

Wprowadzenie do dplyr

Statystyka I z R

Bartosz Maćkiewicz

Wprowadzenie do dplyr

Pakiet `dplyr` jest pakietem, który ma pozwolić nam ułatwić i przyspieszyć wykonywanie różnych często dokonywanych operacji na ramkach danych. Jego podstawowe zalety to przejrzysta składnia oraz szybkość działania (wiele z jego funkcji zostało napisanych w C++ dla lepszej optymalizacji). Przy naprawdę dużej ilości danych użycie tego pakietu może okazać się niezbędne.

Wszystkie te operacje potrafia Państwo wykonać za pomocą “czystego” R, na czym bardzo mi zależało. Być może jednak czasami będzie wygodniej zrobić to używając funkcji dostarczanych przez pakiet `dplyr`. Na pewno traci się pewną elastyczność i kontrolę nad tym, co się robi, nie ulega jednak wątpliwości, że czasami warto zapłacić taką cenę za wygodę pracy.

To wprowadzenie zostało przygotowane na podstawie oficjalnego krótkiego wprowadzenia do pakietu oraz dokumentacji.

Omówimy sobie kilka możliwości jakie daje nam ten pakiet.

Instalujemy pakiet za pomocą standardowego `install.packages` oraz ładujemy go za pomocą `library`.

```
#install.packages('dplyr')  
library(dplyr)
```

Do ćwiczeń wykorzystamy dane pochodzące z badań nad uchowcami. Jak wyglądają te zwierzatka? Tak:



Rysunek 1: Uchowiec

Informacje na temat danych mogą Państwo znaleźć tutaj: [Dokumentacja zbioru Abalone](#)

Wczytajmy dane i zobaczmy jak one wyglądają za pomocą `head`.

```
df <- read.csv('abalone.data', header = FALSE)
col_names <- c('Sex', 'Length', 'Diameter', 'Height',
               'Whole.weight', 'Shucked.weight',
               'Viscera.weight', 'Shell.weight', 'Rings')
colnames(df) <- col_names
head(df)
```

	Sex	Length	Diameter	Height	Whole.weight	Shucked.weight	Viscera.weight
1	M	0.455	0.365	0.095	0.5140	0.2245	0.1010
2	M	0.350	0.265	0.090	0.2255	0.0995	0.0485
3	F	0.530	0.420	0.135	0.6770	0.2565	0.1415
4	M	0.440	0.365	0.125	0.5160	0.2155	0.1140
5	I	0.330	0.255	0.080	0.2050	0.0895	0.0395
6	I	0.425	0.300	0.095	0.3515	0.1410	0.0775

	Shell.weight	Rings
1	0.150	15
2	0.070	7
3	0.210	9
4	0.155	10
5	0.055	7
6	0.120	8

W kolumnie `Sex` znajdujemy informacje o płci (I oznacza “infant”), dalej mamy dane dotyczące rozmiaru i wagi oraz liczby “kręgów” (rings?). Z dokumentacji danych możemy dowiedzieć się, co zmienna ta oznacza: “Rings / integer / - / +1.5 gives the age in years”

filter

Funkcja `filter` pozwala nam wybierać wiersze z ramki danych. Warto zwrócić uwagę na jej składnię. Jako pierwszy argument przyjmuje ramkę danych, kolejne wyrażenia to warunki, jakie musi spełniać obserwacja. **Bardzo ważne** jest, żeby zauważyć, że do nazw kolumn odwołujemy się bez cudzysłowów. Spróbujemy wybrać wszystkie te ślimaczki, które są płci męskiej i mają więcej niż 22 “kręgi”.

```
filter(df, Sex == 'M', Rings > 22)
```

	Sex	Length	Diameter	Height	Whole.weight	Shucked.weight	Viscera.weight
1	M	0.600	0.495	0.195	1.0575	0.3840	0.1900
2	M	0.630	0.485	0.175	1.3000	0.4335	0.2945
3	M	0.665	0.535	0.225	2.1835	0.7535	0.3910
4	M	0.610	0.490	0.150	1.1030	0.4250	0.2025
5	M	0.515	0.400	0.160	0.8175	0.2515	0.1560
6	M	0.690	0.540	0.185	1.6195	0.5330	0.3530

	Shell.weight	Rings
1	0.375	26
2	0.460	23
3	0.885	27
4	0.360	23
5	0.300	23
6	0.555	24

