Funkcje. `apply`, `lapply`, `sapply`, `tapply`

Statystyka I z R

Bartosz Maćkiewicz

W R funkcje pisze się dość podobnie, co w innych językach programowania. W przeciwieństwie do niektórych z nich, w R funkcji nie deklaruje się za pomocą specjalnie do tego celu przeznaczonej składni. Funkcje są w R takimi samymi obiektami jak wszystko inne. Obiekty takie tworzymy za pomocą funkcji `function`.

```
witaj <- function() print('Witaj świecie')
witaj()
```

```
## [1] "Witaj świecie"
```

Zobaczmy, jaki typ ma wartość zmiennej `witaj`

```
class(witaj) # spodziewamy się 'function'
```

```
## [1] "function"
```

Funkcje mogą przyjmować argumenty. Ile i jakie - o tym decydują argumenty, jakie przekażemy `function`. Wywołując daną funkcję możemy przekazać jej argumenty albo pozycyjnie, albo przez nadaną im nazwę.

```
witaj_2 = function(name){
  powitanie = paste('Witaj', name)
  print(powitanie)
}

witaj_2('Bartoszu') # przekazanie wartości argumentu pozycyjnie
```

```
## [1] "Witaj Bartoszu"
```

```
witaj_2(name = 'świecie') # przekazanie wartości argumentu przez nazwę
```

```
## [1] "Witaj świecie"
```

Możemy dość swodobnie mieszać sposoby przekazywania argumentów funkcji. Staramy się jednak tego nie robić, ponieważ bardzo utrudnia to czytanie kodu.

```
dodaj_cztery = function(v,x,y,z) print(v+x+y+z)
dodaj_cztery(2,y=4,2,6)
```

```
## [1] 14
```

Kiedy deklarujemy funkcję, możemy jej argumentom przypisać domyślną wartość, jaką będą przyjmować jej argumenty. Zilustrujmy to na przykładzie. Funkcja `witaj_3` przyjmuje jeden argument (`name`), którego domyślną wartością jest `"świecie"`.

```
witaj_3 = function(name = 'świecie'){
  powitanie <- paste('Witaj', name)
  print(powitanie)
}
```

Jeśli wywołamy tę funkcję bez żadnych argumentów, to `name` przyjmie wartość `"świecie"`.

```
witaj_3() # bez żadnych argumentów, używa domyślnego
```

```
## [1] "Witaj świecie"
```

Możemy jednak wciąż przekazać naszej funkcji argument.

```
witaj_3('Kasiu') # nadpisujemy domyślny argument
```

```
## [1] "Witaj Kasiu"
```

Zwróćmy uwagę na to, że R ewaluuje funkcje leniwie - jeśli nie jest potrzebny jakiś argument, to przy wykonywaniu funkcji R nie zaprotestuje, jeżeli go nie podaliśmy. Dopiero wówczas, gdy "zabraknie" jakiejś wartości w czasie wykonywania funkcji, zwróci błąd. Zilustrujmy to na przykładzie. Funkcja `witaj_4` przyjmuje argument `lesmian`, ale wykorzystuje go jedynie wtedy, gdy argument `name` przyjmuje wartość `"świecie"`.

```
witaj_4 <- function(name = 'świecie', lesmian){
  if(name == 'świecie'){
  powitanie <- ifelse(lesmian,
                      paste('Witaj niedobry', name),
                      paste('Witaj', name))
  }else{
    powitanie <- paste('Witaj', name)
  }
  print(powitanie)
}
```

Jeżeli wywołamy funkcję `witaj_4` przekazując `"Bartoszu"` jako wartość `name` i jednocześnie nie podamy wartości argumentu `lesmian`, to R nie zaprotestuje!

```
witaj_4('Bartoszu')
```

```
## [1] "Witaj Bartoszu"
```

Tak samo R nie zaprotestuje, jeśli nie podamy argumentu `name` a podamy argument `lesmian`, ponieważ `name` przyjmuje domyślną wartość `"świecie"`.

```
witaj_4(lesmian = TRUE)
```

```
## [1] "Witaj niedobry świecie"
```

Problem pojawia się, jeśli nie podamy obu argumentów. Wtedy `name` przyjmuje wartość `"świecie"`. Na skutek tego funkcja sprawdza wartość argumentu `lesmian`. Jeśli wywołując funkcje nie przekaliśmy żadnej wartości, to R zaprotestuje.

```
witaj_4() # proszę sprawdzić, co dzieje się, jeśli wykonamy tę linijkę kodu
```

Do tej pory wynikiem wykonania naszej funkcji było wydrukowanie na ekranie jakiegoś napisu. Z reguły jednak będziemy chcieli, żeby funkcja nie drukowała niczego na ekranie, ale zwracała wartości.

