Wprowadzenie do języka R (cz. 3.)

1) Funkcje anonimowe

1. W konsoli RStudio wpisujemy kolejno:

```
> inc <- function(x) x + 1
> inc(1)

> sumXY <- function(x,y) x + y
> sumXY(1,2)
```

2. Następnie wpisujemy:

```
> (function(x) x + 1)(1) # funkcja anonimowa i jej wywołanie
> (function(x,y) x + y)(1,2)
```

3. ***Zadania***:

    1. Napisz funkcje anonimowe (wyrażenia lambda) odpowiadające funkcjom:

        - $f_1(x) = x - 2$
        - $f_2(x, y) = \sqrt{(x^2 + y^2)}$
        - $f_3(x, y, z) = \sqrt{(x^2 + y^2 + z^2)}$

        a następnie sprawdź ich działanie dla wybranych wartości argumentów

    2. (*opcjonalne*) Napisz funkcje anonimowe odpowiadające:

        ```
        abs(x), id(x), maxOf(x,y), minOf(x,y), isEven(n), isOdd(n)
        ```

        a następnie sprawdź ich działanie

2) Funkcje wyższego rzędu: funkcje jako parametry/argumenty

1. W pliku `s31.R` wpisujemy

```
sumElems <- function(v) {
   sum <- 0
   for (e in v) {
      sum <- sum + e
   }
   sum
}
```

2. Zaznaczamy całą funkcję i klikamy przycisk `Run`

3. Testujemy działanie funkcji, np.

```
> sumElems(1:3)
> sumElems(list(1,2,3))
```

4. W pliku `s31.R` dodajemy

```
# suma elementów podniesionych do kwadratu
sumSqrOfElems <- function(v) {
   sum <- 0
   for (e in v) {
      sum <- sum + e^2
   }
   sum
}
```

5. Testujemy działanie funkcji, np.

```
> sumSqrOfElems(1:3)
> sumSqrOfElems(list(1,2,3))
```

6. W pliku `s31.R` dodajemy

```
# sumElemsWith - funkcja wyższego rzędu (jej parametr 'f' jest funkcj
ą)
sumElemsWith <- function(f, v) {
    sum <- 0
    for (e in v) {
        sum <- sum + f(e)
    }
    sum
}
```

7. Zaznaczamy całą funkcję, klikamy przycisk `Run` i testujemy jej działanie

```
> sumElemsWith(function(e) e, 1:3)
> sumElemsWith(function(e) e^2, 1:3)
> sumElemsWith(function(e) e^3, sample(1:10, 10)) # sprawdź opis funk
cji 'sample'
> sumElemsWith(function(e) sqrt(e), 1:3)
> sumElemsWith(function(e) if (e %% 2 == 0) 0 else e, 1:5) # sumOddEl
ems
```

8. ***Zadania***:

    1. Wykorzystując `sumElemsWith` (bez definiowania nowej funkcji) obliczyć w konsoli RStudio:

        - $\sum_{i=1}^{15} i^5$
        - $\sum_{i=1}^{10} \sqrt{(1+i^2)}$

    2. Napisz funkcję `prodElemsWith`, która oblicza iloczyn elementów przekształcanych zadaną funkcją

    3. (*opcjonalne*) Wykorzystując `sumElemsWith` napisać funkcję oobliczającą długość podanego (jako argument) wektora

    4. (bez definiowania nowej funkcji) obliczyć w konsoli RStudio:

        - $\sum_{i=1}^{15} i^5$
        - $\sum_{i=1}^{10} \sqrt{(1+i^2)}$

3) Funkcje wyższego rzędu: funkcje jako wyniki

1. W pliku `s31.R` dodajemy

```
funcFactory <- function(s) {
  if (s == 1) function(x) x
  else if (s == 2) function(x) x * x
  else if (s == 3) function(x) if (x != 0) 2 / x else 0
  else stop("Enter something between 1 and 3!")
}
```

2. Testujemy działanie funkcji (w konsoli RStudio), np.

```
> funcFactory(1)(5)
> funcFactory(2)(5)
> funcFactory(3)(5)
> funcFactory(4)(5)
```

3. ***Zadania***:

    1. Napisz funkcję

    ```
    expApproxUpTo n = ...
    ```

    zwracającą rozwinięcie funkcji e^x w szereg MacLaurina o długości `n+1, n < 6`, tzn. `expApproxUpTo n` $= \sum_{k=0}^{n} \frac{x^k}{k!}$

2. (*opcjonalne*) Napisz funkcję

```
dfr f h = ...
```

zwracającą dla zadanej funkcji `f` przybliżenie jej pochodnej obliczone wg schematu różnicowego $f'(x_0, h) \approx \frac{f(x_0+h)-f(x_0)}{h}$.

