać wszystkie możliwości metod `.agg`, `.transform` lub `.map`.

Podsumowujące dane statystyczne

Podsumowujące dane statystyczne umożliwiają szybkie zrozumienie podstawowych właściwości i rozkładu danych. W tej recepturze przedstawimy dwie potężne metody biblioteki pandas: `value_counts` i `describe`, które mogą służyć jako użyteczny punkt wyjścia do dalszej analizy danych.

Jak to zrobić?

Metoda `pd.Series.value_counts` przypisuje liczebności do każdej unikalnej wartości danych, co ułatwia sprawdzenie, jak często one występują. Jest to szczególnie przydatne w przypadku danych dyskretnych:

```
ser = pd.Series(["a", "b", "c", "a", "c", "a"], dtype=pd.StringDtype())
ser.value_counts()
a    3
c    2
b    1
Name: count, dtype: Int64
```

W przypadku danych ciągłych metoda `pd.Series.describe` to zestaw obliczeń statystycznych, które zostały zebrane w jednym wywołaniu. Korzystając z tej metody, można łatwo uzyskać informacje o liczebności, średniej, wartościach minimalnej i maksymalnej, a także na temat ogólnego rozkładu danych:

```
ser = pd.Series([0, 42, 84], dtype=pd.Int64Dtype())
ser.describe()
count     3.0
mean     42.0
std      42.0
min       0.0
25%      21.0
50%      42.0
75%      63.0
max      84.0
dtype: Float64
```

Domyślnie rozkład danych jest przedstawiany za pomocą kwartyli: 25%, 50%, 75% oraz wartości maksymalnej (100%). Jeśli analiza koncentruje się na konkretnej części rozkładu, można dostosować prezentowane wartości za pomocą argumentu `percentiles=`.

```
ser.describe(percentiles=[.10, . 44, .67])
count      3.0
mean      42.0
std       42.0
min        0.0
10%        8.4
44%      36.96
50%       42.0
67%      56.28
max       84.0
dtype: Float64
```

Algorytmy grupowania

Kategoryzacja to proces polegający na podziale zmiennej ciągłej na dyskretne przedziały. Jest to przydatne, gdy chcemy przekształcić potencjalnie nieskończoną liczbę wartości w skończoną liczbę „koszyków" do analizy.

Jak to zrobić?

Wyobraźmy sobie, że zebraliśmy dane ankietowe od użytkowników pewnego systemu. Jedno z pytań w ankiecie dotyczy wieku użytkowników, co daje nam dane wyglądające następująco:

```
df = pd.DataFrame([
    ["Jane", 34],
    ["John", 18],
    ["Jamie", 22],
    ["Jessica", 36],
    ["Jackie", 33],
    ["Steve", 40],
    ["Sam", 30],
    ["Stephanie", 66],
    ["Sarah", 55],
    ["Aaron", 22],
    ["Erin", 28],
    ["Elsa", 37],
], columns=["name", "age"])
df = df.convert_dtypes(dtype_backend="numpy_nullable")
df.head()
```

	name	age
0	Jane	34
1	John	18
2	Jamie	22
3	Jessica	36
4	Jackie	33

Zamiast traktować każdy wiek jako osobną liczbę, użyjemy funkcji `pd.cut`, aby przypisać poszczególne rekordy do odpowiedniej grupy wiekowej. Na początek przekażemy serię `pd.Series` oraz liczbę przedziałów, które chcemy utworzyć, jako argumenty funkcji:

```
pd.cut(df["age"], 4)
```

```
0       (30.0, 42.0]
1     (17.952, 30.0]
2     (17.952, 30.0]
3       (30.0, 42.0]
4       (30.0, 42.0]
5       (30.0, 42.0]
6     (17.952, 30.0]
7       (54.0, 66.0]
8       (54.0, 66.0]
9     (17.952, 30.0]
```

```
10    (17.952, 30.0]
11      (30.0, 42.0]
Name: age, dtype: category
Categories (4, interval[float64, right]): [(17.952, 30.0] < (30.0, 42.0] <
(42.0, 54.0] < (54.0, 66.0]]
```

To tworzy typ `pd.CategoricalDtype` z czterema oddzielnymi przedziałami: `(17.952, 30.0]`, `(30.0, 42.0]`, `(42.0, 54.0]` i `(54.0, 66.0]`. Pomijając nieoczekiwane miejsca dziesiętne w pierwszym przedziale, który zaczyna się od `17.952`, wszystkie te przedziały obejmują równy zakres 12 lat. Wynika to z faktu, że różnica między wartością maksymalną (66) i minimalną (18) daje łączną rozpiętość wieku 48 lat, która podzielona równo na 4 daje nam 12-letni zakres dla każdego przedziału.

Wiek `17.952`, który widzimy w pierwszym przedziale, może mieć sens wewnętrznie dla biblioteki pandas ze względu na wybrany algorytm określania przedziałów. Dla nas ostatecznie nie jest istotny, ponieważ wiemy, że operujemy na liczbach całkowitych. Na szczęście można to kontrolować za pomocą argumentu `precision=` i usunąć miejsca dziesiętne:

```
pd.cut(df["age"], 4, precision=0)
0     (30.0, 42.0]
1     (18.0, 30.0]
2     (18.0, 30.0]
3     (30.0, 42.0]
4     (30.0, 42.0]
5     (30.0, 42.0]
6     (18.0, 30.0]
7     (54.0, 66.0]
8     (54.0, 66.0]
9     (18.0, 30.0]
10    (18.0, 30.0]
11    (30.0, 42.0]
Name: age, dtype: category
Categories (4, interval[float64, right]): [(18.0, 30.0] < (30.0, 42.0] < (42.0,
54.0] < (54.0, 66.0]]
```