Tworząc funkcje w R możemy zwracać wartości *explicite* albo *implicite*. W przypadku jawnego zwracania wartości robimy to za pomocą funkcji `return`. W przypadku niejawnego zwróconą wartością jest wartość, do której ewaluuje się ostatnie wyrażenie w naszej funkcji. Zilustrujmy to na przykładzie prostej funkcji, która przyjmuje jako argument wektor i zwraca sumę pierwszego i ostatniego elementu tego wektora.

```
n <- c(3,2,3,4,5,5,5,3,3,5,6)

# Funkcja explicite zwraca y
suma_skrajnych = function(x){
  y <- x[1] + x[length(x)]
```

```
  return(y)
}

# Funkcja zwraca wartość, do której ewaluuje się ostatnie wyrażenie
suma_skrajnych_alt = function(x) {
  x[1] + x[length(x)] # ostatnie wytażenie (pierwsze też...)
}

suma1 <- suma_skrajnych(n) # przypisujemy wartość zwracaną funkcji do zmiennej
suma2 <- suma_skrajnych_alt(n)

print(suma1 == suma2)
```

```
## [1] TRUE
```

Klasa funkcji `apply`

W R dostępne mamy specjalne funkcje o suffiksie `...apply`, których zadaniem jest zastosowanie funkcji do pewnej struktury danych według określonego wzorca. Na przykład możemy chcieć zastosować jakąś funkcję (np. funkcję obliczającą jakąś statystykę deskryptywną, przykładowo: średnią) do każdego wiersza lub kolumny ramki danych. Normalnie musielibyśmy napisać odpowiednią pętlę `for`. Za pomocą funkcji `apply` możemy zrobić to za pomocą jednego polecenia. Funkcje z klasy `...apply` są szczególnie przydatne podczas interaktywnej pracy z R, kiedy nie piszemy dłuższego kodu, ale eksplorujemy zbiór danych, wpisując odpowiednie polecenia w wierszu poleceń.

`apply`

`apply` jest podstawową funkcją służącą do aplikowania innych funkcji do struktur danych. Jako pierwszy argument przyjmuje obiekt macierzopodobny (tzn. strukturę danych, która ma wiersze oraz kolumny) i aplikuje podaną jako trzeci argument funkcję w kolumnach/wierszach, w zależności od wartości drugiego argumentu (`1` = wiersze, `2` = kolumny).

Zobaczmy jak działa funkcja `apply` na przykładzie zbioru danych `beaver1` oraz napisane wcześniej przez nas funkcji `suma_skrajnych`

```
head(beaver1)
```

```
##   day time  temp activ
## 1 346  840 36.33     0
## 2 346  850 36.34     0
## 3 346  900 36.35     0
## 4 346  910 36.42     0
## 5 346  920 36.55     0
## 6 346  930 36.69     0
```

```
print('Oto `suma_skrajnych` w wierszach')
```

```
## [1] "Oto `suma_skrajnych` w wierszach"
```

```
apply(beaver1, # struktura danych, do której chcemy zastosować funkcję
      1, # jak chcemy ją zastosować? 1 = dla każdego wiersza osobno
      suma_skrajnych) # funkcja, którą chcemy zaaplikować
```

```
##   [1] 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346
##  [19] 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346
##  [37] 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 346 347
##  [55] 346 346 346 346 346 346 346 346 346 346 346 346 346 347 346 346 346 346 346
##  [73] 346 346 346 346 346 346 346 347 346 346 347 346 346 347 346 346 346 346 346
##  [91] 346 347 347 347 347 347 347 347 347 347 347 347 347 347 347 347 347 347 347
```

```
## [109] 347 347 347 347 347 348
```

```
print('Oto `suma_skrajnych` w kolumnach')
## [1] "Oto `suma_skrajnych` w kolumnach"
apply(beaver1, 2, suma_skrajnych) # to samo, tylko że w kolumnach
##     day    time    temp   activ
##  693.00 1180.00   73.48    1.00
```