Sprawdź dokładność uzyskiwanych wyników w zależności od wartości `h`

4. (*opcjonalne*) Napisz funkcję

```
dfc f h = ...
```

zwracającą dla zadanej funkcji `f` przybliżenie jej pochodnej obliczone wg schematu różnicowego $f'(x_0, h) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$.

Sprawdź dokładność uzyskiwanych wyników w zależności od wartości `h` i porównaj z poprzednimi

5. (*opcjonalne*) Napisać funkcję

```
d2f f h = ...
```

obliczającą przybliżenie drugiej pochodnej funkcji `f`

4) Funkcje jako elementy struktur danych

1. W pliku `s31.R` dodajemy

```
functVec <- c(function(x) x, function(x) x^2, function(x) x^3)
```

2. Testujemy działanie funkcji, np.

```
> functVec[[1]](2)
> functVec[[2]](2)
> functVec[[3]](2)
```

3. W pliku `s31.R` dodajemy

```
# obliczenie wartości wszystkich funkcji z kolekcji 'fs' w punkcie 'x
'
 evalFuncListAt <- function(x, fs) {
    res <- c()
    for (f in fs) {
      res <- c(res, f(x))
```

```
    }
    res
}
```

4. Testujemy działanie funkcji, np.

```
> evalFuncListAt(3, functVec)
> evalFuncListAt(5, functVec)
```

5. ***Zadania***:

    1. Zdefiniuj nową kolejcję funkcji (podobnie jak w przypadku `functVec`) i (wykorzystując `evalFuncListAt`) oblicz wartości każdej z nich w kilku punktach

    2. Wywołaj funkcję `evalFuncListAt` z listą funkcji zdefiniowaną w argumencie wywołania

5) Funkcje wyższego rzędu: keep i discard

1. Dołączamy do sesji biblioteki `purrr` i `stringr`, np. w konsoli `RStudio`

```
library(purrr)
library(stringr)
```

2. W konsoli `RStudio` wpisujemy

```
> ?keep # analizujemy opis
> ?discard # analizujemy opis
> ?str_length # analizujemy opis
```

3. Następnie wpisujemy

```
> keep(1:10, function(x) x %% 2 == 0) # filtruj zarzymując parzyste
> discard(1:10, function(x) x %% 2 == 0) # filtruj odrzucając parzyst
e
> keep(c("a", "bb", "ccc"), function(s) str_length(s) > 1)
```

4. ***Zadania***:

    1. Napisz funkcje `isEven` i `isOdd`, a następnie wykorzystaj je w wywołaniach funkcji `keep` i `discard`, np.

    ```
    > keep(1:10, isEven)
    > keep(1:10, isOdd)
    ```

    2. (*opcjonalne*) Wykorzystując `keep` i/lub `discard` napisz funkcje `onlyOdd`

and `onlyEven` zwracające jako wynik odpowiednio tylko nieparzyste i parzyste elementy przekazanej jako argument listy

6) Funkcje wyższego rzędu: `map`

1. W konsoli `RStudio` wpisujemy

```
> ?map # analizujemy opis
> ?map2 # analizujemy opis
```

2. Następnie wpisujemy

```
> map(1:10, function(x) x^2)
> map_dbl(1:10, function(x) x^2)
```

```
> map_at(1:10, c(1,2), sqrt)
> map_if(1:5, function(e) e > 3, function(e) e^2)
```

3. W pliku `s31.R` dodajemy

```
dfXY <- data.frame(
   x = c(1, 2, 5),
   y = c(5, 4, 8)
)
```

4. W konsoli wpisujemy

```
> map(dfXY, median)
> map(dfXY, mean)
> map(dfXY, sd)
```

5. Następnie wpisujemy

```
> map2_dbl(df1$x, df1$y, function(x,y) x * y)
> map2_dbl(df1$x, df1$y, function(x,y) if (x <= y) x else y)
```

1. ***Zadania***:
    1. Napisz nową wersję funkcji `evalFuncListAt` , tak aby wykorzystywała funckcję `map`
    2. Przeanalizuj działanie

```
> map(1:10, function(x) sample(1:10, x))
```

7) Funkcje wyższego rzędu: `reduce`

1. W konsoli `RStudio` wpisujemy

```
?reduce # analizujemy opis
```

2. Następnie wpisujemy

```
> reduce(1:5, function(x,y) x + y)
> reduce(1:5, `+`)
> reduce(1:5, function(x,y) x * y)
> reduce(2:4, `^`)
> reduce_right(1:5, `-`)
> reduce(1:5, `-`)
```