Funkcja `pd.cut` nie ogranicza nas do tworzenia koszyków o równej wielkości. Jeśli zamiast tego chcielibyśmy przypisać każdą osobę do przedziałów wiekowych co 10 lat, możemy podać te zakresy jako drugi argument funkcji:

```
pd.cut(df["age"], [10, 20, 30, 40, 50, 60, 70])
0     (30, 40]
1     (10, 20]
2     (20, 30]
3     (30, 40]
4     (30, 40]
5     (30, 40]
6     (20, 30]
7     (60, 70]
8     (50, 60]
9     (20, 30]
10    (20, 30]
```

```
11    (30, 40]
Name: age, dtype: category
Categories (6, interval[int64, right]): [(10, 20] < (20, 30] < (30, 40] < (40,
50] < (50, 60] < (60, 70]]
```

Jednakże takie podejście jest zbyt restrykcyjne, ponieważ nie uwzględnia użytkowników powyżej 70 roku życia. Aby to naprawić, możemy zmienić ostatnią granicę przedziału z `70` na `999`, traktując ją jako kategorię zbiorczą:

```
pd.cut(df["age"], [10, 20, 30, 40, 50, 60, 999])
0      (30, 40]
1      (10, 20]
2      (20, 30]
3      (30, 40]
4      (30, 40]
5      (30, 40]
6      (20, 30]
7     (60, 999]
8      (50, 60]
9      (20, 30]
10     (20, 30]
11     (30, 40]
Name: age, dtype: category
Categories (6, interval[int64, right]): [(10, 20] < (20, 30] < (30, 40] < (40,
50] < (50, 60] < (60, 999]]
```

Takie rozwiązanie z kolei spowodowało wygenerowanie etykiety `(60, 999)`, co pozostawia wiele do życzenia z punktu widzenia prezentacji. Jeśli nie jesteśmy zadowoleni z domyślnie tworzonych etykiet, możemy kontrolować ich wygląd za pomocą argumentu `labels=`:

```
pd.cut(
    df["age"],
    [10, 20, 30, 40, 50, 60, 999],
    labels=["10-20", "20-30", "30-40", "40-50", "50-60", "60+"],
)
0     30-40
1     10-20
2     20-30
3     30-40
4     30-40
5     30-40
6     20-30
7       60+
8     50-60
9     20-30
10    20-30
11    30-40
Name: age, dtype: category
Categories (6, object): ['10-20' < '20-30' < '30-40' < '40-50' < '50-60' <
'60+']
```

Jednak powyższe etykiety nie są *do końca* poprawne. Zwróć uwagę, że mamy przedziały `30-40` i `40-50`, ale co się stanie, jeśli ktoś ma dokładnie 40 lat? Do którego przedziału zostanie przypisany?

Na szczęście możemy to już zaobserwować w naszych danych na przykładzie Steve'a, który idealnie pasuje do tego kryterium. Jeśli przyjrzysz się domyślnemu przedziałowi, do którego został przypisany, zobaczysz, że wygląda on tak: `(30, 40]`.

```
df.assign(age_bin=lambda x: pd.cut(x["age"], [10, 20, 30, 40, 50, 60, 999]))
        name        age     age_bin
0       Jane        34      (30, 40]
1       John        18      (10, 20]
2       Jamie       22      (20, 30]
3       Jessica     36      (30, 40]
4       Jackie      33      (30, 40]
5       Steve       40      (30, 40]
6       Sam         30      (20, 30]
7       Stephanie   66     (60, 999]
8       Sarah       55      (50, 60]
9       Aaron       22      (20, 30]
10      Erin        28      (20, 30]
11      Elsa        37      (30, 40]
```

Domyślnie przedziały są *prawostronnie domknięte*, co oznacza, że każdy przedział można rozumieć jako *do i włącznie z* daną wartością. Jeśli chcielibyśmy mieć *do, ale nie włączając*, moglibyśmy to kontrolować za pomocą argumentu `right`:

```
df.assign(
    age_bin=lambda x: pd.cut(x["age"], [10, 20, 30, 40, 50, 60, 999],
right=False)
)
        name        age   age_bin
0       Jane        34    [30, 40)
1       John        18    [10, 20)
2       Jamie       22    [20, 30)
3       Jessica     36    [30, 40)
4       Jackie      33    [30, 40)
5       Steve       40    [40, 50)
6       Sam         30    [30, 40)
7       Stephanie   66    [60, 999)
8       Sarah       55    [50, 60)
9       Aaron       22    [20, 30)
10      Erin        28    [20, 30)
11      Elsa        37    [30, 40)
```

To zmieniło przedział dla Steve'a z `(30, 40]` na `[40, 50)`. W domyślnej reprezentacji tekstowej nawias kwadratowy oznacza, że dana wartość graniczna jest *włączona* do przedziału, natomiast nawias okrągły wskazuje na jej *wyłączenie*.

Kodowanie „1 z n” za pomocą funkcji pd.get_dummies

W zastosowaniach z zakresu analizy danych i uczenia maszynowego często spotyka się praktykę przekształcania danych kategorycznych na sekwencje wartości `0/1`. Jest to korzystne, ponieważ takie dane są łatwiejsze do interpretacji przez algorytmy numeryczne. Proces ten nazywany jest zwykle kodowaniem *jeden z n* (ang. *one-hot encoding*), a jego wyniki określa się mianem *zmiennych zastępczych*.

Jak to zrobić?