Spróbujmy użyć funkcji `apply`, aby stworzyć dodatkową kolumnę ze średnią z 3 testów w znanym nam zbiorze danych `student-mat.csv`, który dotyczy wyników uczniów na lekcjach matematyki. W kolumnach `G1`, `G2` oraz `G3` znajdziemy oceny z trzech testów. Jeśli chcemy dla każdego ucznia obliczyć średnią z tych trzech kolumn, to powinniśmy:

- wybrać z ramki danych interesujące nas kolumny (`df[, c('G1', 'G2', 'G3')]`)
- zastosować funkcję obliczająca średnią (`mean`) do każdego wiersza (`1`) w wybranym przez nas podzbiorze danych
- przypisać tak stworzony wektor do nowej kolumny w oryginalnej ramce danych `df['GMEAN'] <- ...`

```
df <- read.csv('student-mat.csv', header=TRUE, sep=';') # Wczytujemy dane

# jako pierwszy argument apply przekazujemy trzy kolumny - G1, G2, G3
# jako drugi argument apply przekazujemy wymiar - 1 oznacza wiersze, 2 oznacza kolumny
# jako trzeci argument apply przekazujemy funkcję, którą chcemy "aplikować"
df['GMEAN'] <- apply(df[,c('G1','G2','G3')], 1, mean)

head(df[,c('G1', 'G2', 'G3', 'GMEAN')])
```

```
##   G1 G2 G3     GMEAN
## 1  5  6  6  5.666667
## 2  5  5  6  5.333333
## 3  7  8 10  8.333333
## 4 15 14 15 14.666667
## 5  6 10 10  8.666667
## 6 15 15 15 15.000000
```

W jaki sposób przekazać dodatkowe argumenty funkcji? Załóżmy, że chcemy wyliczyć średnią przyciętą. Normalnie robimy to przekazując dodatkowy argument `trim` w wywołaniu funkcji `mean` (np. `mean(x, trim = 0.5)`. Kiedy używamy funkcji `apply` dodatkowe argumenty przekazujemy po argumencie, w którym przekazujemy funkcję do zaaplikowania:

```
df['GTMEAN'] <- apply(df[,c('G1','G2','G3')], 1, mean, trim=0.5) # przekazujemy
head(df[,c('G1', 'G2', 'G3', 'GTMEAN')])
```

```
##   G1 G2 G3 GTMEAN
## 1  5  6  6      6
## 2  5  5  6      5
## 3  7  8 10      8
## 4 15 14 15     15
## 5  6 10 10     10
## 6 15 15 15     15
```

lapply

`lapply` jest konceptualnie nieco prostszą funkcją niż `apply`. Jako pierwszy argument przyjmuje ona wektor lub listę i zwraca jako wynik listę. Zilustrujmy działanie `lapply` pisząc nową funkcję, która działa dokładnie tak jak znana nam funkcja `class` (zwraca typ przekazanego jej argumentu), ale możemy zastosować ją do

dowolnej liczby argumentów. Podstawowa idea jest taka, że za pomocą `lapply` dla każdego przekazanego argumentu wywołujemy funkcję `class`.

```
# możemy z `...` stworzyć listę, pozwala nam to na tworzenie funkcji przyjmującej dowolną liczbę argume
multi_str <- function(...){
  lapply(list(...), class) # o tutaj jest najważniejsze!
}
multi_str('Hulaj', c('Dusza', 'piekła'), FALSE)
```

```
## [[1]]
## [1] "character"
##
## [[2]]
## [1] "character"
##
## [[3]]
## [1] "logical"
```

Jak widzimy jako wynik działania funkcja `multi_str` zwróciła trzyelementową listę. Każdy element odpowiada typowi przekazanego do `multi_str` argumentu.