Możemy oczywiście użyć bardziej oczywistej składni i posłużyć się operatorem logicznym koniunkcji. Efekt jest dokładnie taki sam, jak w poprzednim przykładzie:

```
filter(df, Sex == 'M' & Rings > 22)
```

	Sex	Length	Diameter	Height	Whole.weight	Shucked.weight	Viscera.weight
1	M	0.600	0.495	0.195	1.0575	0.3840	0.1900
2	M	0.630	0.485	0.175	1.3000	0.4335	0.2945
3	M	0.665	0.535	0.225	2.1835	0.7535	0.3910
4	M	0.610	0.490	0.150	1.1030	0.4250	0.2025
5	M	0.515	0.400	0.160	0.8175	0.2515	0.1560
6	M	0.690	0.540	0.185	1.6195	0.5330	0.3530

	Shell.weight	Rings
1	0.375	26
2	0.460	23
3	0.885	27
4	0.360	23
5	0.300	23

6 0.555 24

Nic nie stoi na przeszkodzie, by używać wszystkich dostępnych w R operatorów logicznych i arytmetycznych. W poniższym przykładzie wybraliśmy te ślimaczki, które są płci żeńskiej lub męskiej, mają wartość `Height` większą niż 0.24 oraz co najmniej 10 “kręgów”. Wszystkie zasady związane z nawiasami jakie znamy ze standardowego R dalej obowiązują.

```
filter(df, (Sex == 'M' | Sex == 'F') & Height > 0.24 & Rings >= 10)
```

	Sex	Length	Diameter	Height	Whole.weight	Shucked.weight	Viscera.weight
1	M	0.705	0.565	0.515	2.2100	1.1075	0.4865
2	F	0.815	0.650	0.250	2.2550	0.8905	0.4200
3	M	0.775	0.630	0.250	2.7795	1.3485	0.7600
4	F	0.595	0.470	0.250	1.2830	0.4620	0.2475

	Shell.weight	Rings
1	0.5120	10
2	0.7975	14
3	0.5780	12
4	0.4450	14

arrange

Za pomocą funkcji `arrange` możemy porządkować ramkę danych zgodnie z wartościami z poszczególnych kolumn. Jako pierwszy argument przekazujemy ramkę danych, kolejnymi są kolumny po których sortujemy. Kolejne kolumny używane są wówczas, gdy w poprzednich mamy do czynienia z “remisem”. W 2 i 3 wierszu nasze ślimaczki mają tyle samo “kręgów”, więc pod uwagę brana jest kolejna kolumna - długość. Za pomocą funkcji `desc` możemy zdecydować, czy porządkować chcemy rosnąco, czy malejąco. W zasadzie funkcja ta robi to samo co `order`, oferując (może) bardziej przejrzystą składnię.

```
head(arrange(df, desc(Rings), desc(Length), Sex), 15)
```

	Sex	Length	Diameter	Height	Whole.weight	Shucked.weight	Viscera.weight
1	F	0.700	0.585	0.185	1.8075	0.7055	0.3215
2	M	0.665	0.535	0.225	2.1835	0.7535	0.3910
3	F	0.550	0.465	0.180	1.2125	0.3245	0.2050
4	M	0.600	0.495	0.195	1.0575	0.3840	0.1900
5	F	0.645	0.490	0.215	1.4060	0.4265	0.2285
6	F	0.700	0.540	0.215	1.9780	0.6675	0.3125
7	M	0.690	0.540	0.185	1.6195	0.5330	0.3530
8	F	0.800	0.630	0.195	2.5260	0.9330	0.5900
9	M	0.630	0.485	0.175	1.3000	0.4335	0.2945

10	F	0.620	0.470	0.200	1.2255	0.3810	0.2700
11	F	0.620	0.520	0.225	1.1835	0.3780	0.2700
12	M	0.610	0.490	0.150	1.1030	0.4250	0.2025
13	F	0.550	0.415	0.135	0.7750	0.3020	0.1790
14	M	0.515	0.400	0.160	0.8175	0.2515	0.1560
15	F	0.490	0.385	0.150	0.7865	0.2410	0.1400