Zacznijmy od utworzenia małego obiektu serii (`pd.Series`), który będzie zawierał zbiór kilku kolorów:

```
ser = pd.Series([
    "green",
    "brown",
    "blue",
    "amber",
    "hazel",
    "amber",
    "green",
    "blue",
    "green",
], name="eye_colors", dtype=pd.StringDtype())
ser
```

```
0    green
1    brown
2     blue
3    amber
4    hazel
5    amber
6    green
7     blue
8    green
Name: eye_colors, dtype: string
```

Przekazanie tego jako argumentu do funkcji `pd.get_dummies` utworzy ramkę danych (obiekt `pd.DataFrame`) o tych samych indeksach, która będzie dla każdego koloru zawierała kolumnę typu logicznego. W każdym wierszu tylko jedna kolumna będzie miała wartość `True`, odpowiadającą oryginalnemu kolorowi, podczas gdy wszystkie pozostałe kolumny w tym samym wierszu będą miały wartość `False`:

```
pd.get_dummies(ser)
```

```
       amber   blue    brown   green   hazel
0      False   False   False   True    False
1      False   False   True    False   False
2      False   True    False   False   False
```

```
3       True    False   False   False   False
4       False   False   False   False   True
5       True    False   False   False   False
6       False   False   False   True    False
7       False   True    False   False   False
8       False   False   False   True    False
```

Jeśli domyślne nazwy kolumn nam nie odpowiadają, możemy je zmodyfikować, dodając prefiks. Powszechną praktyką w modelowaniu danych jest dodawanie prefiksu `is_` do kolumn typu logicznego (boolowskiego):

```
pd.get_dummies(ser, prefix="is")
        is_amber  is_blue  is_brown  is_green  is_hazel
0       False     False    False     True      False
1       False     False    True      False     False
2       False     True     False     False     False
3       True      False    False     False     False
4       False     False    False     False     True
5       True      False    False     False     False
6       False     False    False     True      False
7       False     True     False     False     False
8       False     False    False     True      False
```

Łączenie operacji za pomocą metody .pipe

Podczas tworzenia kodu wykorzystującego bibliotekę pandas programiści zazwyczaj stosują dwa główne style. Pierwsze podejście polega na swobodnym używaniu zmiennych w całym programie, co oznacza tworzenie nowych zmiennych:

```
df = pd.DataFrame(...)
df1 = do_something(df)
df2 = do_another_thing(df1)
df3 = do_yet_another_thing(df2)
```

lub po prostu wielokrotne przypisywanie wartości do tej samej zmiennej:

```
df = pd.DataFrame(...)
df = do_something(df)
df = do_another_thing(df)
df = do_yet_another_thing(df)
```

Alternatywnym podejściem jest wyrażenie kodu w formie *potoku*, w którym poszczególne kroki pobierają i zwracają obiekt `pd.DataFrame`:

```
(
    pd.DataFrame(...)
    .pipe(do_something)
    .pipe(do_another_thing)
    .pipe(do_yet_another_thing)
)
```

W podejściu opartym na zmiennych musisz tworzyć wiele zmiennych w swoim programie lub zmieniać stan obiektu `pd.DataFrame` w trakcie każdego przypisania. Z kolei podejście oparte na potoku nie tworzy nowych zmiennych ani nie zmienia stanu obiektu `pd.DataFrame`.

Wprawdzie podejście oparte na potoku teoretycznie mogłoby być lepiej obsługiwane przez optymalizator zapytań, jednak w chwili powstawania książki pandas nie oferuje takiej funkcjonalności i trudno przewidzieć, jak mogłoby to wyglądać w przyszłości. W związku z tym wybór między tymi dwoma podejściami nie ma prawie żadnego wpływu na wydajność i jest to tak naprawdę kwestia stylu.

Zachęcamy do zapoznania się z obiema metodami. Czasami może być łatwiej wyrazić swój kod jako potok, a innym razem może to być uciążliwe. Nie ma sztywnego wymogu stosowania jednego lub drugiego podejścia, więc możesz swobodnie mieszać style w swoim kodzie.

Jak to zrobić?

Zacznijmy od bardzo prostego obiektu `pd.DataFrame`. Na razie nie są istotne nazwy kolumn ani ich zawartość:

```
df = pd.DataFrame({
    "col1": pd.Series([1, 2, 3], dtype=pd.Int64Dtype()),
    "col2": pd.Series(["a", "b", "c"], dtype=pd.StringDtype()),
})
df
```

	col1	col2
0	1	a
1	2	b
2	3	c

Teraz tworzymy kilka przykładowych funkcji, które będą modyfikować zawartość kolumn. Te funkcje powinny przyjmować i zwracać obiekt `pd.DataFrame`, co można zauważyć w adnotacjach kodu:

```
def change_col1(df: pd.DataFrame) -> pd.DataFrame:
    return df.assign(col1=pd.Series([4, 5, 6], dtype=pd.Int64Dtype()))

def change_col2(df: pd.DataFrame) -> pd.DataFrame:
    return df.assign(col2=pd.Series(["X", "Y", "Z"], dtype=pd.StringDtype()))
```

Jak wspomnieliśmy już wcześniej, jednym z najczęstszych sposobów stosowania tych funkcji jest wypisanie ich jako osobnych kroków w naszym programie i przypisanie wyników każdego kroku do nowej zmiennej po drodze:

```
df2 = change_col1(df)
df3 = change_col2(df2)
df3
```

	col1	col2
0	4	X
1	5	Y
2	6	Z

Gdybyśmy chcieli całkowicie uniknąć użycia zmiennych pośrednich, moglibyśmy również zagnieździć wywołania funkcji jedno w drugim:

```
change_col2(change_col1(df))
      col1   col2
0     4      X
1     5      Y
2     6      Z
```

Jednak w przypadku takiego podejścia kod nie staje się bardziej czytelny, szczególnie biorąc pod uwagę fakt, że wykonanie `change_col1` odbywa się przed `change_col2`.