Rozważmy jeszcze jeden przykład. Pamiętamy, że ramki danych są tak naprawdę listami, dlatego możemy z powodzeniem jako argument funkcji `lapply` przekazać ramkę. Obliczmy więc za pomocą `lapply` średnią dla kazdej kolumny ramki danych `mtcars`:

```
head(lapply(mtcars, mean), 3)
```

```
## $mpg
## [1] 20.09062
##
## $cyl
## [1] 6.1875
##
## $disp
## [1] 230.7219
```

sapply

`sapply` to tak naprawdę "wrapper" dla funkcji `lapply`. To co różni `sapply` od `lapply` to fakt, że `lapply` zwraca *listę*, a `sapply` domyślnie stara się zwrócić *wektor*. Zilustrujmy to zachowanie prostym przykładem:

```
y <- c(1,2,3)

# Zwraca listę
print(lapply(y, function(x) (x+2)/2))
```

```
## [[1]]
## [1] 1.5
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 2.5
```

```
# Zwraca wektor
print(sapply(y, function(x) (x+2)/2))
```

```
## [1] 1.5 2.0 2.5
```

```
print(sapply(y, function(x) (x+2)/2, simplify = FALSE)) # zrobi to samo co lapply
```

```
## [[1]]
## [1] 1.5
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 2.5
```

tapply

Funkcja `tapply` pozwala nam pogrupować wartości w grupy na podstawie poziomów wektora typu `factor` i do każdej grupy zastosować funkcję. W poniższym przykładzie grupujemy wartości z kolumny `GMEAN` według zawodu ojca (`Fjob`) i dla każdej "grupy" obliczamy osobno średnią:

```
tapply(df$GMEAN, df$Fjob, mean)
```

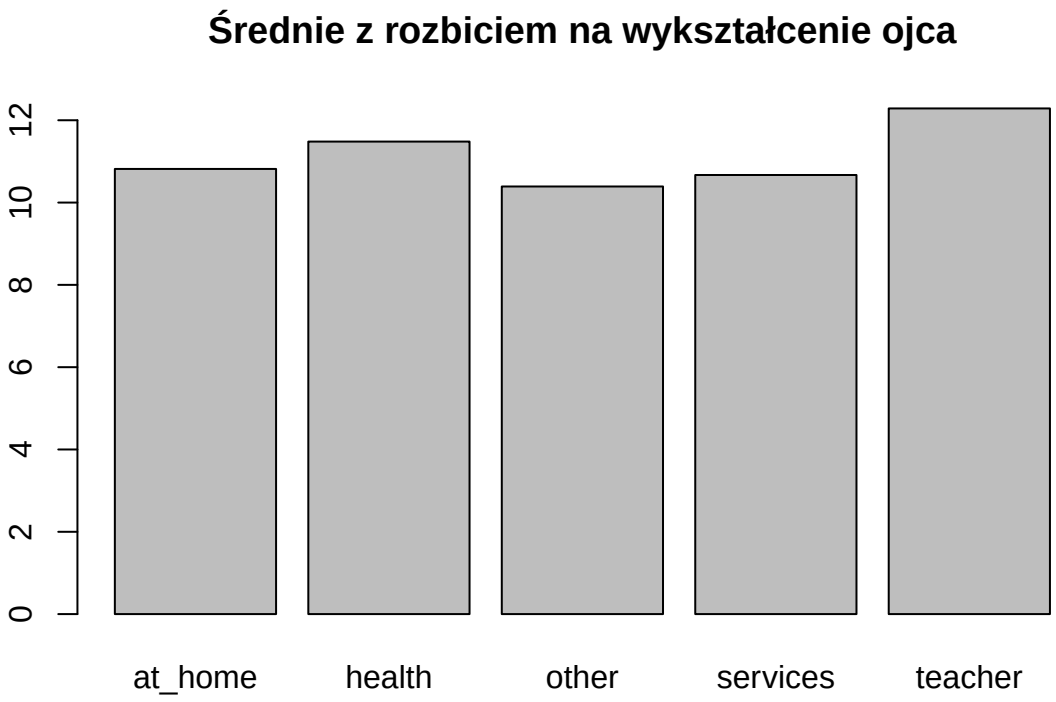
```
##  at_home   health    other services  teacher
## 10.81667 11.48148 10.39017 10.66967 12.28736
```

Możemy przekonać się, że najwyższą średnią z matematyki mają dzieci ojców, którzy pracują jako nauczyciele! Kto by się spodziewał...

