Vysoká škola ekonomická v Praze

R SNADNO A RYCHLE 2

VIZUALIZACE DAT A PROGRAMOVÁNÍ

Karel Šafr, Jakub Danko

2020

OECONOMICA
Nakladatelství VŠE

Autoři:

Ing. Karel Šafr, Ph.D. (Vysoká škola ekonomická v Praze)
Ing. Jakub Danko, PhD. (Vysoká škola ekonomická v Praze)

Recenzenti:

Mgr. Vítězslav Štembera, Ph.D. (Technická univerzita Vídeň)
Ing. Zdeněk Šulc, Ph.D. (Vysoká škola ekonomická v Praze)

ISBN 978-80-245-2381-1

Poděkování

Na tomto místě bychom rádi poděkovali zejména doc. RNDr. Ivaně Malé, CSc. (Vysoká škola ekonomická v Praze) za velké množství času, který nám věnovala při psaní této naší prvotiny. Bez jejích odborných připomínek, komentářů· a dotazů· by zcela jistě tato kniha nevznikla ve stávajícím formátu, rozsahu a kvalitě. Dále bychom chtěli poděkovat recenzentům této knihy, kterými jsou: Mgr. Vítězslav Štembera, Ph.D. (Technická univerzita Vídeň) a Ing. Zdeněk Šulc, Ph.D. (Vysoká škola ekonomická v Praze), za množství a podnětnost připomínek, které oba recenzenti vznesli, a velmi si vážíme toho, že věnovali tolik svého volného času detailnímu recenzování této knihy. Dále bychom rádi poděkovali Ing. Veronice Ptáčkové, MUDr. Leoně Karáskové, Květě a Nele Šafrové, které provedly prvotní gramatickou a slohovou korekturu této knihy. V neposlední řadě patří velký dík našim nejbližším a rodinám, které nás podporovaly a motivovaly při psaní této knihy.

V neposlední řadě je nutno zmínit, že vznik knihy byl podpořen projektem interní rozvojové soutěže VŠE v Praze (IRS) F4/18/2019.

Autoři

Obsah

Úvod

Vážené čtenářky, vážení čtenáři,

dostal se vám do rukou druhý díl knihy „R snadno a rychle“. V prvním díle knihy „R snadno a rychle 1 Úvod do jazyka“ jsme představili základy jazyka R. Primárně jsme se zaměřili na to, jak nadefinovat objekty, jaké mají vlastnosti, a vysvětlili jsme si základy práce v programovacím jazyku R. V knize, kterou jste nyní otevřeli, se již nebudeme zabývat základy, jako je i načítání dat, ale rovnou se budeme věnovat základům programování a vizualizace dat v R. Musíme vás tedy upozornit na to, že kniha se již nezabývá základy práce v jazyku R – definice objektů, import dat, operátor nebo základní funkce. Předpokládáme, že uživatelé zmíněné znalosti bez problémů ovládají. Kniha je tak vhodná pro pokročilé začátečníky, ať už mezi studenty, nebo veřejností, která si chce rozšířit znalosti a usnadnit si tak výpočty v pracovním životě.

V první kapitole vám představíme cykly, které jsou důležité zejména z pohledu automatizace zpracování řešené úlohy. V návaznosti na cykly si ukážeme, jak fungují podmínky. Pomocí nich totiž dokážeme kód tzv. větvit – podmínit chování v určité situaci. Uvedeme příklad: chceme, aby se pro sudá čísla realizoval jiný výpočet než pro lichá. Kapitola „Next, break a repeat cyklus“ uzavře dvě zmíněné kapitoly představením tzv. obecného cyklu „repeat“. V navazující kapitole („Funkce apply, by a aggregate“) vám vysvětlíme, jak fungují velmi oblíbené apply funkce a s nimi spojená agregace dat. Další kapitolou je „Tvorba vlastních funkcí“, která představí, jak lze v programovacím jazyku R vytvořit vlastní uživatelskou funkci. Kapitola „Základy tvorby grafů v jazyce R“ se věnuje tvorbě grafů pomocí základní knihovny a populární knihovny ggplot2. V předposlední kapitole „Funkce merge a knihovna dplyr“ si ukážeme, jak data spojit, agregovat nebo filtrovat. Poslední kapitola se zaměří na formátování kódu jazyka R.

Přejeme vám mnoho úspěchů při programování a analýze dat pomocí jazyka R. Doufáme, že se vám bude kniha líbit a usnadní vám práci v jazyce R.

Hodně štěstí!
Tým autorů

Kapitola 1

Cykly

V rámci předchozích kapitol jsme si představili základy práce s jazykem R. Ukázali jsme si, jak se pracuje s jednotlivými objekty, představili jsme si základní funkce a vysvětlili základy práce zpracování dat. V rámci navazujících kapitol si představíme pokročilejší programovací techniky. Výhodou programovacího jazyka R oproti statistickým softwarům je zejména automatizace, tvorba vlastních funkcí, grafické knihovny a pokročilejší správa dat. V této kapitole si představíme metody pro automatické opakování vykonávaného kódu neboli cykly.

Cyklus (neboli „smyčka“) je cyklicky se opakující sekvence kódu. Cyklicky se opakující část kódu se nachází uvnitř složených závorek za definicí funkcí cyklu. Cyklus je vykonán pro předem definovanou sekvenci hodnot, má stanovený počet opakování (`for` cyklus), anebo pokud je splněná podmínka, která povoluje opakování cyklu (`while` cyklus). Jedna realizace cyklu se nazývá iterací. Iteraci lze též v češtině nazvat průchod cyklem.

Cykly jsou součástí většiny programovacích jazyků a jedná se o jejich nejdůležitější komponentu. Výhodou cyklů je zejména automatizace procesů a zjednodušení zápisu kódu. Například postupné zobrazení hodnot od 1 do 10 bychom bez cyklu mohli zapsat jako postupné zadáváním příkazu `print()`.

```
> print(1)
[1] 1
> print(2)
[1] 2
> print(3)
[1] 3
> print(4)
[1] 4
> print(5)
[1] 5
> print(6)
```

```
[1] 6
> print(7)
[1] 7
> print(8)
[1] 8
> print(9)
[1] 9
> print(10)
[1] 10
```

Pomocí cyklu si můžeme výrazně usnadnit zápis kódu na jeden řádek. Stačí, když použijeme for cyklus a funkci `print()` vložíme do těla cyklu.

```
> for (i in 1:10) {print(i)}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Ne v každé situaci je nutné použít cyklus. Pokud by nám šlo o čisté zobrazení hodnot od jedné do desíti, můžeme použít funkci `seq()`. Funkce `seq()` vytváří sekvenci hodnot od definované první hodnoty (parametr from) do poslední hodnoty (parametr `to`). Délku sekvence můžeme ovlivnit pomocí parametru `length.out` nebo `by`. Parametr `length.out` specifikuje délku výsledného vektoru (velikost kroku si vypočítá funkce sama), anebo můžeme použít parametr `by`, který specifikuje délku kroku mezi jednotlivými prvky vektoru (a délka vektoru je výsledně získaná samotnou funkcí).

```
> seq(from = 2, to = 8, length.out = 4)
[1] 2 4 6 8
> seq(from = 2, to = 8, by = 1.5)
[1] 2.0 3.5 5.0 6.5 8.0
```

Cyklus najde své použití zejména v situaci, kdy chceme v rámci iterace provést více úloh, jež nelze provést pro celý vektor/objekt najednou.

1.1 For cyklus

Cyklus `for` je cyklus, u kterého je předem známý počet opakování. Cyklus je vykonáván pomocí tzv. iterátoru (iterační proměnná, popř. řídicí proměnná), který postupně nabývá hodnot z předem definované množiny. Množinou hodnot se rozumí obvykle prvky atomického vektoru hodnot, ale je možné iterovat i přes prvky listu. Prozatím se ale omezíme na atomický vektor hodnot.

Cyklus vytvoříme pomocí příkazu `for`. V kulatých závorkách za tímto příkazem je nutné definovat iterační proměnnou. Iterační proměnnou definujeme tak, že specifikujeme její jméno a dále její rozsah. Tedy například výraz `i in 1:10` způsobí, že iterační proměnná `i` bude nabývat hodnot od jedné do desíti v jednotlivých průchodech cyklem. Samotné tělo cyklu definujeme až následně. Standardně je tělo cyklu uzavřeno do bloku složených závorek (`{...}`). Tělo cyklu obsahuje kód, který se má v jednotlivých iteracích vykonat.

```
for (# definice iteracni promene) {
# telo cyklu
}
```

V předchozí kapitole jsme si ukázali for cyklus, kde se v každé iteraci zobrazila hodnota iterátoru. Cyklus lze ale rozepsat i na více řádek tak, jak ilustruje následující kód.

```
> for (i in 1:10) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Cyklus `for` lze zapsat i ve zkrácené formě. Jedná se o situaci, kdy lze kód napsat na jeden řádek. V tu chvíli můžeme vynechat složené závorky. Je nutné si ale uvědomit, že se iteruje pouze kód do konce řádku či do prvního středníku na daném řádku.

```
> for (i in 1:10) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

```
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Alternativně lze zapsat cykly i pomocí odsazení (bez složených závorek). Iteruje se zde pouze první odsazená řádka za definicí `for` cyklu.

```
> for (i in 1:10)
+   print(i)
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
>   print("ahoj")
[1] "ahoj"
```

Klíčovou roli v cyklu zastává řídicí proměnná (iterační). Jak již bylo řečeno, řídicí proměnná nabývá postupně hodnot z množiny, přes jejíž prvky iteruje. Tato řídicí proměnná nám může usnadnit přístup k dílčím prvkům jiných objektů. Toto si ukážeme na následujícím příkladu, kde vypíšeme jednotlivé sloupce matice `X`. Nejprve si ale matici i zobrazíme, abychom poznali rozdíl od výsledného výstupu z cyklu (pomocí funkce `print()`).

```
> # Definice matice:
> X <- matrix(1:9, 3)
> print(X)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
>
> # FOR cyklus:
> for (i in 1:ncol(X)) {
+   print(X[, i])
+ }
[1] 1 2 3
[1] 4 5 6
[1] 7 8 9
```

Tento kód provede iterace od jedné do počtu sloupců proměnné (X) a zobrazí hodnoty v těchto sloupcích. Co bychom ale měli udělat, kdybychom chtěli vytvořit cyklus, který bude postupně zobrazovat jednotlivé prvky matice? Máme dvě základní možnosti: 1) převést matici X na vektor a postupně její hodnoty zobrazit, 2) vnořit dva cykly do sebe a postupovat po řádcích nebo po sloupcích. Převedení matice na vektor a iterace od délky daného vektoru (length) je velmi snadná úloha, pojďme si tedy ukázat vnoření dvou cyklů do sebe.

```
> # Definice matice
> X <- matrix(1:9, 3)
> print(X)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
>
> # Cyklus
> for (i in 1:ncol(X)) {
+   #Zobrazeni informace o danem cyklu:
+   print(paste("Vnejsi cyklus:",i))
+   for (j in 1:nrow(X)) {
+     print(X[i, j])
+   }
+ }
[1] "Vnejsi cyklus: 1"
[1] 1
[1] 4
[1] 7
[1] "Vnejsi cyklus: 2"
[1] 2
[1] 5
[1] 8
[1] "Vnejsi cyklus: 3"
[1] 3
[1] 6
[1] 9
```

Před samotný výkon vnitřního cyklu (s iterátorem `j`) je vložená funkce `print()`, která nás informuje o tom, jaké hodnoty nabývá iterátor (`i`) vnějšího cyklu. Cyklus tedy postupuje tak, že nejprve vnější cyklus přiřadí do iterační proměnné `i` hodnotu 1 a následně vnitřní cyklus postupně dosazuje do iterační proměnné `j` hodnoty 1 až 3. Tedy první hodnoty, které cyklus zobrazí, jsou `X[1, 1]`, `X[1, 2]`, `X[1, 3]`. Tyto hodnoty odpovídají první iteraci vnějšího cyklu a jedné až třetí iteraci vnitřního cyklu. Následně cyklus postoupí do druhého kola vnějšího for cyklu a zobrazí: `X[2, 1]`, `X[2, 2]`, `X[2, 3]`. Tedy v druhém kole (kdy iterační proměnná vnějšího cyklu `i` je rovna 2) znovu vnitřní cyk-

lus nabude hodnot 1, 2 a 3 (iterační proměnná `j`). V poslední fázi vnějšího cyklu (kdy `i` je rovna 3) budou zobrazeny prvky: `X[3, 1]`, `X[3, 2]`, `X[3, 3]`. Samozřejmě v případě, kdy si vytváříme vlastní cyklus, není vůbec třeba vypisovat informaci o tom, jaké hodnoty iterační proměnná v určité fázi cyklu nabývá. V našem případě nabývá hodnot `print(paste("Vnejsi cyklus:", i))`. Zde tuto informaci uvádíme pro zpřehlednění výkonu jednotlivých iterací.

Všimněte si, že pro každý cyklus využíváme jinak pojmenovanou řídicí proměnnou. To je z toho důvodu, že řídicí proměnná se po dobu realizace těla cyklu chová jako jakýkoliv jiný objekt v **R**. Teprve až po dokončení kola iterace nabude další hodnoty z přiřazené množiny hodnot. Pokud jste před realizací cyklu vytvořili v **R** proměnnou stejného jména, tak ji cyklus přepíše. Po dokončení cyklu bude nabývat poslední hodnotu přiřazené množiny, přes jejíž prvky iteruje.

Zajímavá situace nastává v případě, kdy vnoříme do sebe dva cykly a oba budou používat stejně pojmenovanou řídicí proměnnou. Vnější cyklus přiřadí do řídicí proměnné první prvek vnější množiny, avšak vnitřní cyklus bude postupně do dané proměnné přiřazovat prvky své (druhé) množiny. Na konci první iterace vnějšího cyklu bude v řídicí proměnné poslední hodnota množiny vnitřního cyklu. Na začátku druhé iterace vnějšího cyklu ale bude již v iterátoru zase prvek (druhé) množiny vnějšího cyklu. A takto se situace bude opakovat až do konce obou cyklů.

```
> for (i in 1:3){
+   print(paste("zacatek vnejsiho cyklu", i))
+
+   for (i in 4:5) {
+   print(paste("vnitrni cyklus", i))
+   }
+
+   print(paste("konec vnejsiho cyklu", i))
+ }

[1] "zacatek vnejsiho cyklu 1"
[1] "vnitrni cyklus 4"
[1] "vnitrni cyklus 5"
[1] "konec vnejsiho cyklu 5"
[1] "zacatek vnejsiho cyklu 2"
[1] "vnitrni cyklus 4"
[1] "vnitrni cyklus 5"
[1] "konec vnejsiho cyklu 5"
[1] "zacatek vnejsiho cyklu 3"
[1] "vnitrni cyklus 4"
[1] "vnitrni cyklus 5"
[1] "konec vnejsiho cyklu 5"
```

Podobná situace bude nastávat i v případě, kdy se rozhodneme předefinovat samotnou iterační proměnnou v samotném běhu cyklu.

```
> for (i in 1:3){
+   print(i)
+   i=1
+   print(i)
+ }
[1] 1
[1] 1
[1] 2
[1] 1
[1] 3
[1] 1
```

Zde je ale nutné upozornit čtenáře, že kód, kde do iterační proměnné jsou přiřazované jakékoliv hodnoty v rámci těla cyklu, je vyloženě nesprávný a uživatel by se měl vyvarovat tomuto použití. Kód se takovouto aplikací znepřehlední a často nastávají neočekávané (a chybné) situace.

Řekli jsme si, že jednotlivé elementy for cyklu jsou vykonávané pro jednotlivé prvky množiny hodnot (obvykle vektoru), přes který má řídicí proměnná iterovat. Cyklus nemusí být proveden pouze přes atomický vektor hodnot, ale lze jej realizovat i přes list (list je přece také vektor), anebo přes jakékoliv jiné objekty, k jejichž vnitřním datům lze přistupovat přes předem definovanou dimenzi (např. data.frame). Obvykle se jedná o objekty, u nichž lze zjistit délku, například pomocí funkce `length`, a přistoupit na této délce k částečné výseči daného objektu.

```
> # Definice listu:
> l <- list(c = 1,
+           b = c(1, 3),
+           c = matrix(1:9, 3))
> # Zobrazime si objekt aby jsme vedeli pres co iterujeme
> print(l)
$c
[1] 1

$b
[1] 1 3

$c
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> # Samotne provedeni for cyklu:
> for (i in l) {
```

```
+   print(i)
+ }
[1] 1
[1] 1 3
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Jak si zcela jistě pamatujete z úvodních kapitol knihy, tak objekt typu `list` je vlastně formou vektoru. Nejprve se tedy zobrazil podlist `$a`, pak `$b` a následně `$c`.

Iterovat můžeme i přes objekt typu `data.frame`. Takové iterování si ukážeme na objektu `mtcars`. Do iterační proměnné jsou v jednotlivých iteracích přiřazované sloupce (proměnné) z datasetu `mtcars`. Délka objektu je zde čerpána z počtu proměnných obsažených v objektu typu `data.frame`, tedy odpovídá počtu sloupců.

```
> for (i in mtcars) {
+   print(mean(i))
+ }
[1] 20.09062
[1] 6.1875
[1] 230.7219
[1] 146.6875
[1] 3.596563
[1] 3.21725
[1] 17.84875
[1] 0.4375
[1] 0.40625
[1] 3.6875
[1] 2.8125
```

Analogický výstup bychom získali i použitím příkazu `colMeans(mtcars)`.

Většina ostatních objektů ale nemá parametr délky (atribut `length`). Například matice má atributy dimenzí, ale ne délky. Kdybychom se ale i tak rozhodli iterovat přes matici (přes sloupce matice), tak získáme nesmyslný výstup.

```
> X <- matrix(1:4, 2)
> print(X)
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> for (i in X) {
+   print(X)
+ }
     [,1] [,2]
```

```
[1,]    1    3
[2,]    2    4
     [,1] [,2]
[1,]    1    3
[2,]    2    4
     [,1] [,2]
[1,]    1    3
[2,]    2    4
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Program **R** není schopen přiřadit iterátoru jednotlivé elementy objektu (řádky, sloupce či jednu buňka matice), a tak se provede iterace přes součet dimenzí, ale v každé iteraci nabývá iterátor hodnoty samotné celé matice. Pokud se tedy uživatel rozhodne iterovat přes celý objekt, měl by se předem ujistit o třídě a dimenzích takového objektu.

Obvykle se v programovacím jazyku **R** volí atomický vektor hodnot, jehož hodnoty iterátor nabývá. Hlavním důvodem je přehlednost, pak i to, že **R** má rodinu `apply()` funkcí, která usnadňuje vykonání cyklické operace přes určité druhy objektů.

Příklad 1

Spočítejte pomocí `for` cyklu variační koeficient pro proměnné datasetu `mtcars`.

Řešení 1

```
> for (i in mtcars) {
+   print(sd(i) / mean(i))
+ }
[1] 0.2999881
[1] 0.2886338
[1] 0.5371779
[1] 0.4674077
[1] 0.1486638
[1] 0.3041285
[1] 0.1001159
[1] 1.152037
[1] 1.228285
[1] 0.2000825
[1] 0.5742933
```

Příklad 2

Máme vektor `x=1:10` a chceme pomocí cyklu vytvořit nový vektor (`k`), který bude obsahovat kumulace vektoru `x`. Tedy kumulovanou sumu hodnot vektoru `x` od prvního prvku do pozice daného prvku, pro kterou sumu provádíme.

Řešení 2

```
> x <- 1:10
> print(x)
 [1]  1  2  3  4  5  6  7  8  9 10
> # Prazdny vektor k:
> k <- c()
> # Cyklus
> for (i in 1:length(x)) {
+   k[i] <- sum(x[1:i])
+ }
> # Vysledek:
> print(k)
 [1]  1  3  6 10 15 21 28 36 45 55
```

Příklad 3

Vytvořte matici `X`, která bude mít pět řádek a čtyři sloupce. Následně pomocí `for` cyklu(ů) umístěte do prvků této matice indexy daného prvku matice (buňky).

Řešení 3

```
> # Vytvorime si prvne matici:
> X <- matrix(ncol=4, nrow=5)
> print(X)
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
[4,]   NA   NA   NA   NA
[5,]   NA   NA   NA   NA
> # Nasledne pomoci dvou cyklu projdeme vsechny prvky matice:
> for (i in 1:nrow(X)){
+   for (j in 1:ncol(X)){
+     X[i, j]<-paste0(i, ",", j)
+   }
+ }
```

```
> print(X)
     [,1]   [,2]   [,3]   [,4]
[1,] "1, 1" "1, 2" "1, 3" "1, 4"
[2,] "2, 1" "2, 2" "2, 3" "2, 4"
[3,] "3, 1" "3, 2" "3, 3" "3, 4"
[4,] "4, 1" "4, 2" "4, 3" "4, 4"
[5,] "5, 1" "5, 2" "5, 3" "5, 4"
```

1.2 While cyklus

`While` cyklus vykonává (opakuje) tělo cyklu do té doby, do které je splněná podmínka pro výkon cyklu. Výsledek této podmínky musí být interpretovatelný jako logická proměnná. Tedy i čísla 0 a 1 lze použít jako vstup do podmínky cyklu. Standardně se ale jedná o logický výraz, který testuje nějakou vlastnost. Následující kód ilustruje použití while cyklu.

```
while (# podminka) {
# telo cyklu
}
```

V případě, kdy je podmínka vyhodnocena jako `FALSE`, tak další kolo cyklu již není vykonáno, a to i v případě, že doposud nebyla žádná iterace vykonána vůbec.

Na následujícím příkladu je ilustrována situace použití while cyklu, která je analogická k `for` cyklu (`for (i in 1:10)`). Tedy v jednotlivých kolech iterace vypíšeme hodnotu proměnné `i` od jedné do desíti.

```
> i = 1
> while (i <= 10) {
+   print(i)
+   i = i + 1
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Výraz v kulaté závorce je logickým testem toho, zda je `i` menší nebo rovno deseti. Pokud

výsledkem tohoto testu je hodnota `TRUE`, vykoná se kód ve složené závorce. Pokud by však výsledek jednou nabyl hodnoty `FALSE`, tak se cyklus úplně zastaví a žádná další iterace se již nevykoná.

V našem případě proměnná `i` plní funkci iterátoru, a tak je celý cyklus velmi podobný `for` cyklu. Na tomto příkladu je hezky vidět, proč je for cyklus považován za formu `while` cyklu. V programovacím jazyku **R** je ale situace o něco složitější, protože je zde ještě obecnější forma cyklu while. Jedná se o cyklus jménem `repeat`. Na tento cyklus se ale podíváme až v následujících kapitolách.

Důležité je zde upozornit, že proměnná `i` musí být zavedená ještě před samotným zápisem cyklu, jinak by cyklus navrátil `error (missing value)`. Protože i pro první spuštění cyklu dochází k testování podmínky toho, zda se má cyklus spustit! Při definici podmínky musí tedy uživatel brát tuto situaci předem v potaz. Stává se, že mnoho lidí definuje cyklus, jehož výchozí podmínka nemůže být pro první kolo iterace nikdy splněná. Podmínka samozřejmě nemusí být jen jedna a logické výrazy můžeme kombinovat.

```
> i = 1
> j = 1
> while (i >= 1 & j <= 1 | abs(i * j) < 10) {
+   i = i * j + 1
+   j = j * i - 1
+   print(paste("i:", i, ", j:", j, "abs(i*j):", abs(i * j)))
+ }
[1] "i: 2 , j: 1 abs(i*j): 2"
[1] "i: 3 , j: 2 abs(i*j): 6"
[1] "i: 7 , j: 13 abs(i*j): 91"
```

V každé iteraci cyklu je proveden předem definovaný výpočet a následně se zobrazí hodnoty jednotlivých proměnných, které se budou před začátkem dalšího cyklu ověřovat v podmínce. Pro zobrazení proměnných využíváme stejné funkce jako minule v předcházejícím příkladu – funkce `print()`.

V praxi se může stát, že nebudeme vědět, zda vůbec nastane situace, kdy a jestli bude podmínka nesplněna (`FALSE`). Tedy nejsme schopni s jistotou říct, zda se cyklus zastaví. Tato situace je velmi podobná tomu, kdy by cyklus splnil podmínku až po velkém množství iterací, a chceme se předem ujistit, že tato situace nenastane. Tedy chceme zastavit cyklus po předem daném počtu iterací.

Tyto dva problémy lze vyřešit dvěma způsoby. Buď musíme očistit tuto situaci pomocí řídicích procedur `NEXT` a `BREAK` v kombinaci s podmínkami typu `if`, nebo přidat do cyklu `while` iterátor, který předem definuje maximální počet iterací. `NEXT` a `BREAK` příkazy probereme až v následujících kapitolách. Teď si ukážeme řešení této situace pro příklad, kdy vytvoříme nekonečný cyklus ošetřený pomocí iterátoru.

```
> i = 0
> x = 1
```

```
> while (x != 2 & i < 100) {
+    x = x + 0.13
+    i = i + 1
+ }
> print(i)
[1] 100
```

Příklad je nadefinovaný tak, že nikdy nebude splněná podmínka o rovnosti x dvou. Z tohoto důvodu je nutné přidat proměnnou iterátoru jako součet podmínky – pro ošetření celkového počtu cyklů. Nezapomeňte inkrementovat iterátor, oproti for cyklu to za vás while cyklus neudělá.

Přesto, že se vám může zdát předchozí příklad relativně nesmyslný (kdo by definoval ukončení cyklu na určitou nerovnost, která nenastává), tak je nutné vědět, že nesmíme zapomenout na to, že **R** má přesnost výpočtu na šestnáct desetinných míst (`1e-16`!). Proto mnoho lidí o hodnotách, které jsou menší než `1e-16`, mluví jako o technicky nulových. Může se pak u některých typů výpočtů snadno stát, že analytickým řešením je nulový výsledek, ale technicky získáme číslo například `1e-18`. Tedy číslo, které je nenulové, ale je již v pásmu zaokrouhlovací chyby softwaru. Pokud by ale while cyklus testoval toto číslo na rovnost nule, nedošlo by k ukončení cyklu, protože číslo je pouze technicky nulové.

Příklad 4

Pomocí while cyklu zobrazte hodnoty proměnné `x` (definované sekvencí hodnot `10:1`) a v případě, kdy bude hodnota menší než 5, tak cyklus zastavte.

Řešení 4

```
> #Definice iteratoru
> i = 1
> #while cyklus se dvema podminkama
> while (i <= 10 & x[i] >= 5) {
+    print(x[i])
+    i = i + 1
+ }
[1] 10
[1] 9
[1] 8
[1] 7
[1] 6
[1] 5
```

Příklad 5

Pomocí `while` cyklu zjistěte, kolikrát musíme vydělit hodnotu 10, aby byla její aktuální hodnota menší než 0.001. Hodnotu 10 budeme dělit 2.

Řešení 5

```
> # Definujeme vychozi hodnotu x a iterator (i)
> x <- 10
> i <- 0
> # Realizace while cyklu
> while (x > 0.001) {
+   x = x / 2
+   i = i + 1
+ }
> # Prvni hodnota x mensi nez 0.001
> print(x)
[1] 0.0006103516
> # Kolikrat bylo potreba hodnotu 10 vydelit - kolikrat probehl
    cyklus
> print(i)
[1] 14
```

Příklad 6

Máte vektor `x`, který je definován jako sinusoida od 0 do hodnoty π s krokem 0,1. Následně zkuste pomocí while cyklu (a vlastnoručně definovaného iterátoru) najít první největší hodnotu, po které začínají hodnoty prvku x klesat (tedy hledáme jakési první lokální maximum).

Řešení 6

```
> # Definice vektoru:
> x <- sin(seq(from = 0, to = pi, by = 0.1))
> print(x, digits = 1)
 [1] 0.00 0.10 0.20 0.30 0.39 0.48 0.56 0.64 0.72 0.78 0.84
[12] 0.89 0.93 0.96 0.99 1.00 1.00 0.99 0.97 0.95 0.91 0.86
[23] 0.81 0.75 0.68 0.60 0.52 0.43 0.33 0.24 0.14 0.04
>
> # Iteracni promenna:
> i = 2
> # while cyklus kde testujeme zda prvek x stale roste:
> while (x[i + 1] > x[i]) {
```

```
+   i = i + 1
+ }
> # Na ktere i-te pozici je prvek nejvetsi:
> print(i)
[1] 17
> # Jaka je hodnota nejvyssiho prvku:
> print(x[i])
[1] 0.9995736
```

Správné by bylo v těchto příkladech vyřešit (ošetřit) různé problematické situace (kdy například iterátor převýší délku x). Toto a jiné techniky si ukážeme v následujících kapitolách.

Kapitola 2

Podmínky

Často nastává situace, kdy potřebujeme realizovat část kódu jen za určité podmínky. V takovém případě pak využijeme právě podmínku. Nejpoužívanější podmínkou je funkce `if` a dále funkce `ifelse()`. Podmínku `if` můžeme dále rozšířit o část `else` a `else if`. Pojďme se podívat na jednotlivou problematiku v následujících kapitolách.

2.1 Podmínka if

Podmínka `if` prvně ověří, zda je logický výraz v kulaté závorce za zápisem `if` pravdivý, a pokud je, tak **R** vykoná kód ve složených závorkách. Díky tomuto postupu jsme schopni vykonávat určité sekvence kódu jen za určité situace. Ukážeme si prvně podmínku se vždy splněným výrazem.

```
> if (TRUE) {
+   print("Splnena")
+ }
[1] "Splnena"
```

Naopak podmínka, jejíž kód se ve složené závorce nikdy nerealizuje, by byla:

```
> if (FALSE) {
+    print("nesplnena, tento kod se Vam nikdy nezobrazi")
+ }
```

Samozřejmě obvykle nebudeme přímo psát výsledek logické operace do kulatých závorek ručně, ale stanovíme logický výraz, který bude testovat danou situaci. Například když chceme, aby se pro čísla od jedné do dvanácti, která jsou celočíselně dělitelná třemi, zobrazil specifický text, můžeme stanovit podmínku pomocí nulového zbytku po dělení.

```
> for (i in 1:12) {
+   if (i %% 3 == 0) {
+     print(paste("Cislo", i, "je celociselne delitene trema"))
+   }
+ }
[1] "Cislo 3 je celociselne delitene trema"
[1] "Cislo 6 je celociselne delitene trema"
[1] "Cislo 9 je celociselne delitene trema"
[1] "Cislo 12 je celociselne delitene trema"
```

Výraz `i%%3 == 0` testuje, zda zbytek po dělení řídicí proměnné for cyklu je nulový. V případě, kdy je, se pak realizuje část kódu příslušící k tělu podmínky. V našem případě se zobrazí hodnota čísla, které daný test splnilo, a vypíše se informace o tom, že daná hodnota je celočíselně dělitelná třemi.

Samozřejmě podmínky mohou být i komplexnější a můžeme testovat celou sérii logických výrazů. Například můžeme pro čísla jedna až sto otestovat, zda je dané číslo celočíselně dělitelné jak třemi, tak pěti, a pokud je liché, tak jej zobrazíme v konzoli. Řešení pak najdeme pomocí spojení několika logických výrazů do jedné podmínky, jak ilustruje následující kód.

```
> for (i in 1:100) {
+   if (i%%3 == 0 & i%%5 == 0 & i%%2 != 0) {
+     print(i)
+   }
+ }
[1] 15
[1] 45
[1] 75
```

Na tomto příkladu je vidět, že pomocí logických operátorů, jako jsou & (and) a nebo i | (or), můžeme zapsat opravdu komplexní testovací úlohy.

Příklad 1

Pro sekvenci hodnot definovanou jako `seq(from = 1, to = 100, by = 3)` zobrazte hodnoty celočíselně dělitelné pěti ve `for` cyklu.

Řešení 1

```
> #Definice promenne
> x <- seq(from = 1, to = 100, by = 3)
> #Vlastni cyklus
> for (i in 1:length(x)) {
```

```
+    #IF podminka
+    if (x[i] %% 5 == 0)
+      print(x[i])
+ }
[1] 10
[1] 25
[1] 40
[1] 55
[1] 70
[1] 85
[1] 100
```

2.2 Větvení podmínek pomocí else

Často nastává situace, kdy v případě, že není podmínka splněná, se má realizovat jiný kód, než když podmínka splněná je. Tuto situaci můžeme vyřešit pomocí rozšíření if podmínky o výraz else. Jedná se tedy o alternativní část kódu, kterou umístíme přímo za tělo podmínky `if`, tedy přímo za druhou složenou závorku. Následující kód bude hodnoty od jedné do desíti testovat na to, zda jsou sudé nebo liché. Tento problém by šel jistě vyřešit i dvěma podmínkami `if`, kdy by jedna testovala, zda je číslo liché, a druhá, zda je sudé. Jedná se ale o velmi špatný zvyk a v případě komplexnějších úloh by byl kód nejen hůře čitelný a delší, ale i by se v něm snáze vyrobila chyba.

```
> for (i in 1:10) {
+    if (i %% 2 == 0) {
+      print(paste("hodnota: ", i, "je suda"))
+    } else{
+      print(paste("hodnota: ", i, "je licha"))
+    }
+ }
[1] "hodnota:  1 je licha"
[1] "hodnota:  2 je suda"
[1] "hodnota:  3 je licha"
[1] "hodnota:  4 je suda"
[1] "hodnota:  5 je licha"
[1] "hodnota:  6 je suda"
[1] "hodnota:  7 je licha"
[1] "hodnota:  8 je suda"
[1] "hodnota:  9 je licha"
[1] "hodnota:  10 je suda"
```

V případě nesplnění podmínky se realizuje kód uvedený v else větvi podmínky. Naopak v případě platnosti podmínky již nedojde k realizaci tohoto kódu, ale realizuje se pouze část první (přímo za `if` podmínkou).

Příklad 2

Pomocí `if...else` v datasetu mtcars vypište do konzole, zda auto má či nemá automatickou převodovku (proměnná `am`). Daná úloha má být provedena pouze pro auta s šesti válci (proměnná `cyl`).

Řešení 2

```
> for (i in 1:nrow(mtcars)) {
+   if (mtcars$cyl[i] == 6) {
+     if (mtcars$am[i] == 1) {
+       print(paste("auto", rownames(mtcars)[i], "ma automatickou
    prevodovku"))
+     } else{
+       print(paste("auto", rownames(mtcars)[i], "nema automatickou
    prevodovku"))
+     }
+   }
+ }
[1] "auto Mazda RX4 ma automatickou prevodovku"
[1] "auto Mazda RX4 Wag ma automatickou prevodovku"
[1] "auto Hornet 4 Drive nema automatickou prevodovku"
[1] "auto Valiant nema automatickou prevodovku"
[1] "auto Merc 280 nema automatickou prevodovku"
[1] "auto Merc 280C nema automatickou prevodovku"
[1] "auto Ferrari Dino ma automatickou prevodovku"
```

2.3 Větvení podmínek pomocí else if

Další možné rozšíření `if` podmínky je `else if`. Tento výraz nám umožní rozšířit podmínku `if` tak, aby v případě, kdy nebude první podmínka splněna (v `if` části), byla testována alternativní podmínka (či podmínky v `else if` části). Tedy po nesplnění první podmínky dochází k testování další podmínky, ale v `else if` části. Pokud ani tato podmínka není splněná, tak se přechází k `else` části, anebo je testování zcela ukončeno. Je možné vytvořit podmínku, která bude mít po if části více else if výrazů. Podstatné je, aby měla pouze jen jednu else větev, která je až poslední (po `else if` výrazech). Při testování podmínek hraje roli i pořadí. Podmínky jsou testované v pořadí, v jakém jsou zapsané. Tedy v podmínce může být splněno více podmínek jednotlivých větví kódu, ale realizuje se pouze ta, která je splněna jako první! Ostatní splněné podmínky se vůbec nerealizují.

```
> for (i in 1:10) {
+   if (i %in% 1:5) {
+     print(paste(i, ": a"))
+   } else if (i %in% 1:7) {
+     print(paste(i, ": b"))
+   } else if (i %in% 1:10) {
+     print(paste(i, ": c"))
+   } else{
+     print("ostatni")
+   }
+ }
[1] "1 : a"
[1] "2 : a"
[1] "3 : a"
[1] "4 : a"
[1] "5 : a"
[1] "6 : b"
[1] "7 : b"
[1] "8 : c"
[1] "9 : c"
[1] "10 : c"
```

Pro čísla od jedné do desíti testujeme následující podmínku. V případě, kdy je číslo přítomné v množině čísel jedna až pět, pak zobrazíme písmeno „a“. Pokud je číslo přítomné v množině čísel jedna až sedm, tak zobrazíme písmeno „b“, a pokud je číslo přítomné v množině jedna až deset, tak zobrazíme písmeno „c“, jinak zobrazíme text „ostatní“. Všimněte si, že například hodnota pět splňuje všechny podmínky, ale zobrazí se u ní pouze písmeno „a“. Toto je dané právě pořadím podmínek.

Jak byste pomocí předchozích znalostí otestovali čísla jedna až deset na to, zda jsou prvočísla? V **R** to lze udělat jednoduše.

```
> for (i in 1:10) {
+   if (i == 2) {
+     print(paste("Cislo: ", i, "je prvocislo"))
+   } else if (any(i %% c(2:(i-1)) == 0)) {
+     print(paste("Cislo: ", i, "neni prvocislo"))
+   } else {
+     print(paste("Cislo: ", i, "je prvocislo"))
+   }
+ }
[1] "Cislo:  1 neni prvocislo"
[1] "Cislo:  2 je prvocislo"
[1] "Cislo:  3 je prvocislo"
[1] "Cislo:  4 neni prvocislo"
[1] "Cislo:  5 je prvocislo"
```

```
[1] "Cislo:  6 neni prvocislo"
[1] "Cislo:  7 je prvocislo"
[1] "Cislo:  8 neni prvocislo"
[1] "Cislo:  9 neni prvocislo"
[1] "Cislo:  10 neni prvocislo"
```

Cyklus prvně otestuje, zda je číslo rovno dvěma, pokud je, tak číslo je prvočíslem. Pokud není, tak se testuje, zda číslo má jiného dělitele než jedna a sebe samého (tedy od dvou do $i-1$). Pokud má (je tedy tímto číslem celočíselně dělitelné – zbytek po dělení je roven nule), tak je číslo prohlášené za hodnotu, která není prvočíslem. Všechna ostatní čísla (`else` část) jsou prvočísly (jako číslo dva).

Příklad 3

Zkuste předchozí příklad (č. 3) vyřešit pomocí jen jedné podmínky s využitím `else if` větve podmínky.

Řešení 3

```
> for (i in 1:nrow(mtcars)) {
+   # Do if casti je nutne pridat dalsi podminku na pocet valcu
+   if (mtcars$am[i] == 1 & mtcars$cyl[i] == 6) {
+     print(paste("auto", rownames(mtcars)[i], "ma automatickou
    prevodovku"))
+     # a dale je nutne i v else if casti osetrit podminku na pocet
    valcu
+   } else if (mtcars$cyl[i] == 6) {
+     print(paste("auto", rownames(mtcars)[i], "nema automatickou
    prevodovku"))
+   }
+ }
[1] "auto Mazda RX4 ma automatickou prevodovku"
[1] "auto Mazda RX4 Wag ma automatickou prevodovku"
[1] "auto Hornet 4 Drive nema automatickou prevodovku"
[1] "auto Valiant nema automatickou prevodovku"
[1] "auto Merc 280 nema automatickou prevodovku"
[1] "auto Merc 280C nema automatickou prevodovku"
[1] "auto Ferrari Dino ma automatickou prevodovku"
```

2.4 Větvení podmínek ifelse

V případě, kdy je kód, který se má realizovat v podmínce, velmi jednoduchý, lze alternativně využít funkci `ifelse()`. Funkce má tyto parametry: `test`, `yes`, `no`. Tedy výraz, který testuje (`test`), `yes` – co se má stát v případě, kdy je podmínka splněna, `no` – co se má stát v případě, kdy podmínka splněna není.

```
> for (i in 1:6) {
+   ifelse(i <= 3, print("mensi nebo rovno trem"), print("vetsi nez
    3"))
+ }
[1] "mensi nebo rovno trem"
[1] "mensi nebo rovno trem"
[1] "mensi nebo rovno trem"
[1] "vetsi nez 3"
[1] "vetsi nez 3"
[1] "vetsi nez 3"
```

Zajímavostí je, že lze v rámci této funkce deklarovat (přiřazovat) hodnoty i do proměnných.

```
> ifelse(TRUE, x <- 10, x <- 100)
[1] 10
> x
[1] 10
```

Nejedná se ale o doporučený postup. V případě deklarací proměnných či komplexnějších kódů je lepší zvolit standardní `if...else` výraz.

Příklad 4

Pomocí `for` cyklu a `ifelse` funkce nahraďte v sekvenci `sin(seq(from = 0, to = pi*4, by = 0.5))` hodnoty větší než nula hodnotou jedna a menší nebo rovno nule hodnotou nula.

Řešení 4

```
> # Deklarace promenne
> x <- sin(seq(from = 0, to = pi * 4, by = 0.5))
> print(x, digit = 1)
 [1]  0.00  0.48  0.84  1.00  0.91  0.60  0.14 -0.35 -0.76 -0.98
     -0.96 -0.71 -0.28  0.22
[15]  0.66  0.94  0.99  0.80  0.41 -0.08 -0.54 -0.88 -1.00 -0.88
    -0.54 -0.07
```

```
> # for cyklus
> for(i in 1:length(x)) ifelse(x[i] > 0, x[i] <- 1, x[i] <- 0)
> print(x)
 [1] 0 1 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0 0
```

Kapitola 3

Příkazy next, break a repeat cyklus

V této kapitole se seznámíme s výrazy `next`, `break` a `repeat` cyklem. `Next` a `break` jsou takzvané řídicí výrazy a umožňují nám ovládat cyklus mimo předem stanovený běh cyklu. Dále se seznámíme s cyklem `repeat`. `Repeat` cyklus je takzvaný nekonečný cyklus. Pokud nestanovíme ukončení pomocí výrazu `break`, tak tento cyklus poběží „do nekonečna“. Z tohoto důvodu je daná problematika spojena do jedné kapitoly.

3.1 Příkaz next a break

Výrazy `next` a `break` nám umožňují řídit cyklus. Výraz `next` způsobí to, že daná iterace cyklu se od místa zápisu next ukončí a začne ihned iterace následující. Naopak výraz `break` ukončí rovnou celý cyklus jako takový. Oba dva výrazy lze kombinovat a uplatnit je v jakémkoliv cyklu. Tedy lze je uplatnit jak ve `for`, tak `while` i `repeat` cyklu.

Představme si, že máme data od jedné do sta a chceme zastavit cyklus na místě, kdy součet všech předchozích hodnot je více než padesát.

```
> for (i in 1:100) {
+   if (sum(1:i) > 50)  break
+   print(paste("iterace:", i, "suma:", sum(1:i)))
+ }
[1] "iterace: 1 suma: 1"
[1] "iterace: 2 suma: 3"
[1] "iterace: 3 suma: 6"
[1] "iterace: 4 suma: 10"
[1] "iterace: 5 suma: 15"
[1] "iterace: 6 suma: 21"
```

```
[1] "iterace: 7 suma: 28"
[1] "iterace: 8 suma: 36"
[1] "iterace: 9 suma: 45"
```

Množina for cyklu, přes kterou má cyklus iterovat, je od jedné do sta, avšak cyklus se zastaví již v desáté iteraci. V deváté iteraci byl součet ještě čtyřicet pět, a proto cyklus pokračoval dále, ale již v desáté iteraci byl větší než padesát, a tak se cyklus od místa podmínky ukončil. Tedy sice do desáté iterace vstoupil, ale již ji nevykonal celou, ještě před funkcí `print` se zastavil. Oba výrazy můžeme i kombinovat. Například přidat podmínku pro případ, že se daná iterace přeskočí, když součet bude lichý.