Wyrażając to jako potok, możemy uniknąć użycia zmiennych i łatwiej przedstawić kolejność wykonywanych operacji. Aby to osiągnąć, skorzystamy z metody `pd.DataFrame.pipe`:

```
df.pipe(change_col1).pipe(change_col2)
        col1   col2
0       4      X
1       5      Y
2       6      Z
```

Jak widać, otrzymaliśmy ten sam wynik co wcześniej, ale bez użycia zmiennych oraz w sposób, który można uznać za bardziej czytelny.

W przypadku gdy którakolwiek z funkcji, którą chcesz zastosować w potoku, wymaga dodatkowych argumentów, metoda `pd.DataFrame.pipe` potrafi je przekazać. Przyjrzyjmy się temu, co się stanie, jeśli dodamy nowy parametr `str_case` do funkcji `change_col2`:

```
from typing import Literal

def change_col2(
        df: pd.DataFrame,
        str_case: Literal["upper", "lower"]
) -> pd.DataFrame:
    if str_case == "upper":
        values = ["X", "Y", "Z"]
    else:
        values = ["x", "y", "z"]

    return df.assign(col2=pd.Series(values, dtype=pd.StringDtype()))
```

Jak widać na przykładzie metody `pd.DataFrame.pipe`, możesz po prostu przekazać ten argument jako pozycyjny lub nazwany, tak jakbyś bezpośrednio wywoływał funkcję `change_col2`:

```
df.pipe(change_col2, str_case="lower")
       col1   col2
0      1      x
1      2      y
2      3      z
```

Warto powtórzyć to, o czym wspomnieliśmy we wstępie do tej receptury: nie ma praktycznie żadnej różnicy funkcjonalnej między tymi stylami. Zachęcamy do poznania obu, ponieważ nieuchronnie spotkasz się z kodem napisanym na oba sposoby. W swojej własnej pracy programistycznej możesz nawet odkryć, że najlepiej sprawdza się mieszanie i dopasowywanie tych podejść.

Wybieranie filmów o najniższym budżecie z listy stu najlepszych

Teraz, gdy omówiliśmy wiele podstawowych algorytmów biblioteki pandas na poziomie teoretycznym, możemy zacząć przyglądać się bardziej „rzeczywistym" zbiorom danych i poznać popularne sposoby ich analizy.

Analiza „*N* najlepszych" to powszechna technika, w której filtrujemy dane na podstawie ich wyników mierzonych według jednej zmiennej. Większość narzędzi analitycznych umożliwia filtrowanie danych, aby odpowiedzieć na pytania typu: *Którzy klienci generują największą sprzedaż?* lub: *Które produkty mają najniższy stan magazynowy?*. Łącząc takie analizy, można tworzyć chwytliwe nagłówki prasowe, na przykład: *Spośród 100 najlepszych uniwersytetów te 5 ma najniższe czesne* albo: *Z 50 najlepszych miast do życia te 10 jest najtańszych*.

Ze względu na powszechność tego typu analiz pandas udostępnia wbudowane funkcje ułatwiające ich przeprowadzenie. W tej recepturze przyjrzymy się metodom `pd.Data➥Frame.nlargest` i `pd.DataFrame.nsmallest` oraz zobaczymy, jak je wykorzystać do odpowiedzi na pytanie typu: *Które ze 100 najlepszych filmów miały najniższy budżet?*.

Jak to zrobić?

Zacznijmy od wczytania zbioru danych o filmach i wybrania kolumn `movie_title` (tytuł filmu), `imdb_score` (ocena filmu w serwisie IMDB), `budget` (budżet filmu) i `gross` (przychód wygenerowany przez film):

```
df = pd.read_csv(
    "data/movie.csv",
    usecols=["movie_title", "imdb_score", "budget", "gross"],
    dtype_backend="numpy_nullable",
)
df.head()
```

	gross	movie_title	budget	imdb_score
0	760505847.0	Avatar	237000000.0	7.9
1	309404152.0	Pirates of the Caribbean: At World's End	300000000.0	7.1
2	200074175.0	Spectre	245000000.0	6.8
3	448130642.0	The Dark Knight Rises	250000000.0	8.5
4	<NA>	Star Wars: Episode VII The Force Awakens	<NA>	7.1

Metody `pd.DataFrame.nlargest` można użyć do wybrania stu najlepszych filmów według wartości kolumny `imdb_score`:

```
df.nlargest(100, "imdb_score").head()
```

	gross	movie_title	budget	imdb_score
2725	<NA>	Towering Inferno	<NA>	9.5
1920	28341469.0	The Shawshank Redemption	25000000.0	9.3
3402	134821952.0	The Godfather	6000000.0	9.2
2779	447093.0	Dekalog	<NA>	9.1
4312	<NA>	Kickboxer: Vengeance	17000000.0	9.1

Mając już wybraną pierwszą setkę filmów, możemy teraz skorzystać z metody `pd.Data➥Frame.nsmallest`, aby wyodrębnić pięć produkcji o najniższym budżecie spośród tej grupy:

```
df.nlargest(100, "imdb_score").nsmallest(5 , "budget")
        gross       movie_title              budget       imdb_score
4804    <NA>        Butterfly Girl           180000.0     8.7
4801    925402.0    Children of Heaven       180000.0     8.5
4706    <NA>        12 Angry Men             350000.0     8.9
4550    7098492.0   A Separation             500000.0     8.4
4636    133778.0    The Other Dream Team     500000.0     8.4
```

To jeszcze nie wszystko...