Przykłady

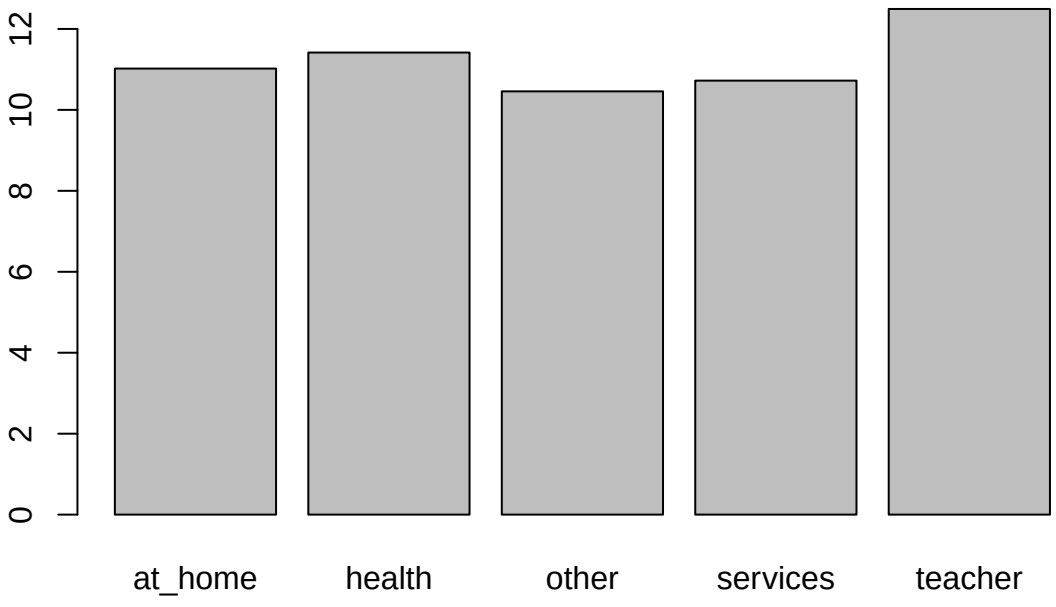
Spróbujmy zrobić coś ciekawego z naszą nowo nabytą wiedza. Zobaczmy jak rozkładają się wyniki ze względu na płeć. Na początek za pomocą `tapply` obliczymy średnie ze średnich wyników egzaminów (`GMEAN`) dla każdego zawodu ojca z osobna. Następnie korzystając z funkcji `barplot` naniesiemy obliczone wartości na wykres słupkowy. To chyba najprostszy sposób tworzenia prostych, eksploracyjnych wykresów!

```
srednie <- tapply(df$GMEAN, df$Fjob, mean) # z poprzedniego przykładu
barplot(srednie, main = "Średnie z rozbiciem na wykształcenie ojca") # skonstruujmy szybki wykres
```