3. ***Zadania***:

    1. (*opcjonalne*) Wykorzystując `reduce_right` napisz implementację funkcji `filter`
    2. (*opcjonalne*) Wykorzystując `reduce_right` napisz implementację funkcji `map`

8) Funkcje anonimowe i formuły

1. W konsoli `RStudio` wpisujemy

```
> map_dbl(1:10, function(x) 2 * x)
> map_dbl(1:10, ~.x*2)
> map_dbl(1:10, ~.*2)
> keep(1:10, ~.x>3)
> keep(1:10, ~.>3)
```

2. Następnie wpisujemy

```
> map2(1:5, 11:15, function(x,y) x + y)
> map2(1:5, 11:15, ~ .x + .y)
```

3. ***Zadania***:

    1. Przepisz poniższe wywołania funkcji wykorzystując formuły (zamiast funkcji anomimowych)

```
map_dbl(1:10, function(x) sqrt(x))
map_dbl(1:10, function(x) 2 * x^2 + x)
keep(1:10, function(x) x %% 2 == 0)
map2(1:5, 11:15, function(x,y) x * y + 2)
```

2. (*opcjonalne*) Odszukaj w dokumentacji sposób obsługi formuł

9) Wzorzec `collection pipeline` , operator `%>%`

1. W konsoli wpisujemy

```
> 1:10 %>% map_dbl(~ .+1)
> 1:10 %>% keep(~.x > 4 && .x < 8)
> 1:10 %>% keep(~. > 4 && . < 8)
> 1:5 %>% reduce(`+`)
```

2. Następnie wpisujemy

```
1:10 %>% keep(~.>5) %>% map_dbl(~.+1)
1:10 %>% keep(~.>5) %>% map_dbl(~.+1) %>% reduce (`+`)
```

3. ***Zadania***:

    1. Przepisz powyższe ‘potoki obliczeniowe’ wykorzystując funkcje anonimowe zamiast formuł

    2. Przepisz powyższe ‘potoki obliczeniowe’ bez użycia operatora `%>%`

10) Prezentacja wybranych możliwości pakietu `ggplot2`

1. W konsoli `RStudio` wpisujemy

```
> library(ggplot2)
> ?diamonds # zapoznajemy się z opisem
> str(diamonds)
> summary(diamonds)
> diamonds
> dsmall <- diamonds[sample(nrow(diamonds), 100), ]
```

2. Następnie wpisujemy

```
> ?qplot
> qplot(dsmall$carat, dsmall$price)
> qplot(log(carat), log(price), data = dsmall)
> qplot(carat, price, data = dsmall, colour = color)
> qplot(carat, price, data = dsmall, shape = cut)
> qplot(carat, price, data = dsmall, colour = I("red"), size = I(3),
alpha = I(1/5))
> qplot(carat, price, data = dsmall, colour = color, geom = "point")
```

3. Następnie wpisujemy

```
 > qplot(carat, price, data = dsmall, geom = "line")
 > qplot(carat, price, data = dsmall, colour = color, geom = "boxplot
")
 > qplot(carat, price, data = dsmall, colour = color, geom = "smooth"
)
 > qplot(carat, price, data = dsmall, geom = c("point", "smooth"))
 > qplot(carat, price, data = dsmall, geom = c("point", "smooth"), sp
an = 0.2)
 > qplot(carat, price, data = dsmall, geom = c("point", "smooth"), sp
an = 1)
 > qplot(color, price / carat, data = diamonds, geom = "jitter", alph
a = I(1 / 5))
 > qplot(carat, data = diamonds, geom = "histogram")
 > qplot(carat, data = diamonds, geom = "density")
 > qplot(color, data = diamonds, geom = "bar")
```

4. Następnie wpisujemy

```
 > ggplot(msleep, aes(sleep_rem / sleep_total, awake)) + geom_point()
 > qplot(sleep_rem / sleep_total, awake, data = msleep)
 > ggplot(mpg, aes(displ, hwy)) + geom_point()
 > qplot(displ, hwy, data = mpg)
 > qplot(displ, hwy, data = mpg) + geom_smooth()
 > ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth()
 > p1 <- ggplot(mpg, aes(displ, hwy))
 > summary(p1)
 > bestfit <- geom_smooth(method = "lm", se = F, colour = alpha("stee
lblue", 0.5), size = 2)
 > qplot(sleep_rem, sleep_total, data = msleep) + bestfit
 >
```

```
```