Możliwe jest przekazanie listy nazw kolumn jako parametru `columns=` metod `pd.Data➥Frame.nlargest` i `pd.DataFrame.nsmallest`. Byłoby to przydatne jedynie w celu rozstrzygnięcia remisów w przypadku, gdy występują zduplikowane wartości dzielące *n*-te miejsce w rankingu w pierwszej kolumnie z listy.

Aby zobaczyć, gdzie ma to znaczenie, spróbujmy wybrać 10 najlepszych filmów według oceny w serwisie IMDB:

```
df.nlargest(10, "imdb_score")
        gross   movie_title     budget  imdb_score
2725    <NA>    Towering Inferno        <NA>     9.5
1920    28341469.0      The Shawshank Redemption        25000000.0     9.3
3402    134821952.0     The Godfather   6000000.0       9.2
2779    447093.0        Dekalog    <NA>    9.1
4312    <NA>    Kickboxer: Vengeance    17000000.0      9.1
66      533316061.0     The Dark Knight 185000000.0     9.0
2791    57300000.0      The Godfather: Part II     13000000.0      9.0
3415    <NA>    Fargo   <NA>    9.0
335     377019252.0     The Lord of the Rings: The Return of the King
94000000.0     8.9
1857    96067179.0      Schindler's List        22000000.0      8.9
```

Jak widać, najniższa ocena IMDB wśród pierwszej dziesiątki wynosi `8.9`. Jednakże istnieje więcej niż 10 filmów, które mają ocenę `8.9` lub wyższą:

```
df[df["imdb_score"] >= 8.9]
        gross   movie_title     budget  imdb_score
66      533316061.0     The Dark Knight 185000000.0      9.0
335     377019252.0     The Lord of the Rings: The Return of the King
94000000.0     8.9
1857    96067179.0      Schindler's List        22000000.0      8.9
1920    28341469.0      The Shawshank Redemption       25000000.0      9.3
2725    <NA>    Towering Inferno        <NA>    9.5
2779    447093.0        Dekalog <NA>    9.1
2791    57300000.0      The Godfather: Part II      13000000.0    9.0
3295    107930000.0     Pulp Fiction    8000000.0       8.9
3402    134821952.0     The Godfather   6000000.0       9.2
3415    <NA>    Fargo   <NA>    9.0
```

```
4312    <NA>    Kickboxer: Vengeance    17000000.0      9.1
4397    6100000.0       The Good, the Bad and the Ugly    1200000.0      8.9
4706    <NA>    12 Angry Men     350000.0      8.9
```

Filmy, które znalazły się w pierwszej dziesiątce, to pierwsze dwa tytuły, które biblioteka pandas napotkała z daną oceną. Można jednak użyć kolumny z przychodami jako dodatkowego kryterium w przypadku remisu:

```
df.nlargest(10, ["imdb_score", "gross"])
        gross   movie_title     budget  imdb_score
2725    <NA>    Towering Inferno        <NA>     9.5
1920    28341469.0      The Shawshank Redemption        25000000.0      9.3
3402    134821952.0     The Godfather   6000000.0       9.2
2779    447093.0        Dekalog    <NA>    9.1
4312    <NA>    Kickboxer: Vengeance    17000000.0      9.1
66      533316061.0     The Dark Knight    185000000.0     9.0
2791    57300000.0      The Godfather: Part II    13000000.0        9.0
3415    <NA>    Fargo   <NA>    9.0
335     377019252.0     The Lord of the Rings: The Return of the King
94000000.0      8.9
3295    107930000.0     Pulp Fiction    8000000.0       8.9
```

W efekcie widać, że w naszej analizie dziesięciu najlepszych filmów *Pulp Fiction* zastąpił *Listę Schindlera*, ponieważ osiągnął wyższe wpływy kasowe.

Obliczanie ceny dla kroczącego zlecenia stop

Istnieje wiele strategii handlu akcjami. Jednym z podstawowych rodzajów zleceń, stosowanych przez wielu inwestorów, jest **zlecenie stop**. Jest to zlecenie składane przez inwestora w celu kupna lub sprzedaży akcji, które zostaje zrealizowane, gdy cena rynkowa osiągnie określony poziom. Zlecenia stop są przydatne zarówno do zapobiegania dużym stratom, jak i do ochrony zysków.

W typowym zleceniu stop cena nie zmienia się przez cały okres ważności zlecenia. Na przykład jeśli kupiłeś akcje po 100 zł za sztukę, możesz ustawić zlecenie stop na poziomie 90 zł za akcję, aby ograniczyć potencjalną stratę do 10%.

Bardziej zaawansowaną strategią jest ciągła modyfikacja ceny sprzedaży w zleceniu stop, aby śledzić wartość akcji, jeśli ta wzrasta. Nazywa się to **kroczącym zleceniem stop** (ang. *trailing stop order*). Konkretnie, jeśli te same akcje o wartości 100 zł wzrosną do 120 zł, to kroczące zlecenie stop ustawione na 10% poniżej aktualnej ceny rynkowej przesunęłoby cenę sprzedaży do 108 zł.

Kroczące zlecenie stop nigdy nie przesuwa się w dół i jest zawsze powiązane z maksymalną wartością od momentu zakupu. Jeśli cena akcji spadłaby ze 120 zł do 110 zł, zlecenie stop pozostałoby na poziomie 108 zł. Wzrosłoby tylko wtedy, gdyby cena przekroczyła 120 zł.

Ta receptura określa cenę kroczącego zlecenia stop na podstawie początkowej ceny zakupu dla dowolnych akcji, wykorzystując metodę `pd.Series.cummax`. Ponadto pokazuje, jak można zastosować metodę `pd.Series.cummin` do obsługi pozycji krótkich. Wyjaśnimy również, jak metoda `pd.Series.idxmax` może być wykorzystana do zidentyfikowania dnia, w którym zlecenie stop zostałoby zrealizowane.