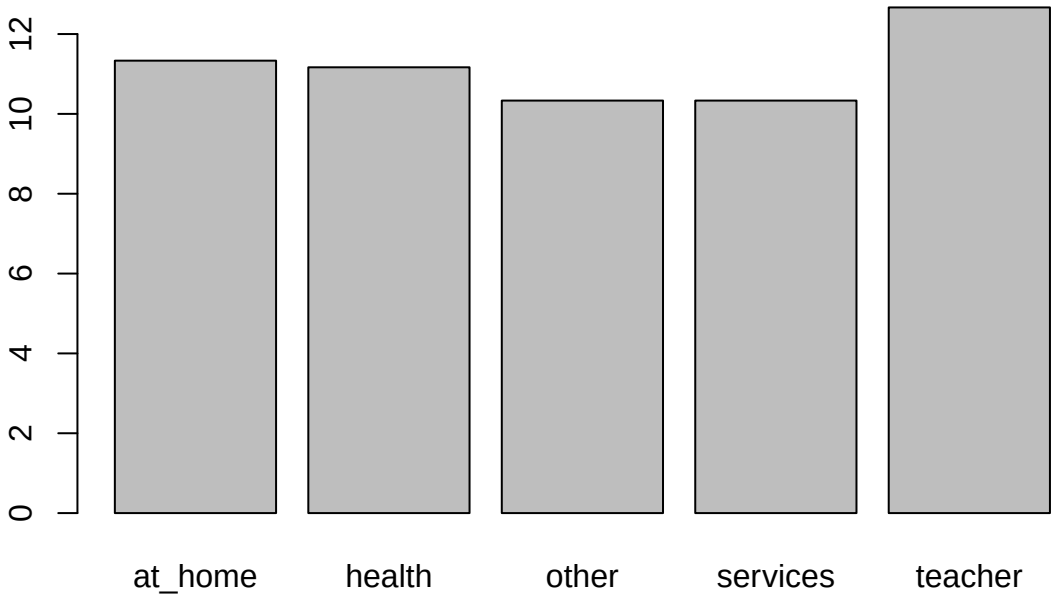
Możemy zrobić to samo używając średniej przyciętej:

```
srednie_przyciete <- tapply(df$GMEAN, df$Fjob, mean, trim = 0.1) # zobaczmy jak to wygląda z przyciętym
barplot(srednie_przyciete, main = "Średnie przyciete z rozbiciem na wykształcenie ojca")
```



A nawet użyć zamiast średniej mediany:

```
mediany <- tapply(df$GMEAN, df$Fjob, median)
barplot(mediany, main = "Mediany z rozbiciem na wykształcenie ojca")
```

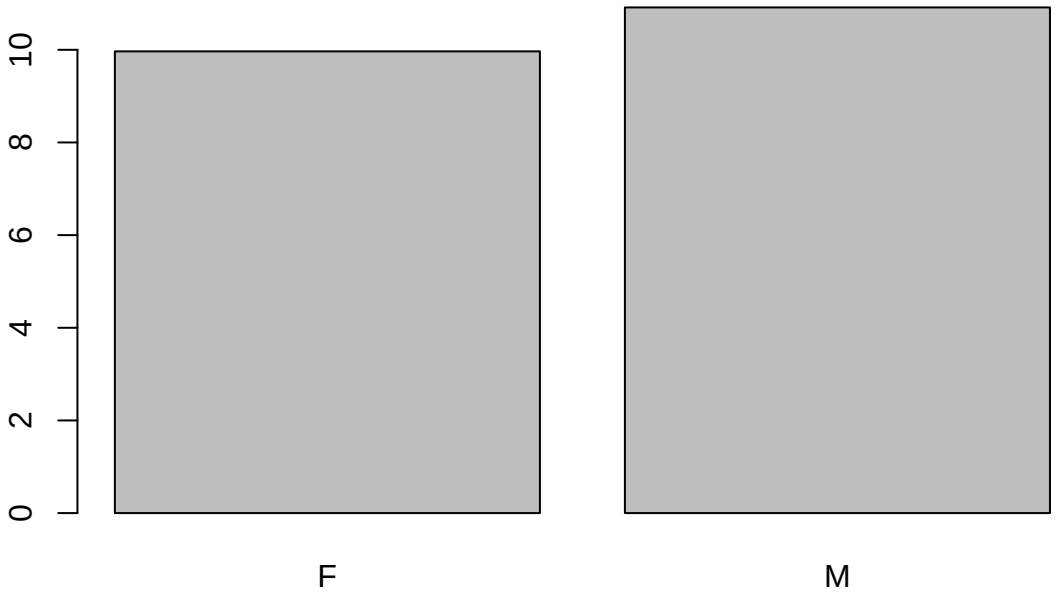


Dzięki temu, że funkcja `tapply` jest tak elastyczna, możemy zmieniając tylko jeden element kodu (przekazywaną funkcję) stworzyć prosty, eksploracyjny wykres dla praktycznie dowolnej statystyki deskryptywnej!