```
> for (i in 1:100) {
+   if (sum(1:i) > 50) {
+     break
+   } else if (sum(1:i) %% 2 != 0) {
+     next
+   }
+   print(paste("iterace:", i, "suma:", sum(1:i)))
+ }
[1] "iterace: 3 suma: 6"
[1] "iterace: 4 suma: 10"
[1] "iterace: 7 suma: 28"
[1] "iterace: 8 suma: 36"
```

Jak z výsledků vidíme, hodnoty 1, 3, 15, 21, a 45 cyklus při informaci o tom, jaké hodnoty splnily podmínky, vynechal. Cyklus pokračoval dále, ale od místa testu `else if` daná iterace vždy skončila. Zkusme se teď podívat na problematický `while` cyklus, který má špatně nastavenou podmínku, a tak nikdy nekončí. Někdy se při implementaci matematických metod může stát, že si nejsme zcela jisti, že může nastat situace, kdy je podmínka nesplněna. Pak se hodí použití `break` řídicího výrazu v kombinaci s vlastním iterátorem. Následující kód by doběhl po velmi dlouhé době, ale díky zapojení iterátoru jsme se na tuto situaci připravili, a v případě, kdy se nám nepodaří při určitém počtu iterací dosáhnout negativní podmínky, tak cyklus ukončí výraz break.

```
> iterator = 1
> x = 0
> while (abs(x) < 10) {
+   x + 1e-5
+   iterator = iterator + 1
+   if (iterator >= 100) break
+ }
>
> print(iterator)
[1] 100
```

3.2 Cyklus repeat

Cykly `for` a `while` nejsou jedinými cykly, které jsou v jazyku **R** dostupné. Alternativou k těmto cyklům je použití obecnějšího cyklu `repeat`. Cyklus `for` má přesně definovaný počet iterací, kolikrát se vykoná. Tuto informaci má cyklus z délky množiny, kterou přebírá jako vstupní parametr. Cyklus `while` naopak před začátkem každého cyklu ověřuje platnost podmínky, pokud není splněna, tak se cyklus ukončí. Tedy oba cykly mají definován způsob ukončení cyklu. Cyklus `repeat` nemá předem definován způsob ukončení cyklu a ani jinak není omezen ve svém opakování. Jedná se záměrně o nekonečnou smyčku. Cyklus lze připodobnit `while` cyklu s nastavenou podmínkou na `TRUE`, tedy `while(TRUE)`. Uživatel pak musí sám definovat způsob ukončení cyklu v samotném těle cyklu. Ukončení cyklu se dosáhne pomocí podmínky if v kombinaci s řídicím výrazem break. Zkusme si tedy zapsat variantu for cyklu pomocí `break` cyklu. Následující kód ilustruje použití `repeat` cyklu ve smyslu for cyklu. Máme zde řídicí proměnou (`i`), kterou iterujeme přes množinu jedna až deset. Navýšení hodnoty proměnné je prováděno o hodnotu jedna a poslední hodnotou je hodnota deset (definovaná proměnnou `pocet_cyklu`).

```
> i <- 0
> pocet_cyklu = 10
> repeat {
+   print(i)
+   if (i >= pocet_cyklu) {
+     break
+   } else {
+     i = i + 1
+   }
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Alternativně si lze představit i variantu while cyklu. Zde provádíme výpočet: `x = (x+pi) * x / pi`, a pokud hodnota x převýší hodnotu 1000, tak se cyklus zastaví.

```
> x <- 1
> repeat {
+   x = (x + pi) * x / pi
```

```
+   print(x)
+   if (x > 1000)
+     break
+ }
[1] 1.31831
[1] 1.871514
[1] 2.986414
[1] 5.825315
[1] 16.62694
[1] 104.6253
[1] 3588.99
```

Zde je důležité si uvědomit, že hodnota `x` je v posledním kole cyklu vyšší než stanovená podmínka. Tato situace je dána tím, že testujeme podmínku `if` s příkazem `break` až po výpočtu nové hodnoty `x`.

Příklad 1

Pomocí cyklu `repeat` spočítejte variační koeficient pro všechny numerické proměnné datasetu `iris`.

Řešení 1

```
> i = 1
> repeat {
+   # zastaveni cyklu
+   if (i > ncol(iris)) break
+   #samotne telo cyklu
+   if (is.numeric(iris[, i])) {
+     print(sd(iris[, i]) / mean(iris[, i]) , digit = 3)
+   }
+   # navyseni hodnoty iteratoru
+   i = i + 1
+ }
[1] 0.142
[1] 0.143
[1] 0.47
[1] 0.636
```

Kapitola 4

Funkce apply, by a aggregate

Rodina funkcí apply slouží k automatizaci určité funkce pro určitý objekt, popřípadě jeho část (např. část dimenze u matice – sloupce, řádky...). Apply funkce nepatří u začátečníků mezi oblíbené funkce, ale pokud se je naučíme používat, zkrátí nám nejen čas věnovaný zápisu kódu, ale také zjednoduší kód a mnohdy ho zrychlí. Jedná se o následující funkce:

- `apply()`,
- `sapply()`,
- `vapply()`,
- `lapply()`,
- `mapply()`,
- `rapply()`,
- `tapply()`.

K apply funkcím se standardně řadí navíc i funkce `by()` a `aggregate()`. Tyto funkce mají trošku odlišné použití, a ukážeme si to na konci této kapitoly.

4.1 Funkce apply

Základní funkcí, kterou musíte poznat, je funkce apply. Všechny ostatní navazující funkce jsou totiž její variantou. Funkce `apply()` má tři povinné parametry. Jedná se o: `X`, `MARGIN` a `FUN`.

Do parametru `X` předáváme objekt, pro který se má funkce provést. Parametr `MARGIN` definuje dimenzi, skrze kterou se má funkce provést. Poslední povinný parametr je `FUN`, do kterého předáváme jméno funkce, která se má přes specifikovanou dimenzi daného objektu provést.

Parametr označující dimenzi (`MARGIN`) je vhodné blíže osvětlit. Vrátíme-li se na chvilku k výběru elementů jednotlivých objektů, například vektor má jednu dimenzi, matice dvě a pole může mít i tři a více dimenzí (ale i jednu nebo dvě). Když provádíme výběr prvků pomocí hranatých závorek, tak ve vektoru `v` (například o deseti prvcích) vybereme druhý pomocí příkazu `v[2]`. V matici (`V`, například o deseti řádkách a deseti sloupcích) bychom výběr prvku v druhém řádku a druhém sloupci provedli pomocí výrazu `V[2,2]`. V případě trojrozměrného pole (pro zjednodušení příkladu si teď uveďme pole o rozměrech $3 \times 3 \times 3$), kdybychom chtěli vybrat prvek v druhém řádku, druhém sloupci a druhé řadě, tak to provedeme pomocí výrazu: `R[2,2,2]`. V případě vektoru `v` se tedy uvádí jen pořadí prvku, jenž chceme vybrat (`v[číslo prvku]`), v případě matice `V` jsme zapsali řádku, sloupec (`V[číslo řádky, číslo sloupce]`) a v případě našeho pole jsme zapsali všechny tři dimenze (`R[číslo řádku, číslo sloupce, číslo řady]`). Do parametru funkce apply dimenze (`MARGIN`) pak zapisujeme pořadí dané dimenze při výběru. V případě aplikace funkce apply na matici, kdy budou hodnoty parametru `MARGIN` 1, tak se daná funkce provede přes řádky daného objektu (řádky se zapisují jako první do hranatých závorek). Naopak, kdybychom zvolili hodnotu dva, tak by se daná funkce aplikovala přes sloupce daného objektu, protože výběr přes sloupce je až druhá hodnota, co se zapisuje do hranatých závorek. S podobnou úvahou funguje tato funkce i pro pole a jiné objekty.

S ohledem na to, že se jedná o dost abstraktní definici toho, jak funkce funguje, uveďme si jednoduchý příklad. Když bychom chtěli spočítat průměr pro každý sloupec matice, tak máme v jazyku **R** několik možností:

- Použijeme funkci `colMeans()`.
- Vytvoříme cyklus, který si postupně vybere sloupec po sloupci dané matice a spočítá z něj průměr.
- Použijeme funkci apply, které předáme parametr hrany (dimenze, přes kterou sčítáme), funkci, kterou použijeme (`mean()`), a objekt, na který chceme funkci aplikovat.

Následující kód ilustruje řešení úlohy výpočtem průměru dat přes sloupce/řádky pomocí výchozích funkcí jazyka **R** `colMeans()`/`rowMeans()`.

```
> V <- matrix(data = c(1:9),
+             nrow = 3,
+             ncol = 3)

> print(V)
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> prumer_sloupcu <- colMeans(V)

> prumer_radku <- rowMeans(V)

> print(prumer_sloupcu)
[1] 2 5 8

> print(prumer_radku)
[1] 4 5 6
```

Alternativní řešení lze samozřejmě provést pomocí for cyklu. Zde je důležité poznamenat, že se jedná o dost neefektivní řešení. V případě, kdy neexistuje výchozí funkce v jazyku **R** (například pro výpočty intervalů spolehlivosti pro celý dataset), tak mnoho uživatelů volí tento postup. Efektivnějším řešením by ale bylo použít `apply()` funkci.

```
> V <- matrix(data = c(1:9),
+             nrow = 3,
+             ncol = 3)

> print(V)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> prumer_sloupcu <- rep(NA, ncol(V))

> prumer_radku <- rep(NA, nrow(V))

> for (i in 1:ncol(V))  prumer_sloupcu[i] <- mean(V[, i])

> for (i in 1:nrow(V))  prumer_radku[i] <- mean(V[i, ])

> print(prumer_sloupcu)
[1] 2 5 8

> print(prumer_radku)
[1] 4 5 6
```

Řešení úlohy pomocí apply funkce pak představuje následující kód. Rozdíl ve výpočtu průměru přes řádek/sloupec pak spočívá v čísle hrany, přes kterou se daná funkce realizuje.

```
> V <- matrix(data = c(1:9),
+             nrow = 3,
+             ncol = 3)
> print(V)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> prumer_sloupcu <- apply(X = V, MARGIN = 2, FUN = mean)
> prumer_radku <- apply(X = V, MARGIN = 1, FUN = mean)
> print(prumer_sloupcu)
[1] 2 5 8
> print(prumer_radku)
[1] 4 5 6
```

Jak jsme si již představili, v případě průměru má **R** výchozí funkci, která danou úlohu provede efektivněji (`colMeans()`/`rowMeans()`). Jazyk **R** má dále obdobnou funkci pro sumu (`rowSums()`/`colSums()`). Existuje ale mnoho dalších funkcí, které **R** nemá převedené pro maticové/tabulkové výpočty – přes všechny sloupce či řádky. Ukažme si například způsob, jak vypočítat medián přes všechny proměnné datasetu `mtcars`.

```
> apply(X = mtcars, MARGIN = 2, FUN = median)
    mpg     cyl    disp      hp    drat      wt    qsec      vs
     am    gear    carb
 19.200   6.000 196.300 123.000   3.695   3.325  17.710   0.000
  0.000   4.000   2.000
```

Obdobně můžeme spočítat i rozptyl či směrodatnou odchylku:

```
> apply(X = mtcars, MARGIN = 2, FUN = sd)
       mpg        cyl       disp         hp
 6.0269481  1.7859216 123.9386938 68.5628685
      drat         wt       qsec         vs
 0.5346787  0.9784574  1.7869432  0.5040161
        am       gear       carb
 0.4989909  0.7378041  1.6152000
```

Apply tedy postupně dosazuje za první parametr funkce, definované v parametru `FUN`, výběr dat specifikovaný dimenzí. Apply umožňuje specifikovat uživateli ostatní parametry funkce, do které dosazuje. Tyto parametry lze zapsat na konec funkce apply, je ale nutné je specifikovat správným jménem. Například si vytvoříme matici, která bude obsahovat `NA` řádek. Bez specifikace parametru na.rm ve funkci mean **R** nevrátí požadovaný průměr, ale `NA` hodnotu.

```
> A <- matrix(1:9, 3)
```

```
> A[2, ]<-NA
> apply(X = A, MARGIN = 2, FUN = mean)
[1] NA NA NA
> apply(X = A, MARGIN = 2, FUN = mean, na.rm = TRUE)
[1] 2 5 8
```

Tak jako v případě matice či data frame objektu lze aplikovat funkci apply i na objekt typu pole (`array`). Vytvoříme si pole velikosti $3 \times 3 \times 3$, tedy pole o třech dimenzích, a zkusíme si provést jednotlivé možné součty. Prvně si zkusíme vytvořit řádkový součet skrze celé pole. Zůstane zachována dimenze řádku a všechny ostatní dimenze budou sečteny. Toto provedeme pomocí parametru `MARGIN = 1`.

```
> pole = array(data = c(1:27), dim = c(3, 3, 3))
> apply(X = pole, MARGIN = 1, FUN = sum)
[1] 117 126 135
```

V případě pole nemusí být aplikace funkce apply na první pohled srozumitelná, proto si ukážeme i řešení dané úlohy pomocí funkcí `rowSums()` a selekcí příslušných dimenzí pole.

```
> pole = array(data = c(1:27), dim = c(3, 3, 3))
> cbind(sum(pole[1,,]), sum(pole[2,,]), sum(pole[3,,]))
[1] 117 126 135
```

Co kdybychom ale chtěli vytvořit matici součtů řádků skrze všechny dimenze a ne součet řádkových sum? V tomto případě je nutné do parametru `MARGIN` specifikovat všechny dimenze, které mají být zachovány. V našem případě půjde o dimenzi, skrze kterou provádíme součet a dále zbývající (třetí) dimenzi úlohy, aby zůstala zachována všechna data.

```
> pole = array(data = c(1:27), dim = c(3, 3, 3))
> apply(X = pole, MARGIN = c(1, 3), FUN = sum)
     [,1] [,2] [,3]
[1,]   12   39   66
[2,]   15   42   69
[3,]   18   45   72
```

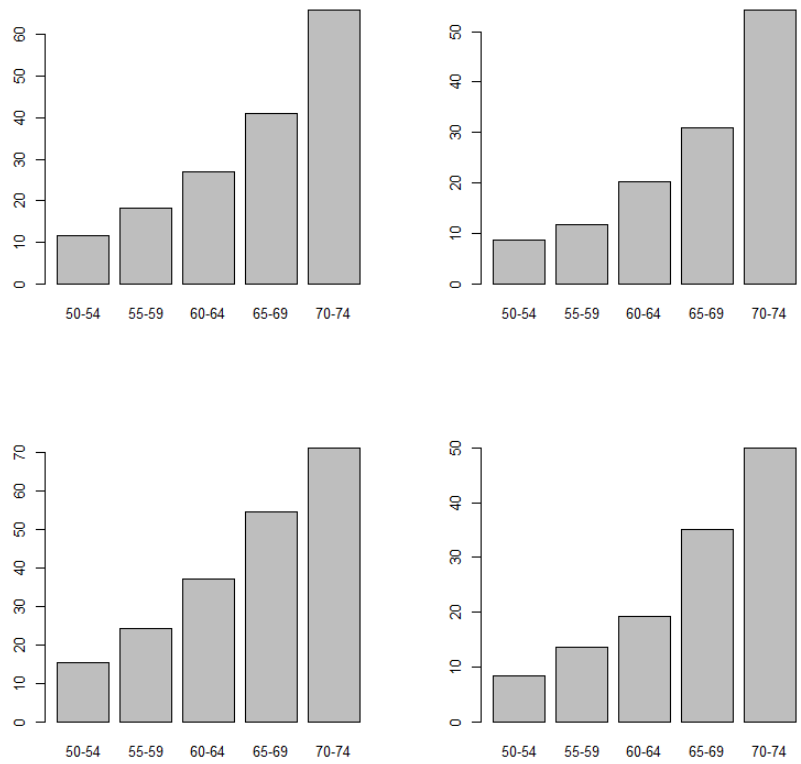
Jaké by bylo alternativní řešení?

```
> pole = array(data = c(1:27), dim = c(3, 3, 3))
> cbind(rowSums(pole[,, 1]), rowSums(pole[,, 2]), rowSums(pole[,,
    3]))
     [,1] [,2] [,3]
[1,]   12   39   66
[2,]   15   42   69
[3,]   18   45   72
```

Zajímavostí je, že lze v **R** zpracovávat pomocí apply funkcí i grafické funkce, například vykreslit sérii grafů. Jen je důležité si uvědomit, že `apply()` funkce navrací návratovou

hodnotu grafické funkce do konzole. Obvyklá návratová hodnota je v případě grafů v jazyku R `NULL` nebo vstupní data. Hodnota se nám pak vypíše v případě grafů do příkazové řádky. Tento postup si ilustrujme na datech o počtu úmrtí na 1000 obyvatel ve Virginii v USA. Data jsou tříděná dle věku a místa, kde daný jedinec bydlí.

```
> print(VADeaths)
      Rural Male Rural Female Urban Male Urban Female
50-54        11.7          8.7        15.4          8.4
55-59        18.1         11.7        24.3         13.6
60-64        26.9         20.3        37.0         19.3
65-69        41.0         30.9        54.6         35.1
70-74        66.0         54.3        71.1         50.0
> par(mfrow = c(2, 2))
> apply(X = VADeaths, 2, FUN = barplot)
     Rural Male Rural Female Urban Male Urban Female
[1,]         0.7          0.7         0.7          0.7
[2,]         1.9          1.9         1.9          1.9
[3,]         3.1          3.1         3.1          3.1
[4,]         4.3          4.3         4.3          4.3
[5,]         5.5          5.5         5.5          5.5
> par(mfrow=c(1, 1))
```



Druhé zobrazení dat v příkazové řádce je pak důsledkem aplikace funkce barplot ve funkci apply. Kdybychom chtěli specifikovat dodatečné parametry grafické funkce `barplot()`, bylo by vhodné použít jinou funkci rodiny apply, například `mapply()`, kterou si představíme v navazující podkapitole.

Do všech funkcí rodiny apply je možné zadávat i anonymní funkce. Tedy funkce, které nemají jméno a uživatel je zadefinuje přímo při volání apply funkce. Více o anonymních funkcích si povíme v kapitole uživatelské funkce. Ukažme si ale dva příklady.

```
> apply(X = mtcars, MARGIN = 2, FUN = function(z) max(z) - min(z))
    mpg     cyl    disp      hp    drat      wt    qsec      vs
             am
 23.500   4.000 400.900 283.000   2.170   3.911   8.400   1.000
  1.000
   gear    carb
  2.000   7.000
```

V následujícím příkladu je vypočtena dolní a horní mez intervalu spolehlivosti pro data z datasetu `mtcars`. Vyjděme z předpokladu, že data mají normální rozdělení bez známého

rozptylu. Zajímavé na této úloze je to, že funkce `apply()` zvládne navrátit i více hodnot pro daný sloupec, ale musí mít stejnou délku. Dále si všimněte, jak byl omezen výběr dat (pomocí selekce sloupců) přímo v těle samotné funkce.

```
> apply(X = mtcars[, c(1, 3:7)], MARGIN = 2,
+   FUN = function(x)
+     mean(x) + c(-1, +1) *
+     (qt(p = 0.975, df = (length(x) - 1)) / sqrt(length(x)))
+ )
           mpg      disp       hp      drat        wt      qsec
[1,] 19.73009 230.3613 146.327 3.236024 2.856712 17.48821
[2,] 20.45116 231.0824 147.048 3.957101 3.577788 18.20929
```

Více si anonymní funkce rozebereme ale až v následujících kapitolách.

Příklad 1

V datasetu `mtcars` spočtěte pro každou proměnnou rozdíl mezi maximální a minimální hodnotou (tedy variační rozpětí).

Řešení 1

```
> apply(mtcars, 2,FUN = function(x) max(x) - min(x))
    mpg      cyl     disp       hp     drat       wt     qsec       vs
             am     gear     carb
 23.500   4.000 400.900 283.000   2.170   3.911   8.400   1.000
    1.000   2.000   7.000
```

Příklad 2

Některé datové objekty jsou občas spravovány jako vícerozměrná pole (`array`). V následujícím příkladu si ukážeme, jak pro třídimenzionální pole provést základní operaci – sečtení dimenzí.

Máte třídimenzionální krychli definovanou jako: `krychle = array(data = 1:27, dim = c(3, 3, 3))`. Jak byste provedli součet skrze dimenze tak, aby byly „nasečteny“ dimenze skrze jednotlivé řady? Tedy aby výsledkem byl součet skrze třetí dimenzi. Tedy chceme pomocí `apply()` funkce a funkce `sum()` získat výstup analogický k následujícímu kódu:

```
> krychle[,, 1] + krychle[,, 2] + krychle[,, 3]
     [,1] [,2] [,3]
[1,]   30   39   48
[2,]   33   42   51
[3,]   36   45   54
```

Řešení 2

Podíváme se nejprve na to, jak naše data vypadají.

```
> krychle = array(data = 1:27, dim = c(3, 3, 3))
> print(krychle)
, , 1

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

, , 2

     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18

, , 3

     [,1] [,2] [,3]
[1,]   19   22   25
[2,]   20   23   26
[3,]   21   24   27
```

Prostým zadáním 3 do parametru `MARGIN` získáme zcela jiný výsledek, než byste očekávali.

```
> # Prostym zadanim 3 dimenze se sectou uplne cele dimenze ale ne po
    prvcich:
> apply(krychle, MARGIN = 3, FUN = sum)
[1]  45 126 207
> Toto je analogicke k:
> c(sum(krychle[,, 1]), sum(krychle[,, 2]), sum(krychle[,, 3]))
[1]  45 126 207
```

Řešení je vložením funkce `apply()` do samotné funkce `apply()` tak, jak ukazuje následující kód.

```
> apply(krychle, MARGIN = 2, FUN = apply, 1, sum)
     [,1] [,2] [,3]
[1,]   30   39   48
[2,]   33   42   51
[3,]   36   45   54
```

4.2 Funkce lapply, sapply, vapply, replicate

4.2.1 Funkce lapply

Funkce `lapply()` oproti `apply()` nepracuje již s maticí/polem, ale naopak s vektorem dat. Může se jednat o atomický vektor nebo o vektor typu list (protože i list je vektorem, ale ne atomickým). Dále funkce `lapply()` pracuje s jakýmkoliv objektem, jenž lze převést na list pomocí funkce `as.list()`. Funkce `lapply()` má parametry: `X` – objekt, pro který se provádí, `FUN` – funkce, která se provádí. Použití funkce je pak identické s funkcí `apply()`, ale s tím rozdílem, že se zde již nespecifikuje dimenze dané úlohy (parametr `margin`), funkce je prováděna pro celé prvky listu. Vytvořme si následující list.

```
> L<-list(a = c(1:10), b = c(2:5),
+          c = c(-5:5), M = matrix(data = c(1:9),ncol = 3))
> print(L)
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$b
[1] 2 3 4 5

$c
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5

$M
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Funkce `sum()` se použije na všechny objekty zvlášť, protože není nikde zaručeno, že lze převést výsledek funkce na homogenní objekt z pohledu dat i dimenzí, tak je návratový objekt typu list.

Když pak použijeme funkci `lapply()` s funkcí `sum()`, tak funkce `lapply()` spočítá sumu přes celé objekty. Zajímavostí je ale i návratová hodnota. Objekt, který se navrátí, je taktéž typu list.

```
> lapply(X = L, FUN = sum)
$a
[1] 55

$b
[1] 14

$c
```

```
[1] 0

$M
[1] 45
```

Zajímavé na funkci `lapply()` je i to, že ji lze kombinovat s funkcí `apply()` a provést úlohu např. pro všechny proměnné objektu typu `data.frame`. Jediným problémem zde je, že funkce apply má stejná jména parametrů jako funkce `lapply()`. Tuto situaci lze vyřešit tak, že parametry pro funkci apply se zadají až po parametrech pro funkci `lapply()` a navíc bez jména parametru (nutná podmínka pro kolizní situace parametrů). Musíme proto vypsat parametry ve správném pořadí. Vytvoříme si list matic, pro který budeme chtít spočítat mediánovou hodnotu přes sloupce.

```
> L2 <- list(M1 = matrix(1:9, 3),
+            M2 = matrix(9:1, 3),
+            M3 = matrix(11:19, 3))
> print(L2)
$M1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$M2
     [,1] [,2] [,3]
[1,]    9    6    3
[2,]    8    5    2
[3,]    7    4    1

$M3
     [,1] [,2] [,3]
[1,]   11   14   17
[2,]   12   15   18
[3,]   13   16   19
> lapply(X = L2, FUN = apply, 2, median)
$M1
[1] 2 5 8

$M2
[1] 8 5 2

$M3
[1] 12 15 18
```

Ne všem může vyhovovat výstupní formát typu list. V tuto chvíli máme dvě možnosti, jak výstup z funkce apply převést např. na matici či pole. První způsob je pomocí funkce

`unlist()`. Funkce „odlistuje“ jakýkoliv list, jednotlivé elementy pojmenuje podle jména listu a podle indexu daného místa na dané pozici v objektu listu.

```
> L2 <- list(M1 = matrix(1:9, 3),
+            M2 = matrix(9:1, 3),
+            M3 = matrix(11:19, 3))
> print(L2)
$M1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$M2
     [,1] [,2] [,3]
[1,]    9    6    3
[2,]    8    5    2
[3,]    7    4    1

$M3
     [,1] [,2] [,3]
[1,]   11   14   17
[2,]   12   15   18
[3,]   13   16   19
> unlist(lapply(X = L2, FUN = apply, 2, median))

M11 M12 M13 M21 M22 M23 M31 M32 M33
  2   5   8   8   5   2  12  15  18
```

Dalším způsobem, jak převést list na jiný objekt, je za použití funkce `simplify2array()`. Pokud je délka jednotlivých elementů listu shodná, pak daný objekt funkce převede na pole.

```
> L2 <- list(M1 = matrix(1:9, 3),
+            M2 = matrix(9:1, 3),
+            M3 = matrix(11:19, 3))
> print(L2)
$M1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$M2
     [,1] [,2] [,3]
[1,]    9    6    3
[2,]    8    5    2
```

```
[3,]    7    4    1

$M3
     [,1] [,2] [,3]
[1,]   11   14   17
[2,]   12   15   18
[3,]   13   16   19
> simplify2array(lapply(X = L2, FUN = apply, 2, median))
     M1 M2 M3
[1,]  2  8 12
[2,]  5  5 15
[3,]  8  2 18
```

4.2.2 Funkce sapply

Funkce `sapply()` je formou `lapply()` funkce s tím rozdílem, že výstupní hodnotou není list, ale přímo pole/matice, pokud to lze. Pokud to možné není, tak funkce `sapply()` navrací stejně jako funkce `lapply()` list. Ukážeme si příklad, kdy první list obsahuje tři matice 3×3 a druhý list (L3) obsahuje místo toho jednu matici o velikosti 9×1.

```
> L2 <- list(M1 = matrix(1:9, 3),
+            M2 = matrix(9:1, 3),
+            M3 = matrix(11:19, 3))
> print(L2)
$M1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$M2
     [,1] [,2] [,3]
[1,]    9    6    3
[2,]    8    5    2
[3,]    7    4    1

$M3
     [,1] [,2] [,3]
[1,]   11   14   17
[2,]   12   15   18
[3,]   13   16   19
> sapply(X=L2,FUN=sum)
 M1  M2  M3
 45  45 135

```

```
> sapply(X=L2,FUN=apply,2, median)
     M1 M2 M3
[1,]  2  8 12
[2,]  5  5 15
[3,]  8  2 18

> L3 <- list(M1 = matrix(1:9, 3),
+            M2 = matrix(9:1, 9),
+            M3 = matrix(11:19, 3))
> print(L3)
$M1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$M2
      [,1]
 [1,]    9
 [2,]    8
 [3,]    7
 [4,]    6
 [5,]    5
 [6,]    4
 [7,]    3
 [8,]    2
 [9,]    1

$M3
     [,1] [,2] [,3]
[1,]   11   14   17
[2,]   12   15   18
[3,]   13   16   19
> sapply(X = L3, FUN = apply, 2, median)
$M1
[1] 2 5 8

$M2
[1] 5

$M3
[1] 12 15 18
```

Oproti funkci `lapply()` jsou parametry `simplify` (výchozí true) a `USE.NAMES` (taktéž) nové. Parametr `simplify` právě způsobí převod na pole namísto listu. Parametr `USE.NAMES` slouží v případě, kdy X je charakter, a způsobí použití X jako jména objektu.

4.2.3 Funkce vapply

Funkce `vapply()` je verzí funkce `sapply()` s tím rozdílem, že oproti `sapply()` je nutné definovat typ objektu, který se má navrátit. To, že definujeme typ vektoru (atomický typ elementů vektoru/pole, co se navrací), má nespornou výhodu při kontrole toho, zda funkce proběhla správně – tedy, že navrací hodnoty, jaké má. Další výhodou funkce oproti `sapply()` je navíc to, že je vykonána znatelně rychleji než funkce `sapply()`. Z těchto důvodů bývá funkce `vapply()` doporučována jako funkce, která se má používat spíše než `sapply()`. Použití funkce `vapply()` je pak následující:

```
> L <- list(
+    a = c(1:10),
+    b = c(2:5),
+    c = c(-5:5),
+    M = matrix(data = c(1:9), ncol = 3)
+    )
> print(L)
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$b
[1] 2 3 4 5

$c
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5

$M
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> vapply(X = L,
+        FUN = mean,
+        FUN.VALUE = numeric(1))
  a   b   c   M
5.5 3.5 0.0 5.0

> vapply(X = L,
+        FUN = mean,
+        FUN.VALUE = complex(1))
      a       b       c       M
5.5+0i 3.5+0i 0.0+0i 5.0+0i

> vapply(X = L,
+        FUN = mean,
```

```
+        FUN.VALUE = character(1))
Error in vapply(X = L, FUN = mean, FUN.VALUE = character(1)) :
  values must be type 'character',
 but FUN(X[[1]]) result is type 'double'
```

Parametr `FUN.VALUE` pak specifikuje typ proměnné, kterou má funkce `vapply()` navrátit. Zatímco double je intepretovatelná jako numerická i jako komplexní číslo, tak ji naopak jako charakter nelze navrátit. Již víme, že lze číslo převést na charakter pomocí funkce `as.character()`, ale zde jde o to, že návratová hodnota funkce mean není v souladu s očekávanou hodnotou zadefinovanou uživatelem.

4.2.4 Funkce replicate

Funkce `replicate()` replikuje aplikaci funkce jako takové. Tedy neprovádí se zde pro prvky určitého objektu jedna funkce, ale primárně se provádí replikace jedné a té samé funkce. Ukažme si použití této funkce na funkci `sample()` v následujícím příkladu.

```
> set.seed(1)
> replicate(n = 5, exp = sample(x = 1:100, size = 3))
     [,1] [,2] [,3] [,4] [,5]
[1,]   27   91   95    7   69
[2,]   37   20   66   21   39
[3,]   57   89   62   18   76
```

Funkce navrací výstup ve formátu matice, pokud je to možné. Parametr `n` určuje počet replikací dané funkce. Funkce `replicate()` je podobná funkci cyklu `for` s tím, že zde není přítomný iterátor. Funkce má zejména použití v simulacích a v uživateli definovanými funkcemi.

Příklad 3

Máte následující list L.

```
> L <- list(
+   L1 = sample(1:100, 10),
+   L2 = sample(1:100, 5),
+   L3 = sample(1:100, 9)
+ )
> print(L)
$L1
 [1] 19 82 66 78 11 69 39 77 60 72

$L2
[1] 56 53 78  3 46
```

```
12
13 $L3
14 [1] 74 69 47 84 43 24  7 10 30
```

Pro tento list chcete zjistit délky vektorů, které se v něm nacházejí. Zkuste to provést funkcemi `sapply()`, `lapply()` a `vapply()`.

Řešení 3

```
1  > lapply(L, FUN = length)
2  $L1
3  [1] 10
4
5  $L2
6  [1] 5
7
8  $L3
9  [1] 9
10
11 > sapply(L, FUN = length)
12 L1 L2 L3
13 10  5  9
14 > vapply(L, FUN = length, FUN.VALUE = numeric(1))
15 L1 L2 L3
16 10  5  9
```

4.3 Funkce mapply

Zvláštním případem `apply()` funkcí je funkce `mapply()`. Jedná se o verzi funkce `sapply()` spojenou s namapováváním parametrů. Funkce je zejména vhodná, když potřebujeme vyzkoušet různé varianty verze nějakého zpracování aj. Jako standardní příklad pro funkci `mapply()` se uvádí použití na funkci `rep()`. Tato funkce má dva hlavní parametry: `x` (to, co se má replikovat) a `times` (kolikrát se to má replikovat).

```
1 > mapply(rep ,x = 1:3, times = c(2, 4, 6))
2 [[1]]
3 [1] 1 1
4
5 [[2]]
6 [1] 2 2 2 2
7
8 [[3]]
9 [1] 3 3 3 3 3 3
```

Funkce `mapply()` nejen „namapuje“ prvky dvou vektorů na sebe, ale také i objekty v listech. Namapování zde pak probíhá prostřednictvím jednotlivých elementů listů. V následujícím případě na sebe napojíme ale list a vektor. První element listu (neatomický vektor) s prvním elementem vektoru (prvek o jedné hodnotě).

```
> mapply(rep, x = list(1:3, 3:4, 4:5), times = c(2, 3, 3))
     [,1] [,2] [,3]
[1,]    1    3    4
[2,]    2    4    5
[3,]    3    3    4
[4,]    1    4    5
[5,]    2    3    4
[6,]    3    4    5
```

```
> data1 <- data.frame(A1 = c(1, 2, 1, 3),
+                     B1 = c(1, 2, 3, 4),
+                     C1 = c(1, 1, 2, 1))
> data2 <- data.frame(A2 = c(1, 2, 3, 4),
+                     B2 = c(1, 2, 2, 5),
+                     C2 = c(2, 2, 3, 1))
> print(data1)
  A1 B1 C1
1  1  1  1
2  2  2  1
3  1  3  2
4  3  4  1
> print(data2)
  A2 B2 C2
1  1  1  2
2  2  2  2
3  3  2  3
4  4  5  1
> mapply(cor, data1, data2)
       A1        B1        C1
0.6741999 0.8944272 0.8164966
> # analogicky vystup by slo ziskat trojim pouziti funkce cor:
> cor(data1$A1, data2$A2)
[1] 0.6741999
> cor(data1$B1, data2$B2)
[1] 0.8944272
> cor(data1$C1, data2$C2)
[1] 0.8164966
```

Příklad 4

Jak byste pomocí funkce `mapply()` vytvořili následující matici?

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    3    4    5    6
[3,]    3    4    5    6    7
```

Řešení 4

Stačí funkci `mapply()` použít spolu s funkcí `seq()`.

```
> mapply(seq, from = 1:5, to = 3:7)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    2    3    4    5    6
[3,]    3    4    5    6    7
```

4.4 Funkce rapply

Fukce `rapply()` je rekurzivní verzí `lapply()` funkce. Výhodou funkce je i parametr how, který nám umožňuje definovat, jak má funkce iterovat. Tento parametr má tři základní možnosti, jak se mají data párovat: `unlist`, `replace` a `list`. Dále umožňuje definovat typ dat, pro která se má funkce provést (atribut class).

```
> L <- list(
+   a = c(1:10),
+   b = c(2:5),
+   c = c(-5:5),
+   M = matrix(data = c(1:9), ncol = 3)
+ )
> print(L)
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$b
[1] 2 3 4 5

$c
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5

$M
     [,1] [,2] [,3]
```

```
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> rapply(L, sum, how = "unlist")
 a  b  c  M
55 14  0 45
> rapply(L, sum, how = "replace")
$a
[1] 55

$b
[1] 14

$c
[1] 0

$M
[1] 45
> rapply(L, sum, how = "list")
$a
[1] 55

$b
[1] 14

$c
[1] 0

$M
[1] 45
```

Hlavní rozdíl mezi funkcí apply a funkcí rapply spočívá v tom, že u funkce rapply je nejprve objekt vyhodnocen (na splnění podmínek – např. typu objektu, pro který se má zpracovat) a až následně je volaná samotná funkce v jazyce C. Nebuďte zde překvapeni, že se jedná jazyk C. Software **R** je naprogramován v jiném jazyce než jazyk **R**. Většina základních funkcí je z důvodu optimalizace naprogramována v jazyce C, protože je výrazně rychlejší než samotný jazyk **R**. Do verze jazyka **R** 3.5.X bylo požadováno, aby vstupním objektem byl typ list, což již u novějších instalací není povinné a může jít i o tzv. list-like objekt či objekt typu „expression“, díky čemuž jsme schopni tuto funkci použít i pro `data.frame` objekt. Ukažme si toto na příkladu datasetu `iris`. Tento dataset obsahuje údaje o měření provedeném na různých druzích kosatců. Dataset obsahuje čtyři numerické proměnné a jednu kategoriální proměnnou uvažovanou jako typ objektu faktor (druh kosatce).

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
```

```
2           4.9         3.0          1.4          0.2  setosa
3           4.7         3.2          1.3          0.2  setosa
4           4.6         3.1          1.5          0.2  setosa
5           5.0         3.6          1.4          0.2  setosa
6           5.4         3.9          1.7          0.4  setosa
> rapply(iris, mean, classes = "numeric")
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333

>rapply(iris, summary, classes = "factor")
    Species.setosa Species.versicolor  Species.virginica
                50                 50                 50
```

4.5 Funkce tapply a by funkce

Funkce `tapply()` je specifickou funkcí z rodiny apply, protože se funkce `tapply()` používá na vektor dat. K tomuto vektoru dat ale dodáváme vektor tříd (obvykle faktorů), podle kterých chceme data ze vstupního vektoru rozdělit pro výpočet. Představme si tedy, že máme vektor pozorování příslušících k různým skupinám faktorů (například měření podle druhů rostlin) a chceme spočítat statistiky za dané skupiny zvlášť (druhy rostlin). Ukažme si dva příklady použití této funkce na datech `iris` a `mtcars`. V prvním případě spočítáme pro proměnnou `Sepal.Length` průměr pro jednotlivé druhy rostlin.

```
> tapply(X = iris$Sepal.Length,
+        INDEX = iris$Species,
+        FUN = mean)
    setosa versicolor  virginica
     5.006      5.936      6.588
```

Funkce `tapply()` nám srozumitelně navrací pojmenované pole (jednorozměrné pole) dat příslušných průměrů pro jednotlivé druhy rostlin. Co kdybychom si chtěli vypočítat průměrnou vzdálenost, kterou ujedou auta na jeden galon benzínu (`mpg`) podle druhu převodovky (`am`) z datasetu mtcars?

```
> tapply(X = mtcars$mpg,
       INDEX = mtcars$am,
       FUN = mean)
       0        1
17.14737 24.39231
```

Výhodou parametru `INDEX` je i to, že umí zpracovat více proměnných, podle kterých chceme výstup třídit. Můžeme si tedy spočítat průměrnou vzdálenost, kterou ujedou daná auta na galon benzínu podle druhu převodovky a počtu válců auta.

```
> > tapply(
+   X = mtcars$mpg,
+   INDEX = list(mtcars$am, mtcars$cyl),
+   FUN = mean
+ )
        4         6     8
0 22.900 19.12500 15.05
1 28.075 20.56667 15.40
```

Z výstupu je pak hezky vidět, že druh převodovky je důležitý z pohledu spotřeby auta, ale jeho rozdíl mezi skupinami klesá při nárůstu počtu válců.

Funkce `tapply()` se provádí pro jeden vektor. Co kdybychom ale chtěli spočítat data za všechny proměnné? Máme dvě možnosti. Buď použijeme funkci `tapply()` uvnitř funkce `apply()`, nebo použijeme tzv. wrapper funkci – `by()`. Zkusme nejprve „vložit“ funkci `tapply()` do funkce `apply()`. První problém, na který narazíme, je to, že funkce mají stejně pojmenované parametry. Toto lze vyřešit například tím, že parametr `FUN` pro funkci `tapply()` předáme nepojmenovaný.

```
> apply(
+   X = iris[, 1:4],
+   MARGIN = 2,
+   FUN = tapply,
+   INDEX = iris$Species,
+   mean
+ )
           Sepal.Length Sepal.Width Petal.Length Petal.Width
setosa            5.006       3.428        1.462       0.246
versicolor        5.936       2.770        4.260       1.326
virginica         6.588       2.974        5.552       2.026
```

Identický výsledek, avšak ve formě jiného objektu (objekt typu `by`), lze získat použitím funkce `by()`. Zkuste se teď podívat na obě řešení a najít zásadní rozdíly.

```
> by(data = iris[, 1:4],
+    INDICES = iris$Species,
+    FUN = colMeans)
iris$Species: setosa
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.006        3.428        1.462        0.246
------------------------------------------------------------
iris$Species: versicolor
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       5.936        2.770        4.260        1.326
------------------------------------------------------------
iris$Species: virginica
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
       6.588        2.974        5.552        2.026
```

Je zřejmé, že prvním rozdílem ve způsobu provedení jsou rozdílné typy výstupních objektů. Funkce `apply()` nám navrátila matici, se kterou se dá dobře pracovat. S objektem typu by lze také pracovat dál, ale není to přímočaré (například převést na list, dále provést `unlist()` a nebo převést na vektor přímo). Za hlavní rozdíl lze ale považovat to, že funkce `tapply()` používá funkci `mean()` a funkce `by()` funkci `colMeans()`. S funkcí `mean()` by funkce `by()` vůbec nefungovala. To je dáno tím, že funkce `by()` zpracovává vstupní objekt jako celek. Pokud vám funkce by nevyhovuje a přijde vám uživatelsky příjemnější použití funkce `tapply()` v `apply()` funkci, tak mám pro vás dobrou zprávu: funkce `aggregate()` je řešením této situace.

Příklad 5

Jak byste pomocí funkce `tapply()` spočítali průměrnou hodnotu proměnné `mpg` z datasetu `mtcars` v třídění pro auta podle toho, zda mají podprůměrnou anebo nadprůměrnou proměnnou `hp`?

Řešení 5

```
> tapply(mtcars$mpg,
+         INDEX = mtcars$hp > mean(mtcars$hp),
+         FUN = mean)
   FALSE     TRUE
24.22353 15.40667
```

Příklad 6

Chcete zjistit počet aut podle počtu válců a podle toho, zda mají podprůměrnou a nadprůměrnou hodnotu `hp`.

Řešení 6

```
> tapply(mtcars$mpg,
+         INDEX = list(mtcars$hp > mean(mtcars$hp), mtcars$cyl),
+         FUN = length)
        4 6  8
FALSE  11 6 NA
TRUE   NA 1 14
```

Příklad 7

Dokázali byste úlohu z příkladu 5 provést pro všechny proměnné v datasetu?

Řešení 7

```
> by(mtcars,
+    INDICES = as.factor(mtcars$hp > mean(mtcars$hp)),
+    FUN = colMeans)
as.factor(mtcars$hp > mean(mtcars$hp)): FALSE
        mpg         cyl        disp          hp
 24.2235294   4.7058824 134.9823529  93.5294118
       drat          wt        qsec          vs
  3.8976471   2.5995882  18.8735294   0.8235294
         am        gear        carb
  0.5882353   3.9411765   2.0588235
------------------------------------
as.factor(mtcars$hp > mean(mtcars$hp)): TRUE
        mpg         cyl        disp          hp
 15.406667   7.866667 339.226667 206.933333
       drat          wt        qsec          vs
  3.255333   3.917267  16.687333   0.000000
         am        gear        carb
  0.200000   3.400000   3.666667
```

Příklad 8

Zvládli byste vypočítat korelační matici pro numerické proměnné v datasetu `iris` tříděného podle druhu rostliny (`Species`) pomocí funkce `by()`?

Řešení 8

```
> by(iris[, 1:4],INDICES = iris$Species, FUN = cor)
iris$Species: setosa
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.7425467    0.2671758   0.2780984
Sepal.Width     0.7425467   1.0000000    0.1777000   0.2327520
Petal.Length    0.2671758   0.1777000    1.0000000   0.3316300
Petal.Width     0.2780984   0.2327520    0.3316300   1.0000000
------------------------------------------------------------
iris$Species: versicolor
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.5259107    0.7540490   0.5464611
```

```
Sepal.Width      0.5259107   1.0000000    0.5605221   0.6639987
Petal.Length     0.7540490   0.5605221    1.0000000   0.7866681
Petal.Width      0.5464611   0.6639987    0.7866681   1.0000000
------------------------------------------------------------------------
iris$Species: virginica
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.4572278    0.8642247   0.2811077
Sepal.Width     0.4572278   1.0000000    0.4010446   0.5377280
Petal.Length    0.8642247   0.4010446    1.0000000   0.3221082
Petal.Width     0.2811077   0.5377280    0.3221082   1.0000000
```

4.6 Funkce aggregate a ts objekt

Funkce `aggregate()` je agregační funkce, která umožňuje pro různé typy objektů vykonávat různé úlohy. Funkce umí, tak jako funkce `lapply()`, vykonat určitou funkci na vektoru, který třídíme do určitých skupin. Výhodou funkce `aggregate()` je, že výstupem v tomto případě je objekt typu `data.frame`. Jako hlavní parametr vložíme požadovaný vektor a do parametru by vložíme list, podle kterého chceme vykonat funkci (parametr `FUN`).