Jak to zrobić?

Na początek zajmiemy się akcjami firmy Nvidia (NVDA) i założymy, że dokonano ich zakupu pierwszego dnia handlowego w 2020 roku:

```
df = pd.read_csv(
    "data/NVDA.csv",
    usecols=["Date", "Close"],
    parse_dates=["Date"],
    index_col=["Date"],
    dtype_backend="numpy_nullable",
)
df.head()
ValueError: not all elements from date_cols are numpy arrays
```

W wersji 2.2 biblioteki pandas występuje błąd, który uniemożliwia wykonanie poprzedniego bloku kodu — prowadzi on do zgłoszenia wyjątku `ValueError`. Jeśli napotkasz ten błąd, możesz alternatywnie wykonać metodę `pd.read_csv` bez argumentu `dtype_backend`, a następnie dodać wywołanie `pd.DataFrame.convert_dtypes`:

```
df = pd.read_csv(
    "data/NVDA.csv",
    usecols=["Date", "Close"],
    parse_dates=["Date"],
    index_col=["Date"],
).convert_dtypes(dtype_backend="numpy_nullable")

df.head()
                Close
Date
2020-01-02  59.977501
2020-01-03  59.017502
2020-01-06  59.264999
2020-01-07  59.982498
2020-01-08  60.095001
```

Aby uzyskać więcej informacji na ten temat, zapoznaj się ze zgłoszeniem błędu #57930 w repozytorium biblioteki pandas (*https://github.com/pandas-dev/pandas/issues/57930*).

Niezależnie od wybranej metody pamiętaj, że funkcja `pd.read_csv` zwraca obiekt `pd.Data`
`↪Frame`, ale w tej analizie potrzebujemy jedynie obiektu `pd.Series`. Aby dokonać wymaganej konwersji, możesz użyć metody `pd.DataFrame.squeeze`, która zredukuje obiekt z dwóch do jednego wymiaru, jeśli to możliwe:

```
ser = df.squeeze()
ser.head()
```

```
Date
2020-01-02    59.977501
2020-01-03    59.017502
2020-01-06    59.264999
2020-01-07    59.982498
2020-01-08    60.095001
Name: Close, dtype: float64
```

Dzięki temu możemy użyć metody `pd.Series.cummax` do śledzenia najwyższej ceny zamknięcia odnotowanej do danego momentu:

```
ser_cummax = ser.cummax()
ser_cummax.head()
Date
2020-01-02    59.977501
2020-01-03    59.977501
2020-01-06    59.977501
2020-01-07    59.982498
2020-01-08    60.095001
Name: Close, dtype: float64
```

Aby utworzyć kroczące zlecenie stop, które ogranicza nasze straty do 10%, możemy dodać operację mnożenia przez `0.9`:

```
ser.cummax().mul(0.9).head()
Date
2020-01-02    53.979751
2020-01-03    53.979751
2020-01-06    53.979751
2020-01-07    53.984248
2020-01-08    54.085501
Name: Close, dtype: float64
```

Metoda `pd.Series.cummax` działa poprzez zachowywanie maksymalnej wartości napotkanej do danego momentu, włącznie z bieżącą wartością. Pomnożenie tej serii przez `0.9` (lub inny wybrany współczynnik) tworzy kroczące zlecenie stop. W tym konkretnym przykładzie cena akcji NVDA wzrosła, a wraz z nią wzrósł również poziom dla kroczącego zlecenia stop.

Z drugiej strony załóżmy, że byliśmy pesymistycznie nastawieni do akcji NVDA w tym okresie i chcieliśmy je sprzedać. Jednak nadal chcieliśmy zastosować zlecenie stop, aby ograniczyć straty do 10% wzrostu wartości.

W tym przypadku możemy po prostu zastąpić metodę `pd.Series.cummax` metodą `pd.Series.cummin` oraz przeprowadzić mnożenie przez `1.1` zamiast przez `0.9`:

```
ser.cummin().mul(1.1).head()
Date
2020-01-02    65.975251
2020-01-03    64.919252
2020-01-06    64.919252
2020-01-07    64.919252
2020-01-08    64.919252
Name: Close, dtype: float64
```

To jeszcze nie wszystko...

Po obliczeniu kroczących zleceń stop możemy łatwo określić dni, w których spadliśmy poniżej skumulowanego maksimum o więcej niż ustalony próg:

```
stop_prices = ser.cummax().mul(0.9)
ser[ser <= stop_prices]
Date
2020-02-24     68.320000
2020-02-25     65.512497
2020-02-26     66.912498
2020-02-27     63.150002
2020-02-28     67.517502
                  ...
2023-10-27    405.000000
2023-10-30    411.609985
2023-10-31    407.799988
2023-11-01    423.250000
2023-11-02    435.059998
Name: Close, Length: 495, dtype: float64
```

Gdybyśmy chcieli zidentyfikować tylko pierwszy dzień, w którym wartość spadła poniżej skumulowanego maksimum, moglibyśmy użyć metody `pd.Series.idxmax`. Metoda ta działa w następujący sposób: najpierw oblicza maksymalną wartość w obiekcie `pd.Series`, a potem zwraca indeks pierwszego wiersza, w którym to maksimum wystąpiło:

```
(ser <= stop_prices).idxmax()
Timestamp('2020-02-24 00:00:00')
```

Wyrażenie `ser <= stop_prices` zwraca serię logiczną `pd.Series`, która zawiera wartości `True`/`False`. W tym kontekście wartość `True` oznacza, że cena akcji jest równa wcześniej obliczonej cenie zlecenia stop lub od niej niższa. Metoda `pd.Series.idxmax` traktuje `True` jako wartość maksymalną w danej serii. Zwraca pierwszy indeks, gdzie wystąpiła wartość `True`, i informuje nas o pierwszym dniu, w którym kroczące zlecenie stop powinno zostać zrealizowane.