	Shell.weight	Rings
1	0.475	29
2	0.885	27
3	0.525	27
4	0.375	26
5	0.510	25
6	0.710	24
7	0.555	24
8	0.620	23
9	0.460	23
10	0.435	23
11	0.395	23
12	0.360	23
13	0.260	23
14	0.300	23
15	0.240	23

select

`select` umożliwia nam wybieranie kolumn z ramki danych. Jako pierwszy argument przyjmuje ramkę danych jako kolejne kolumny, które chcemy wybrać. Z najprostszym przypadkiem mamy do czynienia wówczas, gdy po prostu przekazujemy nazwy kolumn.

```
head(select(df, Sex, Length, Diameter))
```

	Sex	Length	Diameter
1	M	0.455	0.365
2	M	0.350	0.265
3	F	0.530	0.420
4	M	0.440	0.365
5	I	0.330	0.255
6	I	0.425	0.300

Pakiet `dplyr` umożliwia nam radzenie sobie z sytuacjami, gdy w naszej ramce danych mamy bardzo dużo kolumn. Załóżmy, że chcemy wybrać tylko te z nich, których nazwa zawiera ciąg znaków `weight`. Możemy ten problem rozwiązać za pomocą funkcji `contains`.

```
head(select(df, Sex, contains('weight')))
```

	Sex	Whole.weight	Shucked.weight	Viscera.weight	Shell.weight
1	M	0.5140	0.2245	0.1010	0.150
2	M	0.2255	0.0995	0.0485	0.070
3	F	0.6770	0.2565	0.1415	0.210
4	M	0.5160	0.2155	0.1140	0.155
5	I	0.2050	0.0895	0.0395	0.055
6	I	0.3515	0.1410	0.0775	0.120

Dodatkową funkcjonalnością jest fakt, że możemy posłużyć się wyrażeniami regularnymi. Jeżeli nie mieli Państwo z nimi styczności wcześniej, to najkrócej mówiąc są to ciągi znaków wyrażające pewne wzorce, do których dopasowywane są napisy. W poniższym przykładzie posłużyłem się patternem `^...ght`, który dopasowuje wszystkie napisy, które zaczynają się (^) od trzech dowolnych znaków (...), po których następuje ght. W naszym przypadku tylko Height spełnia te warunki, dlatego tylko tę kolumnę wybraliśmy.

```
head(select(df, Sex, matches('^...ght')))
```

	Sex	Height
1	M	0.095
2	M	0.090
3	F	0.135
4	M	0.125
5	I	0.080
6	I	0.095

Możemy również wybrać wszystkie kolumny z wyjątkiem jakichś. Posługując się składnią `nazwa_kolumny:_nazwa_kolumny` wybieramy wszystkie kolejne kolumny pomiędzy dwoma określonymi i używając znaku minusa (-) wybieramy wszystkie kolumny z ramki danych oprócz tych wyszczególnionych. W przykładzie poniżej posłużyliśmy się tą metodą, by wybrać wszystkie kolumny oprócz tych dotyczących wagi naszych zwierzątek.

```
head(select(df, -(Whole.weight:Shell.weight)))
```

	Sex	Length	Diameter	Height	Rings
1	M	0.455	0.365	0.095	15
2	M	0.350	0.265	0.090	7
3	F	0.530	0.420	0.135	9
4	M	0.440	0.365	0.125	10
5	I	0.330	0.255	0.080	7
6	I	0.425	0.300	0.095	8

distinct

Wybiera unikatowe wartości z ramki danych, jako drugi i kolejne argumenty przekazujemy kolumny, które chcemy wziąć pod uwagę, decydując o “unikatowości” obserwacji. W poniższym przykładzie przekazaliśmy kolumny **Sex** i **Rings**, otrzymamy więc maksymalnie 3 (liczba płci) x 29 (liczba “kregów”) wartości (bo tyle jest unikatowych kombinacji w naszym przypadku). W rzeczywistości otrzymaliśmy 68 (ponieważ nie wszystkie kombinacje występują w naszym zbiorze danych)

```
nrow(distinct(df, Sex, Rings))
```

```
[1] 68
```

```
head(distinct(df, Sex, Rings), 15)
```