```
> aggregate(mtcars$mpg, by = list(mtcars$am), FUN = mean)
  Group.1        x
1       0 17.14737
2       1 24.39231
```

Výstupem funkce `aggregate()` je vždy `data.frame`. V případě více klasifikací objektu (kategorií, do kterých prvky objektu třídíme), pro který danou úlohu vykonáváme, nám funkce `lappy()` navrací vícerozměrné pole, kde každý rozměr odpovídá dané klasifikaci dat (`am`, `gear`...). V případě funkce `aggregate()` nám funkce ale vždy navrátí `data.frame`. V případě vícerozměrné úlohy je výstupní objekt stále `data.frame`, ale přibude zde další sloupec proměnné, podle kterého identifikujeme výslednou statistiku ve sloupci `x`.

```
> aggregate(mtcars$mpg,
          by = list(mtcars$am, mtcars$cyl),
          FUN = mean)
  Group.1 Group.2        x
1       0       4 22.90000
2       1       4 28.07500
3       0       6 19.12500
4       1       6 20.56667
5       0       8 15.05000
6       1       8 15.40000
```

Čtenáři doporučuji srovnat tento výstup s výstupem funkce `tapply()`. Hlavní potenciál funkce `aggregate()` ale spočívá v použití na celý objekt jako takový. Toto použití je pro typ objektu `data.frame`, ale výpočet se provede pro jakýkoliv objekt, který lze převést na `data.frame` pomocí funkce `as.data.frame()`.

```
> aggregate(mtcars,
+               by = list(mtcars$gear),
+               FUN = mean)
  Group.1       mpg       cyl      disp        hp      drat
1       3 16.10667 7.466667 326.3000 176.1333 3.132667
2       4 24.53333 4.666667 123.0167  89.5000 4.043333
3       5 21.38000 6.000000 202.4800 195.6000 3.916000
         wt   qsec         vs         am gear      carb
1 3.892600 17.692 0.2000000 0.0000000    3 2.666667
2 2.616667 18.965 0.8333333 0.6666667    4 2.333333
3 2.632600 15.640 0.2000000 1.0000000    5 4.400000
```

Stejně jako do jakékoliv funkce z rodiny `apply()`, tak i do funkce `aggragate()` můžeme přiřadit anonymní funkci. O tomto se více dozvíte v kapitole Uživatelské funkce. Zde ale uvedeme jednoduchý příklad.

```
> aggregate(
+   x = iris[, 1:4],
+   by = list(iris$Species),
+   FUN = function(x)
+     sd(x) / mean(x)
+ )
      Group.1 Sepal.Length Sepal.Width Petal.Length Petal.Width
1      setosa   0.07041344   0.1105789   0.11878522   0.4283967
2  versicolor   0.08695606   0.1132846   0.11030774   0.1491348
3   virginica   0.09652089   0.1084387   0.09940466   0.1355627
```

Pro datový soubor `iris` jsme spočítali variační koeficient. Tedy směrodatnou odchylku dělenou průměrem. **R** nemá funkci, která by toto přímo spočítala v základní instalaci. Proto jsme si do parametru `FUN` dosadili postup výpočtu, který chceme pro dané sloupce provést. Více se o této problematice dozvíte ale až v kapitole věnované tvorbě funkcí. Jediná nevýhoda funkce `aggregate()` je pak v tom, že oproti funkci `by` nezvládne zpracovat maticový výstup (viz příklad 8).

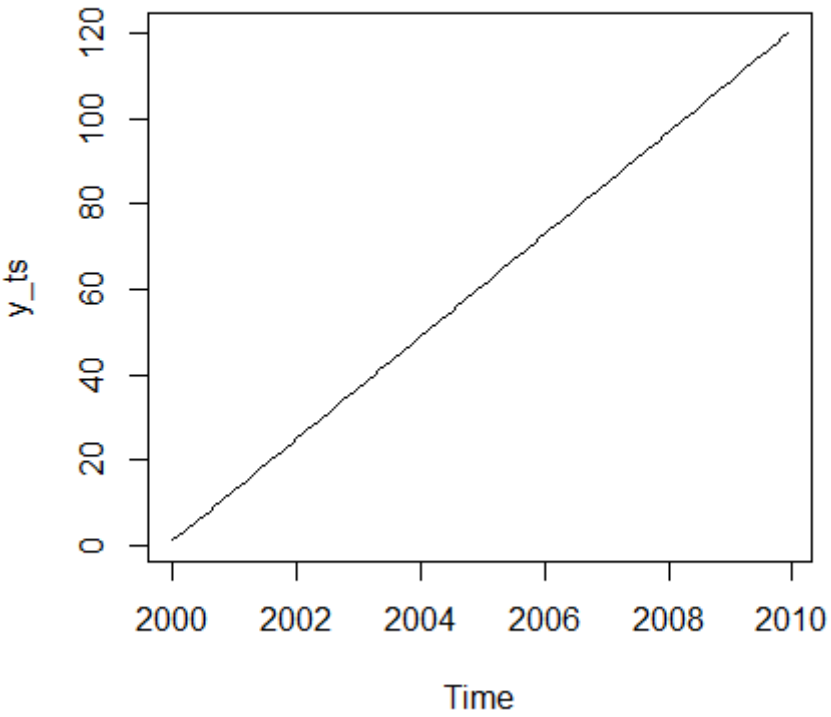
Speciální použití funkce `aggregate()` pak souvisí s objektem typu časové řady. Časové řady vytváříme v jazyku **R** pomocí funkce ts. Tato funkce má jako vstupní parametr data (vektor a nebo i data.frame objekt), dále parametr `start` (počáteční období) a `end` (poslední záznam). Další důležitý parametr je četnost pozorování na jednotku času – `frequency`. Jedná se o to, s jakou periodicitou je daná časová řada pozorována. V případě čtvrtletní časové řady volíme `frequency = 4`, v případě měsíční `frequency = 12`. Vytvoříme si měsíční časovou řadu z vektoru `y`, který nabývá hodnot jedna až sto dvacet. A řekněme, že první pozorování je za leden roku 2000.

```
> y = c(1:120)
> y_ts <- ts(data = y,
+            start = c(2000, 1),
+            frequency = 12
> y_ts
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2000   1   2   3   4   5   6   7   8   9  10  11  12
2001  13  14  15  16  17  18  19  20  21  22  23  24
2002  25  26  27  28  29  30  31  32  33  34  35  36
2003  37  38  39  40  41  42  43  44  45  46  47  48
2004  49  50  51  52  53  54  55  56  57  58  59  60
2005  61  62  63  64  65  66  67  68  69  70  71  72
2006  73  74  75  76  77  78  79  80  81  82  83  84
2007  85  86  87  88  89  90  91  92  93  94  95  96
2008  97  98  99 100 101 102 103 104 105 106 107 108
2009 109 110 111 112 113 114 115 116 117 118 119 120
```

V případě, že využijeme parametru start, již nemusíme použít parametr end a naopak. **R**

má pro objekt `ts` předpřipravené chování několika základních funkcí. Podívejme se, co se stane, když na objekt `ts` zavoláme funkci `plot`.

```
1 plot(y_ts)
```



Obrázek 4.1: Plot funkce

Zdroj: Vlastní zpracování.

Funkce `plot()` má předprogramované procedury pro objekty typu `ts` a vykreslí je předem předpřipraveným způsobem. O tom, jak se dělají a upravují grafy, si však povíme více až v následujících kapitolách. U časových řad často řešíme situaci, kdy chceme agregovat data. Pro tento případ má funkce `aggregate()` parametr `nfrequency` neboli „new frequency“. Do tohoto parametru specifikujete četnost nové časové řady. Ukážeme si to na následujícím příkladu.

```
1 > aggregate(x = y_ts, nfrequency = 4)
2     Qtr1  Qtr2  Qtr3  Qtr4
```

```
2000    6   15   24   33
2001   42   51   60   69
2002   78   87   96  105
2003  114  123  132  141
2004  150  159  168  177
2005  186  195  204  213
2006  222  231  240  249
2007  258  267  276  285
2008  294  303  312  321
2009  330  339  348  357
```

Kam ale zmizel parametr `FUN`? V případě, že vstupním objektem je časová řada, má parametr `FUN` přiřazenou výchozí hodnotu na funkci sum. Automaticky se tedy provede součet. V případě, že chceme provést pro daná čtvrtletí jiný výpočet, je nutné specifikovat parametr `FUN`. Chceme-li například spočítat průměr za daná čtvrtletí, tak toho dosáhneme pomocí následujícího příkladu.

```
> aggregate(x = y_ts,
+           nfrequency = 4,
+           FUN = mean)
     Qtr1 Qtr2 Qtr3 Qtr4
2000    2    5    8   11
2001   14   17   20   23
2002   26   29   32   35
2003   38   41   44   47
2004   50   53   56   59
2005   62   65   68   71
2006   74   77   80   83
2007   86   89   92   95
2008   98  101  104  107
2009  110  113  116  119
```

Zkusme si toto použití ukázat na reálných datech. V **R** se kromě datasetů `mtcars` a `iris` nachází také dataset `AirPassengers`. Jedná se o měsíční počty pasažérů mezi roky 1949 až 1960 v tisících v letecké dopravě. Zkusme si pro ně vypočítat variabilitu v jednotlivých čtvrtletích pomocí funkce `sd`.

```
> aggregate(x = AirPassengers,
+           nfrequency = 4,
+           FUN = sd)
          Qtr1      Qtr2      Qtr3      Qtr4
1949 10.263203  7.023769  6.928203  8.386497
1950 13.051181 12.055428  6.928203 13.453624
1951 17.785762  7.549834  8.660254 10.583005
1952 11.060440 20.808652 16.703293 11.930353
1953 23.094011  7.023769 18.339393 15.821926
```

```
1954 23.895606 19.655364 22.678918 15.011107
1955 17.616280 26.274195 26.514147 22.605309
1956 21.361960 33.867388 31.432467 20.207259
1957 28.583212 40.853396 35.809682 21.779195
1958 22.000000 46.508064 54.720502 24.542480
1959 33.005050 38.850139 52.538874 25.423087
1960 15.620499 39.929104 61.719797 35.697806
```

Příklad 9

Zkuste, jako v příkladu č. 7, vypočítat průměrnou hodnotu jednotlivých proměnných v datasetu `mtcars` tříděného podle toho, zda auto má anebo nemá nadprůměrnou hodnotu proměnné `hp`.

Řešení 9

```
> aggregate(mtcars,
+ by = list(mtcars$hp > mean(mtcars$hp)), FUN = mean)
  Group.1       mpg       cyl      disp         hp      drat        wt
1   FALSE 24.22353 4.705882 134.9824  93.52941 3.897647 2.599588
2    TRUE 15.40667 7.866667 339.2267 206.93333 3.255333 3.917267
      qsec         vs         am      gear      carb
1 18.87353 0.8235294 0.5882353 3.941176 2.058824
2 16.68733 0.0000000 0.2000000 3.400000 3.666667
```

Kapitola 5

Tvorba vlastních funkcí

5.1 Uživatelem definované funkce

Funkci si můžeme představit jako určitou část kódu, kterou někdo zabalil a upravil tak, abychom mohli zadat určité parametry (argumenty) a funkce nám navrátila výsledek. Funkce obvykle interaguje s programovacím prostředím pomocí parametrů funkce a návratové hodnoty funkce. Návratovou hodnotou se rozumí hodnota (neboli výsledek), kterou funkce navrací (více viz. Wickham, 2020a; Long, Teetor, 2019; R studio team, 2018).

Funkce je z teoretického pohledu podprogramem, který umožňuje abstraktnější práci s kódem. Výhodou tvorby funkcí je, že kód lze zpřehlednit a mnoho operací i zautomatizovat. Uživatel jazyka **R** by měl brát na zřetel, že funkce jsou objekty jako jiné v jazyku **R**. S uživatelem definovanými funkcemi lze pracovat jako s jinými objekty – například se dají odstranit. Uživatelem definovaná funkce se uloží po boku ostatních definovaných objektů a lze si je zobrazit pomocí funkce `ls()`. Následující kód ilustruje to, jak taková definice funkce vypadá a z čeho se skládá.

```
1 Jmeno_funkce <- function(#parametry_funkce){
2 # Telo_funkce
3 }
```

Uživatelem definovaná funkce („user defined function“) má následující části:

- jméno,
- parametry funkce neboli parametry,
- tělo funkce (kód, který se realizuje),
- návratovou hodnotu (je součástí těla funkce).

Ukažme si v praxi velmi jednoduchou funkci, která pouze sečte dvě čísla.

```
> secist <- function(x, y) {
+   x + y
+ }
```

Nejprve je nutné funkci vyhodnotit neboli zavést (tedy spustit v příkazové řádce). Příkazová řádka při spuštění kódu funkce nic nenavrátí. Jedná se jen o inicializaci funkce a není třeba (v případě, že nenastala chyba) cokoliv do příkazové řádky navracet. To, že funkce je k dispozici, si například ověříme tak, že zavoláme příkaz ls a podíváme se na objekty, které jsou dostupné.

```
> ls()
[1] "secist
```

Jak vidíte, v mém případě je funkce již vytvořená. Zobrazuje se v seznamu objektů. Funkci používáme tak, že napíšeme její jméno a do kulatých závorek pak parametry funkce.

```
> secist(x = 2, y = 3)
[1] 5
```

V případě, kdy zapomeneme za jméno funkce napsat závorky a zavoláme jen jméno dané funkce, tak jako výsledek se nám do příkazové řádky navrátí celá samotná funkce – její kód.

```
> secist
function(x, y) {
  x + y
}
```

V následujících kapitolách se podíváme na jednotlivé aspekty tvorby funkcí v jazyku **R**.

5.1.1 Jméno funkce

U jmen funkcí, tak jako u jmen objektů, dodržujeme pravidlo, že jméno nesmí začínat číslicí a nesmí obsahovat speciální znak. Dále je nutné se vyhnout rezervovaným jménům. Ne každá uživatelem definovaná funkce ale musí mít jméno. Takovým případem je anonymní funkce. Tento typ funkce má speciální použití zejména v matematických aplikacích nebo ve funkcích typu `apply()`. Obvykle se používá v situaci, kdy se očekává, že funkce bude použita jen na jednom místě v kódu. Následující funkce se jmenuje `rozpeti()` a přebírá jediný parametr `x`, ze kterého spočítá rozpětí (maximum – minimum dat). Jedná se o uživatelem definovanou funkci, která má jméno.

```
> rozpeti <- function(x) {
+   max(x) - min(x)
+ }
```

Alternativně si můžeme ukázat analogickou funkci přímo použitou ve funkci apply.

```
> apply(X = mtcars,
+       MARGIN = 2,
+       FUN = function(x) max(x) - min(x)
+ )
    mpg     cyl    disp      hp    drat      wt    qsec
 23.500   4.000 400.900 283.000   2.170   3.911   8.400
     vs      am    gear    carb
  1.000   1.000   2.000   7.000
```

Samozřejmě lze funkci `rozpeti()` použít i ve funkci `apply()`. Musí být ale funkce `rozpeti` zavedena. Zavedením funkce se zde rozumí to, že daná funkce již existuje v našem prostředí. Funkce se stávají součástí uživatelského prostředí tak jako jiné objekty a můžeme si je zobrazit pomocí `ls` funkce.

```
> apply(X = mtcars, MARGIN = 2, FUN = rozpeti)
    mpg     cyl    disp      hp    drat      wt    qsec
 23.500   4.000 400.900 283.000   2.170   3.911   8.400
     vs      am    gear    carb
  1.000   1.000   2.000   7.000
```

5.1.2 Parametry funkce

U každé funkce rozlišujeme tři druhy parametrů funkce. Jedná se o parametry funkce bez výchozí hodnoty, s výchozí hodnotou a dodatečné parametry. Funkce bez výchozích parametrů jsou ty, do kterých musí být přiřazena hodnota (pokud nastane situace, že se mají v tělu funkce použít). Parametry s výchozí hodnotou jsou ty, do kterých je přiřazena výchozí hodnota, a tedy uživatel nemusí vkládat nějakou hodnotu. Uživatel má ale možnost výchozí hodnotu nahradit svou vlastní hodnotou. Toto se provede prostým přiřazením nové hodnoty do daného parametru při aplikaci funkce – tak jako v případě povinných parametrů. Zajímavostí je, že výchozí hodnota nemusí být vyloženě hodnotou, ale může se jednat i o výpočet. Dokonce existují situace, kdy lze většinu výpočtu realizovat na úrovni parametrů funkce. Třetím typem parametrů jsou dodatečné parametry, tedy zda funkce může přijmout nějaké předem neznámé parametry. Podívejme se nejprve na příklad funkce s povinným parametrem, kdy pro `x` není vložena výchozí hodnota. Tato funkce vypočte variační rozpětí.

```
> rozpeti <- function(x) {
+   max(x) - min(x)
+ }
> rozpeti(1:10)
[1] 9
```

Následující kód ilustruje použití funkce pro mocnění s přiřazenými výchozími hodnotami. Parametr s výchozí hodnotou zadefinujeme pomocí jména daného parametru, rovnítka a hodnoty, které nabývá. Hodnoty přiřazené v rámci definice funkce nazýváme výchozí hodnoty, a pokud je uživatel nevloží do daných parametrů funkce, přebere dané hodnoty, jako by je definoval sám uživatel. Díky tomu, že funkce má pro oba parametry definovanou výchozí hodnotu, lze ji použít i bez jakýchkoli parametrů.

```
> sqxy <- function(x = 1, y = 0) {
+   x ** y
+ }
>
> sqxy()
[1] 1
```

V případě, kdy uživatel nezadá žádné hodnoty, bude hodnota `x` rovna jedné a `y` rovna nule. Alternativně lze pro oba parametry přiřadit hodnoty uživatelem. Uživatelem vložené hodnoty mohou být vložené v pořadí, v jakém je funkce uvádí, anebo pomocí parametrů funkce. Když využíváme přiřazení do parametrů dané funkce, tak nemusíme dodržet pořadí parametrů.

```
> sqxy(2, 3)
[1] 8

> sqxy(x = 2, y = 3)
[1] 8

> sqxy(2, y = 3)
[1] 8

> sqxy(y = 3, x = 2)
[1] 8
```

V případě, kdy chceme vynechat zadání parametru `x` a chceme jen definovat parametr `y`, tak musíme s ohledem na pořadí použít přiřazení hodnoty do příslušného parametru. A to proto, že parametr `x` je definovaný v deklaraci funkce jako první. Ukažme si tento jev v následujícím příkladu na situaci, kdy `y` je rovno třem.

```
> sqxy(y = 3)
[1] 1
```

Všichni známe Pythagorovu rovnici pro pravoúhlý trojúhelník ($c^2 = a^2 + b^2$) ze středoškolského učiva. Následující kód ilustruje implementaci zmíněné rovnice pomocí funkce v jazyku **R**. Schválně jsme zde zvolili velmi specifickou implementaci. Následující příklad ilustruje funkci, jejíž celý výpočet se realizuje na úrovni definice parametrů funkce. Uživatel zadá hodnoty dvou parametrů a funkce zbylé hodnoty dopočítá – v samotné definici parametrů.

```
> pythagoras <- function(a = sqrt(c ^ 2 - b ^ 2),
+                        b = sqrt(c ^ 2 - a ^ 2),
+                        c = sqrt(a ^ 2 + b ^ 2)) {
+   list(a = a, b = b, c = c)
+ }
> pythagoras(a = 1, c = 5)
$a
[1] 1

$b
[1] 4.898979

$c
[1] 5
```

Tento způsob definice parametrů funkce je velmi efektivní zejména v situaci, kdy předpokládáme, že uživatel je znalý dané problematiky (a tak nepředpokládáme, že by zadal sám všechny tři parametry, či že by zadal jen jeden). Daná definice má své výhody. Jedná se zejména o zefektivnění z pohledu rychlosti evaluace (vyhodnocení) dané funkce. Protože v případě, kdy bychom definovali výpočet hodnot chybějící strany trojúhelníku v těle funkce, pak by musela nad rámec výpočtu být spuštěna i `if` podmínka, která by danou situaci ověřila.

5.1.3 Tělo funkce

Tělo funkce představuje část funkce ve složených závorkách (popřípadě výraz následující za definicí parametrů funkce). V rámci těla funkce se realizuje samotný výpočet funkce. Jeho součástí může být i definice jiných funkcí.

```
> statistika <- function(x) {
+   prumer  <- function(x) sum(x) / length(x)
+   sm.odch <- function(x) sqrt(var(x))
+   rozpeti <- function(x) max(x) - min(x)
+
+   list(
+     prumer = prumer(x),
+     sm.odchylka = sm.odch(x),
+     rozpeti = rozpeti(x)
+   )
+ }

> statistika(1:10)
$prumer
[1] 5.5
```

```
16
17 $sm.odchylka
18 [1] 3.02765
19
20 $rozpeti
21 [1] 9
```

Výhodou této definice je to, že funkce nejsou dostupné ve standardním uživatelském prostředí. Jako všechny hodnoty definované uvnitř těla funkce, tak i tyto funkce po realizaci funkce mizí a nejsou nikde uložené, tedy ani uživateli nejsou dostupné.

Prostředí funkce

Uživatelské prostředí obsahuje uživatelem definované proměnné a funkce. Můžeme si je zobrazit například pomocí funkce `ls`. Toto již známe z předcházejících kapitol. Uživatelské funkce se jmenují právě proto, že objekty v nich definuje uživatel. **R** má k dispozici ale více prostředí. Při realizaci jakékoliv funkce si daná funkce sama vytvoří vlastní prostředí s proměnnými. Toto prostředí není pro uživatele běžně přístupné a po tom, co funkce skončí, tak prostředí funkce zaniká. Ukažme si tuto situaci na následujícím příkladu.

```
1  > x <- 1
2  > print(x)
3  [1] 1
4  >
5  > moje_x <- function(y = 10) {
6  +   x <- y
7  +   list(x = x)
8  + }
9  >
10 > print(x)
11 [1] 1
12 > print(moje_x())
13 $x
14 [1] 10
15
16 > print(x)
17 [1] 1
```

Z předcházejícího příkladu je poměrně snadno vidět, že proměnná x zůstává proměnnou x i po tom, co je vyhodnocena funkce, která obsahuje sama definici stejně pojmenované proměnné. Důvodem, proč nedojde ke změně proměnné x, je to, že ta je funkcí vytvořena jen v rámci prostředí funkce, které navíc po vykonání funkce zaniká.

U parametrů funkce jsme schválně vynechali zmínku o tom, že funkce může přebírat i proměnné z nadřazeného prostředí. Tedy dokáže si „sáhnout“ pro proměnnou, která je nad

prostředím, ve kterém se funkce realizuje. Tuto situaci si můžeme ukázat na následujícím příkladu. Proměnná `y` není nijak použita v samotném těle funkce, a tak její výchozí hodnota nemá žádný dopad na samotný výsledek funkce. Co ale má dopad, je proměnná `x` definovaná v uživatelském prostředí.

```
> x <- 1

> moje_x <- function(y = 10) {
+   list(x = x)
+ }
>
> moje_x()
$x
[1] 1

> x <- 10
> moje_x()
$x
[1] 10
```

To, že software a jazyk **R** postupuje ve vyhledávání dané proměnné postupně vždy do nadřazeného prostředí, ve kterém byla funkce definovaná, si můžeme hezky ukázat na následujících dvou příkladech. V tomto příkladu je `funkce_2()` vnořená do `funkce_1()`. Pokud `funkce_2()` nenajde v těle funkce určitou proměnnou, nejprve ji vyhledá v první funkci (`funkce_1()`) a pak až následně v uživatelském prostředí. Tedy přesto, že máme v uživatelském prostředí do hodnoty `x` přiřazenou hodnotu jedna, tak je navrácena hodnota `funkce_1()`.

```
> x <- 1
> funkce_1 <- function() {
+   funkce_2 <- function()
+     list(x = x)
+   x <- "funkce_1"
+   funkce_2()
+ }

> funkce_1()
$x
[1] "funkce_1"
```

Opakem k této situaci je pak situace, kdy funkce 2 je definována mimo funkci 1. Pak přesto, že ve `funkce_1()` je definována proměnná `x`, si `funkce_2()` „nesáhne“ na proměnnou `x` z `funkce_1()`, ale do uživatelského prostředí. Toto chování je způsobené tím, že `funkce_2()` je vyhodnocená v rámci `funkce_1()`, ale přitom sama je definována na stejné úrovni jako `funkce_1()`. Tedy v rámci uživatelského prostředí.

```
> x <- "uzivatelske prostredi"
> funkce_2 <- function() {
+   x
+ }

> funkce_1 <- function() {
+   x <- "funkce_1"
+   list(x = funkce_2())
+ }

> funkce_1()
$x
[1] "uzivatelske prostredi"

> funkce_2()
[1] "uzivatelske prostredi"
```

Lazy evaluation

Lazy evaluation je termín popisující postup vyhodnocení proměnných (a vlastně také to, zda proměnná vůbec existuje). Existence proměnné je ověřována, až když je volána samotným programem. Tedy dokud není potřebná při výpočtu, tak se o ni software nezajímá a je jedno, jestli obsahuje chybnou hodnotu, či zda vůbec existuje. Tuto situaci dobře ilustruje následující kód. Vyjdeme z předpokladu, že nemáte v prostředí jazyka definovanou proměnnou `Y`. Tato proměnná se ale používá v těle funkce. V případě, kdy se nevyhodnocuje část kódu, kde se daná proměnná používá, tak funkce proběhne bez chyby. Naopak, když dojde k realizaci části kódu s touto proměnnou, nastává chyba a výkon funkce se zastaví. Tímto postupem se v jazyku **R** vyhodnocují všechny funkce.

```
> mojefce <- function(x) {
+   if (x == 1) {
+     print(x)
+   } else{
+     print(Y)
+   }
+ }

> mojefce(1)
[1] 1
> mojefce(3)
Error in print(Y) : object 'Y' not found
```

V předcházejícím případě není nikde zadefinovaná proměnná `Y` (ani na úrovni jiného prostředí), tak při druhém pokusu se navrací chybová hláška. Všimněte si ale, že v případě

prvního pokusu proběhne celá realizace kódu bez chyby. To je dáno tím, že se část kódu s proměnnou `Y` při splnění podmínky `x==1` nerealizovala.

5.1.4 Návratová hodnota funkce

Každá funkce v jazyce **R** má návratovou hodnotu. Pokud není návratová hodnota definována, navrací funkce hodnotu `NULL`. Z tohoto důvodu některé grafické funkce navrací `NULL`, pokud se přiřadí do objektu. Návratová hodnota je standardně ta hodnota, která se vyhodnocovala (zavolala) jako poslední. V případě komplexních úloh může nastat situace, kdy si nejsme jisti, jaká hodnota byla poslední vyhodnocená. Z tohoto důvodu umožňuje jazyk **R** vynutit, jaký objekt funkce navrátí. Toto lze provést pomocí příkazu `return`, do něhož se jako parametr vloží objekt, který má daná funkce navrátit. Jak ale postupovat, pokud chceme navrátit několik možných objektů? To lze snadno vyřešit pomocí objektu typu list.

```
> mojefce <- function() {
+   return(list(a = 1, b = matrix(1:4, 2)))
+   x <- 10
+   x
+ }
> mojefce()
$a
[1] 1

$b
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Příklad 1

Vytvořte funkci, která přebere jako parametr numerický vektor dat a provede jeho diferenci. Otestujte na vektoru `x<-1:10`.

Řešení 1

```
> diference <- function(x) {
+   return(x[2:(length(x))] - x[1:(length(x) - 1)])
+ }

> x <- 1:10

> diference(x)
```

```
[1] 1 1 1 1 1 1 1 1 1
```

Příklad 2

Vytvořte funkci, která od vstupního vektoru dat odečte průměr a výsledek vydělí směrodatnou odchylkou. Tento matematický postup se nazývá standardizace dat, někdy taktéž standardizované skóre (z-score). Použijte danou funkci na datech `-2:2` a zkuste pro výsledná data vypočítat průměr a směrodatnou odchylku.

Řešení 2

```
> zscore <- function(x) {
+   return((x - mean(x)) / sd(x))
+ }
> # Vysledek funkce ulozime do promenne y
> # se kterou budeme pak dale pracovat
> y <- zscore(-2:2)
> print(y)
[1] -1.2649111 -0.6324555  0.0000000  0.6324555  1.2649111
> # prumer a smerodatna odchylka:
> mean(y)
[1] 0
> sd(y)
[1] 1
```

Příklad 3

Vytvořte tzv. wrapper funkci. Wrapper funkce je funkce, která obaluje nějakou funkci a automatizuje její použití na jiném datasetu. Obvykle jsme v situaci, kdy máme funkci, jejíž použití je určené na vektor, a chceme vytvořit wrapper funkci, která automatizuje její použití na objektu typu `data.frame`. Vytvořte wrapper funkci pro naši funkci `zscore`.

Řešení 3

```
> zscore <- function(x) {
+   return((x - mean(x)) / sd(x))
+ }

> zscore_dataset <- function(x) {
+   return(apply(x, 2, FUN = zscore))
+ }
```

```
> mtcars_st = zscore_dataset(mtcars)
> head(mtcars_st)
                           mpg        cyl         disp         hp
Mazda RX4           0.1508848 -0.1049878 -0.57061982 -0.5350928
Mazda RX4 Wag       0.1508848 -0.1049878 -0.57061982 -0.5350928
Datsun 710          0.4495434 -1.2248578 -0.99018209 -0.7830405
Hornet 4 Drive      0.2172534 -0.1049878  0.22009369 -0.5350928
Hornet Sportabout  -0.2307345  1.0148821  1.04308123  0.4129422
Valiant            -0.3302874 -0.1049878 -0.04616698 -0.6080186
                          drat         wt        qsec         vs
Mazda RX4           0.5675137 -0.610399567 -0.7771651 -0.8680278
Mazda RX4 Wag       0.5675137 -0.349785269 -0.4637808 -0.8680278
Datsun 710          0.4739996 -0.917004624  0.4260068  1.1160357
Hornet 4 Drive     -0.9661175 -0.002299538  0.8904872  1.1160357
Hornet Sportabout  -0.8351978  0.227654255 -0.4637808 -0.8680278
Valiant            -1.5646078  0.248094592  1.3269868  1.1160357
                           am        gear        carb
Mazda RX4           1.1899014  0.4235542  0.7352031
Mazda RX4 Wag       1.1899014  0.4235542  0.7352031
Datsun 710          1.1899014  0.4235542 -1.1221521
Hornet 4 Drive     -0.8141431 -0.9318192 -1.1221521
Hornet Sportabout  -0.8141431 -0.9318192 -0.5030337
Valiant            -0.8141431 -0.9318192 -1.1221521
```

5.2 Anonymní funkce

Velmi důležitou roli hraje anonymní funkce. Anonymní funkce je funkce, která nemá přiřazené jméno. Tato funkce se používá zejména v situacích, kdy je nutné použít funkci, ale nechceme ji ukládat. Tato situace nastává často v kombinaci s jinými funkcemi, jako jsou funkce apply anebo funkce pro optimalizace či řešení soustav rovnic.

```
> x <- matrix(c(1, 3, 1, 5), 2)
> print(x)
     [,1] [,2]
[1,]    1    1
[2,]    3    5
> apply(x, 2, FUN = function(x) max(x)-min(x))
[1] 2 4
```

Použití anonymní funkce má jeden vstup, a to parametr `x`. Za něj postupně funkce `apply()` dosazuje různé hodnoty – sloupce nebo řádky podle parametru `MARGIN` (druhý parametr funkce apply). Lze samozřejmě vytvořit i anonymní funkci s více parametry. V případě

použití ve funkci `apply()` je nutné dodatečně tyto parametry specifikovat za definicí anonymní funkce.

```
> x <- matrix(c(1, 3, 1, 5), 2)
> print(x)
 [1]  1  2  3  4  5  6  7  8  9 10
> apply(x,2,FUN=function(x,z) sum(x)*z,z=2)
[1]  8 12
```

Příklad 4

Pomocí anonymní funkce a funkce `apply` vypočtěte přes dataset `mtcars` variační koeficient.

Řešení 4

```
> apply(mtcars, MARGIN = 2, FUN = function(x) sd(x)/mean(x))
       mpg        cyl       disp         hp       drat         wt
          qsec
0.2999881 0.2886338 0.5371779 0.4674077 0.1486638 0.3041285
    0.1001159
        vs         am       gear       carb
1.1520369 1.2282853 0.2000825 0.5742933
```

5.3 Rekurzivní funkce

Specifickým typem funkce je tzv. rekurzivní funkce. Funkce je zvláštní tím, že volá sama sebe. Dobře si tuto funkci prohlédněte. Cílem této funkce je najít nejbližší celočíselně dělitelnou hodnotu. Tedy pokud chci najít nejbližší celočíselný dělitel čísla deset číslem tři, tak musím v první fázi zjistit, zda číslo deset je anebo není celočíselně dělitelné číslem tři beze zbytku. Pokud tomu tak není, hledám nižší číslo než číslo deset, které je celočíselně dělitelné třemi beze zbytku. V prvním kroku se tedy ověří, zda je číslo deset celočíselně dělitelné třemi beze zbytku. Jelikož je zbytek po dělení (`10 %% 3`) roven 1, tak funkce vstoupí do `else` části a sníží hodnotu `y` o jednu jednotku. Poté spustí znovu funkci nejbližšího dělitele s hodnotou `y` rovnou devíti. Hodnota devět má již zbytek po dělení roven nule a funkce navrátí hodnotu `y` jako výsledek (v našem případě devět).

```
> Nejblizsi_delitel <- function(y, x) {
+   if (y %% x == 0) {
+     return(y)
+   } else{
+     return(Nejblizsi_delitel(y - 1, x))
```

```
+   }
+ }
>
> Nejblizsi_delitel(10, 3)
[1] 9
> Nejblizsi_delitel(115, 3)
[1] 114
```

V případě realizace `else` části podmínky dojde ke znovu zavolání té samé funkce. Tomuto se říká rekurze. Velkým problémem rekurzivních funkcí je ale situace, kdy počet rekurzí je příliš velký, popř. je nekonečný. V tu chvíli musíme do rekurze vložit podmínku, která kontroluje počet rekurzí. Aby počítadlo počtu rekurzí „prošlo“ do další funkce, je nutné přidat parametr funkce. Ve výchozí deklaraci je počet rekurzí nulový (`pocitadlo = 0`). Při realizaci `else` části kódu se ale k hodnotě počítadla rekurzí přičítá hodnota jedna.

```
> Nejblizsi_delitel <- function(y, x, pocitadlo = 0) {
+   if (pocitadlo < 100) {
+     if (y %% x == 0) {
+       return(y)
+     } else{
+       return(Nejblizsi_delitel(y - 1, x, pocitadlo + 1))
+     }
+   } else{
+     return("zastaveno z titulu velkeho mnozstvi rekurzi")
+   }
+ }

> Nejblizsi_delitel(10, 3)
[1] 9
> Nejblizsi_delitel(10, 3.1)
[1] 0
```

Rekurze vede ke kódu, který je v některých případech velmi elegantní oproti kódu bez rekurze (je kratší a dobře čitelný). Nicméně když se ta samá situace vyřeší bez rekurze, je vždy trochu rychlejší (což samozřejmě nemusí být problém). Proto by se měla rekurze používat jen v případech, kdy je to opravdu vhodné.

Příklad 5

Přidejte do námi vytvořené funkce v příkladu č. 1 nový parametr jménem `d`. Tento parametr bude označovat řád diference. Vytvořte rekurzivní funkci, která spočítá diferenci daného řádu. Př. diference druhého řádu je pak diference z první diference. Aplikujte funkci při 1,2 a 3 diferenci na data $z = (1 : 10)^2$.

Řešení 5

```
> diference <- function(x, d = 1) {
+   # provedu diferenci
+   if (d > 0) {
+     x = (x[2:(length(x))] - x[1:(length(x) - 1)])
+   }
+   # pokudu je diference vysi nez jedna
+   # znovu diference funkce
+   if (d > 1) {
+     x = diference(x, d = d - 1)
+   }
+   return(x)
+ }

> z = (1:10) ^ 2
> diference(x = z, d = 0)
 [1]   1   4   9  16  25  36  49  64  81 100
> diference(x = z, d = 1)
[1]  3  5  7  9 11 13 15 17 19
> diference(x = z, d = 2)
[1] 2 2 2 2 2 2 2 2
> diference(x = z, d = 3)
[1] 0 0 0 0 0 0 0
```

Daná funkce provádí diferenci do té doby, než je hodnota parametru `d` rovna jedné – v tu chvíli provede poslední diferenci a navrátí výsledek. Funkce by šla zapsat ještě jednodušeji, a to pomocí jen jedné podmínky, jak ilustruje kód níže.

```
> diference <- function(x, d = 1) {
+   if (d > 1) {
+     x = (x[2:(length(x))] - x[1:(length(x) - 1)])
+     x = diference(x, d = d - 1)
+   } else if (d == 1) {
+     x = (x[2:(length(x))] - x[1:(length(x) - 1)])
+   }
+   return(x)
+ }
```

První postup řešení pomocí dvou `if` podmínek zde uvádíme z důvodu přehlednosti a lepší čitelnosti.

Kapitola 6

Základy tvorby grafů v jazyce R

6.1 Úvodem ke grafům v R

Jednou z hlavních předností jazyka **R** jsou grafické možnosti vizualizace dat. V **R** lze vytvářet velmi kvalitní grafické výstupy, a to jak po stránce technické, tak i vizuální. Grafy v **R** představují samy o sobě velkou oblast, kde o každé zde prezentované kapitole vycházejí samostatné publikace. Z tohoto důvodu se omezíme na základy daných technologií. Obě prezentované knihovny jsou samostatným nezávislým systémem pro tvorbu grafů, který stojí osamoceně od ostatních prezentovaných knihoven. To je důležité si uvědomit zejména proto, že postupy, které lze aplikovat, například na graf knihovny `ggplot2`, nelze aplikovat na grafy z knihovny `base graphics` či jiné. Velkou výhodou těchto prezentovaných knihoven je i to, že existuje mnoho balíčků, které je rozšiřují a přidávají nové možnosti zpracování grafů (více viz. Rahlf, 2019; Venables, Smith, R Core Team, 2020). Vždy ale mějme na paměti, že nelze použít balíček na rozšíření jedné knihovny pro úpravu grafu z knihovny jiné.

Nelze říci, že jedna z prezentovaných knihoven na vizualizaci dat je jedinou vhodnou na používání a ostatní není třeba znát. Každá z těchto knihoven má výhody a nevýhody při běžném použití. Bez ohledu na použitou knihovnu (`ggplot`, `base graphic`, `lattice`) je základ grafiky vytvořen pomocí grafického engine `grDevices`. Tato knihovna je součástí výchozí instalace **R** a při každém zapnutí je automaticky zavedená. Základní grafy jazyka **R** (`Base graphics`) jsou postavené na knihovně `graphics` (grafický systém). Druhým grafickým systémem je pak systém grid, na kterém je postavená grafická knihovna `ggplot2` (nebo například i knihovna `lattice`). Z těchto důvodů jsou tyto knihovny navzájem nekompatibilní a nelze jejich výstupy ani nijak spojovat.

Velkým rozdílem je chování těchto knihoven z pohledu přístupu k objektům a použití funkcí. `Base graphics` nevytváří prioritně objekt, který obsahuje graf. Naopak knihovna `ggplot2` vytváří objekt, do kterého lze graf uložit a dále s ním i pracovat. S knihovnou

`ggplot2` se pracuje více po stránce objektového programování než funkcionálního. Knihovna `ggplot2` navíc pracuje oproti ostatním knihovnám ve vrstvách, tedy každý bod na grafu je tvořen samostatnou grafickou vrstvou.

Shrneme-li systém tvorby grafů v jazyku **R**, tak rozlišujeme:

- grafický engine (`grDevices`),
- grafické systémy (`grid` a `graphics`),
- grafické balíčky (`base graphics`, `lattice`, `ggplot2`...),
- balíčky rozšiřující grafické balíčky,
- balíčky rozšiřující grafický engine.

Dalším důležitým pojmem je tzv. nízkoúrovňová a vysokoúrovňová funkce. Vysokoúrovňové funkce jsou odpovědné za tvorbu grafů (vytváří grafické okno a jádro samotného grafu), zatímco nízkoúrovňové funkce jsou takové funkce, které jsou odpovědné za editaci grafu či jakoukoli následnou práci s ním.

6.2 Base graphics

Základní knihovnou pro tvorbu grafů je `base graphics` (R Core Team 2018, 2020). Jedná se o nativní funkce jazyka **R**, které jsou součástí výchozí instalace. Tyto grafy poskytují základní grafické nástroje pro vizualizaci. Jejich velkou výhodou je jednoduchost na ovládání a rychlost tvorby grafického výstupu. Nevýhodou je naopak nemožnost komplexního zobrazení dat mimo standardní postupy, popřípadě je to velmi složité (v porovnání například s `ggplot2`). Velkou nevýhodou je již zpětná, nemožná editace už vizualizovaných prvků grafu. Jakékoliv ex-post úpravy grafu se dějí překrytím/přetištěním. Base graphics má pro každý typ grafu specifickou funkci. Avšak různých výstupů lze dosáhnout i pomocí hlavní funkce (`plot()`), která podle typu objektu změní druh výstupu. Základní sada grafů v base graphics je následující:

- `plot()` – vytváří základní grafy, jako je liniový graf, scatter plot či lze použít i např. pro zobrazení map,
- `barplot()` – sloupcový graf,
- `boxplot()` – krabičkový diagram/graf,
- `hist()` – histogram,
- `dotchart()` – bodový graf neboli cleveland plot,
- `pie()` – výsečový (koláčový) graf,

- `stripchart()` – bodový graf pro vizualizaci vektorů dat.

R umožňuje v základu i více druhů grafů, kterými se zde nebudeme zabývat, ale jedná se například o grafy pro vizualizaci dvou a více proměnných jako jsou hvězdicové/pavoučí diagramy (`stars()`), zobrazení obrázku (`image()`), zobrazení 3D prostoru (např. `contour()`, `persp()`) či mozaikové grafy (`mosaicplot()`) anebo grafy pro vizualizaci změn distribucí, četností a jiného určení (jako například `cdplot()`, `spineplot()`, `fourfoldplot()`).

Důležité je mít na paměti to, že mnohé funkce se chovají jinak při jiné třídě datového vstupu. Tuto problematiku si ukážeme následně u jednotlivých typů grafů.