Ten przykład daje nam tylko przedsmak tego, jak użyteczna może być biblioteka pandas w handlu papierami wartościowymi.

Wyszukiwanie najlepszych baseballistów

Baseball, amerykański sport narodowy, od dawna jest przedmiotem intensywnych badań analitycznych, a zbieranie danych sięga początków ubiegłego wieku. Zespołom MLB (Major League Baseball) zaawansowana analiza danych pomaga odpowiedzieć na pytania takie jak: *Ile powinienem zapłacić za zawodnika X?* albo: *Jaką taktykę przyjąć w obecnej sytuacji meczowej?*. Dla kibiców te same dane stanowią pożywkę dla niekończących się dyskusji o tym, *kto jest najlepszym zawodnikiem wszech czasów*.

W tej recepturze wykorzystamy dane pochodzące z serwisu *retrosheet.org*. Zgodnie z wymogami licencyjnymi Retrosheet należy pamiętać o następującej informacji prawnej:

Uwaga

Wykorzystane tu informacje zostały uzyskane bezpłatnie i są chronione prawem autorskim przez Retrosheet. Zainteresowane strony mogą skontaktować się z Retrosheet poprzez witrynę *https://www.retrosheet.org*/.

Dane w pierwotnej postaci zostały przetworzone, aby przedstawić popularne statystyki baseballowe dla podejść do bazy (`ab`), uderzeń (`h`), zdobytych punktów (`r`) i home runów (`hr`) dla zawodowych graczy w latach 2020 – 2023.

Jak to zrobić?

Zacznijmy od wczytania podsumowanych danych i ustawienia kolumny `id` (reprezentującej unikalnego gracza) jako indeksu:

```
df = pd.read_parquet(
    "data/mlb_batting_summaries.parquet",
).set_index("id")

df
```

id	ab	r	h	hr
abadf001	0	0	0	0
abboa001	0	0	0	0
abboc001	3	0	1	0
abrac001	847	116	208	20
abrea001	0	0	0	0
…	…	…	…	…
zimmk001	0	0	0	0
zimmr001	255	27	62	14
zubet001	1	0	0	0
zunig001	0	0	0	0
zunim001	572	82	111	41

```
2183 rows × 4 columns
```

W baseballu rzadko się zdarza, aby jeden zawodnik dominował we wszystkich kategoriach statystycznych. Często gracz z dużą liczbą home runów (`hr`) ma większą siłę uderzenia i potrafi wybić piłkę dalej, ale może to robić rzadziej niż zawodnik wyspecjalizowany w zdobywaniu wielu uderzeń (`h`). Dzięki bibliotece pandas nie musimy oddzielnie analizować poszczególnych wskaźników. Proste wywołanie funkcji `pd.DataFrame.idxmax` sprawdzi każdą kolumnę, znajdzie maksymalną wartość i zwróci indeks wiersza związany z tą maksymalną wartością:

```
df.idxmax()
```

```
ab    semim001
r     freef001
h     freef001
hr    judga001
dtype: string
```

Jak widać, zawodnik `semim001` (Marcus Semien) miał najwięcej podejść do odbicia, `freef001` (Freddie Freeman) zdobył najwięcej punktów i odbić, a `judga001` (Aaron Judge) zdobył najwięcej home runów w tym okresie.

Jeśli chcesz głębiej przeanalizować, jak ci świetni gracze radzili sobie we wszystkich kategoriach, możesz wykorzystać wynik działania funkcji `pd.DataFrame.idxmax`, następnie wywołać funkcję `pd.Series.unique` dla otrzymanych wartości oraz użyć otrzymanego wyniku jako maski dla wartości całego obiektu `pd.DataFrame`:

```
best_players = df.idxmax().unique()
mask = df.index.isin(best_players)
df[mask]
```

id	ab	r	h	hr
freef001	**1849**	**368**	**590**	**81**
judga001	**1487**	**301**	**433**	**138**
semim001	**1979**	**338**	**521**	**100**

To jeszcze nie wszystko...

Aby te dane przedstawić w znacznie atrakcyjniejszej postaci wizualnej, możesz użyć funkcji `pd.DataFrame.style.highlight_max`, która dokładnie pokaże (rysunek 5.1), w której kategorii ci gracze byli najlepsi:

```
df[mask].style.highlight_max()
```

id	ab	r	h	hr
freef001	1849	368	590	81
judga001	1487	301	433	138
semim001	1979	338	521	100

Rysunek 5.1. Dane wyjściowe notatnika Jupyter, które zawierają wynik wykonania funkcji pd.DataFrame.style.highlight_max

Ustalanie pozycji zdobywającej najwięcej punktów dla drużyny

W baseballu drużyny mają prawo do 9 pałkarzy w składzie, gdzie 1 oznacza pierwszego odbijającego, 9 zaś ostatniego. W trakcie meczu drużyny przechodzą przez kolejnych pałkarzy, zaczynając od nowa od pierwszego po tym, jak ostatni zakończył swoje odbicie.

Zazwyczaj drużyny umieszczają swoich najlepszych pałkarzy bliżej początku składu (czyli na niższych pozycjach numerycznych), aby zmaksymalizować ich szanse na zdobycie punktów. Nie oznacza to jednak, że zawodnik na pozycji 1 zawsze zdobędzie punkt jako pierwszy.

W tej recepturze przyjrzymy się wszystkim drużynom Major League Baseball z lat 2000 – 2023 i sprawdzimy, która pozycja w składzie zdobyła najwięcej punktów dla drużyny w każdym sezonie.