	Sex	Rings
1	M	15
2	M	7
3	F	9
4	M	10
5	I	7
6	I	8
7	F	20
8	F	16
9	M	9
10	F	19
11	F	14
12	M	11
13	F	10
14	M	12
15	I	10

mutate

Za pomocą funkcji **mutate** możemy łatwo dodawać kolumny. Składnia jest analogiczna jak przy poprzednich funkcjach - pierwszym argumentem jest ramka danych, kolejne to kolumny, które chcemy dodać. Określając wartości w kolumnach możemy posługiwać się operatorami arytmetycznymi ale też funkcjami R (np. **sqrt**). Korzystając z wiedzy dotyczącej relacji między ilością “kregów” oraz wiekiem dodajmy nową kolumnę **Age**. Dla demonstracji dodajmy również dwie dodatkowe kolumny ujmujące relację między wagą a wymiarami.

```
head(mutate(df,
  Age = Rings + 1.5,
  Height.to.weight.ratio = Height/Whole.weight,
  Body.mass.ratio = sqrt(Height*Diameter)/Whole.weight))
```

	Sex	Length	Diameter	Height	Whole.weight	Shucked.weight	Viscera.weight
1	M	0.455	0.365	0.095	0.5140	0.2245	0.1010
2	M	0.350	0.265	0.090	0.2255	0.0995	0.0485
3	F	0.530	0.420	0.135	0.6770	0.2565	0.1415
4	M	0.440	0.365	0.125	0.5160	0.2155	0.1140
5	I	0.330	0.255	0.080	0.2050	0.0895	0.0395
6	I	0.425	0.300	0.095	0.3515	0.1410	0.0775

	Shell.weight	Rings	Age	Height.to.weight.ratio	Body.mass.ratio
1	0.150	15	16.5	0.1848249	0.3622806
2	0.070	7	8.5	0.3991131	0.6848534
3	0.210	9	10.5	0.1994092	0.3517247
4	0.155	10	11.5	0.2422481	0.4139537
5	0.055	7	8.5	0.3902439	0.6967247
6	0.120	8	9.5	0.2702703	0.4802829

Funkcja `transmute` różni się od `mutate` tylko tym, że zwraca wyłącznie nowododane kolumny:

```
head(transmute(df,
  Age = Rings + 1.5,
  Height.to.weight.ratio = Height/Whole.weight,
  Body.mass.ratio = sqrt(Height*Diameter)/Whole.weight))
```

	Age	Height.to.weight.ratio	Body.mass.ratio
1	16.5	0.1848249	0.3622806
2	8.5	0.3991131	0.6848534
3	10.5	0.1994092	0.3517247
4	11.5	0.2422481	0.4139537
5	8.5	0.3902439	0.6967247
6	9.5	0.2702703	0.4802829

summarise

`summarise` pozwala nam stworzyć ramkę danych, w której w jakiś sposób “podsumowujemy” wiele wartości do jednej. Możemy użyć tutaj standardowych funkcji R takich jak `mean` czy `sd`. W przypadku poniżej stworzyliśmy dwie nowe kolumny, które sprowadzają wartości w kolumnach `Rings` i `Whole.weight` do jednej wartości (do średniej oraz odchylenia standardowego).

```
summarise(df,
  ring_mean = mean(Rings),
  wieight_sd = sd(Whole.weight))
```

```
  ring_mean wieight_sd
1  9.933684  0.490389
```

sample_n/frac

Funkcje `sample_n` i `sample_frac` pozwalają nam losować próbę z ramki danych. Różnią się tym, że `sample_n` pozwala wylosować n obserwacji, a `sample_frac` pozwala wylosować liczbę obserwacji stanowiącą jakiś ułamek wszystkich (w przykładzie - 0.5 wszystkich obserwacji). Funkcje te mogą być bardzo przydatne przy pracy z dużymi zbiorami danych.

```
sample_n(df, 15)
```

	Sex	Length	Diameter	Height	Whole.weight	Shucked.weight	Viscera.weight
1	I	0.440	0.325	0.090	0.3500	0.1480	0.0670
2	F	0.575	0.480	0.165	1.0780	0.5110	0.2095
3	M	0.330	0.235	0.090	0.1630	0.0615	0.0340
4	I	0.445	0.345	0.105	0.4090	0.1675	0.1015
5	I	0.385	0.290	0.085	0.2505	0.1120	0.0610
6	I	0.325	0.245	0.070	0.1610	0.0755	0.0255
7	M	0.645	0.490	0.175	1.3200	0.6525	0.2375
8	F	0.625	0.485	0.175	1.3620	0.6765	0.2615
9	I	0.280	0.220	0.080	0.1315	0.0660	0.0240
10	I	0.410	0.325	0.105	0.3610	0.1605	0.0665
11	M	0.565	0.435	0.185	1.0320	0.3540	0.2045
12	I	0.365	0.255	0.080	0.1985	0.0785	0.0345
13	M	0.560	0.435	0.180	0.8890	0.3600	0.2040
14	F	0.585	0.415	0.155	0.6985	0.3000	0.1460
15	F	0.625	0.495	0.160	1.1115	0.4495	0.2825

