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 potrafią 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:

Figure 1: Uchowiec

Informacje na temat danych mogą Państwo znaleźć tutaj: http://archive.ics.uci.edu/ml/datasets/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 składnie. 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. Spróbujmy 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 więcej niż 0.24mm wysokości oraz więcej niż 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 porzadkować 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 przyjmuję 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 pomoca 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 funkcjonalnościa 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 dwóch 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 Whole.weight Shucked.weight Viscera.weight Shell.weight
## 1   M  0.095       0.5140         0.2245         0.1010        0.150
## 2   M  0.090       0.2255         0.0995         0.0485        0.070
## 3   F  0.135       0.6770         0.2565         0.1415        0.210
## 4   M  0.125       0.5160         0.2155         0.1140        0.155
## 5   I  0.080       0.2050         0.0895         0.0395        0.055
## 6   I  0.095       0.3515         0.1410         0.0775        0.120
```

Możemy również wybrać wszystkie kolumny z wyjątkiem jakichś. Posługując się składnią `nazwa_kolumny:_nazwa_kolumny` wybieramy wszystkie kolejne kolumny pomiędzy dwoma okreslonymi 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 zwierzatek.

```
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 "kręgów") 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`). Korzystajac z wiedzy dotyczącej relacji między ilością "kręgów" 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 wyłacznie 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 probę z ramki danych. Różnią się tym, że `sample_n` pozwala wylosować n obserwacji, a `sample_frac` pozwala wylosować liczę obserwacji stanowiącą jakiś ułamek wszystkich (w przykładzie - 0.05 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    M  0.390    0.285  0.095       0.2710         0.1100         0.0600
## 2    F  0.575    0.475  0.160       0.8950         0.3605         0.2210
## 3    M  0.540    0.405  0.125       0.8910         0.4815         0.1915
## 4    I  0.445    0.320  0.120       0.3780         0.1520         0.0825
## 5    M  0.645    0.495  0.190       1.5390         0.6115         0.4080
## 6    F  0.580    0.445  0.135       0.9500         0.4840         0.1820
## 7    I  0.360    0.265  0.075       0.1845         0.0830         0.0365
## 8    M  0.630    0.490  0.165       1.2005         0.5750         0.2730
## 9    I  0.315    0.230  0.090       0.1285         0.0430         0.0400
## 10   F  0.610    0.485  0.210       1.3445         0.5350         0.2205
## 11   M  0.590    0.470  0.135       1.1685         0.5390         0.2790
## 12   M  0.365    0.295  0.080       0.2555         0.0970         0.0430
## 13   M  0.530    0.410  0.125       0.7690         0.3460         0.1730
## 14   I  0.395    0.295  0.090       0.3025         0.1430         0.0665
## 15   F  0.595    0.480  0.200       0.9750         0.3580         0.2035
##    Shell.weight Rings
## 1        0.0800     8
## 2        0.2710     9
## 3        0.2020     9
## 4        0.1200     8
## 5        0.4450    12
## 6        0.2325     8
## 7        0.0550     7
## 8        0.2940    10
## 9        0.0400     7
## 10       0.5150    20
## 11       0.2800     8
## 12       0.1000     7
## 13       0.2150     9
## 14       0.0765     5
## 15       0.3400    15
```

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

```
##    Sex Length Diameter Height Whole.weight Shucked.weight Viscera.weight
## 1    F  0.580    0.465  0.165       1.1015         0.4040         0.2095
## 2    F  0.535    0.420  0.130       0.6990         0.3125         0.1565
## 3    F  0.720    0.575  0.215       2.2260         0.8955         0.4050
## 4    F  0.525    0.390  0.135       0.6005         0.2265         0.1310
## 5    M  0.750    0.595  0.205       2.2205         1.0830         0.4210
## 6    I  0.465    0.355  0.090       0.4325         0.2005         0.0740
## 7    M  0.535    0.410  0.120       0.6835         0.3125         0.1655
## 8    M  0.280    0.200  0.080       0.0915         0.0330         0.0215
## 9    F  0.595    0.470  0.165       1.0155         0.4910         0.1905
## 10   I  0.515    0.420  0.150       0.6725         0.2555         0.1335
## 11   M  0.690    0.515  0.180       1.8445         0.9815         0.4655
```

```
## 12   F  0.600    0.460  0.145       0.9325          0.3985          0.2245
## 13   I  0.430    0.345  0.115       0.4295          0.2120          0.1080
## 14   F  0.505    0.390  0.120       0.5725          0.2555          0.1325
## 15   I  0.300    0.220  0.065       0.1235          0.0590          0.0260
## 16   F  0.705    0.560  0.205       2.3810          0.9915          0.5005
## 17   M  0.715    0.525  0.200       1.8900          0.9500          0.4360
## 18   I  0.415    0.305  0.120       0.3360          0.1650          0.0760
## 19   F  0.550    0.430  0.140       0.8105          0.3680          0.1610
## 20   F  0.725    0.580  0.185       1.5230          0.8045          0.3595
## 21   I  0.355    0.280  0.085       0.2905          0.0950          0.0395
##    Shell.weight Rings
## 1        0.3500    11
## 2        0.2035     8
## 3        0.6200    13
## 4        0.2100    16
## 5        0.6300    12
## 6        0.1275     9
## 7        0.1590     8
## 8        0.0300     5
## 9        0.2890     9
## 10       0.2350    10
## 11       0.3410    13
## 12       0.2480     8
## 13       0.1090     8
## 14       0.1460     8
## 15       0.0315     5
## 16       0.6240    10
## 17       0.4305    10
## 18       0.0805     7
## 19       0.2750     9
## 20       0.4375     9
## 21       0.1150     7
```

group_by

Funkcja `group_by` odpowiada w moim przekonaniu za całą magię pakietu `dplyr`. Można o niej myśleć jako o funkcji `split` na dopingu. 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 na ramki ze wzgledu na płeć naszych zwierzatek. 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 międzyćwiartkowy dla wysokości i wagi całkowitej z podziałem na wiek
4. Uporządkowaliśmy uzyskane wyniki malejąco ze względu na kolumny z rozstępami międzyćwiartkowymi 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
## # ... with 18 more rows
## # i Use `print(n = ...)` to see more rows
```

Tak skonstruowany kod czyta się i pisze jednak dość trudno. Aby zrekonstruować kolejność wykonywanych operacji, musimy zaczać "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 międzyćwiartkowy dla wysokości i wagi całkowitej z podziałem na wiek
4. Uporządkowaliśmy uzyskane wyniki malejąco ze względu na kolumny z rozstępami międzyćwiartkowymi 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
## # ... with 18 more rows
## # i Use `print(n = ...)` to see 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 czesto nazywany jest w praktyce szerokim (*wide*) formatem danych. W tym formacie

jeden wiersz tabeli odpowiada jednemu uczestnikowi eksperymentu (albo badanemu 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 nasza 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 odwracalana. 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
```