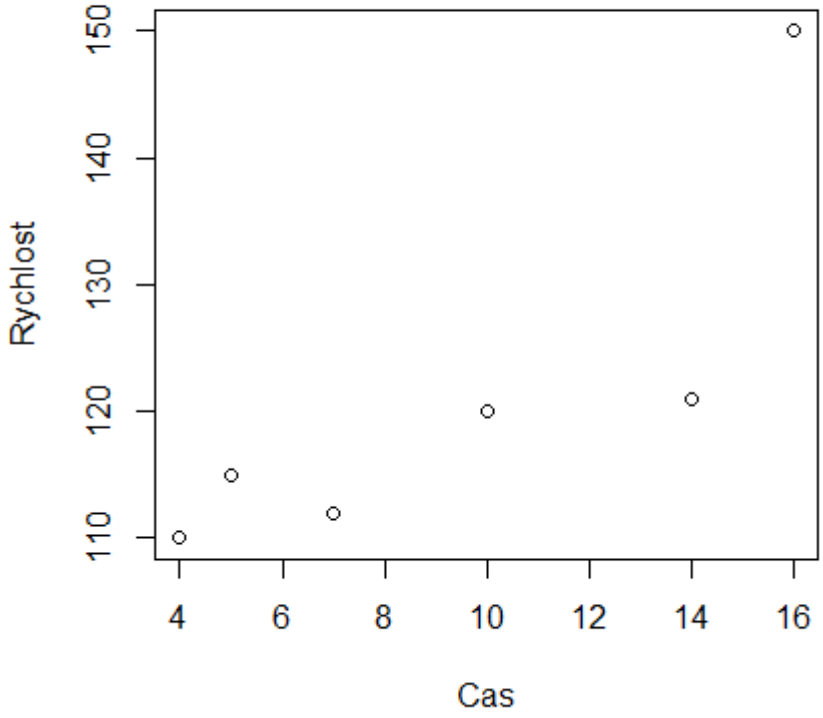
6.2.1 Funkce plot()

Vytvořme si prvně datový soubor, který chceme vizualizovat. Bude se jednat o data popisující rychlost a čas blíže neurčeného vozu.

```
1 > Auto <- data.frame(Cas = c(4, 5, 7, 10, 14, 16),
2 +                    Rychlost = c(110, 115, 112, 120, 121, 150))
```

Pokud do funkce `plot()` vložíme objekt typu `data frame` o dvou sloupcích, tak vykreslí bodový graf těchto proměnných. Na ose x bude uvedená první proměnná a na ose y bude uvedená druhá. **R** dále automaticky přebere názvy těchto proměnných a vloží je jako popisek jednotlivých os. Automaticky se též změní(„přeškálují“) osy grafu podle rozsahu daných proměnných. Graf lze vizualizovat pomocí příkazu `plot()`.

```
> plot(Auto)
```

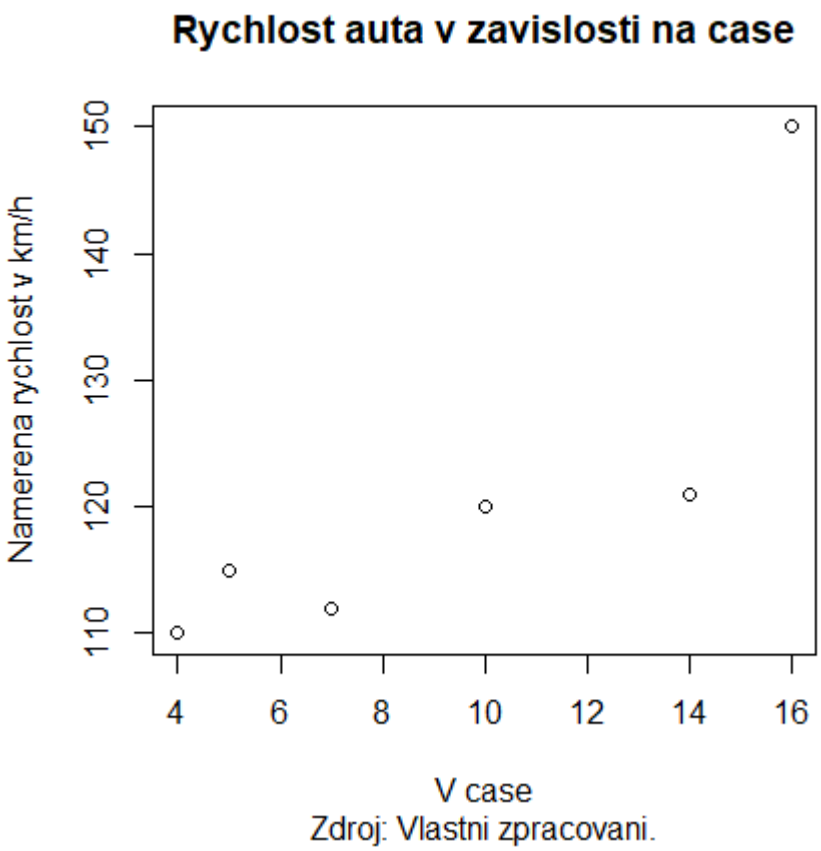


Obrázek 6.1: Funkce plot()

Zdroj: Vlastní zpracování.

Graf lze vizualizovat i pomocí příkazu `plot(x = Auto$Cas, y = Auto$Rychlost)`. Avšak nevýhodou tohoto zápisu je to, že místo rozumného názvu os se nám vloží text volání dané proměnné z data framu (tedy `Auto$Cas` a `Auto$Rychlost`). Zde je pak nutné všechny ostatní popisky vložit pomocí proměnných funkce plot.

```
> plot(Auto, main = "Rychlost auta v zavislosti na case",
+       sub = "Zdroj: Vlastni zpracovani.", xlab = "V case",
+       ylab = "Namerena rychlost v km/h")
```



Obrázek 6.2: Funkce plot() – úprava popisků

Na předcházejícím příkladu je pak vidět, že lze i oba přístupy kombinovat. Sice jsme předali data ve formátu data frame, ale řekli jsme jazyku R, jaké hodnoty má vykreslit do grafu. Parametr `main` určuje hlavní nadpis grafu, `sub` pak podnadpis. Popisky os určují parametry `xlab` a `ylab`. U většiny grafů mají jednotlivé parametry analogické použití. Až na situace, jako jsou parametry pro editaci atributů grafu, který daný graf nemá – například osy u koláčového grafu aj. To, jaké máme základní atributy, si ukážeme hned v následující podkapitole.

Formule

Většina grafů podporuje krom standardního zápisu, kdy přiřazujeme parametrům funkce odpovídající proměnné i zápis pomocí vzorce. Vzorec je zde chápán jako typ proměnné – formule. Ve formuli obvykle používáme základní aritmetická znaménka a místo rovná se symbol vlnovky ~. Toto znaménko má využití zejména v regresi, kdy potřebujeme

zapsat rovnici, kterou odhadujeme. Kdybychom použili znaménko rovná se (=), nedošlo by k vytvoření objektu formule, ale ke snaze o přiřazení uprostřed funkce. V případě grafů slouží znaménko tilda (vlnovka) k rozlišení toho, co má být vůči čemu vykreslené. V některých aplikacích se můžeme setkat i s tečkou, tečka zastupuje všechny ostatní proměnné datasetu. Máme tedy tři způsoby, jak vytvořit identický graf, který se ale bude lišit ve výchozích popiscích os.

```
> plot(Auto)
> plot(x = Auto$Cas, y = Auto$Rychlost)
> plot(Rychlost ~ Cas, data = Auto)
```

6.2.2 Základní editace grafu a barvy

Parametry ovlivňující tvorbu grafů můžeme rozdělit do dvou skupin. Ty, které ovlivňují jak vysokoúrovňové funkce, tak i nízkoúrovňové, a ty, které lze použít pouze v nízkoúrovňových funkcích (pomocí editace funkcí par). Následující tabulka shrnuje základní vysokoúrovňové grafické parametry funkcí jazyka **R**.

Parametr	Popis
`x, y`	parametr pro předání hodnot jednotlivých os
`type`	druh grafu
`cex`	velikost písma (v násobkách)
`main, sub`	hlavní nadpis a podnadpis
`xlab, ylab`	popisky os
`fg`	barva osy grafu
`lwd`	tloušťka linie
`lty`	druh linie
`col`	barva linie či jiných vizualizovaných prvků v oblasti grafu
`pch`	druh symbolu, kterým se mají data vizualizovat
`family, font`	font a typ písma
`xlim, ylim`	rozsah os grafu
`las`	natočení/rotace popisků os

Tabulka 6.1: Parametry funkcí

Zdroj: Vlastní zpracování.

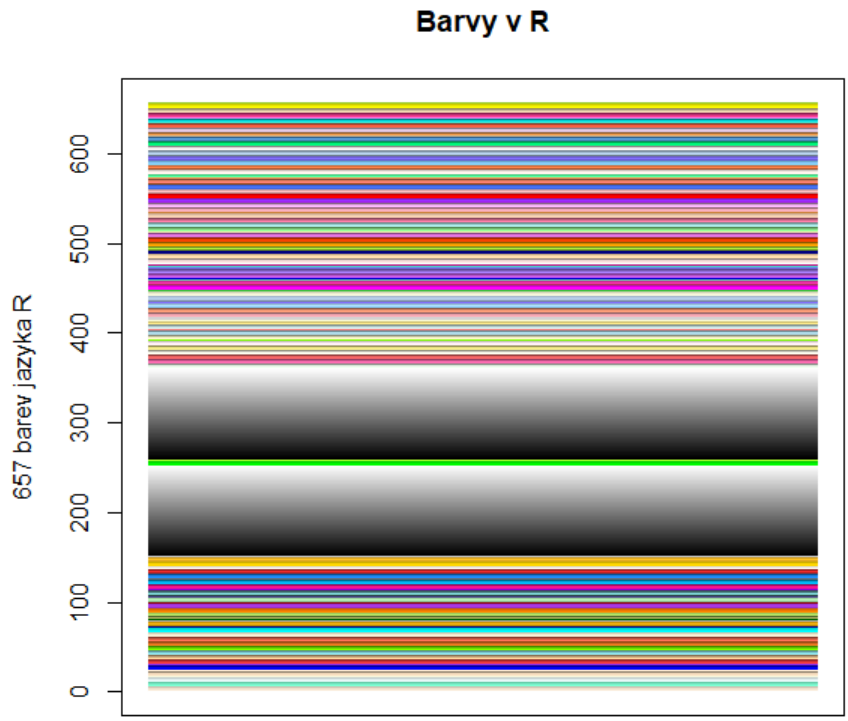
Parametr `cex` ovlivňuje relativní velikost objektů ve vztahu k velikosti řádky. Barvu os můžeme ve funkci plot změnit pomocí `fg` parametru. Zde je ale důležité poznamenat, že **R** dokáže přebrat různé definice barev. Základní barvy lze získat pomocí řadových číslovek

0–8. Dále lze využít i anglická jména barev. Seznam základních jmen 657 barev lze získat pomocí příkazu `colors()`.

```
> colors()
  [1] "white"           "aliceblue"       "antiquewhite"
  [4] "antiquewhite1"   "antiquewhite2"   "antiquewhite3"
  [7] "antiquewhite4"   "aquamarine"      "aquamarine1"
[10] "aquamarine2"     "aquamarine3"     "aquamarine4"
[13] "azure"           "azure1"          "azure2"
[16] "azure3"          "azure4"          "beige"
[19] "bisque"          "bisque1"         "bisque2"
[22] "bisque3"         "bisque4"         "black"
[25] "blanchedalmond" "blue"            "blue1"
[28] "blue2"           "blue3"           "blue4"
...
...
```

Jak je vidět z kódu výše, R má pro každý typ základních 657 barev své jméno. Obvykle jsou dané barvy seskupené podle odstínu a valéru. Pomocí následujícího kódu si můžeme dané barvy i zobrazit. To, jak následující kód funguje a co jaké parametry způsobují, si představíme postupně v rámci této kapitoly.

```
> plot(
+   y = c(1:657),
+   x = c(1:657),
+   main = "Barvy v R",
+   type = "n",
+   xlab = "",
+   ylab = "657 barev jazyka R",
+   sub = "Zdroj: vlastni prace autoru",
+   xaxt = 'n'
+ )
> for (i in 1:657) {
+    lines(
+     x = c(1:657),
+     y = c(rep(i, 657)),
+     col = colors()[i],
+     lwd = 0.01
+   )
+ }
```



Zdroj: vlastni zpracovani.

Obrázek 6.3: Základních 657 barev jazyka R

Jazyk **R** ale podporuje i rgb, hsv a hlc reprezentaci barev. Tyto reprezentace lze použít pomocí analogických funkcí: `rgb()`, `hsv()` a `hlc()`.

To, jakou grafickou formou se budou data vizualizovat, ovlivňuje parametr `type`. Zde máme na výběr mezi více druhy reprezentace dat:

p	body
l	linie (čára)
b	kombinace bodů a linie
c	jako b, ale s vynecháním bodů
o	vykreslí body a pak přes ně vykreslí linii
h	jako histogram
s	schody
n	nic

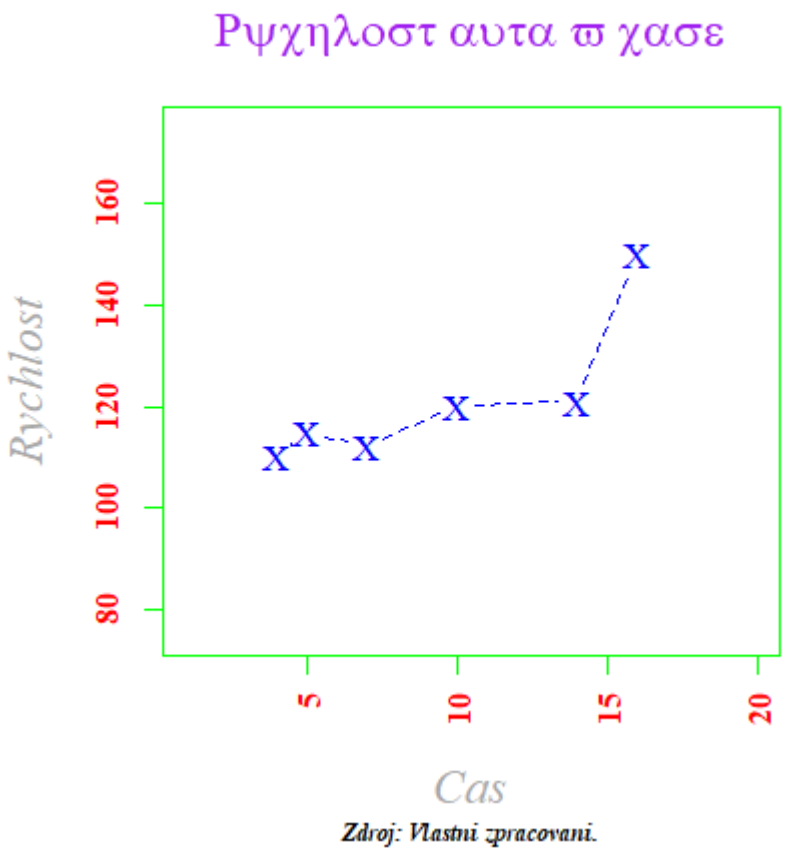
Tabulka 6.2: Druhy čar

Zdroj: Vlastní zpracování.

Některé z těchto parametrů (`cex`, `col`,`font`) mají případné varianty pro editaci specifických parametrů (`main`, `sub`, `axis`, `lab`) a editují se pomocí spojení těchto dvou názvů s tečkou, například `cex.main` bude ovlivňovat pouze relativní velikost nadpisu, `col.lab` jen barvu popisků os. Fonty, které si můžeme vybrat pomocí parametru family, jsou: „sans“, „serif“, „mono“ a „symbol“. Tuto sadu fontů lze samozřejmě rozšířit pomocí dodatečných balíčků. Parametr font pak specifikuje druh písma: 1 – plain, 2 – bold, 3 – italic, 4 – bold a italic, 5 – symbol. Parametry `xlim` a `ylim` se zadávají jako atomický numerický vektor o délce dva, kde první hodnota označuje minimum a druhá hodnota maximum osy grafu. V následujícím obrázku zkusíme použít větší množství těchto parametrů.

```
> plot(
+   Auto,
+   type = "b",
+   lty = "dashed",
+   pch = "X",
+   cex = 1.2,
+   cex.main = 1.5,
+   cex.sub = 0.75,
+   cex.lab = 1.5,
+   family = "serif",
+   font.main = 5,
+   font.sub = 4,
+   font.lab = 3,
+   font.axis = 2,
+   col = "blue",
+   col.axis = "red",
+   col.lab = "darkgrey",
+   col.main = "purple",
+   fg = "green",
+   main = "Rychlost auta v case",
```

```
+    sub = "Zdroj: Vlastni zpracovani",
+    xlim = c(1, 20),
+    ylim = c(75, 175),
+    las = 3
+ )
```

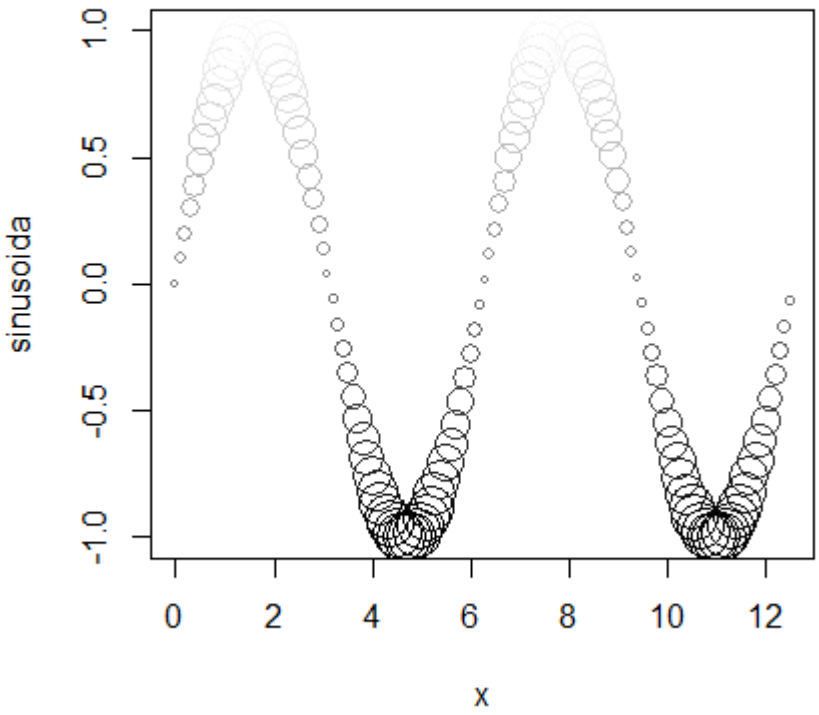


Obrázek 6.4: Funkce plot() – úprava parametrů

R obsahuje celou škálu palet barev, které lze využít při tvorbě grafů. Jedná se například o funkce `rainbow()`, `hlc.colors()`, `heat.colors()`, `terrain.colors()` či `topo.colors()`. Tyto funkce přebírají jako hlavní parametr počet barev, které se z dané palety mají vygenerovat (v rgb kódu). Alternativou je pak například funkce `grey()`, která pro hodnoty mezi 0 a 1 přiřadí odpovídající barvu na škále šedé. Použití funkce `grey()` je výrazně jednodušší, například v následujícím kódu si vytvoříme sekvenci hodnot od 0 do čtyřnásobku pí s krokem 0,1 (obrázek 6.4). Vypočítáme si hodnoty sinusoidy pomocí funkce sin a vykreslíme graf. V tomto grafu budeme chtít, aby: 1) hodnoty byly na škále šedi (tmavší hodnoty pod nulou a světlejší nad) a 2) aby hodnoty, které jsou vzdálenější od 0, měly

větší symbol. Toho lze snadno dosáhnout tak, že do parametru `col` přiřadíme ne jednu hodnotu, ale počet hodnot odpovídajících jednotlivým pozorováním. Pak se každá daná hodnota vykreslí požadovaným atributem. Funkce grey pro hodnoty blízké 0 vykresluje černou barvu a pro hodnoty blízké jedničce bílou. Je tedy nutné vypočítané hodnoty sinusoidy vhodně přeškálovat. Toto uděláme tak, že k její hodnotě přičteme jedna a vydělíme dvěma. Tímto dosáhneme požadované škály šedi. V případě velikosti nám pak stačí zvolit výchozí (nejmenší) velikost bodu (zde jsme zvolili 0,5) a pak zvolit vhodné škálování. My jsme absolutní hodnotu sinusoidy násobili třemi, aby byla změna velikosti velmi patrná.

```
> x <- seq(from = 0, to = pi * 4, by = 0.1)
>
> sinusoida <- sin(x)
>
> plot(sinusoida ~ x,
+      col = grey(c(sinusoida + 1) / 2),
+      cex = 0.5 + abs(sinusoida) * 3)
```



Obrázek 6.5: Graf sinusoidy

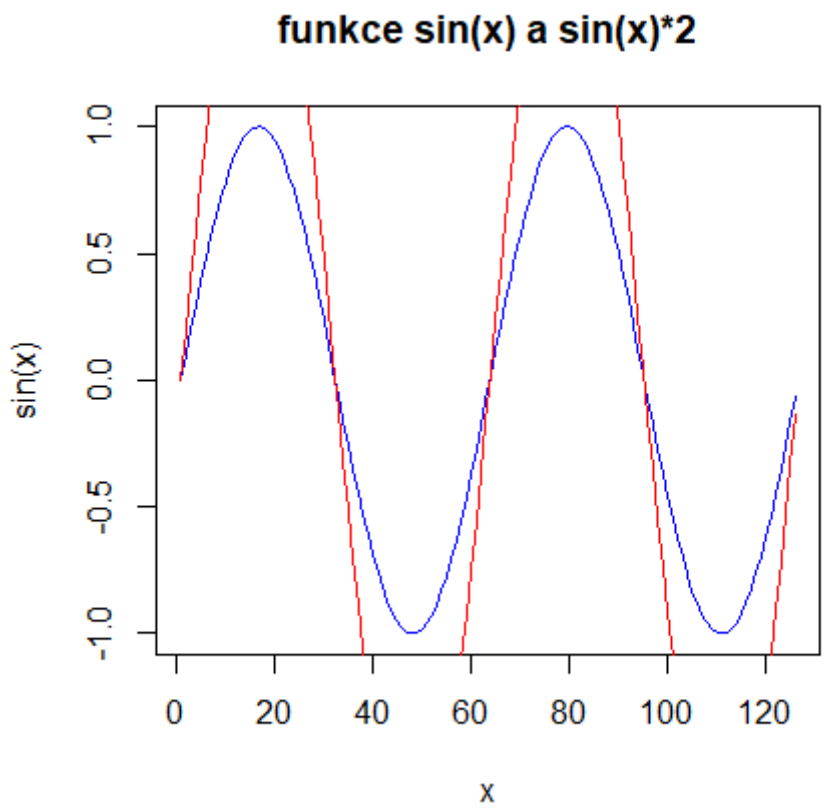
Zdroj: Vlastní zpracování.

6.2.3 Základní typy grafů

Funkce plot() – více proměnných

Než si představíme všechny ostatní typy grafů, tak se ještě vrátíme k základní funkci `plot()`. Ukázali jsme si, jak vykreslit jednoduchý graf, ale co když chceme vykreslit více proměnných v rámci jednoho grafu? Máme dvě možnosti. Buď použijeme funkci `points()`, anebo `lines()`. Obě tyto funkce použijeme až poté, co vykreslíme samotný graf. A obě fungují velmi podobně jako funkce `plot()`. Avšak s tím, že neinicializují, oproti funkci `plot()`, novou grafickou oblast, ale překreslují stávající. S tímto chováním se ale pojí jeden zásadní problém. Pokud jsme již něco „vytiskli“ v rámci funkce `plot()`, pak to již nejsme schopni opravit funkcí `lines` či `points`. Často se pak uživatelům stává, že vykreslí jednu proměnnou, a když chtějí vykreslit druhou, tak zjistí, že se nevejde do grafické oblasti grafu a jaksi „mizí“ mimo rámec grafu (nebo se rovnou vůbec nezobrazí, protože se tyto dvě proměnné ani úrovňově neprotínají). Tomuto lze předejít vhodným nastavením parametrů `xlim` a `ylim` v rámci funkce `plot()`. Tedy když plánujeme použít funkci `lines` či `points`, je nutné již s tímto počítat při funkci plot. Ukažme si toto na funkci sin(x), 2*sin(x). Následující příklad ukazuje, co se stane, pokud nenastavíme vhodně proměnné `xlim` a `ylim`.

```
> # Tvorba promene
> x = seq(from = 0, to = pi * 4, by = 0.1)
> # Vypocet sinusoidy
> sinusoida = sin(x)
> # Graficka funkce plot
> plot(sinusoida,
+      type = "l",
+      col = "blue",
+      xlab = "x",
+      main = "funkce sin(x) a sin(x)*2"
+ )
> # Pomoci funkce lines
> # pridame dalsi caru do grafu
> lines(sin(x) * 2,
+       type = "l",
+       col = "red")
```

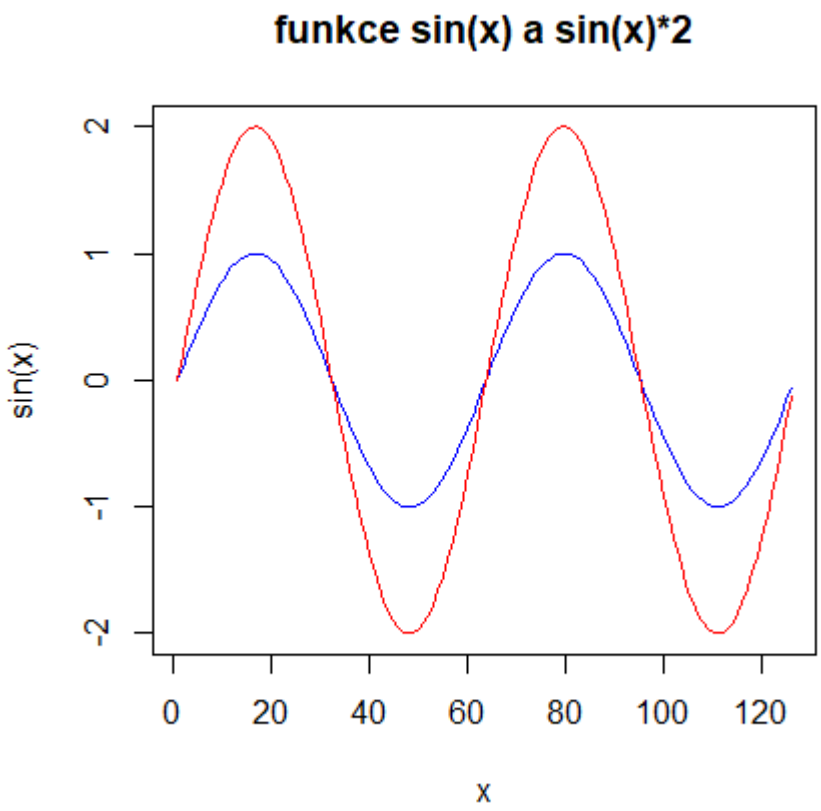


Obrázek 6.6: Funkce plot() a více proměnných

Zdroj: Vlastní zpracování.

Poměrně snadno lze toto vyřešit pomocí parametru `ylim`. Najdeme minimální a maximální hodnotu v obou proměnných a tu předáme do požadovaného parametru.

```
> x = seq(from = 0, to = pi * 4, by = 0.1)
>
> plot(sin(x), type = "l", col = "blue", xlab = "x",
+      main = "funkce sin(x) a sin(x) * 2",
+      ylim = c(min(c(sin(x), sin(x) * 2)),
+               max(c(sin(x), sin(x) * 2))))
>
> lines(sin(x) * 2, type = "l", col = "red")
```



Obrázek 6.7: Funkce plot() – ylim a xlim parametry

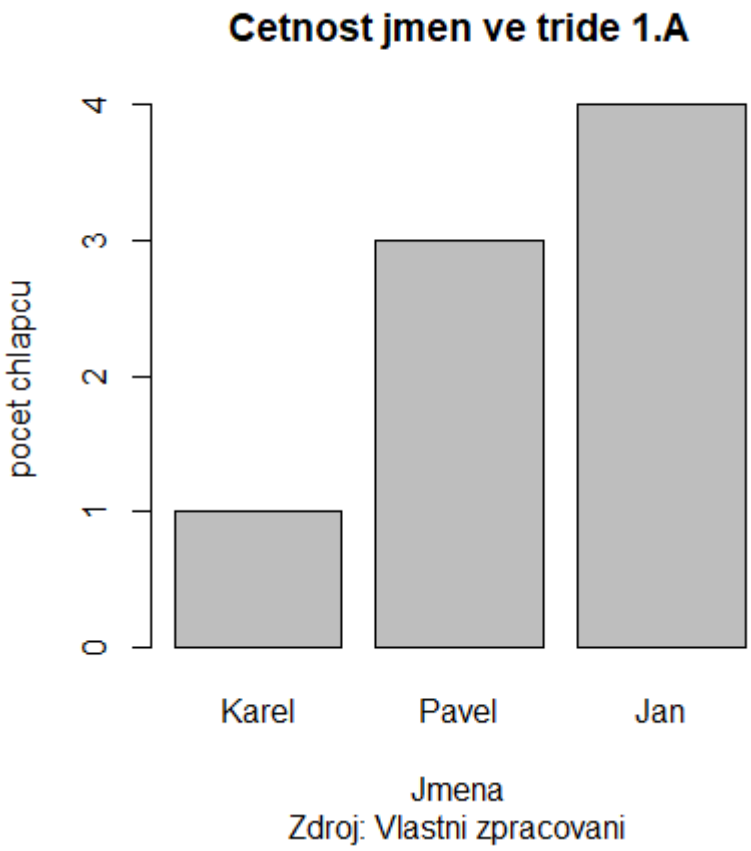
Zdroj: Vlastní zpracování.

Funkce barplot()

Sloupcový graf neboli anglicky „barplot“ je typ grafu, kde jsou data rozdělená do určitých skupin (obvykle na ose x) a vizualizována absolutní (či relativní) četností (nebo součtem) v dané skupině, kterou lze vyčíst na ose y. Oproti příkazu plot nepřebírá tato funkce parametr `x` či `y`, ale parametr `height`, kterému mají odpovídat výšky daných sloupců. Výšku daného sloupce (četnost či jiný údaj) si musíme spočítat sami. Funkce barplot dokáže přebrat mnoho typů objektů tak jako funkce plot. Standardní postup je vložit do této funkce pojmenovaný atomický numerický vektor.

```
1 > n <- c(1, 3, 4)
2 > names(n) <- c("Karel", "Pavel", "Jan")
3 > barplot(n, main = "Cetnost jmen ve tride 1.A",
4 +         sub = "Zdroj: Vlastni spracovani",
```

```
+          ylab = "pocet chlapcu", xlab = "Jmena")
```

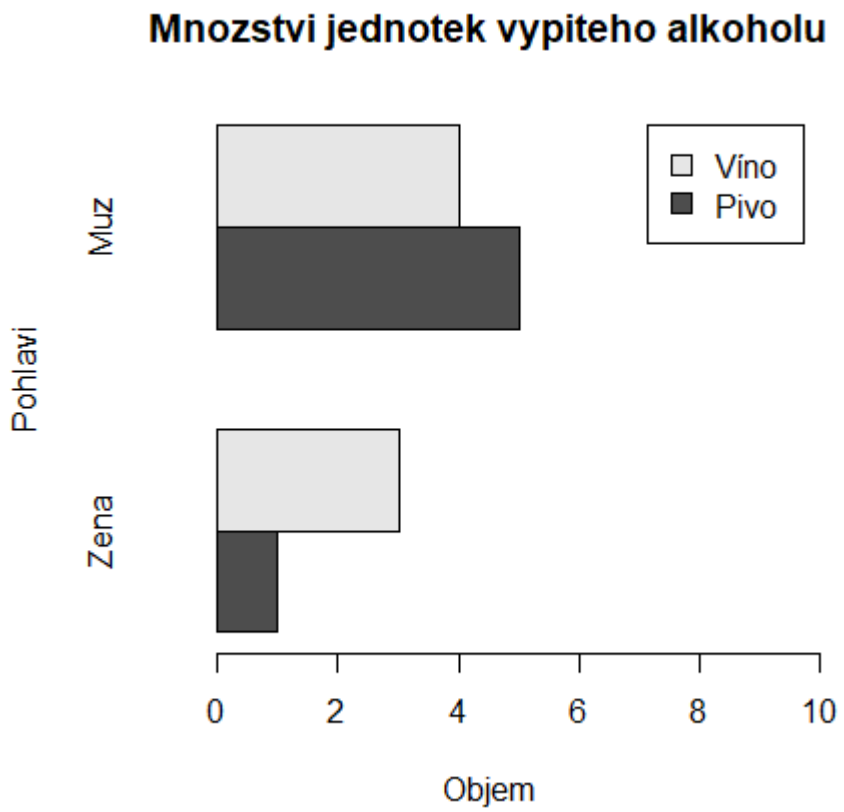


Obrázek 6.8: Funkce barplot()

Jak může pozorný čtenář vidět, funkce barplot má mnoho společných parametrů, které lze použít k editaci grafu. Tato funkce má navíc mnoho dalších vlastních parametrů – např. `space` ovlivňuje množství místa mezi sloupci, `width` ovlivňuje šířku sloupce (musí být použit ale zároveň i parametr `xlim`) do parametru `names.arg` (funguje i parametr `namesn`, ale ostatní nastavení nemusí fungovat zcela správně) můžeme specifikovat alternativní jména sloupců, `beside` v případě zobrazení dat více kategorií pro každou skupinu (z matice) umožní volit typ vizualizace dat podkategorií (vedle sebe/na sobě). Parametr `col` přebírá barvu, popř. barvy sloupců.

```
> Alkohol <- matrix(c(1, 3, 5, 4), nrow = 2,
+                   dimnames = list(c("Pivo", "Vino"),
+                                   c("Zena", "Muz")))
> barplot(Alkohol, beside = TRUE, horiz = TRUE, xlim = c(-1, 10),
```

```
5 +          main = "Mnozstvi jednotek vypiteho alkoholu",
6 +          ylab = "Pohlavi", xlab = "Objem",
7 +          legend.text = c("Pivo", "Vino"))
```



Obrázek 6.9: Úprava parametrů funkce barplot()

Zdroj: Vlastní zpracování.

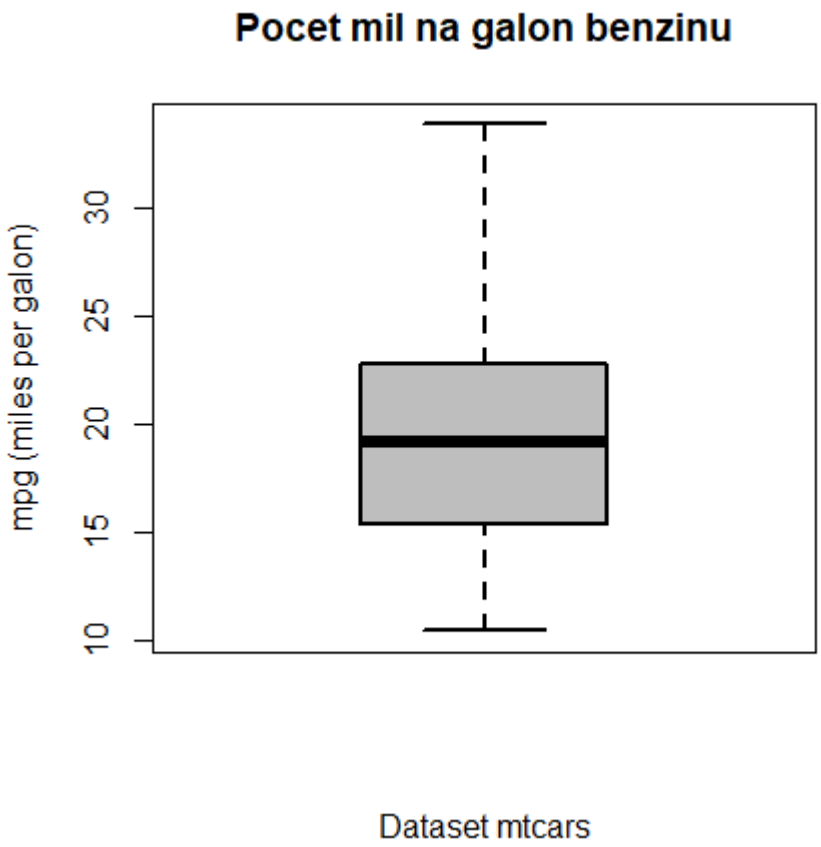
V případě, kdy chceme vytvořit horizontální orientaci grafu, pak parametru `horiz` přiřadíme hodnotu `TRUE` a graf se nám otočí.

Funkce boxplot()

Krabičkový diagram vytvoříme snadno pomocí funkce `boxplot()`. Funkce `boxplot()` umožňuje i vytváření více krabičkových diagramů. Toho lze dosáhnout buď předáním objektu typu `data.frame`, anebo vzorcem, který nám řekne, jak má data dělit. Funkce

`boxplot()` přebírá primárně data ve formátu `data.frame`, anebo jakýkoliv vektor (atomický vektor nebo list). Ukážeme si použití tohoto grafu na datovém souboru `mtcars`. Nejprve si zkusíme vykreslit krabičkový diagram počtu mil, které ujedou jednotlivá auta na galon benzínu (proměnná `mpg`).

```
> boxplot(
+   x = mtcars$mpg,
+   main = "Pocet mil na galon benzinu",
+   sub = "Dataset mtcars",
+   ylab = "mpg (miles per galon)",
+   lwd = 2,
+   col = "grey"
+ )
```

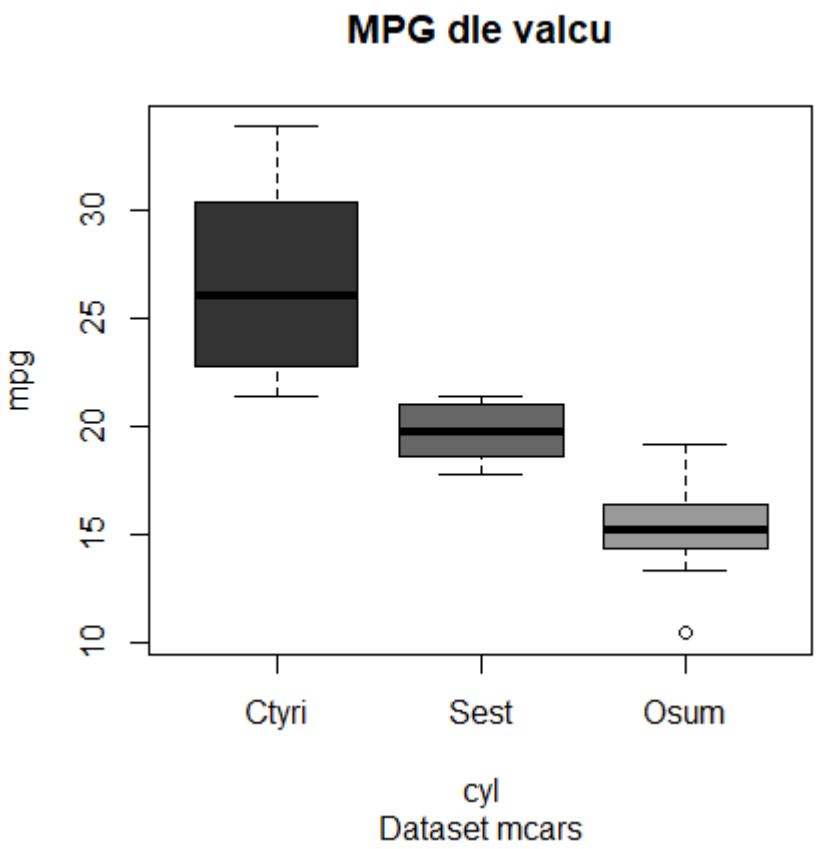


Obrázek 6.10: Funkce boxplot()

Zdroj: Vlastní zpracování.

Funkce `boxplot()` umí efektivně vykreslovat i více krabičkových diagramů v jedné grafické oblasti. Řekněme, že si budeme chtít vykreslit krabičkové diagramy toho, kolik mil ujede auto na jeden galon benzínu (`mpg`) pro jednotlivé varianty počtů válců auta (`cyl`). Máme dvě možnosti, jak takovýto graf vytvořit. Buď vyrobíme nový data frame objekt, který bude obsahovat data o tom, kolik ujedou daná auta při jednotlivých počtech válců, nebo využijeme nám již známé formule.

```
> boxplot(
+   mpg ~ cyl,
+   data = mtcars,
+   col = grey(c(0.2, 0.4, 0.6)),
+   names = c("Ctyri", "Sest", "Osum"),
+   lwd = 1.5,
+   main = "MPG dle valcu",
+   sub = "Dataset mcars"
+ )
```



Obrázek 6.11: Funkce boxplot() – úprava parametrů funkce

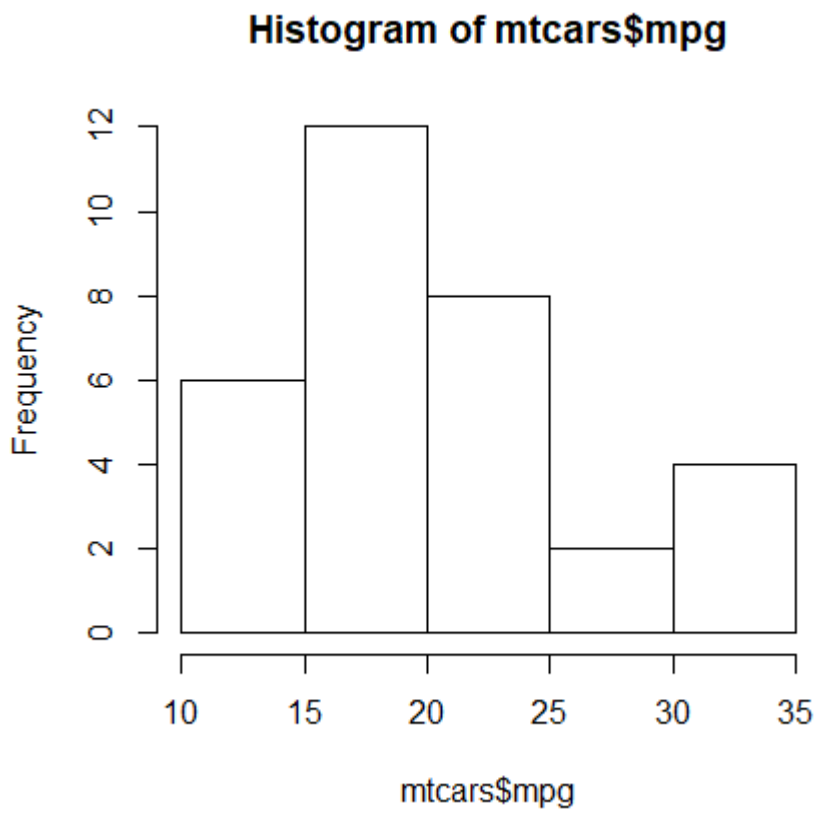
Zdroj: Vlastní zpracování.

Jak vidíme, i v případě boxplotu lze nastavit individuální parametry pro dílčí krabičkové diagramy. Zde jsme pomocí vektoru na škále šedi nastavili každému grafu jinou barvu. Jména na ose x se řídí parametrem `names` (ne `names.arg` jako v předchozím případě).