	Shell.weight	Rings
1	0.1050	7
2	0.3060	9
3	0.0550	10
4	0.1170	7
5	0.0800	8
6	0.0450	6
7	0.3385	11
8	0.3705	10

9	0.0300	5
10	0.1030	8
11	0.3100	20
12	0.0530	5
13	0.2500	11
14	0.1950	12
15	0.3450	11

```
sample_frac(df, 0.005)
```

	Sex	Length	Diameter	Height	Whole.weight	Shucked.weight	Viscera.weight
1	F	0.460	0.365	0.125	0.4785	0.2060	0.1045
2	F	0.580	0.450	0.195	0.8265	0.4035	0.1730
3	F	0.580	0.500	0.165	0.9250	0.3700	0.1850
4	M	0.415	0.305	0.100	0.3250	0.1560	0.0505
5	F	0.715	0.565	0.180	1.7900	0.8440	0.3535
6	M	0.695	0.530	0.210	1.5100	0.6640	0.4095
7	F	0.655	0.520	0.200	1.5475	0.7130	0.3140
8	M	0.480	0.370	0.135	0.6315	0.3445	0.1015
9	F	0.625	0.480	0.200	1.3235	0.6075	0.3055
10	M	0.565	0.390	0.125	0.7440	0.3520	0.1300
11	F	0.680	0.505	0.170	1.3435	0.6570	0.2970
12	M	0.635	0.515	0.170	1.2750	0.5090	0.2860
13	F	0.605	0.490	0.150	1.1345	0.4305	0.2525
14	I	0.395	0.290	0.095	0.3040	0.1270	0.0840
15	F	0.660	0.500	0.165	1.1905	0.4585	0.2980
16	M	0.570	0.460	0.140	0.9535	0.4465	0.2065
17	F	0.695	0.535	0.175	1.8385	0.8035	0.3960
18	F	0.490	0.380	0.130	0.5390	0.2290	0.1355
19	F	0.455	0.350	0.125	0.4485	0.1585	0.1020
20	M	0.575	0.400	0.155	0.9325	0.3605	0.2445
21	I	0.415	0.310	0.095	0.3110	0.1125	0.0625

	Shell.weight	Rings
1	0.1410	8
2	0.2250	9
3	0.3005	10
4	0.0910	6
5	0.5385	9
6	0.3850	10
7	0.4660	9
8	0.1610	7
9	0.3550	9
10	0.1685	11

11	0.3550	12
12	0.3400	16
13	0.3500	10
14	0.0770	6
15	0.3700	12
16	0.2450	12
17	0.5030	10
18	0.1650	12
19	0.1335	16
20	0.3000	17
21	0.1150	8

group_by

Funkcja `group_by` odpowiada w moim przekonaniu za całą magię pakietu `dplyr`. Można o niej myśleć jako o funkcji `split` na doping. Pozwala ona podobnie jak `split` podzielić ramkę danych ze względu na wartości w jakiejś kolumnie. Jej przewagą jest to, że na obiekcie, który zwraca ta funkcja możemy dokonywać wszystkich operacji, jakich dokonywaliśmy za pomocą pakietu `dplyr` na ramkach danych.

W poniższym przykładzie wykorzystaliśmy funkcję `group_by` aby podzielić naszą ramkę danych na ramki ze względu na płeć naszych zwierzątek. Następnie użyliśmy funkcji `summarise` by uzyskać informacje o średniej wadze ze skorupą, po zdjęciu skorupy i średniej wadze skorupy. Dzięki temu, że użyliśmy jej na obiekcie zwracanym przez funkcję `group_by`, dokonywaliśmy wszystkich operacji osobno na każdej “podramce”.

```
by_sex <- group_by(df, Sex)
weight_means <- summarise(by_sex,
  count = n(), # specjalna funkcja, która zwraca liczbę
    ↪ obserwacji
  w.weight.mean = mean(Whole.weight),
  s.weight.mean = mean(Shucked.weight),
  v.weight.mean = mean(Shell.weight))
weight_means
```

```
# A tibble: 3 x 5
  Sex    count w.weight.mean s.weight.mean v.weight.mean
<chr> <int>      <dbl>         <dbl>         <dbl>
1 F      1307      1.05          0.446          0.302
2 I      1342      0.431          0.191          0.128
3 M      1528      0.991          0.433          0.282
```