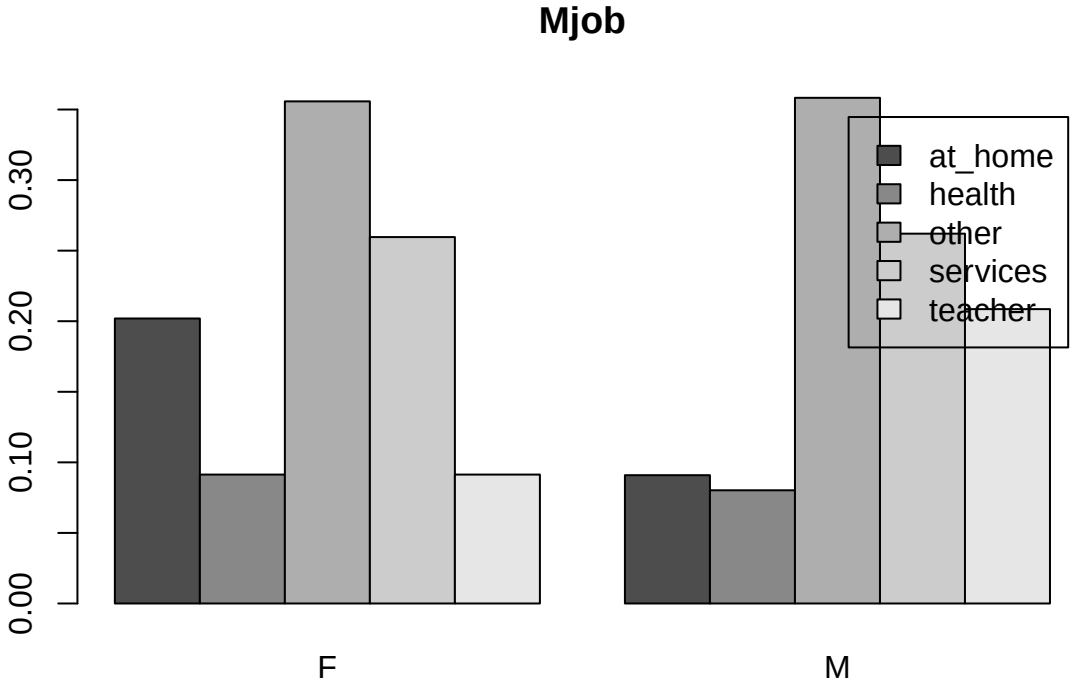
Spróbujmy użyć naszej wiedzy, żeby napisać sobie prosty skrypt, który wyświetla średnie lub proporcje wszystkich kolumn z podziałem na płeć. Nie będę wywoływał tej funkcji w notatniku, ponieważ produkuje ona BARDZO dużo wykresów, zachęcam jednak do uruchomienia ten funkcji na swoich komputerach!

```
# funkcja przyjmuje dwa argumenty - ramke danych, oraz nazwę kolumny
plot_z_plcia = function(df, kolumna){
    # jeżeli mamy wektor typu `numeric`, to chcemy obliczać średnie
    if (is.numeric(df[,kolumna])){
      # wyliczamy średnie z podziałem na płeć
      srednie <- tapply(df[,kolumna], df[,'sex'], mean)
      barplot(srednie, main = kolumna) # tworzymy wykres
    } else {
      # jeśli w kolumnie nie ma wartości liczbowych, to chcemy zliczać wartości i obliczyć proporcje
      # tworzymy więc tabele poznanym sposobem
      tabela <- prop.table(table(df[,kolumna], df[,'sex']), 2)
      # wykres z proporcjami
      barplot(tabela, main = kolumna, beside = TRUE, legend = rownames(tabela))
    }
}

# Dla kolumn z wartościami liczbowymi (wynik ostatniego testu z rozbiciem na płeć)
plot_z_plcia(df, "G3")
```




```
# Dla kolumn z wartościami nieliczbowymi (zawód matki z rozbiciem na płeć)
plot_z_plcia(df, "Mjob")
```




```
# odkomentować, aby zobaczyć wykresy...
#for (kolumna in colnames(df)){
    #plot_z_plcia(df, kolumna)
#}
```

Przeanalizujmy teraz napisaną przez nas funkcję krok po kroku.