Funkce hist()

V **R** můžeme histogram vytvořit velmi snadno a rychle pouze jedním příkazem – `hist()`. **R** se za vás bude snažit spoustu věcí vyřešit samo - jako například zvolit počet sloupců histogramu. Pojďme si nejprve ukázat, jak vytvořit snadno histogram, a pak si jej zkusíme upravit.

```
1 hist(mtcars$mpg)
```

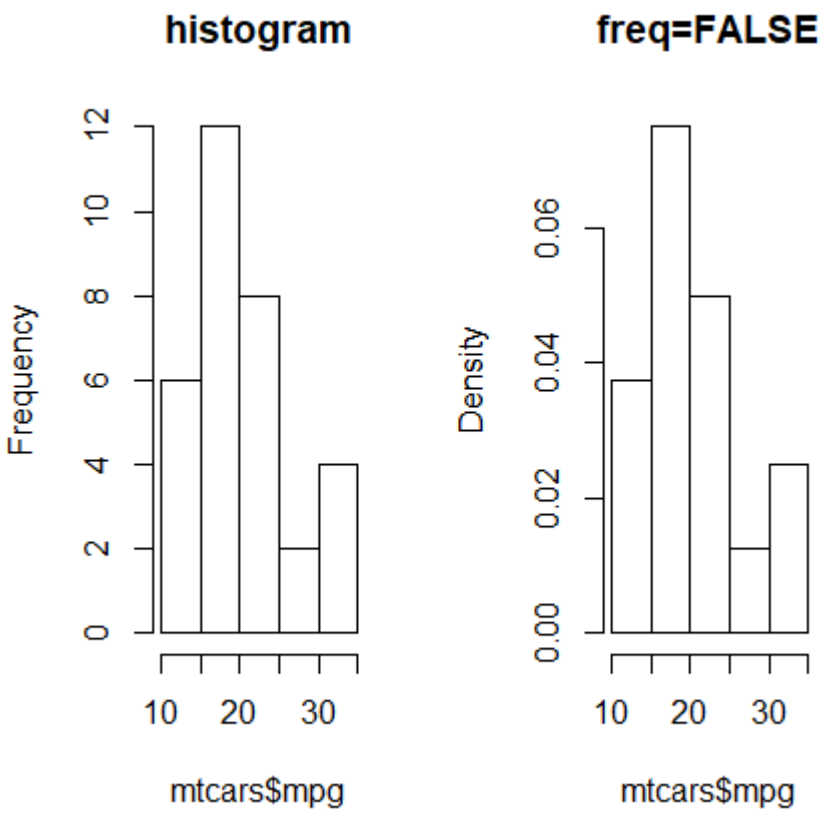


Obrázek 6.12: Funkce hist()

Zdroj: Vlastní zpracování.

Z příkladu vidíme, že **R** za nás volí automaticky nadpis histogramu. Tak jako v případě funkce plot ale můžeme nadpis a popisky os editovat pomocí parametrů `main`, `xlab` a `ylab`. Histogram, který **R** ve výchozím nastavení vykreslí, obsahuje na ose y absolutní četnosti. Pokud bychom chtěli vytvořit graf hustoty, stačí vložit do parametru `freq` hodnotu `FALSE`. Na následujícím obrázku si porovnáme oba grafy.

```
> par(mfrow = c(1, 2))
> hist(mtcars$mpg, main = "histogram")
> hist(mtcars$mpg, freq = FALSE, main = "freq=FALSE")
```



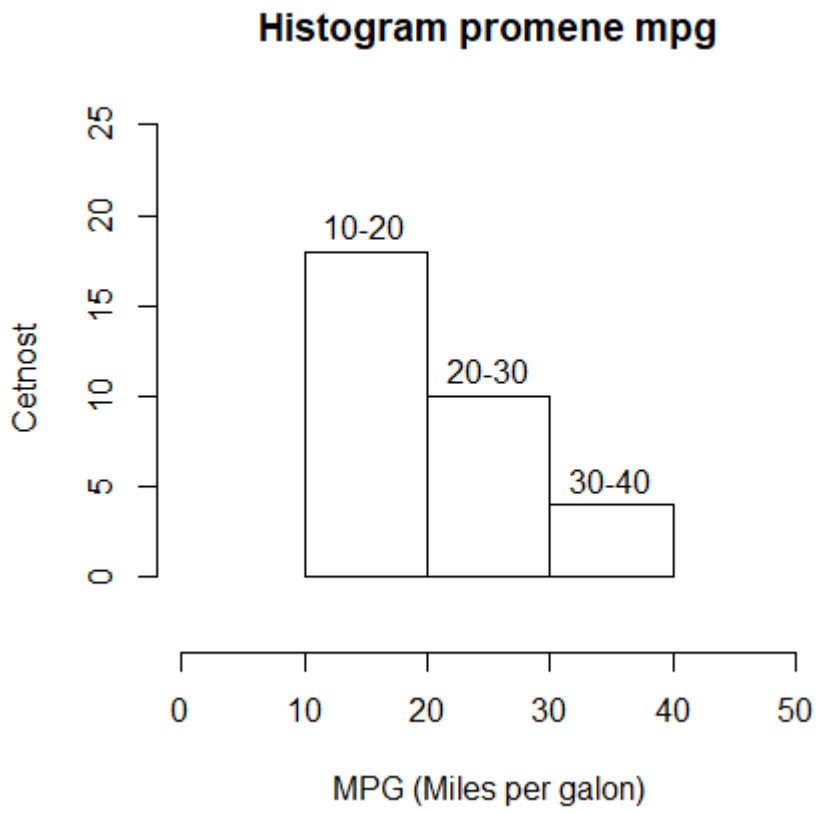
Obrázek 6.13: Funkce hist() a hist(,freq=FALSE)

Zdroj: Vlastní zpracování.

Pozorný čtenář si jistě všimne, že oba grafy jsou „tvarem“ sloupců identické. Rozdíl mezi „density“ histogramem a „frequency“ histogramem spočívá v ose y – tedy v tom, v jakých jednotkách je výsledný graf reprezentován. Počet sloupců můžeme ovlivnit pomocí parametru `breaks`. Tento parametr je schopen ovlivnit širokou škálu vstupů. Pokud mu předáme jedno číslo, tak ho použije na počet intervalů, které má vytvořit, pokud mu předáme vektor několika čísel, pak tyto hodnoty bude považovat za hraniční hodnoty jednotlivých intervalů sloupců histogramu (podobně jako u funkce `cut()`). Dále je schopen tento parametr přebrat i funkci, která mu řekne, jak má vytvořit intervaly pro sloupce histogramu. V neposlední řadě přebírá jméno matematické metody, jež se má použít na výpočet počtu sloupců a histogramu. Tento parametr je při tvorbě histogramů klíčový a ovlivní výslednou vypovídající hodnotu dat. Dále je důležitý v situaci, kdy chceme mezi sebou porovnat jednotlivé histogramy. Pokud bychom měli data do sloupců rozdělena jiným způsobem, bylo by porovnání velmi obtížné. Dalším důležitým parametrem je `freq`.

Tento parametr definuje, zda se mají data vizualizovat absolutní četností. Parametry `xlim`, `ylim` fungují zcela identicky jako v případě funkce `plot()`.

```
> hist(
+   mtcars$mpg,
+   breaks = c(10, 20, 30, 40),
+   labels = c("10-20", "20-30", "30-40"),
+   main = "Histogram promenne mpg",
+   ylim = c(-3, 25),
+   xlim = c(0, 50),
+   xlab = "MPG (Miles per galon)",
+   ylab = "Cetnost"
+ )
```



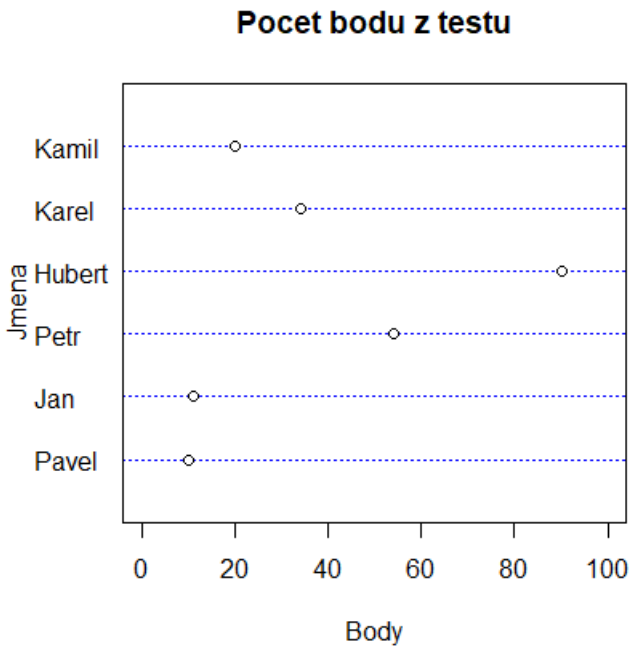
Obrázek 6.14: Funkce hist() – úprava parametrů funkce

Zdroj: Vlastní zpracování.

Funkce dotchart()

Bodový diagram (Cleveland bodový graf) vytvoříme pomocí funkce `dotchart()`. Tato funkce přebírá data, která vizualizuje do parametru `x`. Funkce umí zpracovat jak vektor, tak matice. Pokud funkce přebere jako hlavní parametr vektor, tak vizualizuje jeden bodový diagram. V případě, že přebere matici, tak **R** vytvoří více samostatných bodových diagramů v rámci jednoho grafu – rozdělený do skupin podle sloupců matice. Následující příklad znázorňuje počet bodů, které získali jednotliví studenti v rámci školního turnaje.

```
> body <- c(10, 11, 54, 90, 34, 20)
> names(body) <- c("Pavel", "Jan", "Petr", "Hubert", "Karel", "Kamil
    ")
>
> dotchart(body, main = "Pocet bodu z testu",
+          xlim = c(0, 100), lcolor = "blue",
+          xlab = "Body",  ylab = "Jmena"
+ )
```



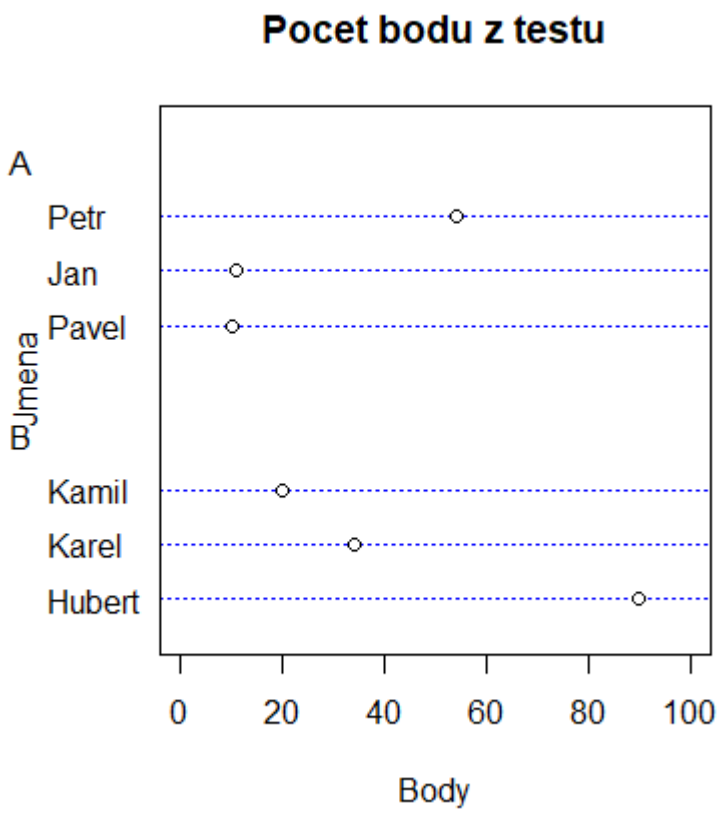
Obrázek 6.15: Funkce dotchart()

Zdroj: Vlastní zpracování.

Pokud bychom chtěli daná data rozdělit do skupin dle jejich třídy, tak bychom si museli vytvořit nový vektor (nejlépe proměnné typu faktor), který nám pomůže identifikovat, kdo

do jaké třídy patří. Výhodou tohoto postupu je, že nám zabezpečí snadnou srovnatelnost obou grafů.

```
> body <- c(10, 11, 54, 90, 34, 20)
> names(body) <- c("Pavel", "Jan", "Petr", "Hubert", "Karel", "Kamil
    ")
> tridy <- as.factor(c("A", "A", "A", "B", "B", "B"))
> dotchart(
+   body,
+   main = "Pocet bodu z testu",
+   xlim = c(0, 100),
+   lcolor = "blue",
+   xlab = "Body",
+   ylab = "Jmena",
+   groups = tridy
+ )
```



Obrázek 6.16: Funkce dotchart() – úprava parametrů funkce

Zdroj: Vlastní zpracování.

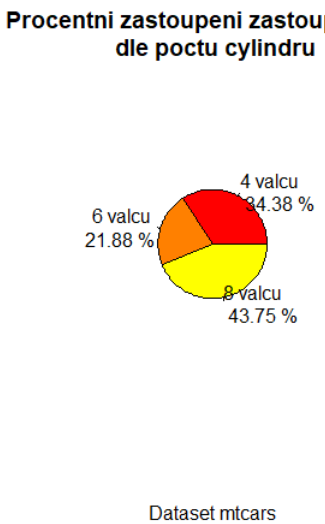
V případě, kdy by proměnná třídy (`tridy`) nebyla faktorem (a byla numerickou proměnnou!), budou data sice opticky rozdělena do skupin, ale bez popisku těchto skupin. V tomto případě je popiskem pouze A a B, ale lze vytvořit faktorovou proměnnou podle své potřeby.

Funkce pie()

Výsečový graf znázorňuje data poměrným zastoupením výseče kruhu na celém jeho obsahu. Jedná se o velmi populární graf, avšak musíme si uvědomit, že tento typ grafu může výsledky opticky zkreslovat – zejména, když nejsou k dispozici popisky výsečí s jejich hodnotou. Proto si vytvoříme graf, který bude obsahovat jak popisky, tak i procentní zastoupení jednotlivých výsečí na celku. V následujícím příkladu si budeme vizualizovat zastoupení aut v data setu dle počtu válců motoru auta. Pro rychlé shrnutí nám postačí

použít funkci `table`.

```
> p.cyl <- table(mtcars$cyl) / length(mtcars$cyl) * 100
> pie(
+     x = p.cyl,
+     main = "Procentni zastoupeni zastoupeni aut \n dle poctu cylindru",
+     sub = "Dataset mtcars",
+     labels = paste(names(p.cyl), "valcu \n", round(p.cyl, 2), "%"),
+     col = heat.colors(3)
+   )
```



Obrázek 6.17: Funkce pie()

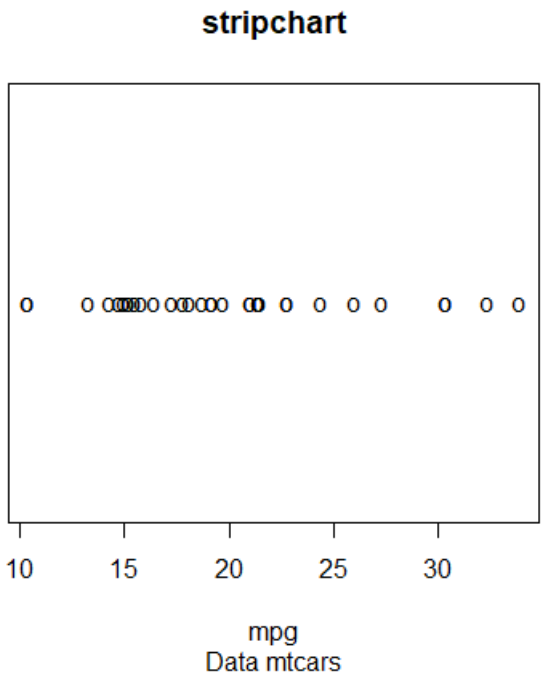
Zdroj: Vlastní zpracování.

Pozorný čtenář si jistě všimne, že zde pro vykreslení barev není již použita funkce `grey()`, ale `heat.colors()`. Funkce `heat.colors()` navrátí přechod „ohnivých“ barev při zadaném počtu barev. Barvy zde neslouží k identifikaci toho, zda má jedna výseč větší hodnotu, ale k jejich odlišení. Stejně fungují funkce `grey.colors()`, `topo.colors()`, `terrain.colors()`, `cm.colors()` nebo `rainbow()`. Další sady barev můžeme získat například prostřednictvím knihovny `ColorBrewer`. Tato knihovna disponuje širokou škálou palet pro různá použití. Například zde najdeme jak gradientní přechody barev, tak ale i divergenční škály. Divergenční škály jsou takové, kdy máme plynulý přechod z jedné barvy do druhé. Například z modré do červené.

Funkce stripchart()

Funkce `stripchart()` vytváří v základu pro uživatele možná na první pohled velmi zvláštní graf, ale pokud si dáme práci a nastavíme jednotlivé parametry této funkce správně, můžeme získat velmi efektní graf s velkou informační hodnotou. Stripchart je vlastně 1-dimenzionální verze scatter plotu – verze bodového grafu jen s jednou proměnnou. Ukažme si nejprve výchozí graf.

```
> stripchart(x = mtcars$mpg, main = "stripchart",
+   xlab = "mpg", sub = "Data mtcars", pch = "o")
```

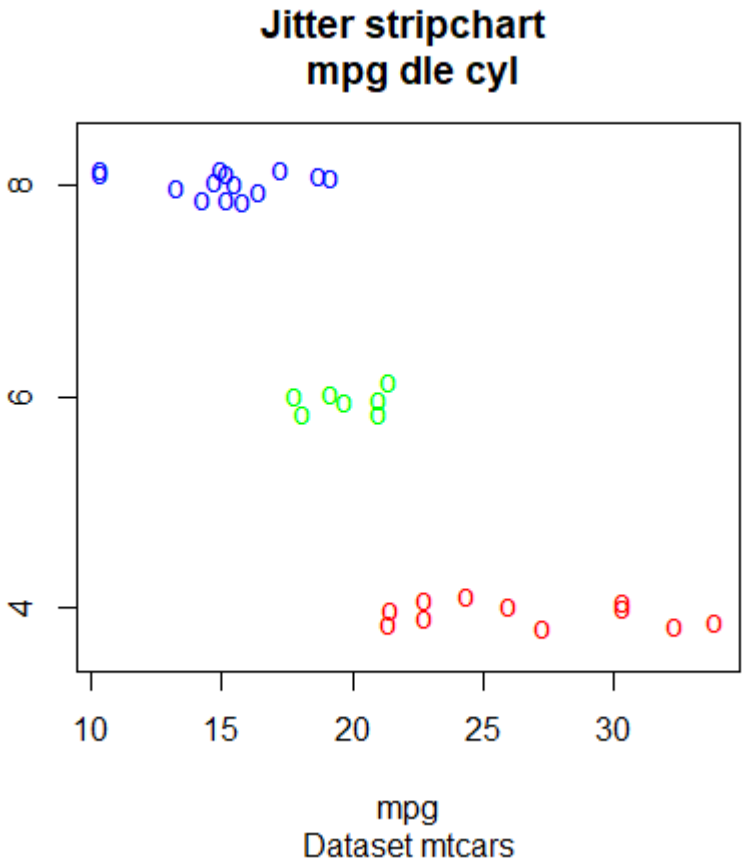


Obrázek 6.18: Funkce stripchart()

Zdroj: Vlastní zpracování.

Na první pohled se může jednat o nepřehledný graf. Máme zde jen jednu osu – x. Na této ose jsou vizualizované hodnoty z předaného vektoru `mpg`. Z grafu je vidět, že se mnoho hodnot navíc překrývá, a pokud by nabývaly přibližně stejné hodnoty, tak nejsme schopni opticky posoudit „zahuštění“ bodů v dané části grafu. Důležitým parametrem je zde `jitter`. Tento parametr ovlivňuje to, jak jsou body zahuštěné v místě jejich překrytí. Pokud nastavíme metodu z `overplot` na `jitter`, tak se aktivuje vertikální rozhození bodů. Další výhodou tohoto grafu je, že umí přebrat vzorec, a tak jsme schopni rozdělit data například pomocí proměnné `mpg` a podle počtu válců (cyl proměnná).

```
> stripchart(mtcars$mpg ~ mtcars$cyl,
+   xlab = "mpg", pch = "o", lwd = 1.5, jitter = 0.1,
+   method = "jitter", col = rainbow(3),
+   main = "Jitter stripchart \n mpg dle cyl",
+   sub = "Dataset mtcars"
+ )
```



Obrázek 6.19: Funkce stripchart() se specifikací jitter

Zdroj: Vlastní zpracování.

Na tomto příkladu je vidět, že nastavení parametrů funkce pro tvorbu grafů je velmi důležité a ovlivňuje, jak čtenář graf vnímá a jakou informační hodnotu pro něj má. Z prvního grafu vytvořeného pomocí funkce `stripchart()` nešlo mnoho údajů vyčíst. Naopak z druhého je snadno patrné významné rozdělení spotřeby benzínu v závislosti na počtu válců.

6.2.4 Funkce par() a ostatní nízkoúrovňové funkce

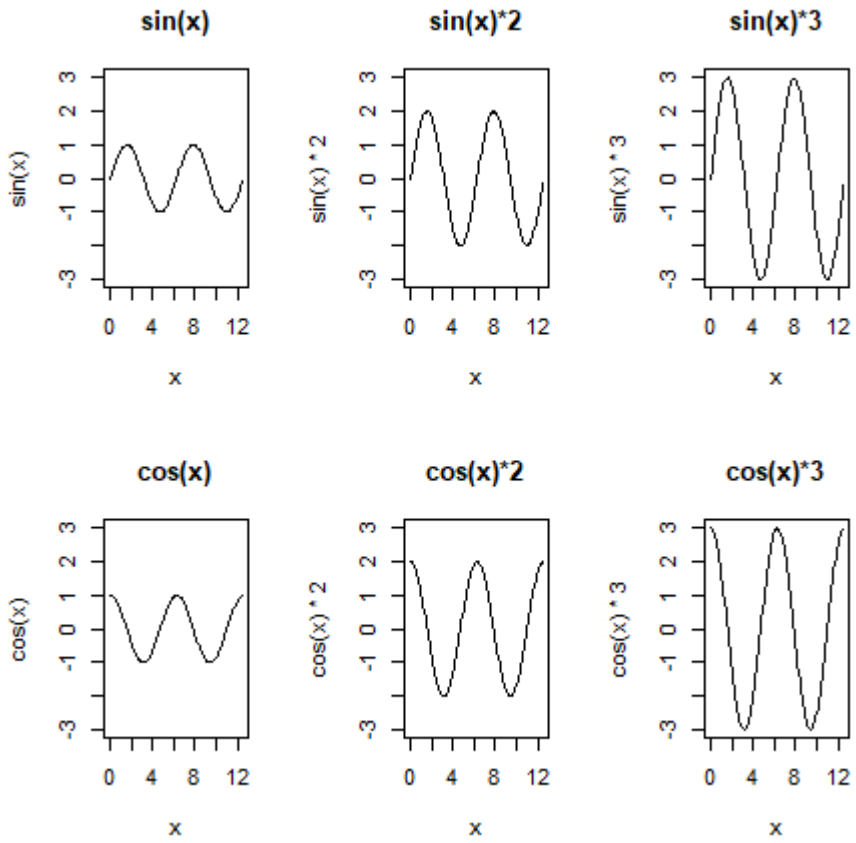
Na začátku kapitoly jsme si představili parametry funkcí, které jsou použitelné zejména pro vysokoúrovňové grafické funkce (tedy funkce tvořící graf). Nedílnou součástí jazyka **R** jsou ale i nízkoúrovňové funkce. Z nich se teď podrobně podíváme na tu nejdůležitější – funkci `par()`. Funkce par je funkce, která, když se zavolá, tak navrátí vektor typu list, na jehož prvcích jsou klíčová grafická nastavení pro tvorbu grafu. Nízkoúrovňová funkce nikdy neovlivňuje graf, který je již zobrazen, popřípadě grafické okno, které je otevřeno. Zde je důležité upozornit na významný rozdíl mezi výchozím prostředím **R** a RStudiem.

Více grafů v jedné tiskové oblasti

V případě, kdy chceme do jedné tiskové oblasti vytvořit více grafů, máme na výběr mezi dvěma možnostmi. Buď chceme vytvořit grafické okno, kde každý graf bude mít stejné zastoupení z pohledu prostoru a tyto grafy budou srovnané v řádkách/sloupcích (parametry `mfrow/mfcol` z par funkce), anebo musíme využít layout funkce. Třetí možností by bylo použít funkci `split.screen()`. Avšak této poslední možnosti je lepší se vyvarovat. Vytváří mnohé problémy s editací a následným ukládáním grafů.

Nízkoúrovňová funkce par má dva parametry, které umožňují vložit více grafů do jedné grafické oblasti. Parametr `mfrow` přebírá vektor, kde je na prvním místě počet řádků grafů a na druhém místě počet sloupců grafů, které má budoucí grafická oblast obsahovat. Parametr `mfcol` má pořadí prvků ve vektoru, který přebírá jen přehozené – na prvním místě počet sloupců a na druhém počet řádků grafů. Nastavením jednoho parametru automaticky funkce `par()` přenastaví ten druhý. Uživatel se nikdy nesmí snažit oba parametry nastavit najednou. Například budu-li si chtít vykreslit funkce sin(x), cos(x) a jejich násobky dvěma a třemi v jednom grafickém okně, ale každý graf zvlášť. Výsledkem budou tedy tři grafy vedle sebe a dva řádky, jeden pro funkci sinus a jeden pro funkci cosinus.

```
> x = seq(from = 0, to = pi * 4, by = 0.1)
> par(mfrow = c(2, 3))
> plot(sin(x) ~ x, main = "sin(x)",
+      type = "l", ylim = c(-3, 3))
> plot(sin(x) * 2 ~ x, main = "sin(x)*2",
+      type = "l", ylim = c(-3, 3))
> plot(sin(x) * 3 ~ x, main = "sin(x)*3",
+      type = "l", ylim = c(-3, 3))
> plot(cos(x) ~ x, main = "cos(x)",
+      type = "l", ylim = c(-3, 3))
> plot(cos(x) * 2 ~ x, main = "cos(x)*2",
+      type = "l", ylim = c(-3, 3))
> plot(cos(x) * 3 ~ x, main = "cos(x)*3",
+      type = "l", ylim = c(-3, 3))
```

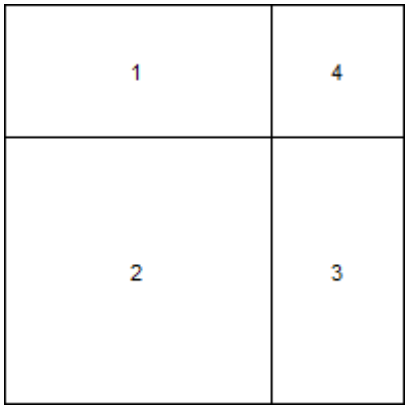


Obrázek 6.20: Více grafů v jedné grafické oblasti – funkce par()

Zdroj: Vlastní zpracování.

Další důležitý rozdíl mezi parametrem `mfrow` a `mfcol` spočívá v pořadí, v jakém jsou jednotlivé oblasti pro grafy naplňovány. Parametr `mfrow` postupuje po řádcích. **R** začíná vyplňovat v levém horním rohu a postupuje po jednotlivých řádcích. Naopak parametr `mfcol` způsobí, že grafy jsou umísťované do buněk po sloupcích. Pro vyplňování grafů lze použít i cykly. Alternativou k tomuto postupu je vytvoření si vlastního grafického modelu pomocí funkce `layout()`. Funkce layout přebírá jako hlavní parametr matici, kde jednotlivé buňky matice představují grafy. Buňky této matice můžeme graficky spojit tak, že jim přiřadíme stejnou hodnotu. Zároveň pořadí čísel definuje pořadí naplňování grafické oblasti grafy.

```
m <- matrix(c(1, 2, 2, 1, 2, 2, 4, 3, 3), ncol = 3)
my.layout <- layout(m)
layout.show(my.layout)
```

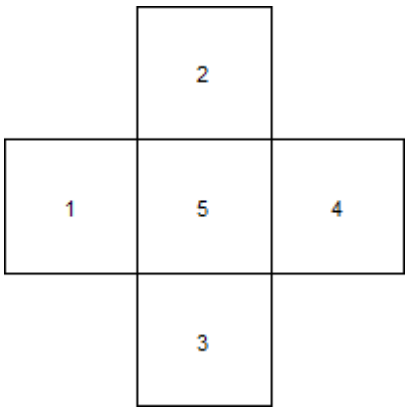


Obrázek 6.21: Layout model grafu

Zdroj: Vlastní zpracování.

Funkce `layout.show()` nám pak umožní vytvořit graf našeho grafického modelu. Tedy funkce `layout()`. show nevytváří budoucí graf, ale ukazuje nám, jaký grafický model se snažíme vytvořit. Dalším důležitým parametrem funkce layout je `width` a `height` – šířka a výška. Tyto dva parametry nám pomohou ovlivnit šířku jednotlivých sloupců a řádků layout modelu. Parametr `n` pak definuje celkový počet grafů, kterým naplníme grafickou oblast.

```
1 > m <- matrix(c(0, 1, 1, 0, 2, 5, 5, 3, 2, 5, 5, 3, 0, 4, 4, 0),
      nrow = 4)
2 > my.layout <- layout(m, height = c(1, 2, 2, 1), width = c(1, 2, 2,
      1))
3 layout.show(my.layout)
```

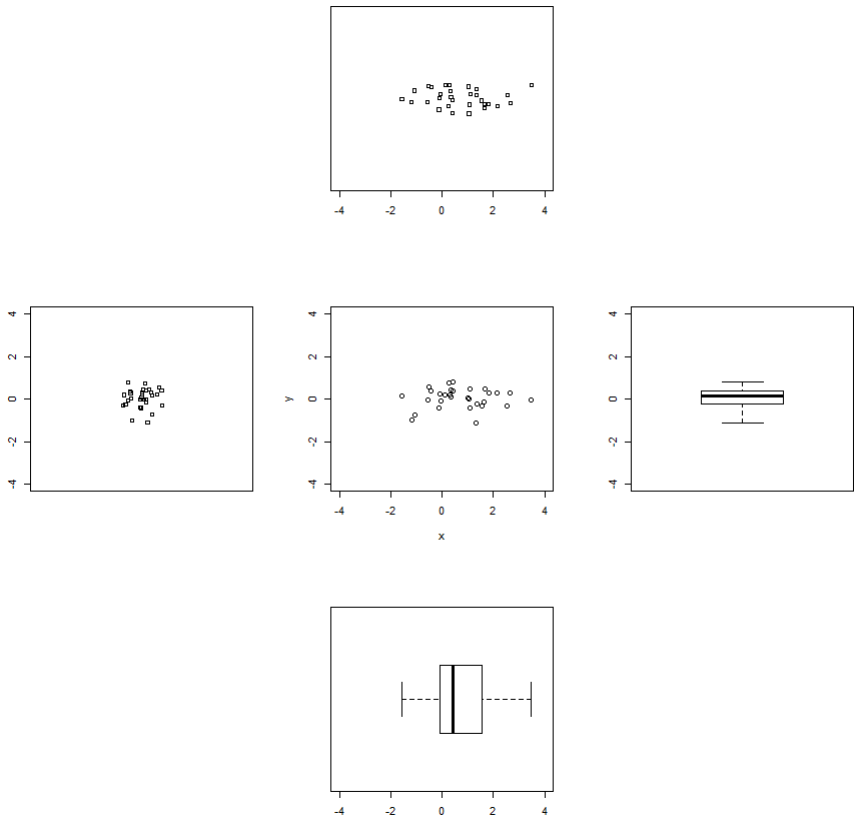


Obrázek 6.22: Layout model grafu II

Zdroj: Vlastní zpracování.

Na tomto grafickém modelu je zejména zajímavé použití nuly. Nula označuje oblasti, které nemají být zaplněny grafy. Oblast grafů nemusí být vůbec souvislá. Zkusíme si vytvořit pět grafů uspořádaných do kříže. Budeme popisovat dvě náhodně generované proměnné, kdy nalevo a nahoře budou jednodimenzionální bodové grafy a napravo a dole budou krabičkové diagramy. Uprostřed se bude nacházet standardní bodový graf obou proměnných.

```
> set.seed(1)
> m <- matrix(c(0, 1, 1, 0, 2, 5, 5, 3, 2, 5, 5, 3, 0, 4, 4, 0),
    nrow = 4)
> my.layout <- layout(m, height = c(2, 1, 1, 2), width = c(2, 1, 1,
    2))
> y <- rnorm(n = 30, mean = 0, sd = 0.5)
> x <- rnorm(n = 30, mean = 0.5, sd = 1.5)
> stripchart(y,
+            method = "jitter",
+            vertical = TRUE,
+            ylim = c(-4, 4))
> stripchart(x, method = "jitter", xlim = c(-4, 4))
> boxplot(x, ylim = c(-4, 4), horizontal = TRUE)
> boxplot(y, ylim = c(-4, 4))
> plot(x, y, xlim = c(-4, 4), ylim = c(-4, 4))
```



Obrázek 6.23: Více grafů v jedné grafické oblasti – layout funkce

Zdroj: Vlastní zpracování.

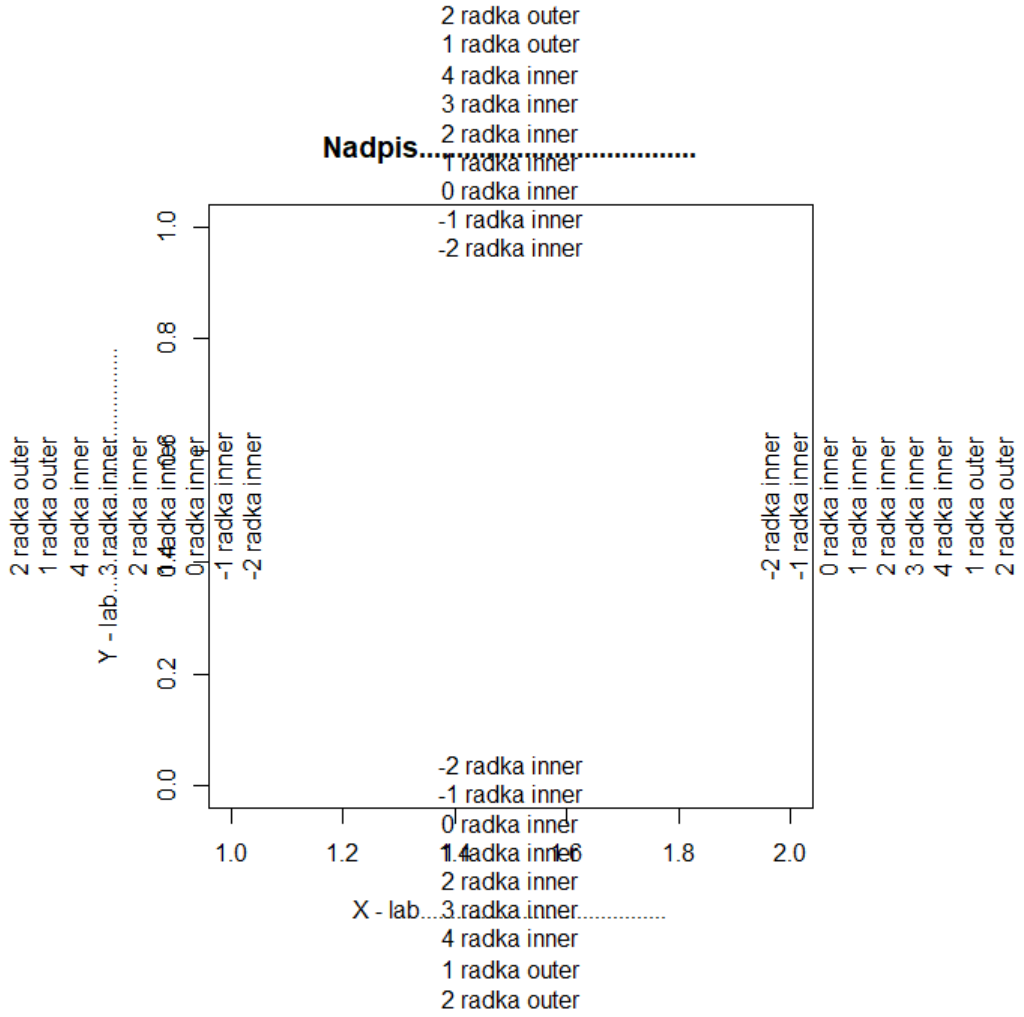
V případě, když vám graf nejde vytvořit, je to nejspíš tím, že máte malou grafickou oblast. Pak stačí zvětšit grafickou oblast (okno, ve kterém se má graf vizualizovat). Nevyřešeným problémem u grafu 6.23 jsou okraje. Možná jste si všimli, že mezi grafy samotnými jsou příliš široké mezery. Na tento problém se podíváme hned v následující kapitole.

Nastavení okrajů grafu

Grafy v **R** mají dva druhy okrajů – vnitřní a vnější. Vnitřní okraj nastavujeme pomocí parametru `mar` (násobky řádků) či `mai` (v palcích). Vnější okraje nastavujeme pomocí `oma` (v řádkách) či `omi` (v palcích). Ve výchozím nastavení jsou vnější okraje nulové a vnitřní mají hodnotu parametru `mar`: 5.1, 4.1, 4.1 a 2.1. Tyto údaje jsou zaznamenány formou atomického numerického vektoru. První hodnota je spodní okraj, druhá levý okraj, třetí hodnota je horní okraj a poslední hodnota je pravý okraj. Hodnoty zapisujeme ve

směru hodinových ručiček od šesté hodiny. Dále je vhodné znát výchozí nastavení grafu v **R**. Řádky pod osou se číslují od nuly. Na nulté řádce jsou značky na osách, na první řádce pod osou jsou umístěné hodnoty příslušící ke značkám os. Další řádka se nechává standardně prázdná a na třetí řádce pak jsou popisky os (`xlab`/`ylab`). S ohledem na to, že se řádky číslují od nuly, je nutné mít okraje minimálně nastavené na hodnotu 4 u os (první a druhý okraj). V následujícím příkladu si na tyto jednotlivé řádky okolo grafu umístíme popisky. Popisky mimo `sub` a `main` lze vytvořit pomocí funkce `mtext()`. Do funkce se zadává text popisku (parametr text), strana (parametr side), řádka, na kterou se má text napsat (parametr `line`), a zda jde o vnější anebo vnitřní okraj (outer).

```
> # Funkci par nastavime okraje
> par(mar = rep(4.1, 4), oma = rep(3, 4))
>
> # Zakladni ramec grafu
> plot(
+   c(0, 1),
+   type = "n",
+   xlab = "X - lab.....................................",
+   ylab = "Y - lab.....................................",
+   main = "Nadpis....................................."
+ )
>
> # Inner okraj
> for (i in-2:4) {
+   mtext(
+     text = paste(i, "radka inner"),
+     side = 1:4,
+     line = i,
+     outer = FALSE
+   )
+ }
> # Outer okraj
> for (i in 1:3) {
+   mtext(
+     text = paste(i, "radka outer"),
+     side = 1:4,
+     line = i,
+     outer = TRUE
+   )
+ }
```

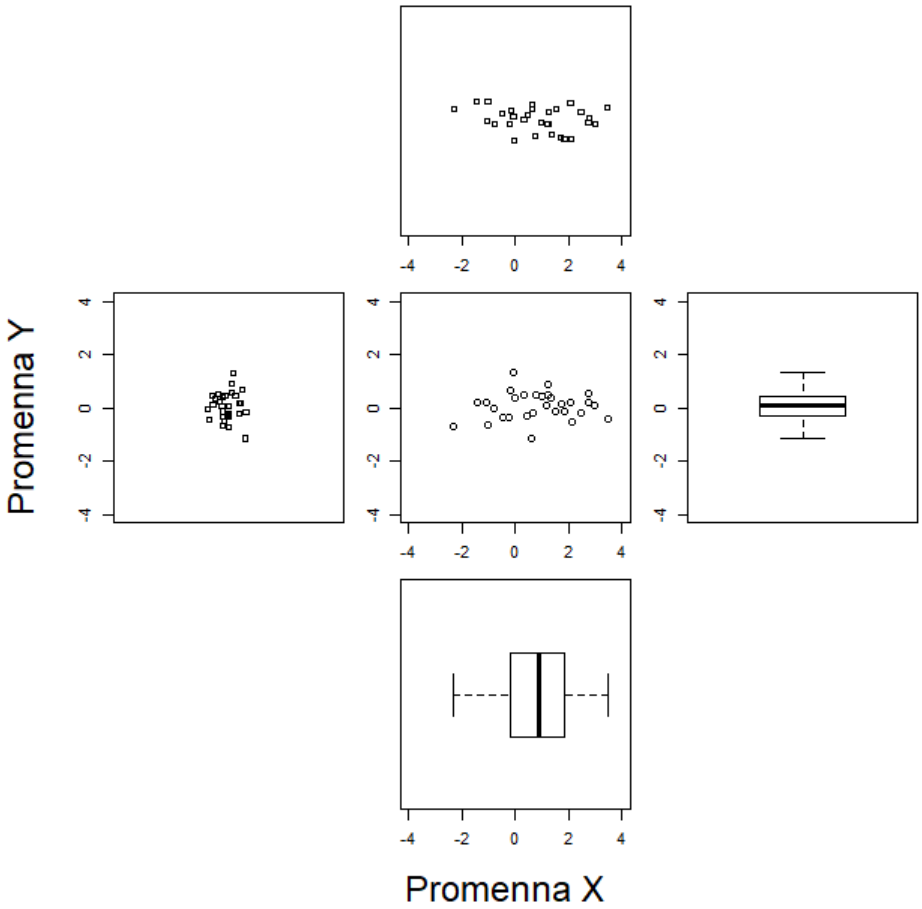


Obrázek 6.24: Okraje grafu – outer a inner okraj

Zdroj: Vlastní zpracování.

Můžeme se ptát, proč si ale v jazyku **R** komplikujeme situaci tím, že máme dva druhy okrajů? Navíc vnější druh okraje (outer) je nulový? Je to zejména proto, abychom mohli nastavit společný okraj (a text) pro situaci, kdy máme více grafů v grafické oblasti. Proto je ve výchozím nastavení hodnota outer nulová. V následujícím příkladu vylepšujeme graf vytvořený pomocí funkce `layout()`.

```
> set.seed(1)
>
> m <- matrix(c(0, 1, 1, 0, 2, 5, 5, 3, 2, 5, 5, 3, 0, 4, 4, 0),
+             nrow = 4)
>
> my.layout <- layout(m, height = c(2, 1, 1, 2), width = c(2, 1, 1,
    2))
>
> y <- rnorm(n = 30, mean = 0, sd = 0.5)
> x <- rnorm(n = 30, mean = 0.5, sd = 1.5)
>
> # Nastaveni par() funkce
> par(mar = c(2, 2, 1, 1), oma = rep(4, 4))
>
> # Tisk grafu
> stripchart(y,
+            method = "jitter",
+            vertical = TRUE,
+            ylim = c(-4, 4))
> stripchart(x, method = "jitter", xlim = c(-4, 4))
>
> boxplot(x, ylim = c(-4, 4), horizontal = TRUE)
>
> boxplot(y, ylim = c(-4, 4))
>
> plot(x, y, xlim = c(-4, 4), ylim = c(-4, 4))
>
> # Spolecne popisky
> mtext(
+   text = "Promenna X",
+   side = 1,
+   line = 2,
+   outer = TRUE,
+   cex = 1.5
+ )
>
> mtext(
+   text = "Promenna Y",
+   side = 2,
+   line = 2,
+   outer = TRUE,
+   cex = 1.5
+ )
```



Obrázek 6.25: Úprava okrajů grafu

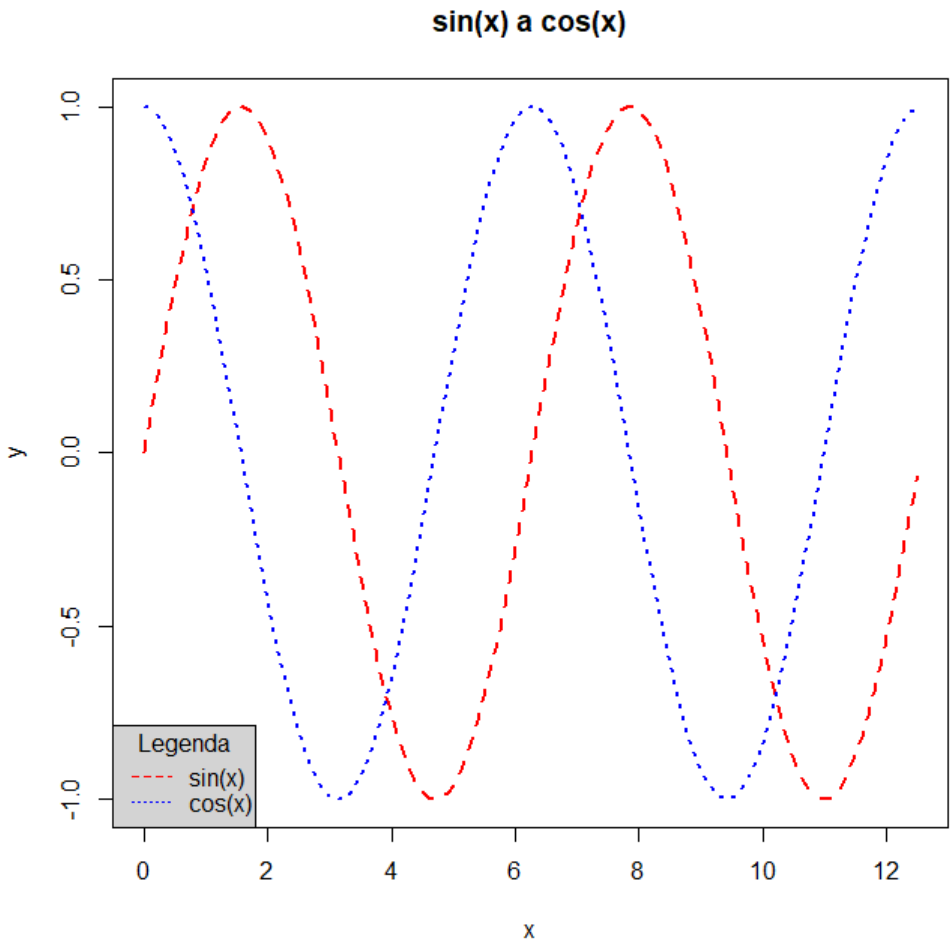
Zdroj: Vlastní zpracování.