Jak to zrobić?

Podobnie jak w recepturze dotyczącej znalezienia najlepszych baseballistów wykorzystamy dane z serwisu *retrosheet.org*. W tym konkretnym zbiorze danych ustawimy kolumny roku (`year`) i drużyny (`team`) jako indeks wierszy, pozostawiając pozostałe kolumny do pokazania pozycji w kolejności odbijania:

```
df = pd.read_parquet(
    "data/runs_scored_by_team.parquet",
).set_index(["year", "team"])

df
```

year	team	1	2	3	…	7	8	9
2000	ANA	124	107	100	…	77	76	54
	ARI	110	106	109	…	72	68	40
	ATL	113	125	124	…	77	74	39
	BAL	106	106	92	…	83	78	74
	BOS	99	107	99	…	75	66	62
…	…	…	…	…	…	…	…	…
2023	SLN	105	91	85	…	70	55	74
	TBA	121	120	93	…	78	95	98
	TEX	126	115	91	…	80	87	81
	TOR	91	97	85	…	64	70	79
	WAS	110	90	87	…	63	67	64

720 rows × 9 columns

Dzięki funkcji `pd.DataFrame.idxmax` możemy dla każdego roku i drużyny sprawdzić, która pozycja zdobyła najwięcej punktów. Jednak w tym zbiorze danych etykieta indeksu, która miałaby zostać zidentyfikowana przez wywołanie `pd.DataFrame.idxmax`, znajduje się w kolumnach, a nie w wierszach. Na szczęście pandas może to łatwo obliczyć za pomocą argumentu `axis=1`:

```
df.idxmax(axis=1)
```

```
year  team
2000  ANA     1
      ARI     1
      ATL     2
      BAL     1
      BOS     4
             ...
2023  SLN     1
      TBA     1
      TEX     1
      TOR     2
      WAS     1
Length: 720, dtype: object
```

Na tej podstawie możemy użyć funkcji `pd.Series.value_counts`, aby sprawdzić, ile razy dana pozycja w kolejności reprezentowała największą liczbę zdobytych punktów dla drużyny. Wykorzystamy również argument `normalize=True`, który zamiast sumy dostarczy częstotliwość występowania danej wartości:

```
df.idxmax(axis=1).value_counts(normalize=True)
1    0.480556
2    0.208333
3    0.202778
4    0.088889
5    0.018056
6    0.001389
Name: proportion, dtype: float64
```

Nie jest zaskoczeniem, że pierwszy pałkarz w składzie najczęściej zdobywał najwięcej punktów. Taki wynik otrzymaliśmy w przypadku 48% drużyn.

To jeszcze nie wszystko...

Warto byłoby zgłębić temat i odpowiedzieć na następujące pytanie: *W drużynach, w których pierwszy pałkarz zdobył najwięcej punktów, kto zdobył drugą największą liczbę?*.

Aby to obliczyć, możemy utworzyć maskę filtrującą drużyny, w których pierwszy pałkarz zdobył najwięcej punktów, usunąć tę kolumnę ze zbioru danych, a następnie powtórzyć to samo podejście z zastosowaniem funkcji `pd.DataFrame.idxmax`, aby zidentyfikować kolejną pozycję w rankingu:

```
mask = df.idxmax(axis=1).eq("1")
df[mask].drop(columns=["1"]).idxmax(axis=1).value_counts(normalize=True)
2    0.497110
3    0.280347
4    0.164740
5    0.043353ta
6    0.014451
Name: proportion, dtype: float64
```

Jak widać, jeśli pierwszy pałkarz drużyny nie jest liderem w zdobytych punktach, to drugi pałkarz staje się liderem prawie w 50% przypadków.

Doskonałe źródło praktycznych rozwiązań typowych problemów, z którymi spotkasz się w swojej pracy analitycznej w Pythonie!
— Wes McKinney, twórca projektów pandas i Ibis

Pandas to najpopularniejsza biblioteka Pythona do przetwarzania danych. Jednak nawet doświadczeni użytkownicy tego darmowego narzędzia często nie znają jego wszystkich imponujących, a przy tym wyjątkowo przydatnych funkcji. Choć oficjalna dokumentacja pandas jest obszerna, brakuje w niej praktycznych przykładów pokazujących, jak łączyć wiele poleceń — a to właśnie okazuje się kluczowe!

Książka powstała z myślą o wszystkich, którzy zajmują się analizą danych — bez względu na poziom doświadczenia. Została pomyślana tak, aby w klarowny i praktyczny sposób, krok po kroku wyjaśnić wykonywanie różnych operacji na danych: od podstawowych czynności przetwarzania danych po zaawansowane techniki obsługi dużych zbiorów. Poszczególne receptury przygotowano w czytelnej konwencji: *Jak to zrobić? — Jak to działa? —To jeszcze nie wszystko...* Każda receptura jest niezależna od innych, a układ treści pozwala na łatwe i szybkie odnalezienie potrzebnego zagadnienia.

W książce między innymi:

- system typów pandas
- eksploracja danych za pomocą biblioteki pandas
- grupowanie, agregowanie, przekształcanie i łączenie danych z różnych źródeł
- niezawodne szeregi czasowe i skalowanie operacji w pandas
- ekosystem biblioteki pandas

William Ayd
od 2018 roku opiekuje się projektem pandas. Jest też doświadczonym konsultantem w dziedzinie zastosowania biblioteki pandas i Pythona w danologii.

Matthew Harrison
korzysta z Pythona od 2000 roku. Jest autorem popularnych podręczników, takich jak *Uczenie maszynowe w Pythonie. Leksykon kieszonkowy.*

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-3173-2
9 788328 931732

Cena: 89,00 zł

packt