Tym, co stanowi o sile pakietu `dplyr` jest to, że poszczególne operacje możemy łączyć w łańcuszki. W tym krótkim fragmencie kodu wykonaliśmy następujące operacje (w kolejności):

1. Dodaliśmy nową kolumnę do ramki danych `Age`, która jest liczbą “kręgów” powiększoną o 1.5
2. Pogrupowaliśmy nasze obserwacje ze względu na wiek (kolumna `Age`)
3. Wyliczyliśmy rozstęp międzykwartkowy dla wysokości i wagi całkowitej z podziałem na wiek
4. Uporządkowaliśmy uzyskane wyniki malejąco ze względu na kolumny z rozstępami międzykwartkowymi wzrostu i wagi (proszę zobaczyć 2 i 3 wiersz - w kolumnie `height.iqr` jest remis, więc kolejność między nimi ustala wartość z kolumny `weight.iqr`).

```
arrange(  
  summarise(  
    group_by(  
      mutate(  
        df,  
        Age = Rings+1.5  
      ),Age),  
    height.iqr = IQR(Height),  
    weight.iqr = IQR(Whole.weight)),  
  desc(height.iqr), desc(weight.iqr))
```

```
# A tibble: 28 x 3  
  Age height.iqr weight.iqr  
  <dbl>      <dbl>      <dbl>  
1  24.5     0.045      0.409  
2  23.5     0.0425     0.916  
3  14.5     0.0425     0.601  
4  18.5      0.04      0.554  
5  17.5      0.04      0.532  
6  13.5     0.0400     0.675  
7  11.5     0.0400     0.537  
8  15.5     0.0400     0.535  
9  16.5     0.0400     0.495  
10 22.5     0.0363     0.584  
# i 18 more rows
```

Tak skonstruowany kod czyta się i pisze jednak dość trudno. Aby zrekonstruować kolejność wykonywanych operacji, musimy zacząć “od środka”, co jest niewygodne. Z tego względu preferowanym w `dplyr` sposobem łączenia operacji w łańcuchy jest specjalny operator `%>%`.

Cała magia tego operatora polega na tym, że przekazuje argument po lewej stronie jako pierwszy argument funkcji po prawej stronie operatora. Zilustrujmy to na najprostszym przykładzie:

```
x <- c(1, 2, 4, 3, 5, 3, 2, 1, 5, 6, 1, 0, 12, 3, 4, 6)
mean(x)
## [1] 3.625
x %>% mean()
## [1] 3.625
```

Wywołaliśmy funkcję `mean` bez żadnego argumentu, ale operator `%>%` pozwolił nam przekazanie lewego operandu jako pierwszego argumentu funkcji stanowiącej prawy operand. Dzięki temu możemy zapisywać łańcuszki funkcji w bardziej czytelny sposób, który odpowiada rzeczywistej kolejności operacji. Poniższy kod robi dokładnie to samo, co kod wyżej, jest jednak bardziej czytelny, ponieważ odpowiada faktycznej kolejności operacji:

1. Dodaliśmy nową kolumnę do ramki danych `Age`, która jest liczbą “kręgów” powiększoną o 1.5
2. Pogrupowaliśmy nasze obserwacje ze względu na wiek (kolumna `Age`)
3. Wyliczyliśmy rozstęp międzykwartkowy dla wysokości i wagi całkowitej z podziałem na wiek
4. Uporządkowaliśmy uzyskane wyniki malejąco ze względu na kolumny z rozstępami międzykwartkowymi wzrostu i wagi.

```
df %>%
  mutate(Age = Rings+1.5) %>%
  group_by(Age) %>%
  summarise(height.iqr = IQR(Height),
            weight.iqr = IQR(Whole.weight)
  ) %>%
  arrange(desc(height.iqr), desc(weight.iqr))
```

```
# A tibble: 28 x 3
   Age height.iqr weight.iqr
<dbl>      <dbl>      <dbl>
1  24.5     0.045     0.409
2  23.5     0.0425    0.916
3  14.5     0.0425    0.601
4  18.5      0.04     0.554
5  17.5      0.04     0.532
6  13.5     0.0400    0.675
7  11.5     0.0400    0.537
```

```

8  15.5      0.0400      0.535
9  16.5      0.0400      0.495
10 22.5      0.0363      0.584
# i 18 more rows