Tento způsob editace vnějších okrajů („outer regions“) lze uplatnit jak na grafy vytvořené pomocí parametrů `mfrow/mfcol` z funkce `par()`, tak ale i na `layout()` funkci. Tato metoda nefunguje u grafů vytvořených funkcí `split.screen()`.

6.2.5 Tvorba legendy

Vytváření legendy v jazyku **R** je relativně komplikované. Tato situace souvisí s tím, že legenda není nijak propojená s metodou a atributy pro vizualizaci dat v samotném grafu. Je tedy nutné ručně přiřadit danému popisku správnou barvu a značku. Další problém s legendou souvisí s její pozicí v rámci grafu. Legenda je umístěná uvnitř samotného grafu. Nejprve si vytvořme standardní legendu pro graf dvou proměnných pomocí funkce `legend()`.

```
> x <- seq(from = 0, pi * 4, by = .1)
>
> # Zakladni graf
> plot(
+   sin(x) ~ x,
+   type = "l",
+   col = "red",
+   main = "sin(x) a cos(x)",
+   ylab = "y",
+   xlab = "x",
+   lty = 2,
+   lwd = 2
+ )
>
> # Pridani car
> lines(
+   cos(x) ~ x,
+   type = "l",
+   col = "blue",
+   lty = 3,
+   lwd = 2
+ )
>
> # Tvorba legendy
> legend(
+   "bottomleft",
+   title = "Legenda",
+   legend = c("sin(x)", "cos(x)"),
+   lty = c(2, 3),
+   col = c("red", "blue"),
+   bg = 'lightgrey'
+ )
```



Obrázek 6.26: Tvorba legendy grafu

Zdroj: Vlastní zpracování.

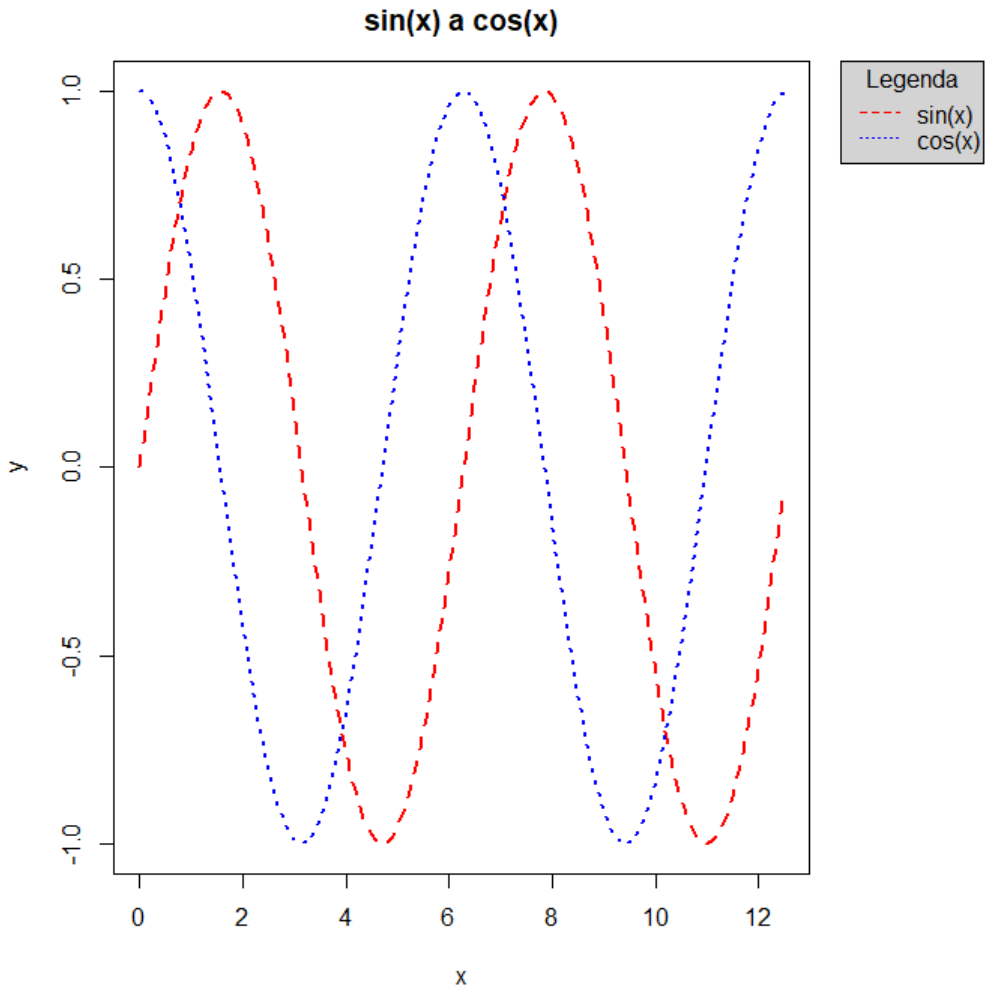
Z příkladu je vidět, že legenda se vytváří stejným způsobem, jako kdybychom vykreslovali graf. Jako první parametr máme pozici legendy. **R** umožňuje textově zadat 9 pozic: „bottomright“, „bottom“, „bottomleft“, „left“, „topleft“, „top“, „topright“, „right“ a „center“. Alternativně lze zadat souřadnice levého horního rohu legendy pomocí parametrů `x` a `y`, které udávají stejné souřadnice, jako jsou na osách grafu pro zobrazení dat. Parametr `title` určuje nadpis legendy. Do parametru `legend` vkládáme jména proměnných (atomický vektor charakterů). V případě, kdy máme spojnicový graf, tak pomocí `lty` vložíme typ čáry. Aby se nám dobře všechny proměnné párovaly, tak je nutné dodržet stejné pořadí (odpovídající popiskům z parametru legend). Pokud by data měla být vykreslena pomocí bodů, tak využijeme parametr `pch`. Pomocí parametru `col` zadáme

barvy odpovídající vykresleným proměnným. Parametr `bg` ovlivňuje barvu pozadí grafu. V případě, kdy bychom chtěli vynechat okraje legendy, využijeme parametr `box.lty=0` (přiřadíme šířce boxu 0 bodů). Parametr `box` má i analogie `lwd` a `col` (`box.lwd`, `box.col`).

Jak ale vyřešit situaci, kdy bychom chtěli legendu umístit mimo vnitřní oblast grafu? My sice můžeme legendě dát ručně souřadnice mimo oblast zobrazení dat, ale legenda se mimo tuto oblast nevykreslí. Maximálně bude kus legendy přesahovat dovnitř grafu, ale nic ven. Tato zdánlivě neřešitelná situace má snadné řešení pomocí funkce `layout()` pro více grafů, kdy jsme druhý graf vykreslili bez bodů a os. Toho jsme dosáhli tak, že jsme parametry `xaxt` a `yaxt` nastavili na `"n"` – žádné okraje. Dále celý okraj (mimo osy i) odstraníme znegováním parametru `frame.plot`.

```
> layout(matrix(c(1, 2), ncol = 2), width = c(4, 1))
> x <- seq(from = 0, pi * 4, by = .1)
> par(mar = c(4, 4, 3, 0))
> plot(sin(x) ~ x,
+      type = "l",
+      col = "red",
+      main = "sin(x) a cos(x)",
+      ylab = "y",
+      xlab = "x",
+      lty = 2,
+      lwd = 2
+ )
> lines(cos(x) ~ x,
+       type = "l",
+       col = "blue",
+       lty = 3,
+       lwd = 2
+ )
>
> # Tvorba prazdne grafikce oblasti pro legendu
> par(mar = c(4, 0, 3, 1))
> plot(c(1),
+      type = "n",
+      frame.plot = FALSE,
+      xaxt = 'n',
+      yaxt = 'n',
+      xlab = "",
+      ylab = ""
+ )
> legend("topright",
+        title = "Legenda",
+        legend = c("sin(x)", "cos(x)"),
+        lty = c(2, 3),
+        col = c("red", "blue"),
```

```
+         bg = 'lightgrey'
+ )
```



Obrázek 6.27: Tvorba legendy okrajů grafu mimo oblast grafu

Zdroj: Vlastní zpracování.

6.2.6 Práce s grafickými okny v softwaru R

V případě, kdy nepracujeme v RStudiu ale v základním prostředí jazyka R, se nám nová okna grafů zobrazují v samostatném okně. Zde můžeme snadno nové okno (bez grafu) otevřít pomocí příkazu `dev.new()`, zavřít pomocí `dev.off()`. Pozor – **R** zavírá aktivní okno! Tedy to, se kterým jsme naposledy pracovali. Příkaz `dev.cur()` vrátí do příkazové

řádky číslo okna, které používáme, a naopak pomocí příkazu `dev.set()` je můžeme přepnout do jiného okna. Příkaz `dev.next()` nám přepne do následujícího grafického okna a příkaz `dev.prev()` do předchozího. Pomocí příkazu `graphics.off()` všechna okna zavřeme. Pokud již máme nějaký graf vytištěný (zobrazený) v samostatném grafickém okně, můžeme toto okno „okopírovat“ pomocí příkazu `dev.copy()` a vložit jej do jiného. Tato situace je výhodná zejména ve chvíli, kdy jsme vytvořili graf a poté jsme si uvědomili, že jej chceme i uložit. Příkaz `dev.print()` vloží okopírované příkazy do nového grafického okna. Výhodou této práce je i to, že můžeme ovlivňovat dopředu to, jak bude grafická oblast velká. Například příkaz `dev.new(height=6, width=3)` vytvoří nové grafické okno o výšce 6 palců a šířce 3 palce.

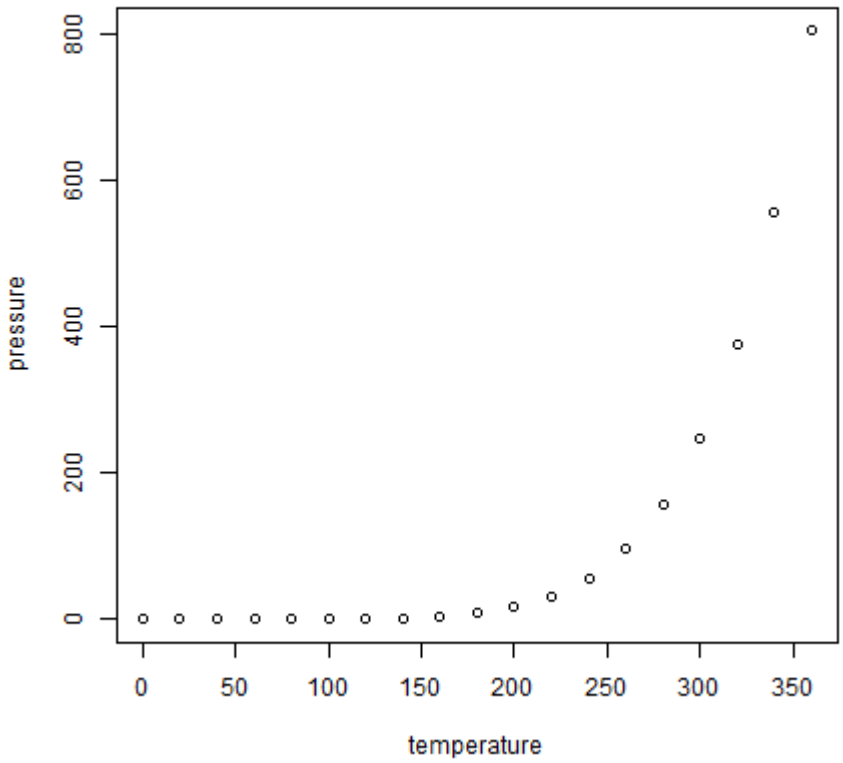
6.2.7 Jak grafy ukládat

V jazyce **R** máme několik způsobů, jak uložit graf. Úplně nejjednoduší je v RStudiu kliknout na tlačítko Export a zvolit požadovaný formát, velikost a umístění. V případě, kdy ale potřebujeme ukládání grafů automatizovat, anebo chceme uložit více grafů do jednoho souboru (např. každý graf na jednu stranu souboru pdf), se hodí znát základní příkazy, kterými toho můžeme dosáhnout. Základní způsob ukládání grafu funguje ve třech fázích. V první fázi se vytvoří soubor a otevře se spojení se souborem. Soubor v této fázi vidíme v počítači, jmenuje se tak, jak požadujeme, ale obsahuje 0 bitů dat. Vytvořený soubor je tedy rezervovaný pro naši úlohu. V druhé fázi soubor začínáme naplňovat. Naplňování souboru provádíme tak, jako bychom vytvářeli graf zcela normálně. Voláme tak například funkci `plot()` či `par()` a jiné. V této fázi se všechna data ukládají do dočasné paměti počítače. Graf se nikde nezobrazuje a v **R** se ani neotevře okno. Funkce je tedy nutné si předem připravit a vyzkoušet, protože v této fázi vlastně uživatel nevidí grafický výstup. V třetí fázi pak uzavíráme spojení s grafem (souborem). Při uzavírání spojení dojde k přesunu dat z mezipaměti počítače do daného souboru. Soubor nebude již rezervován softwarem **R** a uživatel si jej může otevřít v jakémkoliv prohlížeči obrázků/fotek/PDF či editoru.

V první fázi je nutné definovat, kde se v počítači soubor bude nacházet, jak se bude jmenovat a jaké bude mít parametry. Za parametry souboru považujeme rozměry obrázku, rozlišení a jiné vlastnosti. První fázi lze provést pomocí funkcí: `bmp()`, `jpeg()`, `png()`, `tiff()`, `pdf()`, `postscript()`, `bitmap()`, `pictex()`, `xfig()`, `win.metafile()`. Jména funkcí napovídají typům souborů, ve kterých chceme graf uložit. Tyto funkce lze rozšířit o další formáty (`devGTK`, `devJava`, `devSVG`...) pomocí dodatečných balíčků. Ukažme si jednoduchý způsob, jak uložit graf vztahu mezi tlakem a teplotou rtuti.

```
1 > getwd()
2 "C:/Users/Documents/R"
3 > #Prvni faze - otevreni spojeni se souborem
4 > png(filename = "pressure.png",width=450,height=450)
5 > #druha faze - tvorba grafu (popripade jakekoliv jine graficke
    funkce)
```

```
> plot(pressure)
> dev.off() #treti faze - uzavreni spojeni
> dir()
[1] "pressure.png"
```



Obrázek 6.28: Ukládání grafů

Zdroj: Vlastní zpracování.

Alternativou k tomuto postupu je uložit již zobrazený graf. Pro tento účel má jazyk **R** funkci `savePlot()`. Hlavní parametr této funkce je adresa a jméno souboru. Dále `type` – specifikace typu souboru (`wmf, emf, png, jpg, jpeg, bmp, tif, tiff, ps, eps, pdf`). Parametr device slouží k identifikaci okna grafu, které se má uložit. Tento postup ale bude fungovat pouze v případě, kdy je graf vytvořen do samostatného okna. Proto v případě RStudia musí uživatel zavolat před tvorbou grafu příkaz `dev.new()` – který budoucí graf vytvoří do nového okna.

```
> dev.new()
> plot(pressure)
> savePlot("pressure2.png")
```

Závěrem k ukládání grafů je dobré například poznamenat, že absolutní/relativní nastavení parametrů ovlivňující rozměry objektů v grafu má velký vliv na výslednou podobu grafu při ukládání. V případě, kdy bude čtenář vyrábět větší množství grafů, je vhodné si předem rozmyslet rozlišení, ve kterém se budou výsledné grafy ukládat, a následně je držet, neboť stejně nastavená tloušťka čáry při rozlišení grafu 450×450 px bude mít úplně jinou podobu v grafu při 900×900 px.

6.3 Ggplot2

Knihovna `ggplot2` je jednou z nejpoužívanějších knihoven pro vizualizaci dat v softwaru **R**. Tato knihovna je postavená na konceptu „grammar of graphics" (Wickham, 2009; Wickham a kol. 2020), který říká, že jakýkoliv graf může být vytvořen z několika málo komponent (data set, vizuální reprezentace dat a systém souřadnic). Prvenství této knihovny je dáno tím, že se snaží co nejvíce usnadnit práci uživateli a provádět co nejvíce úloh automaticky za něj.

Práce s knihovnou je pro uživatele dost intuitivní, například v base graphics jsme složitě přidávali legendu do grafu. Museli jsme definovat ideální polohu, přiřadit grafické atributy a spojit je se jmény proměnných. Knihovna ggplot ji vytvoří automaticky a všechny atributy grafu správně sama páruje. Knihovna disponuje předem definovanou barevnou paletou, která je kontrastní a pomáhá rozeznávat jednotlivé kategorie vizualizovaných dat. Křivky vykreslené touto knihovnou nepůsobí „kostrbatě/hranatě", tak jako například z knihovny base graphics. Knihovna ggplot pracuje ve vrstvách. Postupně na sebe vrstvíme jednotlivé komponenty grafu – od surových dat, přes vizualizaci jednotlivých proměnných, až po vizualizaci statistických veličin.

Knihovna má samozřejmě i nevýhody. S ohledem na to, že každý grafický objekt se vizualizuje ve vlastní vrstvě, tak občas vytvoření grafů z velkého množství dat může i pár sekund trvat a může mít větší velikost z pohledu místa na disku. Grafici ale na oplátku ocení uložení dat ve vrstvách a křivkách pro další zpracování. Knihovna není nijak stará, ale i tak již prošla vývojem (proto `ggplot2` a ne `ggplot`). Proto je dobré pro uživatele, kteří mají zájem se dozvědět více o tvorbě grafů v knihovně ggplot, aby se nespoléhali na starší publikace a vyhledali si zejména aktuální.

6.3.1 Základní model grafu

Hlavní funkcí, kterou budeme v této kapitole používat, je funkce ggplot. Knihovna `ggplot2` disponuje ještě funkcí `qplot()` – „quick plot". Více o této funkci si povíme až v poslední

podkapitole ke knihovně ggplot2.

Jak již bylo zmíněno v úvodu, každý graf má tři základní prvky:

- data,
- aesthetics,
- geometry.

Nejprve musíme mít data, která chceme vizualizovat. Tato data musí být formátu ve `data.frame`. Pokud tomu tak není, tak se je pokusí funkce `ggplot()` na tento formát převést. Knihovna `ggplot2` předpokládá, že uživatel pracuje s čistými daty. Tedy s objektem typu `data.frame`, kde jsou všechny proměnné vhodně pojmenované a data jsou vyčištěná. Při vizualizaci dat se jména proměnných propíšou do popisků grafů, legend.

Aesthetics definuje, jaké atributy (velikost, barvu...) mají vizualizované grafické objekty, které reprezentují data. Proměnné z dat je nutné na tyto vlastnosti namapovat. Toto provádíme pomocí funkce `aes()`, do níž přiřadíme potřebné napojení na data. Funkce `aes()` pak vstupuje do parametru `mapping()`.

Geometry představují grafickou reprezentaci dat, tedy jakým grafickým objektem jsou data prezentovaná (body, čáry, sloupce...). Grafickou interpretaci přiřadíme do grafu pomocí `geoms` funkcí. Všechny funkce pro přidání vrstvy s grafickou interpretací dat začínají `geom_` (popř. na `stats_` - o nich se zmíníme dále). Použití funkce `ggplot()` je následující:

```
> ggplot(data = <data>, mapping = aes(<aesthetics>)) + <geoms_funkce
  >
```

Alternativně se můžeme setkat i s variantou, kdy aesthetics je deklarované („namapované“) uvnitř geom funkce.

```
> ggplot(data = <data>) + <geoms_funkce>(mapping = aes(<aesthetics>)
  )
```

Zejména u grafů, které se skládají z různých grafických vrstev, má toto použití své opodstatnění. Nic nám ale nebrání přenést i přiřazení vlastního zdroje dat (parametr data) do funkce „geom_“. V případě jednoduchého grafu je ale přehlednější definovat tyto parametry v hlavní funkci ggplot. Aesthetic definujeme pomocí funkce `aes()`. Uvnitř této funkce přiřadíme jednotlivé proměnné z dat do příslušných parametrů. Uživatel může využít jakéhokoli způsobu zmiňované deklarace parametrů, ale může i tyto způsoby kombinovat.

V případě `ggplot()` funkce je vhodné ještě rozlišit další důležité prvky vizualizace grafu. Prvním jsou souřadnice (`coordinates`). Není tím myšleno souřadnice ve smyslu geografickém, ale souřadnice bodů v grafu – tedy obvykle kartézské. Funkce, které začínají na

`coord_`, ovlivňují způsob, jakým jsou souřadnice grafu vizualizované. Můžeme použít lineární souřadnice, nelinární souřadnice, fixovat poměr os, zvětšit vybranou část grafu či prohodit osy.

Funkce začínající na `scale_` nám ovlivňují způsob, jakým jsou namapované proměnné vizualizované z pohledu atributů geometrických objektů (barva, velikost...). Pomocí nich můžeme ovlivnit rozsah os, rozdělení škál, ale i přidat jména na jednotlivé osy.

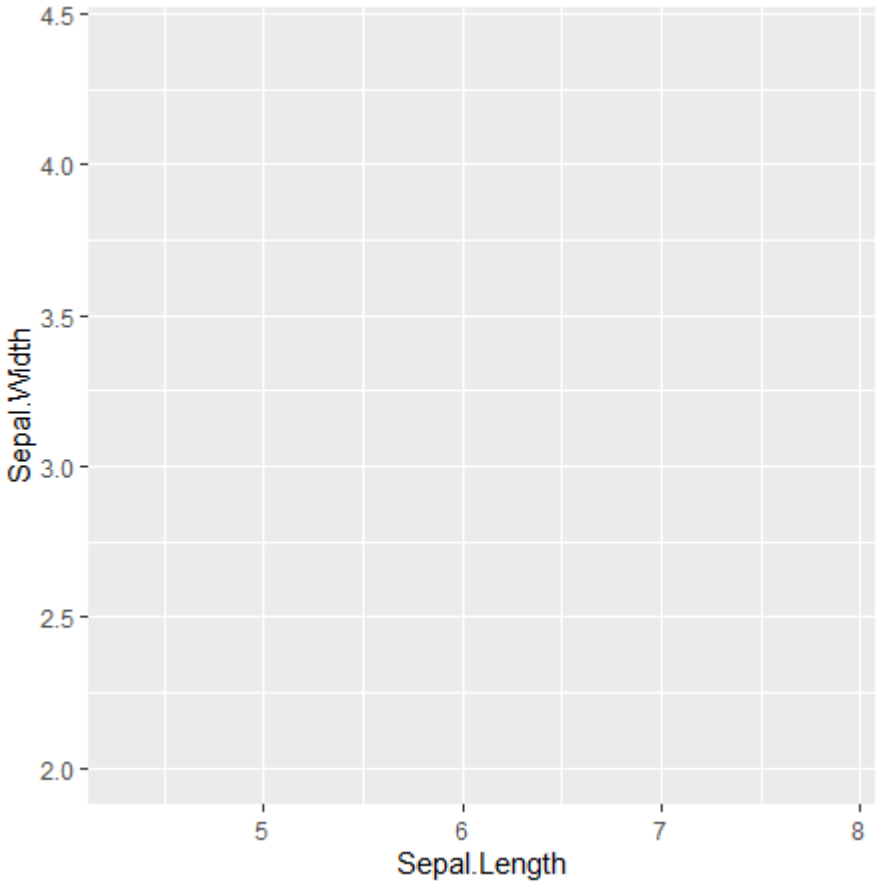
V případě, kdy chceme do grafu přidat grafickou vizualizaci statistické aplikace, využijeme tzv. `stats` funkcí. Jedná se tedy o funkce, které začínají na `stat_`, a jako funkce `geom_` vytváří i funkce `stats` vlastní grafickou vrstvu. I `stats` funkce vytváří vlastní geom objekt, který je vztažený k aesthetics. U některých funkcí jsme schopni dosáhnout velmi podobného či až stejného výsledku (`stats` vs `geoms`), avšak primární rozdíl spočívá v tom, že stats funkce nejprve statisticky interpretují aesthetics a až následně vytvářejí geom objekt.

Facets funkce plní velmi podobnou roli jako `mfrow/mfcol` parametr v `par()` funkci u základních grafů (base graphics). Umožní nám tedy rozdělit grafickou oblast na „buňky“, které následně vyplníme jednotlivými grafy.

Guides a themes funkce nám umožní vytvořit popisky os a upravit další atributy grafu, jako jsou typ a velikost písma. Jak `guides`, tak `theme` funkce vytváří legendu grafu.

Zkusme si udělat základní graf. Ve funkci ggplot přiřadíme data a „namapujeme“ jednotlivé proměnné na osy grafu.

```
1 > ggplot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal.
      Width))
```

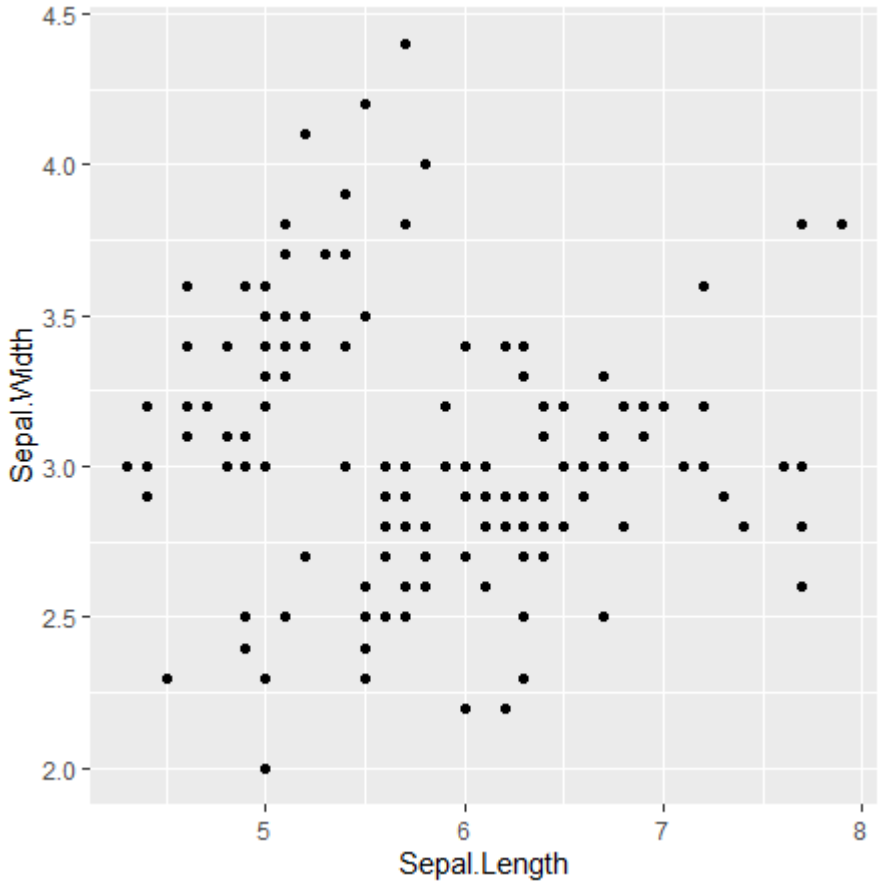


Obrázek 6.29: Funkce ggplot()

Zdroj: Vlastní zpracování.

Pokud postrádáte data v grafu, tak je postrádáte správně – nejsou zde. A to proto, že není přiřazená geom či stat funkce, která by daná data reprezentovala. Zajímavé na tomto grafu je ale to, že jsou zde již přiřazená jména proměnných na obou osách a že funkce `ggplot()` vyřešila rozsah obou os (limity os).

```
> ggplot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal.
    Width)) + geom_point()
```



Obrázek 6.30: Funkce ggplot: geom_point

Zdroj: Vlastní zpracování.

Grafická vrstva geom_point zobrazí příslušná data prostřednictvím bodů. Zcela identický graf můžeme získat i pomocí stats funkce pro identifikaci unikátních dat, tedy pomocí `stats_unique()`.

```
1 > ggplot(data=iris) + geom_point(mapping = aes(x = Sepal.Length, y =
      Sepal.Width))
```

A jak jsme zmínili na začátku kapitoly, tak namapování aesthetics lze provést i v grafické části (a vlastně tedy i v `stat_ funkci()`). Následující dva příklady vygenerují zcela identický graf, jaký jsme již vykreslili.

```
1 > ggplot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal.
    Width)) + stat_unique()
2 > ggplot(data = iris) + stat_unique(mapping = aes(x = Sepal.Length,
    y = Sepal.Width))
```

Knihovna ggplot umožňuje práci s grafy jako s objekty. Je tedy možné si uložit graf do proměnné a pak si jej teprve zobrazit. Dále je možné s touto proměnnou pracovat, přičítat k této proměnné další grafické vrstvy či ji jinak upravovat. Předchozí graf můžeme získat i následujícím způsobem:

```
> p <- ggplot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal
    .Width))
> p + geom_point()
```

Totéž dostaneme i postupným nasčítáním a následně zobrazením.

```
> p <- ggplot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal
    .Width))
> p <- p + geom_point()
> p
```

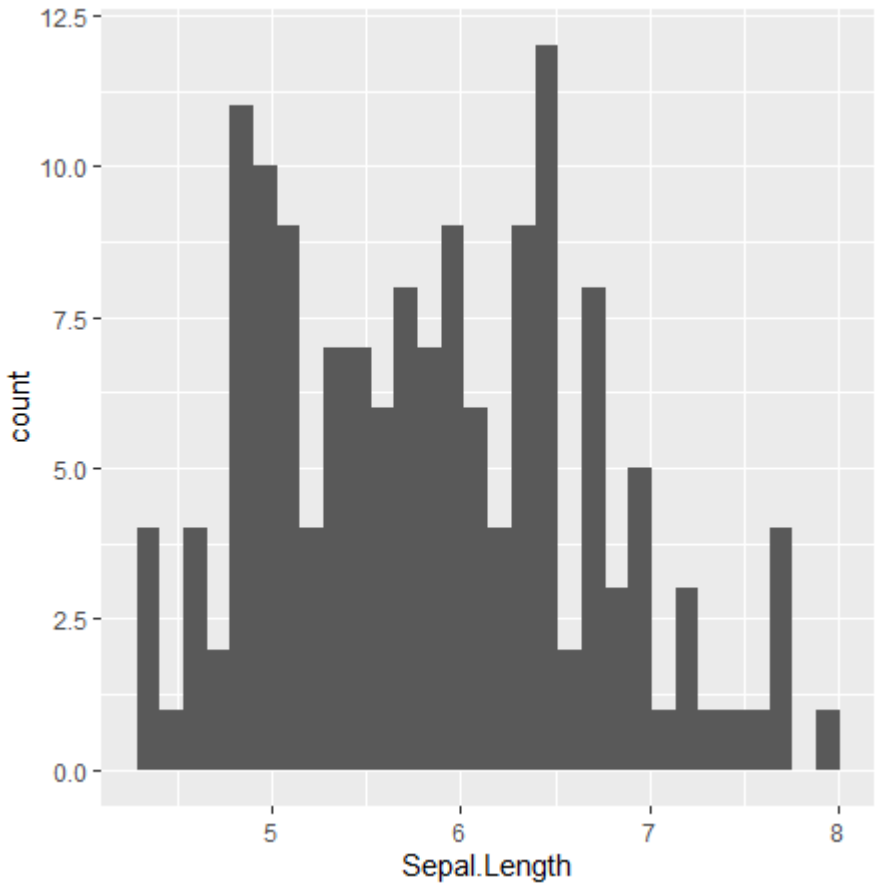
6.3.2 Grafická reprezentace dat – geom a stats

Geoms a stats funkce jsou odpovědné za grafickou vizualizaci dat z pohledu objektu, který je reprezentuje. Definují to, zda objekt bude vizualizován jako čáry, body, sloupce anebo jinou formou. Stat funkce neboli statistická transformace se od geoms odlišuje tím, že před grafickou vizualizací data procházejí transformací, a to obvykle nějakým předem definovaným statistickým postupem. Existuje široká škála geom a stats funkcí. Zaujatému uživateli doporučujeme si projít jednotlivé funkce postupně dle doporučené literatury či dokumentace.

Geoms

V případě grafické reprezentace dat pomocí geoms se obvykle dělí funkce podle počtu proměnných, které zpracovávají. Pro grafickou reprezentaci jedné proměnné máme na výběr funkce, jako je například `geom_area(stat="bin")`. Tato funkce vykreslí plochu grafu pod body, které jsou předané v aesthetics. Funkce geom_density() zase vypočte a vykreslí křivku hustoty. Funkce `geom_dotplot()` vykreslí četnostní sloupcový bodový graf dat, který je rozdělený do skupin. Formou obyčejných sloupců (a ne bodů ve sloupcích) můžeme data vykreslit i pomocí funkce geom_bar(). Funkce `geom_freqpoly()` a `geom_histogram()` vykreslí rozdělení dat (jejich četnost). První pomocí polygonu (linie) a druhou funkci pomocí klasického histogramu. Zkusme si teď tedy vytvořit graf histogramu proměnné `Sepal.Length` z datasetu `iris`.

```
> p <- ggplot(data = iris, mapping = aes(Sepal.Length))
> p <- p + geom_histogram()
> p
```



Obrázek 6.31: Funkce ggplot: geom_histogram

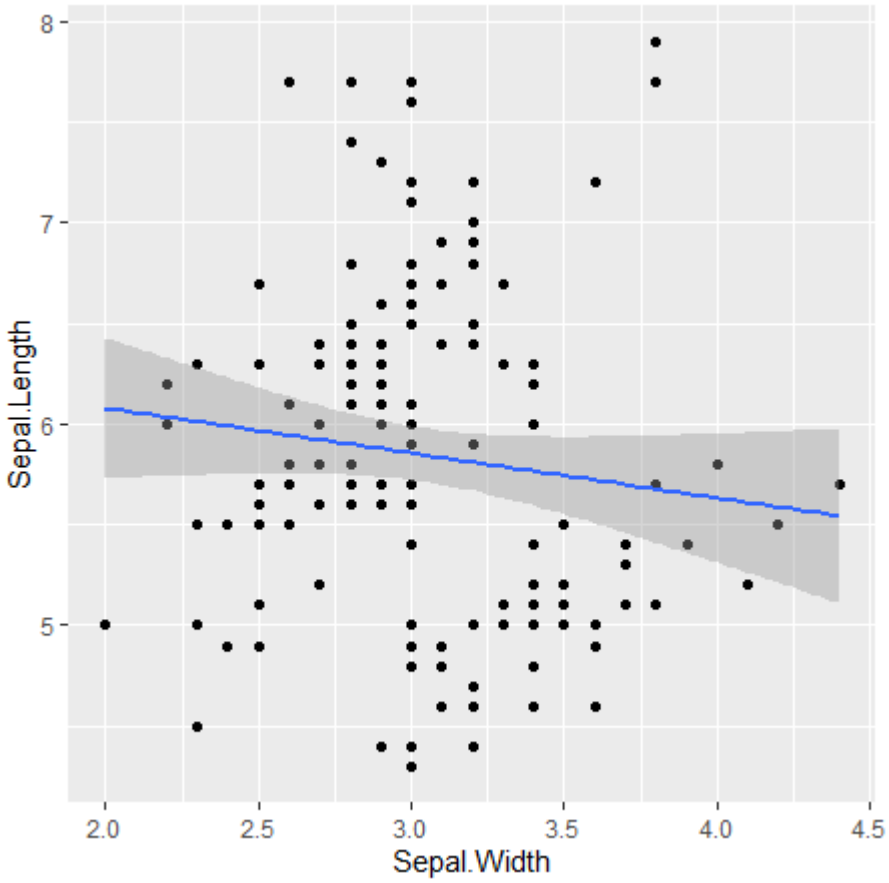
Zdroj: Vlastní zpracování.

V případě, kdy vykreslujeme jednorozměrný objekt (jednu proměnnou), není třeba v aesthetic definovat x a y, ale stačí sem vložit tuto proměnnou. Všechny tyto funkce mají mnoho vlastních parametrů, jako jsou například barva (color) či šířky sloupců aj.

Pro dvě spojité proměnné máme na výběr celou škálu reprezentací dat pomocí geoms. Standardní bodový graf získáme pomocí funkce `geom_point()`. Funkce `geom_jitter()` vykreslí bodový graf, který je „zašumělý“. Body jsou schválně lehce rozhozené od přesného místa jejich polohy. Cílem je zpřehlednit graf v případě, kdy se mnoho bodů zcela překrývá. Kvantily můžeme vykreslit pomocí funkce `geom_quantile()` a pomocí funkce `geom_smooth()` můžeme data protnout vybraným statistickým modelem - například regresí. V následujícím příkladu vykreslíme data `iris` a přidáme vrstvu `geom_smooth()`, která nám umožní přidat regresní funkci do dat.

```
1 > p <- ggplot(data = iris, mapping = aes(y = Sepal.Length, x = Sepal
```

```
    .Width))
> p + geom_point() + geom_smooth(method = "lm")
```

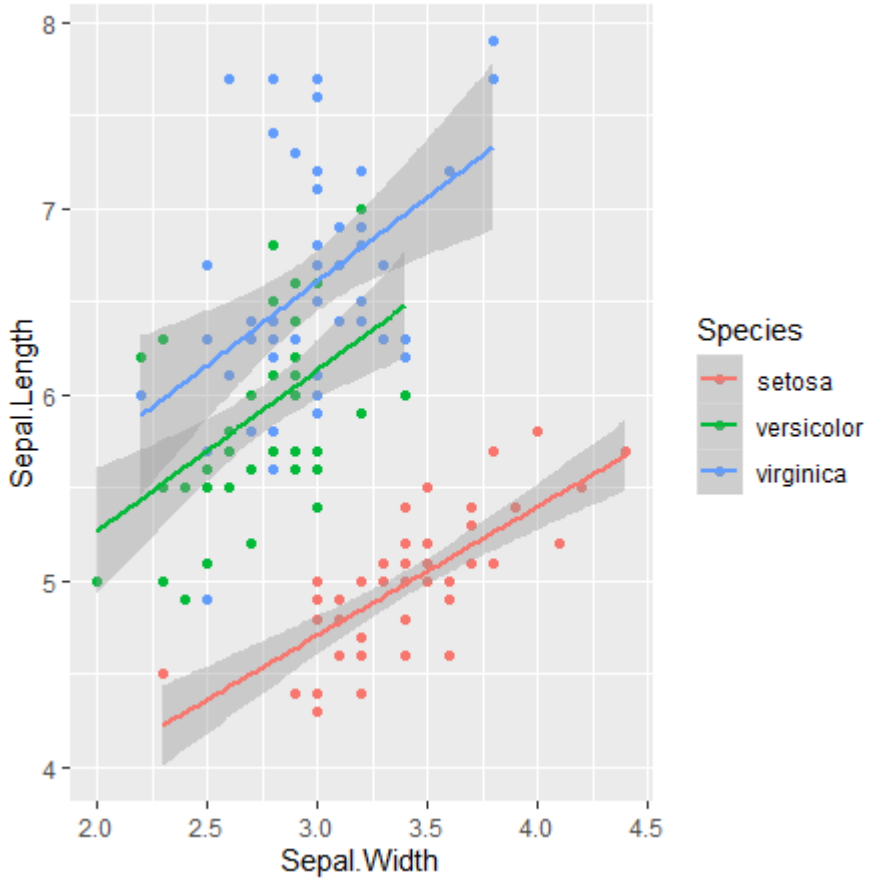


Obrázek 6.32: Funkce ggplot: geom_point, geom_smooth

Zdroj: Vlastní zpracování.

Pozorného čtenáře znalého statistických metod jistě daná regrese nepřesvědčí o své vhodnosti. Co kdybychom ale data rozdělili do skupin a následně za ně provedli vykreslení regresních funkcí? V knihovně ggplot to lze snadno. Data můžeme rozdělit do skupin pomocí několika způsobů, a to buď v aesthetics přiřadíme skupiny do proměnné group, fill, anebo do proměnné color. V případě, že přiřadíme skupiny (vektor skupin, vektor faktorů) do parametru `colour`, data budou i rozdílně vykreslena (pomocí barev).

```
> p <-ggplot(data = iris,
>           mapping = aes(y = Sepal.Length, x = Sepal.Width, color =
    Species))
> p + geom_point() + geom_smooth(method = "lm")
```



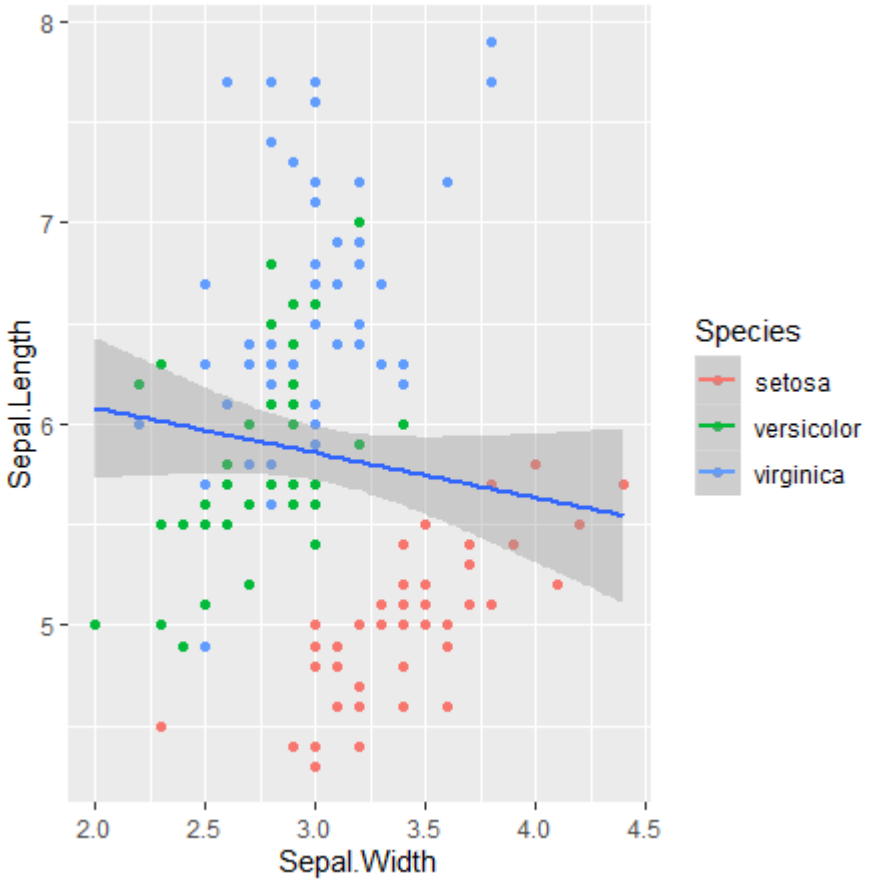
Obrázek 6.33: Funkce ggplot: geom_point, geom_smooth, pro data rozdělená do skupin

Zdroj: Vlastní zpracování.