Nasza funkcja przyjmuje dwa argumenty - pierwszy z nich to ramka danych (`df`), a drugi (`kolumna`) jest

nazwą kolumny. Zadaniem funkcji celem jest narysowanie wykresu z podziałem na płeć.

```
plot_z_plcia = function(df, kolumna){} #
```

Podstawowa struktura naszej funkcji opiera się na konstrukcji warunkowej. Co innego zrobimy z danymi, kiedy będą to liczby (rysujemy wykres słupkowy dla średnich), co innego, gdy będzie to `factor` (rysujemy wykres słupkowy dla proporcji).

```
plot_z_plcia = function(df, kolumna){
    if (is.numeric(df[,kolumna])){
    #...
    }else{
    #...
    }
}
```

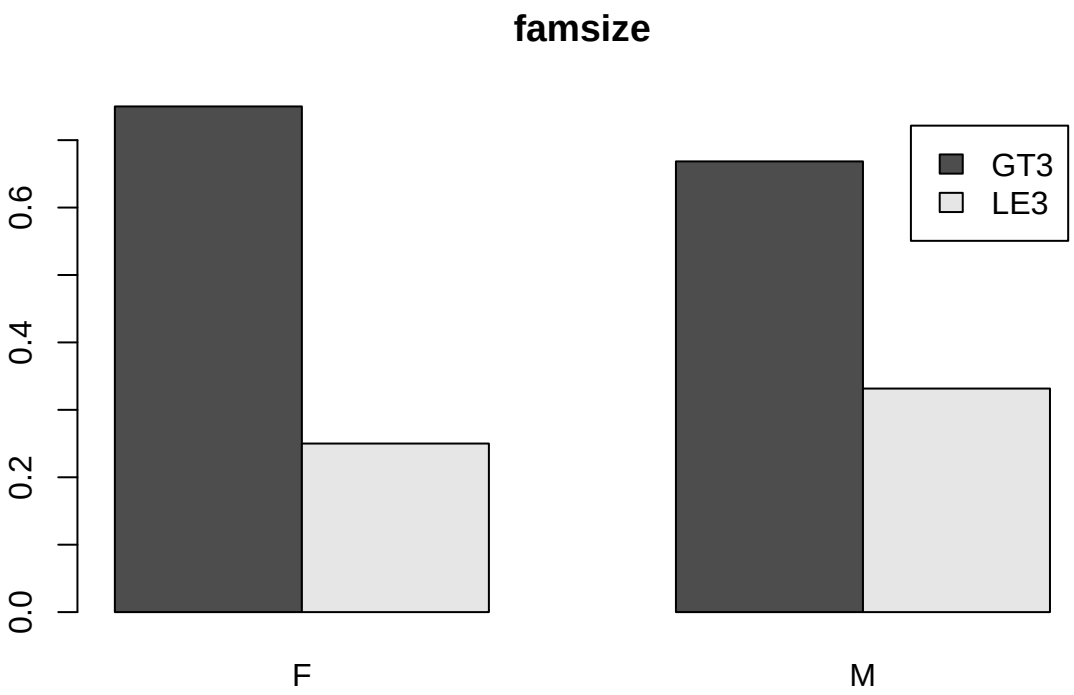
Przeanalizujmy, co dzieje się w bloku `if`, to znaczy wtedy gdy kolumna zawiera dane liczbowe:

```
df <- df # w naszej funkcji przekazujemy jako jeden argument
kolumna <- 'age' # w tej kolumnie mamy wartości liczbowe
srednie <- tapply(df[,kolumna], df[,'sex'], mean) # wyliczamy średnie z podziałem na płeć
barplot(srednie, main = kolumna) # tworzymy wykres
```

Jeśli kolumna zawiera dane nieliczbowe, musimy zmienić nasze podejście:

```
df <- df # w naszej funkcji przekazujemy jako jeden argument
kolumna <- 'famsize' # w tej kolumnie mamy factor

tabela <- prop.table(table(df[,kolumna], df[,'sex']),2) # tworzymy więc tabele znanym spsoobem
barplot(tabela, main = kolumna, beside = TRUE, legend = rownames(tabela)) # wykres z proporcjami
```



Ostatnim elementem jest (zakomentowana!) pętla for, która iteruje po nazwach wszystkich kolumn wywołując naszą funkcję.

```
for (kolumna in colnames(df)){
   #...
}
```

Jak widzimy R może nam posłużyć do pisania bardzo ciekawych oraz przydatnych funkcji!