```

pivot_longer i pivot_wider

Całkiem często podczas pracy z danymi będziemy potrzebować zmiany formatu danych. Wyobraźmy sobie sytuację, w której przeprowadziliśmy eksperyment, w którym chcieliśmy zbadać wpływ spożycia kawy na kompetencje matematyczne. W tym celu zarekrutowaliśmy 5 badanych i daliśmy im do rozwiązania test matematyczny trzy razy: przed wypiciem kawy, po wypiciu pierwszej filiżanki oraz po wypiciu dwóch filiżanek.

```

data <- data.frame(participant = c(1, 2, 3, 4, 5),
                   gender      = c("M", "M", "K", "K", "M"),
                   pre_coffee   = c(3, 4, 3, 4, 5),
                   first_coffee = c(5, 4, 4, 3, 6),
                   second_coffee = c(7, 5, 7, 5, 8)
                   )
data

```

	participant	gender	pre_coffee	first_coffee	second_coffee
1	1	M	3	5	7
2	2	M	4	4	5
3	3	K	3	4	7
4	4	K	4	3	5
5	5	M	5	6	8

Taki format danych często nazywany jest w praktyce szerokim (*wide*) formatem danych. W tym formacie jeden wiersz tabeli odpowiada jednemu uczestnikowi eksperymentu (albo badalnemu przedmiotowi, etc.). Zwróćmy uwagę, że każdy wiersz zawiera więcej niż jeden pomiar. Wiele funkcji w R wymaga jednak, aby w przekazywanej do niej ramce danych jeden wiersz odpowiadał jednemu pomiarowi. Taki format nazywa się formatem długim (*long*).

Za pomocą funkcji `pivot_longer` z biblioteki `tidyr` możemy przekształcić naszą ramkę danych do formatu długiego. Funkcja ta przyjmuje wiele argumentów, wśród których najważniejsze to:

- `data` - ramka danych, którą chcemy przetransformować do postaci długiej
- `cols` - kolumny, w których znajdują się nasze obserwacje
- `names_to` - nazwa nowej kolumny, w której znajdą się nazwy kolumn z obserwacjami
- `values_to` - nazwa nowej kolumny, w której znajdą się wartości z kolumn z obserwacjami

```
library(tidyr)
data_long <- pivot_longer(data = data,
                           cols = c("pre_coffee", "first_coffee",
                                   ↪ "second_coffee"),
                           names_to = "measurement_point",
                           values_to = "value"
                           )

data_long
```

```
# A tibble: 15 x 4
  participant gender measurement_point value
      <dbl> <chr>    <chr>          <dbl>
1         1 M      pre_coffee        3
2         1 M      first_coffee       5
3         1 M      second_coffee      7
4         2 M      pre_coffee        4
5         2 M      first_coffee       4
6         2 M      second_coffee      5
7         3 K      pre_coffee        3
8         3 K      first_coffee       4
9         3 K      second_coffee      7
10        4 K      pre_coffee        4
11        4 K      first_coffee       3
12        4 K      second_coffee      5
13        5 M      pre_coffee        5
14        5 M      first_coffee       6
15        5 M      second_coffee      8
```

Operacja ta jest odwracalna. Do przetransformowania formatu danych z długiego na szeroki służy funkcja `pivot_wider`. Aby jej użyć musimy przekazać tej funkcji kilka argumentów:

- `data` - ramka danych, którą chcemy przetransformować do postaci szerokiej
- `id_cols` - kolumny identyfikujące nową jednostkę obserwacji (tutaj: badanego)
- `names_from` - nazwa kolumny, w której znajdują się nazwy poszczególnych kolumn w nowej ramce danych
- `values_from` - nazwa kolumny, w których znajdują się wartości poszczególnych pomiarów

```
data_wide <- pivot_wider(data = data_long,
                          id_cols = c("participant", "gender"),
                          names_from = "measurement_point",
                          values_from = "value"
                          )

data_wide
```

```
# A tibble: 5 x 5
  participant gender pre_coffee first_coffee second_coffee
      <dbl> <chr>      <dbl>      <dbl>      <dbl>
1           1 M         3         5         7
2           2 M         4         4         5
3           3 K         3         4         7
4           4 K         4         3         5
5           5 M         5         6         8
```