Regresní funkce získané pomocí `geom_smooth()` za jednotlivé skupiny mají hned i jiný sklon. Toto je velmi hezký příklad toho, jak nám může grafická analýza pomoci s pochopením vztahů v datech. Co ale dělat, když chceme mít data rozdělena do barevných skupin, ale výsledek chceme přesto protnout jednou regresní funkcí? Pak nám stačí ve vrstvě smooth změnit hodnotu deklarace parametru skupin, popř. vyrušit deklaraci barev pro danou vrstvu (pro `geom_smooth()`). Oba následující způsoby vedou ke stejnému výsledku.

```
> p <-ggplot(
+      data = iris,
+      mapping = aes(y = Sepal.Length, x = Sepal.Width, colour =
    Species))
```

```
> p + geom_point() + geom_smooth(aes(group = 1), method = "lm")
> p + geom_point() + geom_smooth(aes(colour = NULL), method = "lm")
```

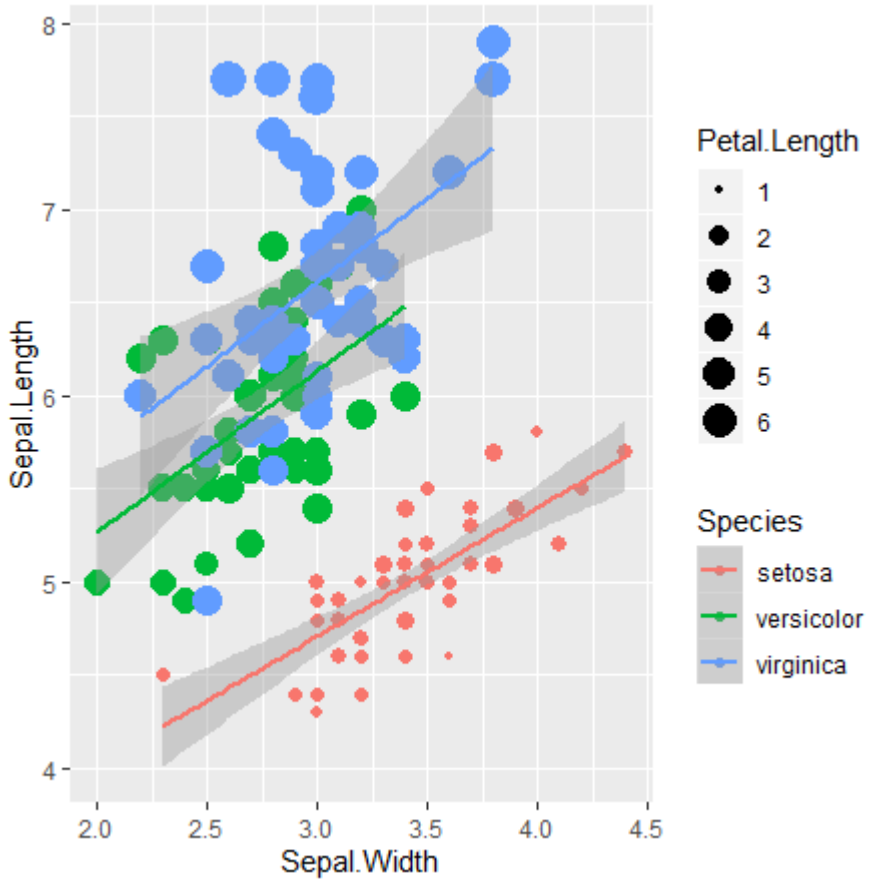


Obrázek 6.34: Funkce ggplot: geom_point, geom_smooth pro data celkem

Zdroj: Vlastní zpracování.

Graf s více regresními funkcemi můžeme vylepšit také například tím, že pro jednotlivé body na grafu zvolíme specifické velikosti. Do vrstvy `geom_point()` přiřadíme vlastní aesthetics, kde do atributu velikosti bodů (size) vložíme i doposud nepoužitou proměnnou z datasetu `iris`. Volba proměnné je zcela na nás, například `Petal.Length`.

```
> p <-ggplot(
+     data = iris,
+     mapping = aes(y = Sepal.Length, x = Sepal.Width, colour =
    Species))
> p + geom_point(aes(size = Petal.Length)) + geom_smooth(method = "
    lm")
```

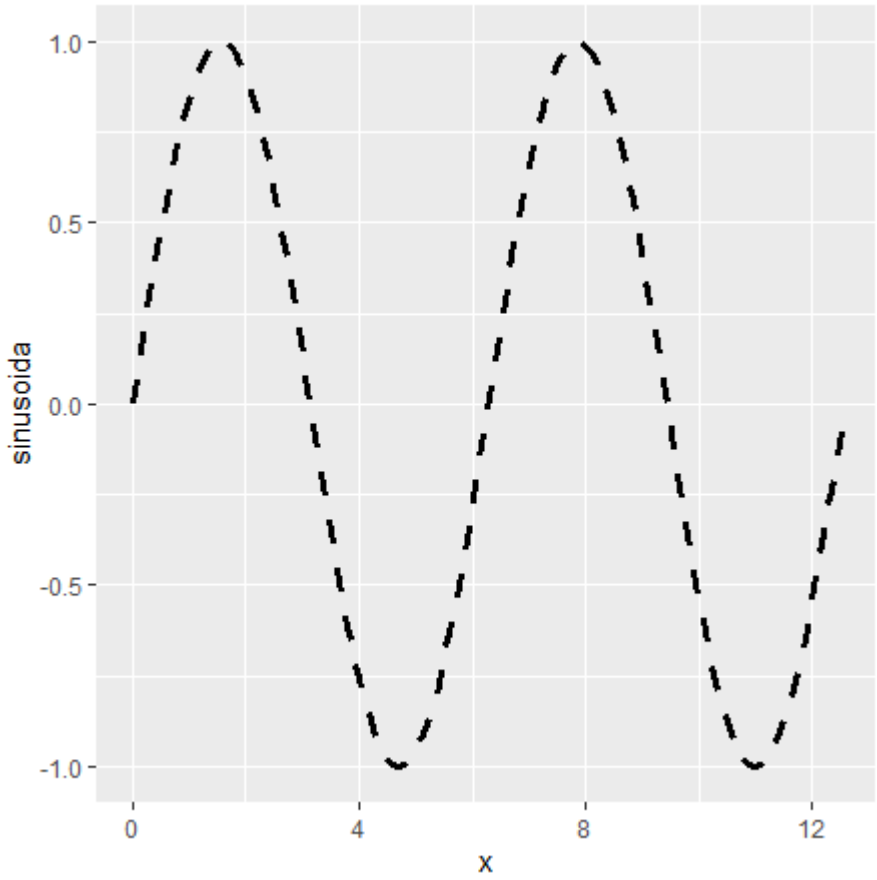


Obrázek 6.35: Funkce ggplot: geom_point, geom_smooth pro skupiny

Zdroj: Vlastní zpracování.

Za povšimnutí stojí zejména to, že **R** automaticky generuje požadované legendy pro dané třídy dat. Intervaly můžeme zrušit přiřazením hodnoty FALSE do parametru `se` ve funkci `geom_smooth()`. Pomocí funkcí `geom_line()` a `geom_area()` můžeme vykreslit liniový graf. V případě funkce `geom_area()` je oblast pod linií vyplněna. Zkusme si jako v base graphics vykreslit i sinusoidu a vykreslit ji do grafu pomocí `geom_line()`.

```
> ggplot(data = datasin, mapping = aes(y = sinusoida, x = x)) +
+         geom_line(size = 1.2, linetype = 2)
```

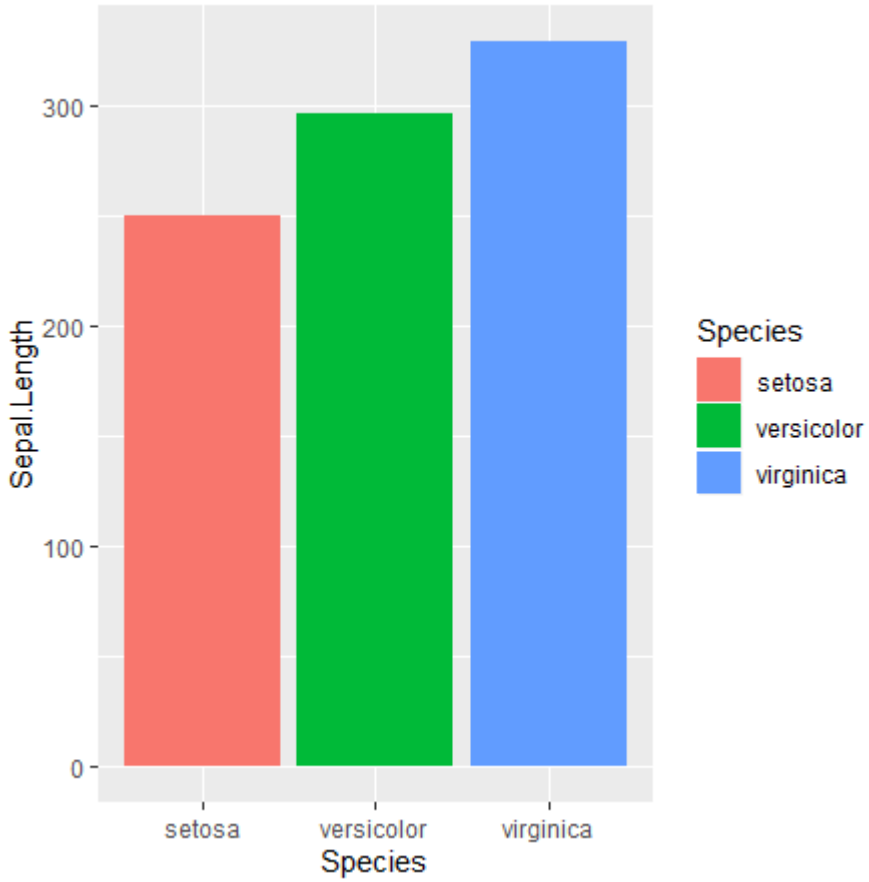


Obrázek 6.36: Funkce ggplot: geom_line

Zdroj: Vlastní zpracování.

Pomocí parametrů ve funkci `geom_line()` můžeme nastavit druh, velikost čáry a průhlednost (alfa). Klasický sloupcový graf můžeme vytvořit pomocí funkce `geom_bar()`. Data rozdělíme do skupin podle druhu rostliny. Zde je důležité použít místo parametru group parametr `fill`. Volba group by znamenala rozdělení do daných skupin, ale nevykreslila by sloupce rozdílně. Pokud bychom zvolili argument `colour`, tak by funkce vykreslila okolo každého pozorování rámeček v barvě dané skupiny, ale nikterak by to nepomohlo odlišit barevně dané slupce. Nastavením parametru `stat` na funkci `identity` získáme klasický sloupcový graf. Funkce `identity` navrací ten samý objekt, který přebírá.

```
1 > p <- ggplot(data = iris, mapping = aes(y = Sepal.Length, x =
      Species, fill = Species))
2 > p + geom_bar(stat = "identity")
```

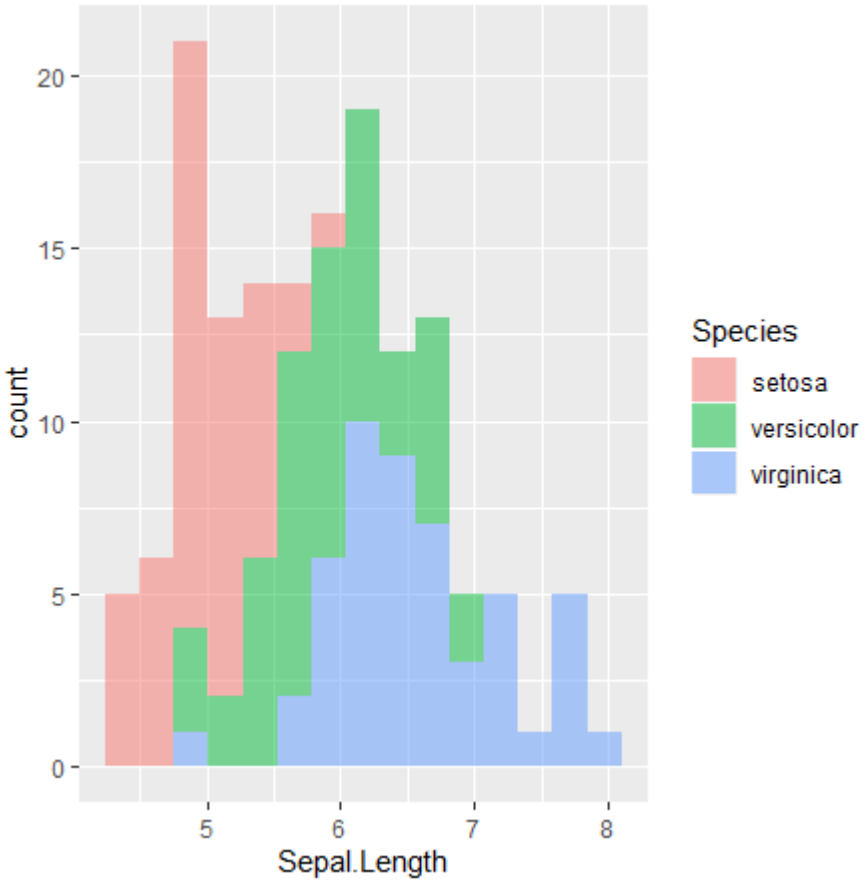


Obrázek 6.37: Funkce ggplot: geom_bar

Zdroj: Vlastní zpracování.

Parametr `alfa` definuje míru průhlednosti vizualizovaných dat, například si můžeme zkusit vykreslit tři histogramy pro všechny skupiny `Species` datasetu `iris` do jednoho grafu. Využijeme parametru průhlednosti, abychom zprůhlednili sloupce a bylo vidět „mřížku“ grafu. Dále zvolíme 15 sloupečků (parametr bins) pro daný histogram.

```
1 > ggplot(data = iris, mapping = aes(x = Sepal.Length, fill = Species
    ))+
2 +   geom_histogram(bins = 15, alpha = 0.5)
```



Obrázek 6.38: Funkce ggplot: geom_histogram

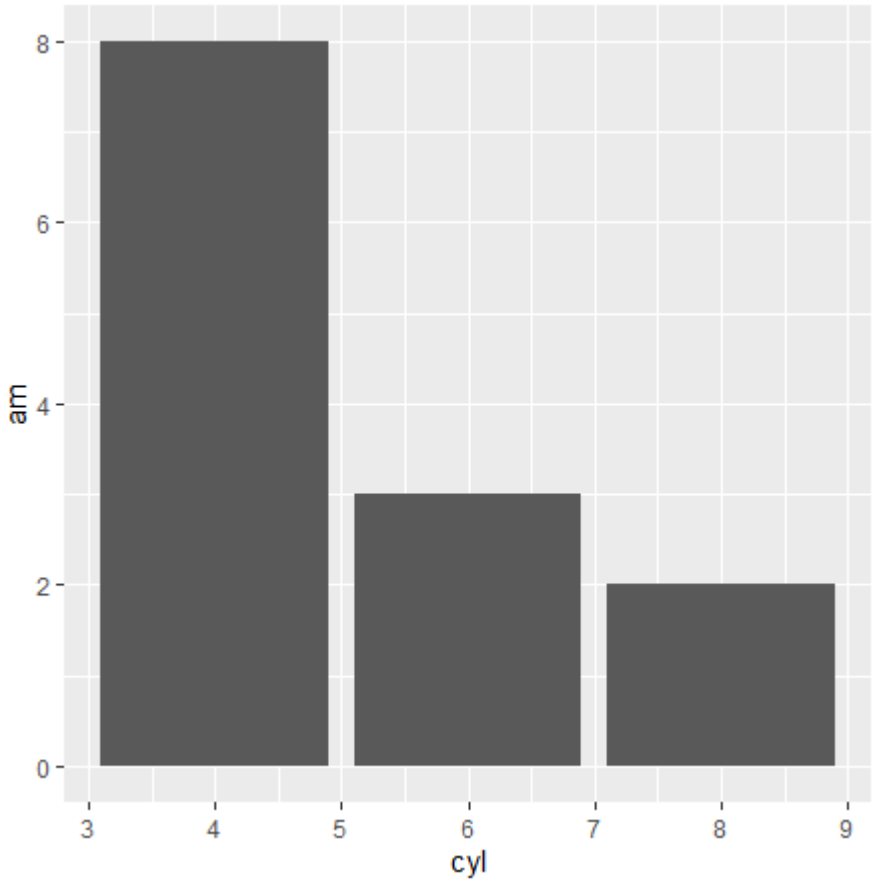
Zdroj: Vlastní zpracování.

6.3.3 Funkce scale

Pomocí `scal_` funkcí můžeme upravovat způsob, jakým jsou aesthetics namapované na jednotlivé grafické vrstvy (geoms). Můžeme ovlivnit barvy, velikosti, ale i rozsah a popisky os (k popiskům se ale dostaneme v následující podkapitole). Po přiřazení proměnných ve funkci `aes()` `ggplot()` sám vybírá nejvhodnější nastavení aesthetics. Způsob výběru těchto nastavení dělá z ggplotu velmi oblíbenou knihovnu – ve výchozím nastavení jsou scales funkce velmi dobře a přehledně nastavené. Avšak může nastat situace, kdy budeme chtít vykreslit některé parametry jinak, popřípadě kdy budeme chtít použití jiné barevné sady či přímo transformovat škálu osy (např. `log, sqrt`...). Scale funkce mají obvykle tři společné základní parametry a mnoho dalších, které se vztahují ke konkrétní funkci. Prvním společným parametrem je `name`. Parametr name ovlivňuje jméno osy/legendy, ke které se vztahuje daná funkce. Dále se jedná o funkci limits. V případě spojité škály

na dané ose, kterou bychom chtěli transformovat, `limits` přebírá minimální a maximální hodnotu dané osy. V případě kategoriální/diskrétní osy pak tento parametr přebírá vektor kategorií, které má vykreslit. Třetím parametrem je `break`. Tento parametr ovlivňuje zlomy na dané ose či v legendě. Lze jej kombinovat s parametrem `labels`, který definuje, jaké popisky se mají u zlomů vizualizovat. Následující sloupcový graf vizualizuje počty aut s automatickou převodovkou (`am=1`) pro počty válců. Co je na tomto grafu špatně?

```
> p <- ggplot(mtcars, aes(y = am, x = cyl))
> p + geom_bar(stat = "identity")
```



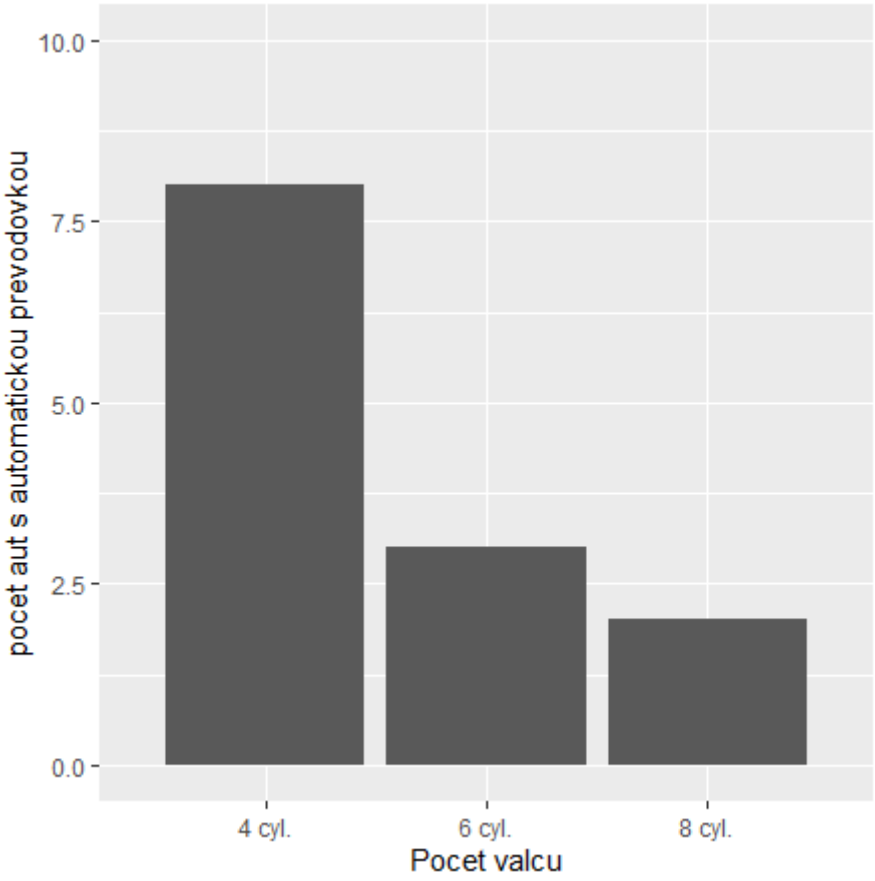
Obrázek 6.39: Funkce ggplot: scale funkce I

Zdroj: Vlastní zpracování.

Možná jste si všimli, že popisky na ose x jsou spojité. Tato osa je chápána jako spojitá veličina. Jsou ale počty válců na motorech aut spojité? Nejsou. V tomto případě máme dvě možnosti, jak daný graf upravit. Buď předáme proměnnou `cyl` jako faktor již v aesthetics,

anebo použijeme funkce scale. Pro převod dané osy na diskrétní můžeme využít funkce: `scale_x_discrete()` a `scale_y_discrete()`. Naopak pro převod na spojitou můžeme využít funkce: `scale_x_continuous()` a `scale_y_continuous()`.

```
> p <- ggplot(mtcars, aes(y = am, x = cyl))
> p + geom_bar(stat = "identity") +
+  scale_x_discrete(
+     name = "Pocet valcu",
+     limits = unique(mtcars$cyl),
+     labels = paste(unique(mtcars$cyl), "cyl.")) +
+     scale_y_continuous(name = "pocet aut s automatickou
    prevodovkou",
+     limits = c(0, 10))
```

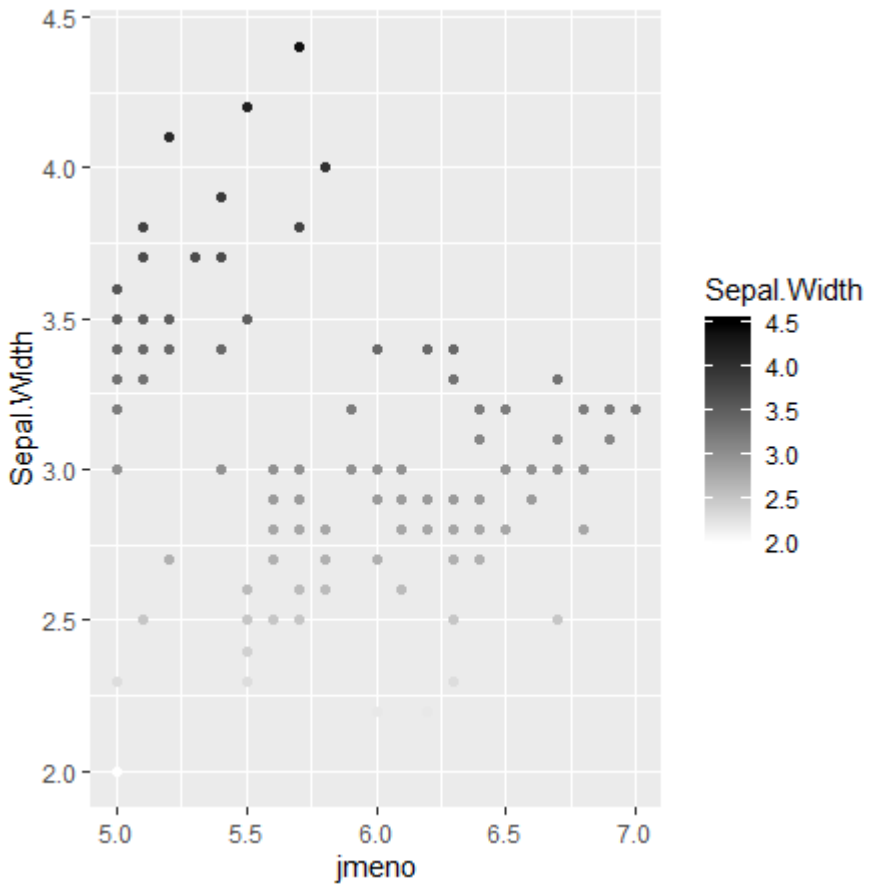


Obrázek 6.40: Funkce ggplot: scale funkce II

Zdroj: Vlastní zpracování.

Obdobně můžeme editovat i barvy, avšak musíme je volit přes parametr, kterým jich bylo dosaženo. Pokud to bylo přes `colour`, tak použijeme funkci `scale_colour()`, pokud to bylo přes fill, tak funkci `scale_fill()`. Záleží vždy na specifické situaci a způsobu tvorbu grafu. Barva bodů grafu je předána pomocí parametru `colour` ve funkci ggplot. Proto dále volíme funkci `scale_colour()` na změnu barvy. Ve funkci `scale_colour_gradient()` volíme minimální a maximální barvu. Tato funkce existuje ve verzi i pro parametr `fill`.

```
> ggplot(data = iris,
+    mapping = aes(x = Sepal.Length, y = Sepal.Width, colour = Sepal.
    Width)) +
+    geom_point() + scale_colour_gradient(limits = c(2, 4.5),
+                                         low = "white",
+                                         high = "black") +
+    scale_x_continuous("jmeno", limits = c(5, 7))
```

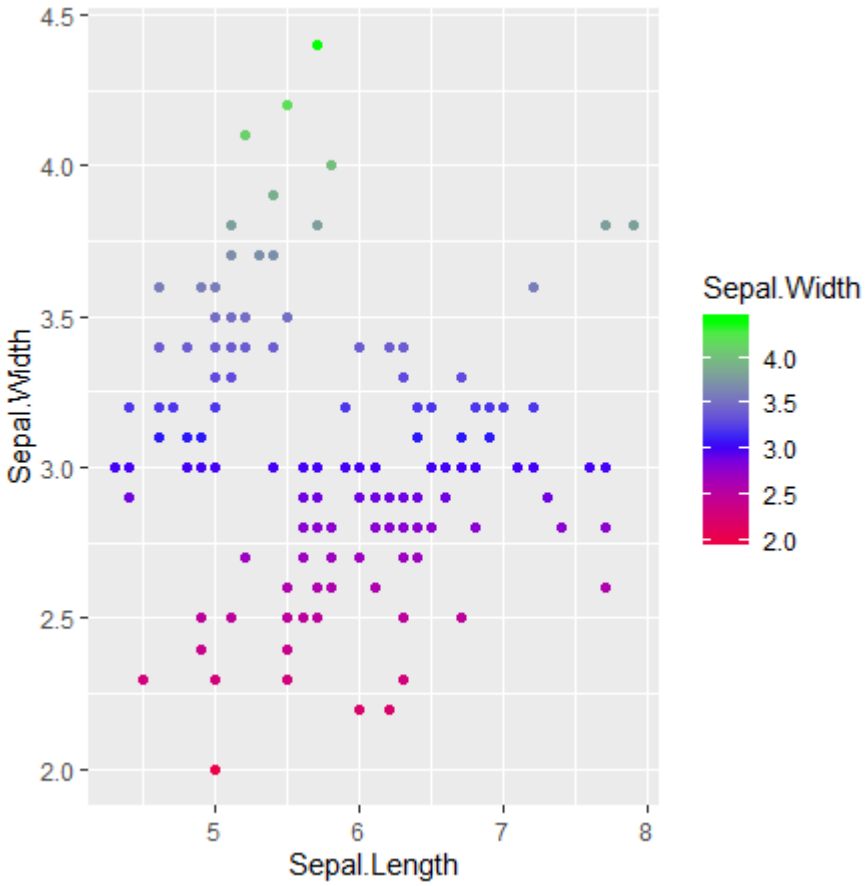


Obrázek 6.41: Funkce ggplot: scale funkce III

Zdroj: Vlastní zpracování.

Ggplot disponuje i funkcemi pro vícepřechodové škály. Tedy pro dva přechody: `scale_colour_gradier` a `scale_fill_gradient2()`. Dále pro n-tý počet přechodů mezi barvami můžeme použít funkce: `scale_colour_gradientn()` a `scale_fill_gradientn()`.

```
> plot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal.Width,
+    colour = Sepal.Width))+ geom_point() +
+    scale_colour_gradient2(midpoint = mean(iris$Sepal.Width),
+    low = "red", mid = "blue", high = "green")
```



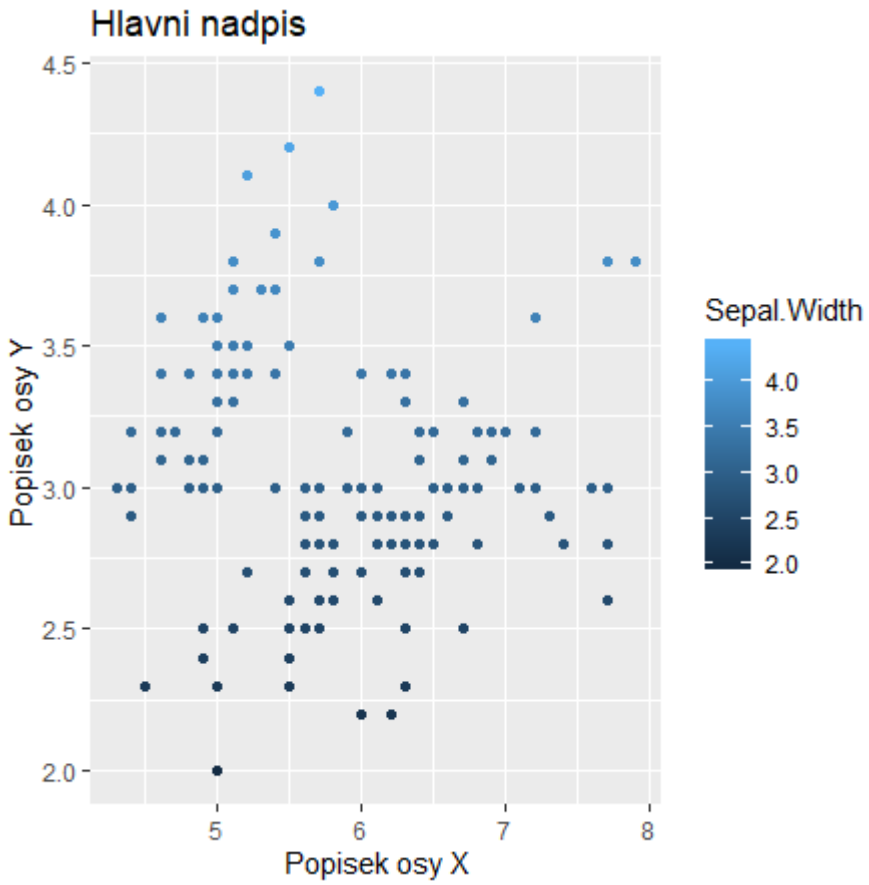
Obrázek 6.42: Funkce ggplot: scale funkce IV

Zdroj: Vlastní zpracování.

6.3.4 Nadpisy a popisky os

Základní editaci nadpisů a popisků grafu můžeme provést pomocí funkcí: `ggtitle()` (hlavní nadpis), `xlab()` a `ylab()` (popisky os).

```
> p <-ggplot(
+     data = iris,
+     mapping = aes(x = Sepal.Length, y = Sepal.Width, colour = Sepal
    .Width)
+   ) + geom_point()
> p + ggtitle("Hlavni nadpis") + xlab("Popisek osy X") + ylab("
    Popisek osy Y")
```

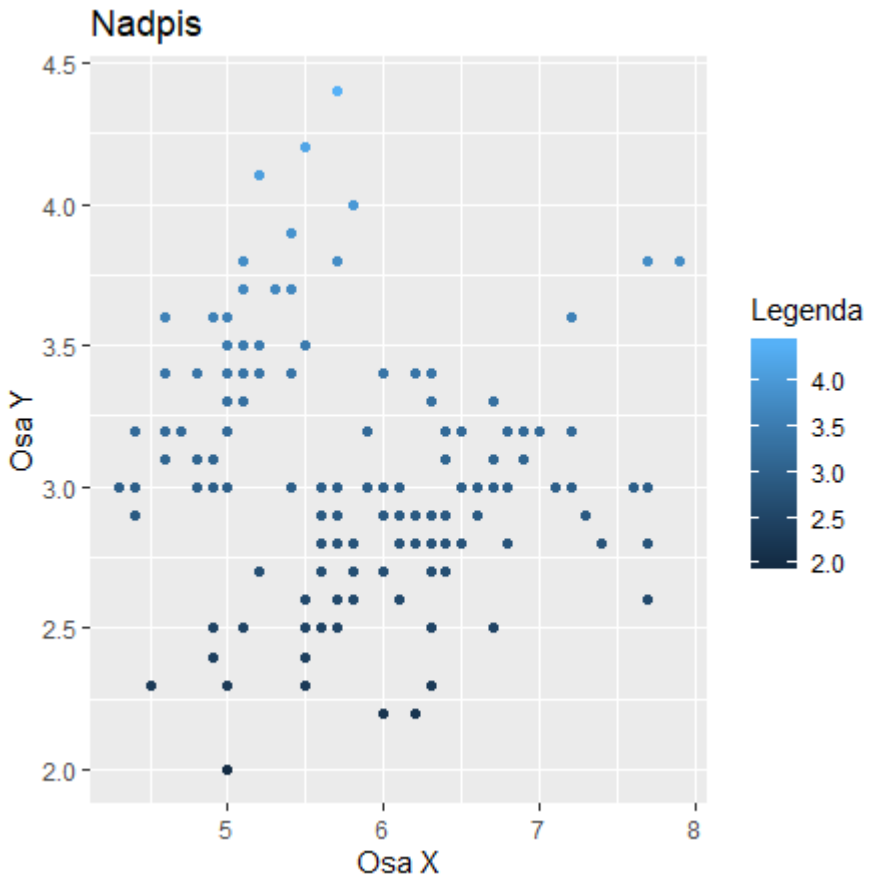


Obrázek 6.43: Funkce ggplot: ggtitle, xlab a ylab

Zdroj: Vlastní zpracování.

Alternativně lze použít funkci `labs()`, která přiřazuje všechny popisky grafu najednou. V případě, kdy chceme editovat legendu či popisky skupin, do kterých jsou data rozdělená, tak to provedeme přes příslušný způsob namapování aesthetics. Pro editaci popisků legendy zde lze použít parametr legend.

```
> p + labs(title = "Nadpis", x = "Osa X", y = "Osa Y", colour = "
    Legenda")
```



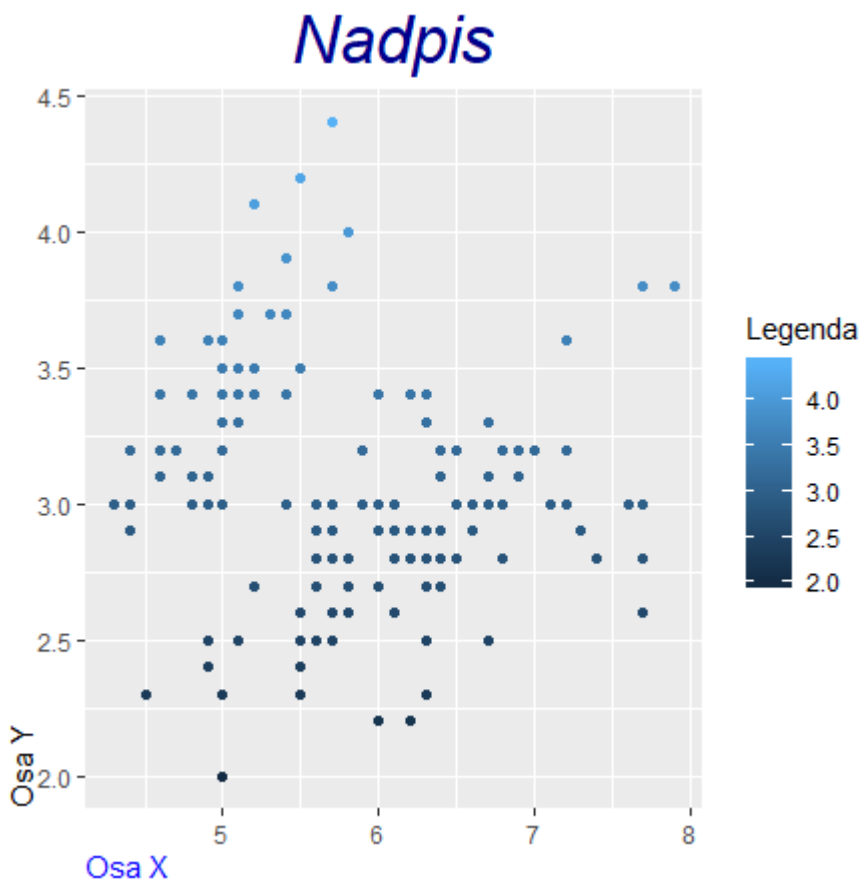
Obrázek 6.44: Funkce ggplot: labs

Zdroj: Vlastní zpracování.

Nejkomplexnější způsob editace atributů grafu lze ale provést pomocí funkce `theme`. Pomocí `theme` můžeme editovat typ, barvu, velikost popisků os. Jejich naklonění, barvu pozadí a ohraničení. Funkce `theme` nám snadno umožní si vytvořit vlastní styl grafu a pak jej můžeme lehce přenášet i na jiné typy grafů. Přes `theme` můžeme editovat velké množství parametrů. Ukažme si základní editaci barev popisků grafu.

```
> p <- ggplot(
+   data = iris,
+   mapping = aes(x = Sepal.Length, y = Sepal.Width, colour = Sepal.
    Width)
+ ) + geom_point()
```

```
> p <- p + labs(
+   title = "Nadpis",
+   x = "Osa X",
+   y = "Osa Y",
+   colour = "Legenda"
+ )
> p <- p + theme(plot.title = element_text(
+   colour = "darkblue",
+   size = 25,
+   face = "italic",
+   hjust = 0.5
+ ))
> p <- p + theme(axis.title.x = element_text(colour = "blue", hjust
    = 0))
> p <- p + theme(axis.title.y = element_text(colour = "black", hjust
    = 0))
> p
```



Obrázek 6.45: Funkce ggplot: theme

Zdroj: Vlastní zpracování.

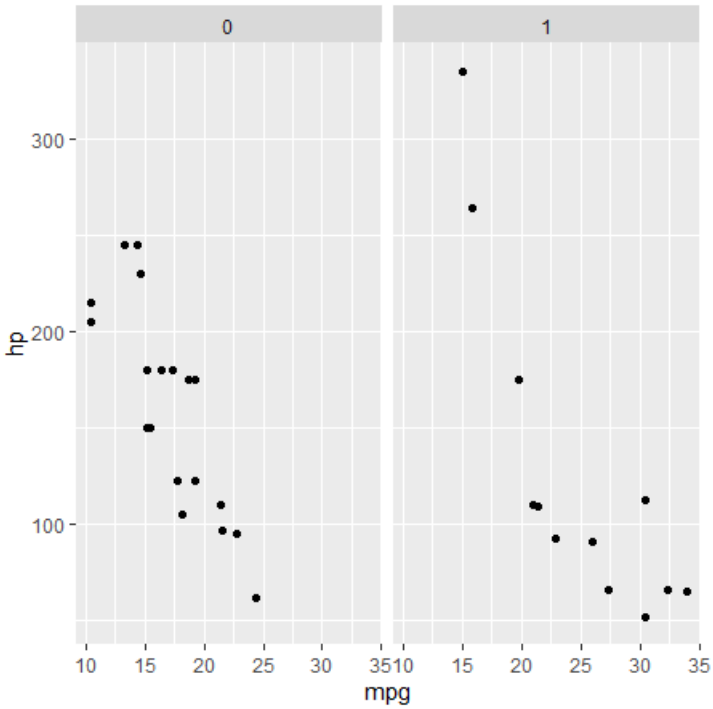
Pomocí parametrů `plot.title` a `axis.title` jsme nastavili formát pro jednotlivé popisky grafu. Pro jejich editaci je nutné využít funkcí `element_`. Každý typ parametru z funkce `theme` má přiřazen druh vstupu. U některých se jedná přímo o specifickou třídu objektu, u jiných je možné více druhů vstupů. V našem případě je pro editaci typu textu nutné použít funkce `element_text`.

6.3.5 Více grafů v jedné grafické oblasti

Knihovna ggplot disponuje podobným způsobem tvorby více grafů v jedné grafické oblasti jako base graphics. Způsob rozdělení grafů je velmi podobný „mřížce“ či „buňkám“, se kterými jsme pracovali pomocí par funkce v base graphics. Pro tuto úlohu disponuje ggplot dvěma funkcemi. Jedná se o funkci `facet_grid` a `facet_wrap`. První funkce `facet_grid` rozděluje data do 2D pole podle vzorce. Proměnná před tildou definuje počet řádek a

proměnná za tildou definuje počet sloupců, do kterých se grafy rozdělí. Zkusme si v následujícím příkladu vykreslit bodový graf pro data `mtcars` (`mpg` a `hp`) podle druhu převodovky do sloupců.

```
> p <- ggplot(data = mtcars, mapping = aes(x = mpg, y = hp))
> p + geom_point() + facet_grid(. ~ am)
```

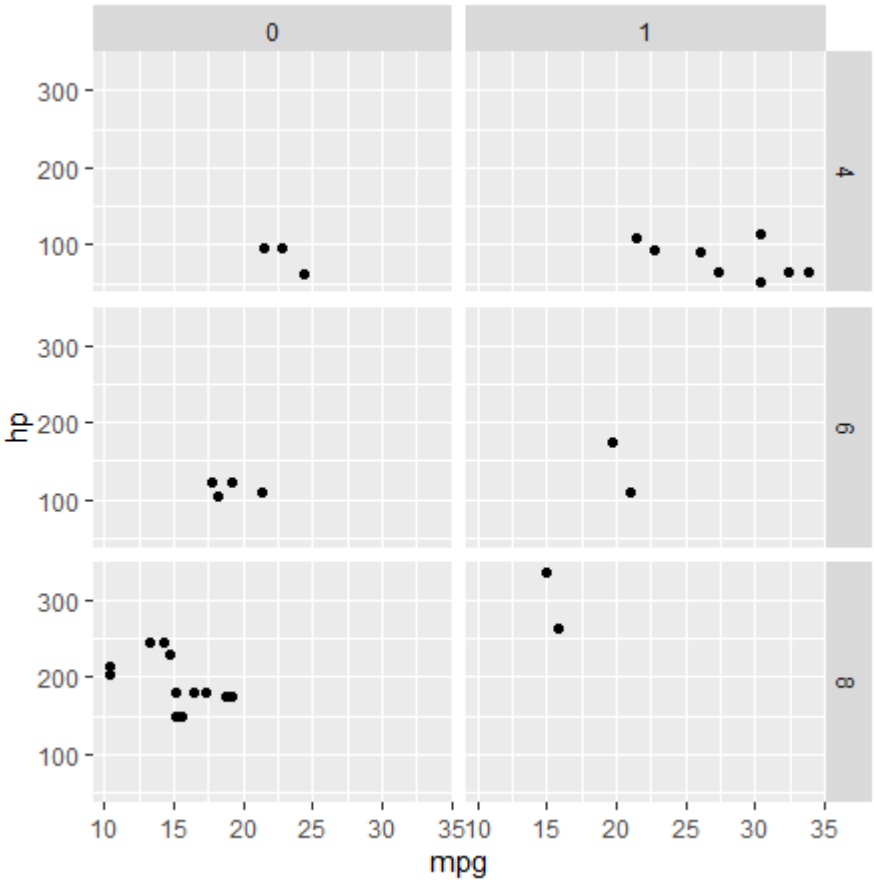


Obrázek 6.46: Funkce ggplot: facet_grid funkce I

Zdroj: Vlastní zpracování.

Alternativně můžeme data rozdělit jak pomocí proměnné `am` (druh převodovky), tak i pomocí proměnné `cyl` (počet válců motoru auta).

```
> p <- ggplot(data = mtcars, mapping = aes(x = mpg, y = hp))
> p + geom_point() + facet_grid(cyl ~ am)
```

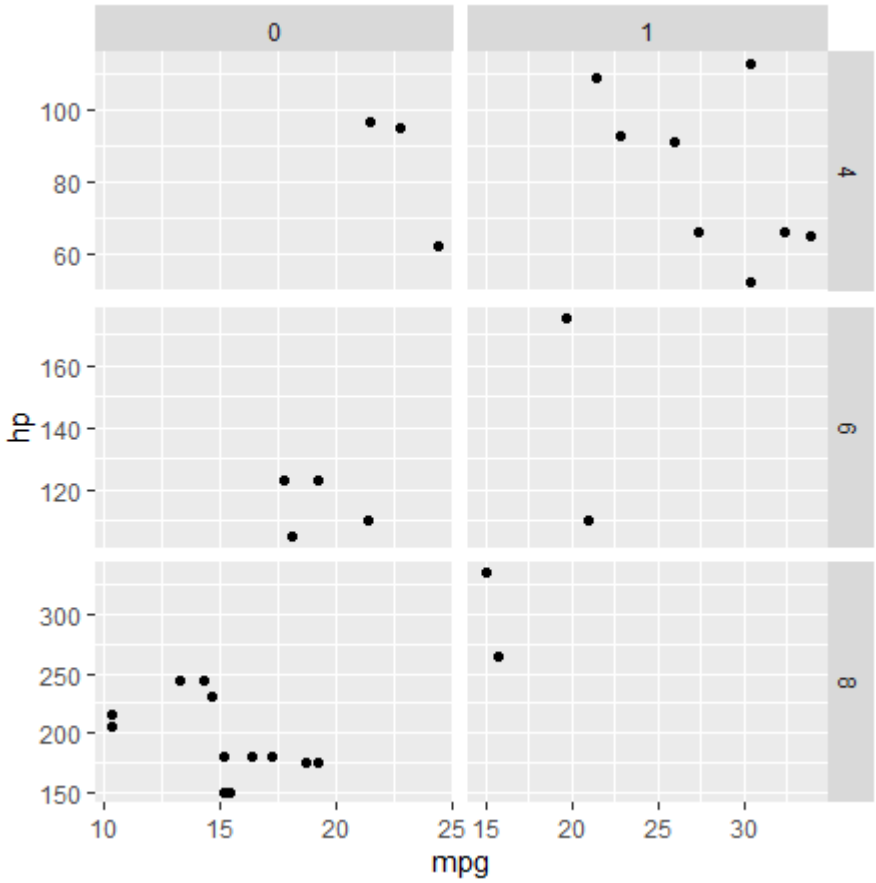


Obrázek 6.47: Funkce ggplot: facet_grid funkce II

Zdroj: Vlastní zpracování.

Jistě jste si všimli, že všechny dílčí grafy mají stejné osy. To se dá ovlivnit pomocí parametru `scales`, který má výchozí parametr „fix“. Na všech grafech jsou stejné rozsahy maxim/minim na osách. Pokud bychom do parametru vložili parametr `"free"`, tak budou všechny osy rozdílné (svázané jsou jen osy x v sloupcích a osy y v řádkách). Dalšími parametry jsou `free_x` a `free_y`, které uvolňují jen specifickou osu.

```
1 > p <- ggplot(data = mtcars, mapping = aes(x = mpg, y = hp))
2 > p + geom_point() + facet_grid(cyl ~ am, scales = "free")
```

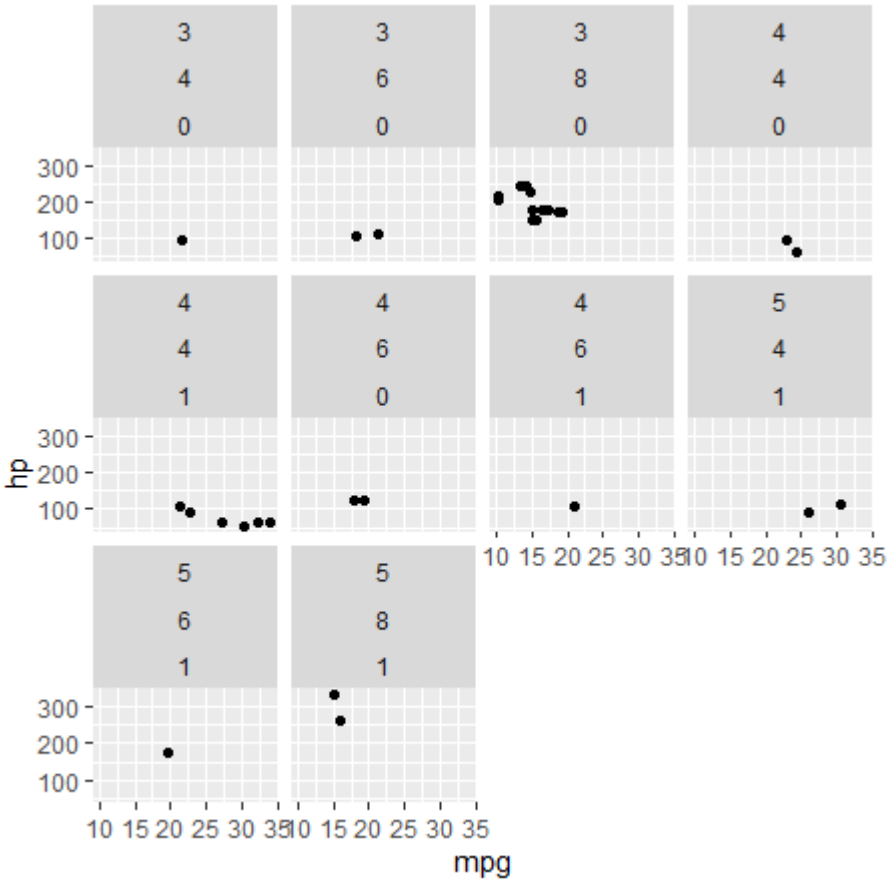


Obrázek 6.48: Funkce ggplot: facet_grid funkce, scales=free

Zdroj: Vlastní zpracování.

Funkce `facet_wrap()` negeneruje cíleně grafy do mřížky podle daných parametrů, ale do panelu grafů. Pro tento panel můžeme zadat i maximální počet sloupců/řádků (`ncol` a `nrow`). Výhodou funkce `facet_wrap()` je to, že můžeme data rozdělit i podle více než dvou proměnných.

```
> p <- ggplot(data = mtcars, mapping = aes(x = mpg, y = hp))
> p + geom_point() + facet_wrap(gear ~ cyl + am)
```



Obrázek 6.49: Funkce ggplot: facet_wrap funkce

Zdroj: Vlastní zpracování.

6.3.6 Ukládání grafu

Ukládání grafů je v ggplot knihovně velmi podobné jako v případě base graphics grafů. Oproti této knihovně je ale nutné tzv. provést „tisk“ daného grafu pomocí funkce `print()`. Ukážeme si tento postup na grafu datasetu `iris`.

```
> p <-ggplot(
+   data = iris,
+   mapping = aes(x = Sepal.Length, y = Sepal.Width, colour = Sepal.
    Width)
+ ) +
+   geom_point()
>
> png(filename = "GGplot.png",
```

```
+      height = 450,
+      width = 450)
> print(p)
> dev.off()
```

Alternativou k tomuto postupu je již zobrazený graf přímo uložit pomocí funkce `ggsave()`. Hlavní parametr funkce `ggsave()` je jméno grafu. Funkce si ze jména grafu přebere sama druh souboru, do kterého ho má uložit, a vybere tak správnou proceduru.

```
> ggplot(data = iris,
+   mapping = aes(x = Sepal.Length, y = Sepal.Width, colour = Sepal.
    Width)) + geom_point()

> ggsave("ggplot.pdf")
Saving 5.53 x 7.77 in image
```

Tato funkce ale funguje jen pro zobrazené grafy. Je tedy nutné si v RStudiu/R evironmentu graf nejprve zobrazit do nového grafického okna. Ve funkci `ggsave()` můžeme specifikovat množství atributů o daném grafu, který ukládáme, například jeho výšku, šířku, poměr stran, rozlišení či změnit okno, ze kterého se má graf uložit, pokud máme otevřeno více grafických oken.

Kapitola 7

Funkce merge a knihovna dplyr

V rámci této kapitoly si představíme funkci `merge()` a knihovnu `dplyr`. Funkce `merge()` slouží ke spojování datových souborů. Knihovna `dplyr` je pak komplexní knihovnou funkcí, které slouží k pokročilé práci s datasety (více viz. Spector 2008).

7.1 Funkce merge

Často nastává situace, kdy potřebujeme dva datasety spojit podle určitého klíče. Ve výchozí instalaci softwaru **R** je k dispozici funkce `merge()`. Tato funkce má více parametrů, které mají nastavené důležité výchozí parametry. Postupně si zkusíme tuto funkci projít. Nejprve si vytvoříme dva testovací datasety ve formátu `data.frame`. První dataset obsahuje seznam pěti jmen žáků a jejich počet bodů ze zeměpisu.

```
> data.A <- data.frame(
+             jmeno = c("Karel", "Pavel", "Jan", "Aneta", "Jakub"),
+             zemepis = c(20, 10, 12, 43, 20),
+             stringsAsFactors = FALSE)
> print(data.A)
  jmeno zemepis
1 Karel       20
2 Pavel       10
3   Jan       12
4 Aneta       43
5 Jakub       20
```

Pomocí argumentu `stringsAsFactors = FALSE` si zajistíme, že jména nebudou interpretována jako faktory. V případě, že by tomu tak bylo, tak bychom se mohli potýkat s problémy, protože faktor má charakter jen jako vizuální reprezentaci pořadové hodnoty faktoru, ale všechny výpočty jsou prováděné s touto pořadovou hodnotou. Druhý dataset

obsahuje analogický seznam žáků a jejich počet bodů z matematiky.

```
> data.B <- data.frame(
+             jmeno = c("Pavel", "Karel", "Jakub" , "Aneta", "Pepa"),
+             matematika = c(42, 55, 84, 87, 45),
+             stringsAsFactors = FALSE)
> print(data.B)
  jmeno matematika
1 Pavel          42
2 Karel          55
3 Jakub          84
4 Aneta          87
5  Pepa          45
```

V případě, kdy na oba dva datasety aplikujeme prostou funkci `merge()`, tak sice dojde ke spojení datasetů podle předem zvoleného klíče, ale(!) o část dat přijdeme. Je to tím, že ve výchozím nastavení se na sebe napojí data, která jsou průsečíkem klíčů v obou datasetech. Za klíč zde považujeme proměnnou jméno. Jak na to ale funkce přišla, že se jedná o klíč? Je to z toho důvodu, že se jedná o jedinou proměnnou, která má stejné jméno napříč oběma soubory a obsahuje charaktery.

```
> merge(x = data.A, y = data.B)
  jmeno zemepis  matematika
1 Aneta       43          87
2 Jakub       20          84
3 Karel       20          55
4 Pavel       10          42
```

Pokud bychom nastavili parametr all na hodnotu `TRUE`, získali bychom všechna data z obou souborů.

```
> merge(x = data.A, y = data.B, all = TRUE)
  jmeno zemepis  matematika
1 Aneta       43          87
2 Jakub       20          84
3   Jan       12          NA
4 Karel       20          55
5 Pavel       10          42
6  Pepa       NA          45
```

V případě chybějících hodnot proměnných funkce dosadí do výsledného datasetu `NA` hodnoty. V případě datasetu, kde jsou rozdílně pojmenované proměnné nesoucí klíč, tak přiřadíme jména parametrů `by.x` a `by.y`.

```
> merge(x = data.A, y = data.B, all = TRUE, by.x = "jmeno", by.y = "
    jmeno)
  jmeno zemepis  matematika
1 Aneta       43          87
2 Jakub       20          84
3   Jan       12          NA
4 Karel       20          55
5 Pavel       10          42
6  Pepa       NA          45
```

V situaci, kdy se sice klíč jmenuje stejně, ale v datasetech existuje více stejně pojmenovaných proměnných, využijeme parametru `by`. V takové situaci by se u shodných jmen proměnných objevily přípony. Jméno přípony můžeme specifikovat v parametru `suffixes`. Pro ilustraci tohoto přístupu vydefinujeme nový dataset `data.C`, který bude obsahovat stejně jako `data.B` body z matematiky.

```
> data.C <- data.frame(
+   jmeno = c("Petr", "Samuel", "Jakub", "Aneta", "Pepa"),
+   matematika = c(47, 89, 81, 87, 45),
+   stringsAsFactors = FALSE
+  .... [TRUNCATED]

> merge(x = data.B, y = data.C,
+   all = TRUE, by = "jmeno",
+   suffixes = c(".prvnni", ".druhe")
+ )
   jmeno  matematika.prvnni  matematika.druhe
1  Aneta                 87                87
2  Jakub                 84                81
3  Karel                 55                NA
4  Pavel                 42                NA
5   Pepa                 45                45
6   Petr                 NA                47
7 Samuel                 NA                89
```

Parametry `all.x` a `all.y` nám umožňují specifikovat, zda mají být z daného datasetu (přiřazeného do proměnné `x` anebo `y`) přebrané všechny hodnoty, anebo jen ty hodnoty, které jsou průsečíkem na druhý dataset. Pokud nastavíme parametr `all` na hodnotu `TRUE`, tak automaticky oba parametry `all.x` a `all.y` přeberou hodnotu `TRUE`. Alternativně můžeme nastavit způsob průniku dat pro dílčí datasety zvlášť. Ukažme si postupně všechny varianty.

Tímto způsobem získáme jen data, která jsou podle klíče v obou datasetech.

```
> merge(x = data.A, y = data.B, all.x = FALSE, all.y = FALSE)
  jmeno zemepis  matematika
1 Aneta       43          87
2 Jakub       20          84
3 Karel       20          55
4 Pavel       10          42
```

V případě, kdybychom chtěli získat všechna data z datasetu `data.A` a zároveň k nim napárovat data z datasetu `data.B` (ale jen ta, co jsou obsažená v datasetu A), tak to provedeme pomocí následujícího příkazu.

```
> merge(x = data.A, y = data.B, all.x = TRUE, all.y = FALSE)
  jmeno zemepis  matematika
1 Aneta       43          87
2 Jakub       20          84
3   Jan       12          NA
4 Karel       20          55
5 Pavel       10          42
```

Opak — tedy pokud chceme získat všechna data z datasetu `data.B` a k nim přidat jen data, co mají stejný klíč z datasetu `data.A`, získáme je podle následujícího příkazu.

```
> merge(x = data.A, y = data.B, all.x = FALSE, all.y = TRUE)
  jmeno zemepis  matematika
1 Aneta       43          87
2 Jakub       20          84
3 Karel       20          55
4 Pavel       10          42
5  Pepa       NA          45
```

Analogii k parametru `all=TRUE` získáme nastavením obou hodnot na `TRUE`. Získáme tím dataset, kde budou obsazené všechny záznamy.

```
> merge(x = data.A, y = data.B, all.x = TRUE, all.y = TRUE)
  jmeno zemepis  matematika
1 Aneta       43          87
2 Jakub       20          84
3   Jan       12          NA
4 Karel       20          55
5 Pavel       10          42
6  Pepa       NA          45
```

7.2 Knihovna dplyr

Balíček `dplyr` obsahuje širokou škálu na práci s datasety. Použití této knihovny je vhodné zejména v situaci, kdy máme velké datové soubory. Knihovna pracuje jak s objektem `data.frame`, tak ale zároveň navrací objekt typu `tibble`. Objekt typu `tibble` je velmi podobný objektu typu `data.frame`. Prvním rozdílem je to, že pokud zavoláme jméno daného objektu v příkazové řádce, tak se nám automaticky vypíše jen hlavička souboru. Tedy provede se obdoba funkce `head()`. Dalším rozdílem je to, že v objektu typu `data.frame` jsou všechny proměnné typu `character` ve výchozím nastavení převáděné na `factor`, v objektu typu `tibble` zůstávají proměnné characterem.

7.2.1 Pipeline operátor

Knihovna `dplyr` zavádí nový operátor jménem pipeline (`%>%`). Ukážeme si nejprve na následujícím příkladu jeho použití na funkci `mean()`.

```
> znamky<-c(10, 20, 30)
> mean(znamky)
[1] 20
> znamky %>% mean
[1] 20
```

Standardní použití funkce `mean` spočívá v tom, že do kulaté závorky za jménem funkce vložíme do parametrů příslušné hodnoty. V tomto případě je ale objekt vlevo před operátorem %>% vložen jako hlavní (první) parametr do funkce napravo od operátoru. Nenechte se zmást tím, že byste si mysleli, že se jedná o porovnávání – vůbec nic zde neporovnáváme. Ukážeme si ještě jeden příklad s použitím funkce `colMeans()` na objektu `mtcars`.

```
> mtcars %>% colMeans
       mpg        cyl       disp         hp
 20.090625   6.187500 230.721875 146.687500
      drat         wt       qsec         vs
  3.596563   3.217250  17.848750   0.437500
        am       gear       carb
  0.406250   3.687500   2.812500
```

V situaci, kdy potřebujeme výsledek jedné funkce vložit jako vstup do jiné, nachází tento operátor použití. Takovouto situaci zpřehlední a čtenář příslušného kódu ihned uvidí posloupnost operací. Ukážeme si to na následujícím příkladu výpočtu průměrů.

```
> mtcars %>% colMeans %>% mean
[1] 39.60853
```

```
> mean(colMeans(mtcars))
[1] 39.60853
```

7.2.2 Filtrování pozorování

Pro filtrování pozorování máme vícero možností. Nejprve můžeme filtrovat pomocí funkce `filter()`. Funkce `filter()` přebírá jako parametr logickou podmínku, kterou musí dané řádky splnit. Ukažme si to na datasetu `mtcars`.

```
> mtcars %>% filter(cyl == 6)
   mpg cyl  disp  hp drat    wt  qsec vs am
1 21.0   6 160.0 110 3.90 2.620 16.46  0  1
2 21.0   6 160.0 110 3.90 2.875 17.02  0  1
3 21.4   6 258.0 110 3.08 3.215 19.44  1  0
4 18.1   6 225.0 105 2.76 3.460 20.22  1  0
5 19.2   6 167.6 123 3.92 3.440 18.30  1  0
6 17.8   6 167.6 123 3.92 3.440 18.90  1  0
7 19.7   6 145.0 175 3.62 2.770 15.50  0  1
  gear carb
1    4    4
2    4    4
3    3    1
4    3    1
5    4    4
6    4    4
7    5    6
```

Tímto způsobem jsme si vybrali řádky datasetu, které splňují podmínku, že mají 6 cylindrů. Podmínky můžeme i kombinovat pomocí standardních logických operátorů.

```
> mtcars %>% filter(cyl == 6 & disp > 170)
   mpg cyl disp  hp drat    wt  qsec vs am
1 21.4   6  258 110 3.08 3.215 19.44  1  0
2 18.1   6  225 105 2.76 3.460 20.22  1  0
  gear carb
1    3    1
2    3    1
```

Funkce distinct nám z datasetu odstraní duplicitní záznamy. Ukažme si to na následujícím příkladu.

```
> data.X <- data.frame(x1 = c(1, 1, 2, 1, 2),
+                      x2 = c(1, 1, 2, 3, 2))
> print(data.X)
  x1 x2
```

```
1  1  1
2  1  1
3  2  2
4  1  3
5  2  2
> data.X %>% distinct()
  x1 x2
1  1  1
2  2  2
3  1  3
```

Náhodný výběr pozorování lze provést pomocí dvou funkcí. První funkce je `sample_frac()`, která definuje podíl (v procentním čísle) dat, který se má vybrat z datasetu, druhá funkce jménem `sample_n()` definuje počet řádek, který se má vybrat z datasetu.

```
> iris %>% sample_n(2)
  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1          7.2         3.0          5.8         1.6 virginica
2          5.4         3.7          1.5         0.2    setosa
> iris %>% sample_frac(0.025)
  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1          5.1         3.5          1.4         0.2    setosa
2          6.9         3.1          5.1         2.3 virginica
3          5.2         3.4          1.4         0.2    setosa
4          5.1         3.3          1.7         0.5    setosa
```

Filtrování funkcí můžeme provést i pomocí kombinace s matematickými funkcemi.

```
> mtcars %>% filter(hp > mean(hp) & wt < mean(wt))
   mpg cyl disp  hp drat   wt qsec vs am gear carb
1 15.8   8  351 264 4.22 3.17 14.5  0  1    5    4
2 19.7   6  145 175 3.62 2.77 15.5  0  1    5    6
```

7.2.3 Setřídění datasetu

Dataset můžeme snadno i setřídit. Stačí použít funkci `arrange()`. Funkce `arrange()` ve výchozím nastavení třídí data vzestupně.

```
> setridene = mtcars %>% arrange(hp)
> head(setridene)
   mpg cyl  disp hp drat    wt  qsec vs am gear carb
1 30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2
2 24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
3 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1
4 32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1
```

```
5 27.3   4  79.0 66 4.08 1.935 18.90  1  1    4    1
6 26.0   4 120.3 91 4.43 2.140 16.70  0  1    5    2
```

Pro sestupní třídění je nutné proměnnou uvnitř funkce `arrange()` vložit do funkce `desc` – descending.

```
> setridene = mtcars %>% arrange(desc(hp))
> head(setridene)
   mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 15.0   8  301 335 3.54 3.570 14.60  0  1    5    8
2 15.8   8  351 264 4.22 3.170 14.50  0  1    5    4
3 14.3   8  360 245 3.21 3.570 15.84  0  0    3    4
4 13.3   8  350 245 3.73 3.840 15.41  0  0    3    4
5 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
6 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
```

Data můžeme třídit i podle více proměnných, stačí je vložit za sebou do funkce `arrange()`.

```
> setridene = mtcars %>% arrange(gear,carb)
> head(setridene)
   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
1 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
2 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
3 21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
4 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
5 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
6 15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
```

Pořadí proměnných ve funkci `arrange()` ovlivňuje i prioritu, podle které jsou výsledná data setříděná. Prvně jsou data setříděna podle skupin proměnné `gear` a následně je v těchto skupinách je pořadí řízené pomocí proměnné `carb`.

7.2.4 Filtrování proměnných

Funkce `select()` nám umožní filtrovat dataset z pohledu proměnných. Z datasetu `mtcars` provedeme selekci výběr `vs` a `am`, dále si zobrazíme hlavičku souboru.

```
> selekt = mtcars %>% select(vs, am)
> head(selekt)
                  vs am
Mazda RX4          0  1
Mazda RX4 Wag      0  1
Datsun 710         1  1
Hornet 4 Drive     1  0
Hornet Sportabout  0  0
Valiant            1  0
```

Do funkce `select()` stačí vypsat jména proměnných, které chceme z datasetu vybrat. Pokud bychom chtěli vybrat všechna data, tak jako parametr vložíme funkci `everything()`.

```
> selekt = mtcars %>% select(everything())
> head(selekt)
                   mpg cyl disp  hp drat
Mazda RX4         21.0   6  160 110 3.90
Mazda RX4 Wag     21.0   6  160 110 3.90
Datsun 710        22.8   4  108  93 3.85
Hornet 4 Drive    21.4   6  258 110 3.08
Hornet Sportabout 18.7   8  360 175 3.15
Valiant           18.1   6  225 105 2.76
                     wt  qsec vs am gear carb
Mazda RX4         2.620 16.46  0  1    4    4
Mazda RX4 Wag     2.875 17.02  0  1    4    4
Datsun 710        2.320 18.61  1  1    4    1
Hornet 4 Drive    3.215 19.44  1  0    3    1
Hornet Sportabout 3.440 17.02  0  0    3    2
Valiant           3.460 20.22  1  0    3    1
```

Funkce `everything()` má zejména použití v dalších funkcích, které plní určitou úlohu, ale zároveň provádí i výběr proměnných. Popřípadě v situaci, kdy chceme přetřídit proměnné datasetu. Funkce `everything()` pak sama dovybere zbývající proměnné.

```
> mtcars %>% select(am, cyl, gear, everything()) %>% head
                  am cyl gear  mpg disp  hp drat    wt
Mazda RX4          1   6    4 21.0  160 110 3.90 2.620
Mazda RX4 Wag      1   6    4 21.0  160 110 3.90 2.875
Datsun 710         1   4    4 22.8  108  93 3.85 2.320
Hornet 4 Drive     0   6    3 21.4  258 110 3.08 3.215
Hornet Sportabout  0   8    3 18.7  360 175 3.15 3.440
Valiant            0   6    3 18.1  225 105 2.76 3.460
                   qsec vs carb
Mazda RX4         16.46  0    4
Mazda RX4 Wag     17.02  0    4
Datsun 710        18.61  1    1
Hornet 4 Drive    19.44  1    1
Hornet Sportabout 17.02  0    2
Valiant           20.22  1    1
```

Pokud bychom chtěli vybrat proměnné, které obsahují ve svém jménu specifický řetězec, tak použijeme funkci `contains()`.

```
> selekt = mtcars %>% select(contains("a"))
> head(selekt)
                  drat am gear carb
Mazda RX4         3.90  1    4    4
```

```
Mazda RX4 Wag      3.90  1    4    4
Datsun 710         3.85  1    4    1
Hornet 4 Drive     3.08  0    3    1
Hornet Sportabout  3.15  0    3    2
Valiant            2.76  0    3    1
```

Alternativně můžeme vybírat i podle konců a začátků jmen. Pomocí funkcí – `starts_with()` a `ends_with()`.

```
> selekt = mtcars %>% select(ends_with("p"))
> head(selekt)
                  disp  hp
Mazda RX4          160 110
Mazda RX4 Wag      160 110
Datsun 710         108  93
Hornet 4 Drive     258 110
Hornet Sportabout  360 175
Valiant            225 105
```

7.2.5 Tvorba nové proměnné

Novou proměnnou vytvoříme velmi snadno. Zadefinujeme ji uvnitř funkce `mutate()`. V následujícím příkladu normujeme pomocí z-skóre proměnnou `mpg`, kterou si uložíme nově jako „Z_mpg“.

```
> data.x <- mtcars
> data.x <- data.x %>% mutate(Z_mpg = (mpg - mean(mpg))/sd(mpg))
> head(data.x)
   mpg cyl disp  hp drat    wt  qsec vs am gear carb       Z_mpg
1 21.0   6  160 110 3.90 2.620 16.46  0  1    4    4  0.1508848
2 21.0   6  160 110 3.90 2.875 17.02  0  1    4    4  0.1508848
3 22.8   4  108  93 3.85 2.320 18.61  1  1    4    1  0.4495434
4 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1  0.2172534
5 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2 -0.2307345
6 18.1   6  225 105 2.76 3.460 20.22  1  0    3    1 -0.3302874
```

Pomocí funkce `mutate_each()` můžeme přepočítat všechny proměnné. Tato funkce přebírá jako parametr jméno funkce, která se má na dané sloupce použít, popřípadě deklaraci takové funkce. V našem případě od proměnné odečteme průměr a zaokrouhlíme hodnoty na celá čísla.

```
> data.x <- mtcars
> data.x <- data.x %>% mutate_each(function(x) round(x - mean(x)))
> head(data.x)
  mpg cyl disp  hp drat wt qsec vs am gear carb
```

```
1   1    0   -71  -37     0  -1    -1   0   1     0     1
2   1    0   -71  -37     0   0    -1   0   1     0     1
3   3   -2  -123  -54     0  -1     1   1   1     0    -2
4   1    0    27  -37    -1   0     2   1   0    -1    -2
5  -1    2   129   28     0   0    -1   0   0    -1    -1
6  -2    0    -6  -42    -1   0     2   1   0    -1    -2
```

Funkce `transmute()` pak provádí změnu jen jedné konkrétní proměnné. Díky ní můžeme transformovat/přepočítat jednu či více zvolených proměnných. V následujícím příkladu zaokrouhlíme proměnnou `mpg` na celá čísla. Důležité je si uvědomit, že funkce `transmute()` funguje zároveň jako funkce `selekt()` a vybírá sloupce datasetu.

```
> data.x <- mtcars
> data.x <- data.x %>% transmute(mpg = round(mpg))
> head(data.x)
  mpg
1  21
2  21
3  23
4  21
5  19
6  18
```

7.2.6 Statistické shrnutí dat

Knihovna `dplyr` disponuje i možností tvorby základních statistik pro daný dataset. Jednu proměnnou z datasetu můžeme shrnout pomocí funkce `summarise()`. Vstupem do této funkce je deklarace proměnné provádějící statistické shrnutí. Dostupné funkce pro statistické shrnutí jsou: `first()` – první hodnota v datasetu, `last()` – poslední hodnota, `min()` – minimální hodnota, `max()` – maximální hodnota, `nth()` – n-tá hodnota datasetu (např. pořadové číslo řádky), `mean()` – průměrná hodnota, `n()` – počet hodnot z datasetu, `median()` – mediánová hodnota, `var()` – rozptyl, `sd()` – směrodatná odchylka, `IQR()` – mezikvartilové rozpětí.

```
> data.x <- mtcars
> data.x %>% summarise(wt_mean = mean(wt))
  wt_mean
1 3.21725
```

Pomocí funkce `summarise_all` můžeme shrnout i celý dataset. Ukažme si alternativu funkce `colMeans()` v následujícím příkladu.

```
> data.x %>% summarise_all(mean)
       mpg     cyl      disp        hp      drat
1 20.09062 6.1875 230.7219 146.6875 3.596563
```

```
        wt      qsec      vs       am    gear    carb
1 3.21725 17.84875 0.4375 0.40625 3.6875 2.8125
```

7.2.7 Práce se skupinami dat

Princip práce se skupinami dat v knihovně dplyr spočívá v tom, že každému řádku v datasetu přiřadíme identifikaci k určité skupině. Následně všechny operace provedené na celém datasetu jsou prováděné podle těchto skupin zvlášť. Funkce, která slouží k rozdělení dat do skupin, se jmenuje `group_by()`. Zkusme si tuto úlohu provést pro výpočet průměrů v datasetu `mtcars`, který roztřídíme do skupin podle válců.

```
> data.x %>% group_by(cyl) %>% summarise_all(mean)
# A tibble: 3 x 11
    cyl   mpg  disp    hp  drat    wt  qsec
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1     4  26.7  105.  82.6  4.07  2.29  19.1
2     6  19.7  183. 122.   3.59  3.12  18.0
3     8  15.1  353. 209.   3.23  4.00  16.8
# ... with 4 more variables: vs <dbl>,
#   am <dbl>, gear <dbl>, carb <dbl>
```

Tato úloha je velmi podobná funkci `aggregate()` s parametrem `by` nastaveným na počet válců.

7.2.8 Spojování datasetů pomocí balíčku dplyr

Knihovna `dplyr` disponuje vlastními funkcemi na spojení datasetů. Definujme si stejná data jako v případě funkce `merge()`.

```
> data.A <- data.frame(
+   jmeno = c("Karel", "Pavel", "Jan", "Aneta", "Jakub"),
+   zemepis = c(20, 10, 12, 43, 20),
+   stringsAsFactors = FALSE
+ )
> data.B <- data.frame(
+   jmeno = c("Pavel", "Karel", "Jakub", "Aneta", "Pepa"),
+   matematika = c(42, 55, 84, 87, 45),
+   stringsAsFactors = FALSE
+ )
> print(data.A)
  jmeno zemepis
1 Karel      20
2 Pavel      10
3   Jan      12
```

```
4 Aneta       43
5 Jakub       20
> print(data.B)
  jmeno matematika
1 Pavel          42
2 Karel          55
3 Jakub          84
4 Aneta          87
5  Pepa          45
```

Funkce `left_join()` provede spojení všech dat z datasetu `data.A` a přiřadí k nim data z datasetu `data.B`.

```
> data.A %>% left_join(data.B, by = "jmeno")
  jmeno zemepis matematika
1 Karel       20          55
2 Pavel       10          42
3   Jan       12          NA
4 Aneta       43          87
5 Jakub       20          84
```

Funkce `right_join()` naopak provede spojení všech dat z datasetu data.B a k nim přiřadí data z datasetu `data.A`.

```
> data.A %>% right_join(data.B, by = "jmeno")
  jmeno zemepis matematika
1 Pavel       10          42
2 Karel       20          55
3 Jakub       20          84
4 Aneta       43          87
5  Pepa       NA          45
```

Funkce `inner_join()` spojí data podle klíče, který se vyskytuje v obou datasetech.

```
> data.A %>% inner_join(data.B, by = "jmeno")
  jmeno zemepis matematika
1 Karel       20          55
2 Pavel       10          42
3 Aneta       43          87
4 Jakub       20          84
```

Pokud chceme spojit data z obou datasetů a zachovat všechny hodnoty (obdoba parametru `all=TRUE` u funkce `merge`), použijeme funkci `full_join()`.

```
> data.A %>% full_join(data.B, by = "jmeno")
  jmeno zemepis matematika
1 Karel       20          55
```

```
2 Pavel       10          42
3   Jan       12          NA
4 Aneta       43          87
5 Jakub       20          84
6  Pepa       NA          45
```

Kapitola 8

Formátování kódu

Při psaní kódu je vhodné dodržet základní pravidla formátování. Existuje vícero druhů pravidel, jak formátovat kód v jazyce R. Zde si představíme formát sestavený Hadley Wickhamem (Wickham, H. 2020) v rámci knihovny tidyverse (odkaz zde). Další formáty jsou například od Jean Fan (Fan, J., 2020; odkaz zde) či Google comunity style guide (odkaz zde). Časem jistě zjistíte, že existuje poměrně mnoho formátů. Čtenáři doporučujeme vybrat si jeden formát, a pokud je to možné, tak se ho dále držet. Reformátování vašeho kódu můžete v RStudiu provést velmi snadno pomocí tlačítka na reformátování kódu. Standard zde použitý odpovídá standardu Hadley Wickhama `Code -> Reformat Code` (či `CTRL+SHIFT+A`).

Proč ale kód specificky formátovat? Formátování kódu má mnoho výhod. Za hlavní výhodu se obecně považuje snadnější srozumitelnost a čitelnost. Zde bychom rádi čtenáře varovali, aby si nemyslel, že když nějaký kód naprogramuje, tak mu bude vždy rozumět. Pokud jste nedodrželi rozumné zásady formátování, tak zjistíte, že je vám váš vlastní kód nesrozumitelný.

Pokud je to možné, tak je vhodné dodržet zásady formátování všude, kde to jde. Dále by se uživatel měl snažit psát srozumitelný kód a volit postupy, co jsou srozumitelné. Může se vám ale stát, že snaha psát srozumitelný a přehledný kód narazí na jeho výkonnost. Tedy, že jedna věc napsána přehledně nemusí být napsána výkonně z pohledu rychlosti běhu samotného kódu. Je tedy na uživateli, aby zvážil tyto všechny aspekty programování: formátovací styl, použité funkce a algoritmy a srozumitelnost.

V následujících pár řádcích se vám pokusíme nastínit základy formátování představené již zmíněným Hadleym Wickhamem.

8.1 Jména

8.1.1 Jména objektů a funkcí

V případě jmen objektů a funkcí volíme vhodné pojmenování, které vystihuje daný jev. Je správné volit krátké, ale výstižné názvy. Pro oddělování slov v názvech je vhodné použít podrtžítko (_) a vyvarovat se tzv. maďarské notace, kdy jsou slova spojována k sobě bez oddělovacího znaku a rozlišena malým/velkým písmenem. Použití tečky pro oddělení slov není vyloženě špatné, ale ani doporučované. Ukážeme si na názvu, který má být pro objekt, co ponese průměr z prvočísel.

```
# Dobre
prumer_prvocisla


# Ani dobre ani spatne:
prumer.prvocisla
prum_prvoc

# Spatne
PrumerPrvocisla
prumerPrvocisla
prumerprvocisla
prumer_vypocitany_z_mejch_prvocisel
pp
p
```

8.1.2 Jména souborů

Stejná pravidla platí i pro pojmenování souborů, s tou výjimkou, že v názvu souboru můžeme na prvním místě použít i čísla a nově i další speciální znak pomlčku (-). Vyvarujte se použití mezery a tečky v názvu souboru (netýká se přípony).

```
# Dobre
"01-uvod.R"
"02-funkce.R"
"03-analyza_dat.R"
...
"10-export_dat.R"

# Spatne
"01 uvod.R"
"2.funkce.R"
"3-analyzaDat.R"
```

V případě, kdy soubory číslujete, je vhodné dodržet délku znaků odpovídající maximální hodnotě čísla.

8.2 Obecná syntaxe

8.2.1 Psaní mezer

V případě výběrů a parametrů funkcí byste měli za čárkou psát mezeru.

```
# Dobre
X[1, 2]
X[1, 2, 45]
X[,, 3]
mean(x, na.rm = TRUE)

# Spatne
X[1,2]
X[1,]
X[ , , 3]
mean(x,na.rm = TRUE)
```

Toto pravidlo platí i pro všechny operátory. Za a po operátoru byste měli psát mezeru. Na začátku závorek a na konci se mezera nepíše a mezi jménem funkce a závorkou se mezera nepíše také!

```
# Dobre
X %% Y
x <- matrix(1:9, ncol = 3)

# Spatne
X%%Y
x<-matrix(1:9,ncol=3)
x<-matrix( 1:9, ncol = 3 )
x <- matrix (1:9, ncol = 3)
```

Výjimka je v případě následujících operátorů.

```
 :, ::, :::, $, @, [, [[ a ^.
```

```
# Dobre
1:10
fit$coef
X[, 3]
X[[1]]

```

```
# Spatne
1 : 10
fit $ coef
X [, 3]
X [[1]]
```

V případě funkcí, kdy vynecháváme nějaký argument, tak nevytváříme prázdné místo pro tento argument pomocí čárek.

```
# Dobre
seq(from = 1, to = 10, length.out = 5)

# Spatne
seq(from = 1, to = 10, , length.out = 5)
```

V případě cyklů a podmínek píšeme mezeru mezi jménem cyklu a závorkou. Tato mezera se týká i případných složených závorek.

```
# Dobre
for (i in 1:10) {
  print(i)
}

if (i > 4) {
  print(i)
}
# Spatne
for(i in 1:10) {
  print(i)
}

if (i > 4){
  print(i)
}
if(i > 4){
  print(i)
}
```

8.2.2 Logické operátory

U logických operátorů se vyvarujte použití zkrácených názvů, kterými jsou `T` a `F`, ale používejte plné termíny.

```
# Dobre
TRUE
```

```
FALSE

# Spatne
T
F
```

8.2.3 Uvozovky

V případě používání uvozovek se doporučuje vyvarovat se použití znaku ’ a místo toho používat jen znak ".

```
# Dobre
print("nas text")

# Spatne
print('nas text')
```

Pozor, tyto dva znaky nejsou zcela zastupitelné, proto existují situace, kdy musíte znak ’ použít.

8.3 Bloky kódu a delší zápis kódu

Bloky kódu, které jsou dané do složených závorek, by měly být osazené tabulátorem. Toto se týká zejména cyklů, podmínek a uživatelem definovaných funkcí. Zároveň mějme na paměti doporučení o tom, že řádek kódu by neměl přesáhnout 80 znaků.

```
# Dobre
for (i in 1:10) {
  print(i)
  if (i > 5){
    print("je vetsi")
  }
}

moje_funkce <- function(X) {
  print(X)
}

moje_funkce <- function(X = "vychozi argument 1",
                        Y = "vychozi argument 2") {
    print(X)
}

```

```r
# Spatne

for (i in 1:10) {
print(i)
if (i > 5){
print("je vetsi")
}
}


moje_funkce <-  function(X="vychozi argument 1", Y="vychozi argument
     2") {
  print(X)
}

moje_funkce <-function(X = "vychozi argument 1",
    Y = "vychozi argument 2") {
    print(X)
}

moje_funkce <- function(X = "vychozi argument 1",
                        Y = "vychozi argument 2") {
print(X)
}
```

Formátování kódu je velmi důležité i pro spolupráci s ostatními programátory a dobře formátovaným kódem mnoho lidí potěšíte. Naopak špatně formátovaným kódem můžete mnoho lidí rozčílit. Časem si na formátování kódu zvyknete a bude mít pro vás stejný význam jako gramatika v českém jazyce.

Seznam obrázků

Seznam tabulek

Literatura

[1] FAN, J. *R Style Guide: Best practices for readable, sharable, and verifiable R code*[online]. JEFworks 2015-2020, 2020 [cit. 2020-02-02]. Dostupné z: `http://jef.works/R-style-guide/`

[2] GOOGLE. *Google's R Style Guide* [online]. Google Style Guides, 2020 [cit. 2020-02-05]. Dostupné z: `https://google.github.io/styleguide/Rguide.html`

[3] LONG, J. D., TEETOR, P., 2019. *R Cookbook: Proven Recipes for Data Analysis, Statistics, and Graphics*. O'Reilly Media, 2. vydání. ISBN: 1492040681.

[4] RAHLF T., 2019. *Data Visualisation with R – 111 Examples*, Cham: Springer Nature, 2 vydání. ISBN: 978-3-030-28444-2.

[5] R CORE TEAM. *A language and environment for statistical computing. R Foundation for Statistical Computing* [online]. Vienna, Austria: R Foundation, 2017, [cit. 2020-01-17]. Dostupné z: `https://www.R-project.org/`

[6] R CORE TEAM. *R: A language and environment for statistical computing. R Foundation for Statistical Computing*[online]. Vienna, Austria: R Foundation, 2020 [cit. 2020-01-20]. Dostupné z: `https://www.R-project.org/`

[7] RStudio Team. *RStudio: Integrated Development for R* [online]. Boston, MA: RStudio, Inc., 2018 [cit. 2020-01-12]. Dostupné z:`http://www.rstudio.com/`

[8] SPECTOR, P., 2008. *Data manipulation with R*. New York: Springer. Use R! ISBN 978-0-387-74730-9.

[9] VENABLES, W. N. SMITH, D. M., R Core Team. *An Introduction to R. Notes on R: A Programming Environment for Data Analysis and Graphics 2020* [online]. Cran R: The Comprehensive R Archive Network, 2020 [cit. 2020-04-14].
Dostupné z: `https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf`

[10] WICKHAM, H., 2009. *Ggplot2: elegant graphics for data analysis*. Dordrecht: Springer. Use R! (Springer). ISBN 978-0-387-98140-6.

[11] WICKHAM, H. *Advanced R*[online]. Hadley Wickham, 2020a [cit. 2020-04-16]. Dostupné z: http://adv-r.had.co.nz/

[12] WICKHAM, H.. *The tidyverse style guide*[online]. Tidiverse 2020b [cit. 2020-02-01]. Dostupné z: https://style.tidyverse.org/_main.pdf

[13] WICKHAM H., ROMAIN F., LIONEL H., MULLER, K., RStudio. *dplyr: A Grammar of Data Manipulation* [online]. Cran R: The Comprehensive R Archive Network, 2020 [cit. 2020-04-16]. Dostupné z: https://cran.r-project.org/web/packages/dplyr/index.html

Název **R snadno a rychle 2**
Vizualizace dat a programování
Autoři Ing. Karel Šafr, Ph.D.
Ing. Jakub Danko, PhD.
Vydavatel Vysoká škola ekonomická v Praze
Nakladatelství Oeconomica
Vydání první vydání v elektronické podobě
Redakční úprava Mgr. Ludmila Doudová
Grafický návrh Daniel Hamerník, DiS.
Počet stran 174
DTP Vysoká škola ekonomická v Praze
Nakladatelství Oeconomica
Doporučená cena zdarma

ISBN 978-80-245-2381-1