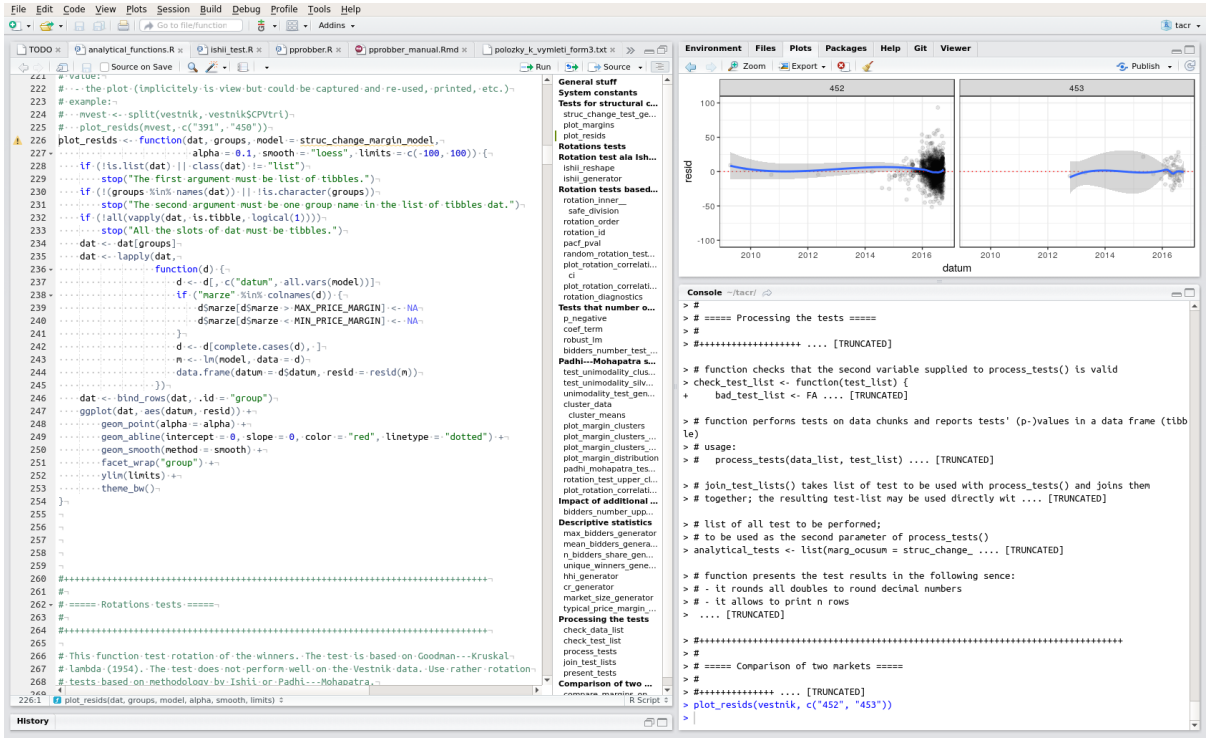


Analýza a vizualizace dat v jazyce R

Michal Kvasnička a Štěpán Mikula

2021-09-10

Obsah

Předmluva

Cílem této knihy je pomoci čtenářům naučit se provádět datovou analýzu v jazyce R. R je volně šiřitelný software pro statistickou a výpočetní analýzu, jehož popularita v současné době výrazně roste. R se používá jak k výzkumu v univerzitním prostředí, tak v komerčních firmách jako je Microsoft, Google, Facebook apod., viz 1. kapitola knihy.

Jednotlivá témata jsme do knihy vybírali tak, aby splňovala dvě kritéria: 1) aby čtenář po jejím přečtení dokázal prakticky připravit k analýze téměř jakákoli data, prozkoumat je a vizualizovat bez potřeby čtení jakékoli další knihy a aby se naučil dost na to, aby mohl dál samostatně rozvíjet své znalosti z volně dostupných zdrojů; 2) zaměřili jsme se na techniky a znalosti, které sami nejčastěji používáme ve vlastním výzkumu.

Tomu odpovídá struktura knihy: V její první části se naučíte základy jazyka R (s jasným zaměřením na datovou analýzu), a to v takové míře, abyste mohli sami "programovat s daty" a učit se dále konkrétní datové analýze. Druhá část se soustřeďuje na získání dat. Zde se naučíte získat data z celé řady zdrojů počínaje textovými soubory CSV přes data z databází až po data získaná strojově z webových stránek. Ve třetí části se naučíte transformovat data pomocí moderních nástrojů ze skupiny **tidyverse**, zejména převádět data do analyticky přívětivé podoby, filtrovat a sumarizovat dat a spojovat data z různých zdrojů. Čtvrtá část se zaměřuje na vizualizaci dat. V této části se naučíte prezentovat svá mnohorozměrná data přehledným a estetickým způsobem v publikační kvalitě. Následující část se zabývá základy statistické a ekonometrické analýzy. V poslední části se naučíte strojově generovat výzkumné zprávy a prezentace a automaticky aktualizovat tyto dokumenty v případě, že se data změní.

V celém textu klademe důraz na praktické zvládnutí probírané látky. Každá kapitola proto obsahuje velké množství příkladů. Většina pokročilejších kapitol zahrnuje i větší případové studie. Po přečtení této knihy byste tedy měli být schopní sami provádět datovou analýzu a samostatně se učit další věci z dokumentace příslušných balíků.

Text knihy není zcela lineární. Někdy v příkladech používáme i funkce a koncepty, kterou jsou důkladněji vysvětlené až v pozdějších kapitolách. Tento přístup jsme zvolili záměrně ze stejného důvodu, z jakého se tento přístup používá v moderních učebnicích jazyků. Tento přístup jednak umožňuje už relativně brzy ukázat příklady skutečného využití probírané látky, jednak usnadňuje pochopení pokročilejších konceptů, protože budete mít správný pocit, že "už jste to viděli". Předpokládáme také, že budete kromě této knihy číst i dokumentaci jednotlivých funkcí. To nejen umožní ušetřit v knize místo vynecháním některých méně podstatných detailů, ale zejména vás bude motivovat naučit se používat dokumentaci. Při vlastní praktické práci je schopnost efektivně používat dokumentaci neocenitelná.

1 Ochutnávka místo úvodu

Tato kniha vysvětluje, jak analyzovat a vizualizovat data ve výpočetním prostředí R. R je programovací jazyk a výpočetní prostředí (nejen) pro statistické výpočty a grafiku. Umožňuje dva režimy práce. Prvním je interaktivní režim, kdy zapíšete zvolené výrazy do konzole a R je okamžitě vyhodnotí a na obrazovku vypíše výsledky a vykreslí obrázky. Druhým režimem je psaní skriptů. Skripty jsou jednoduché programy, které je možné spouštět opakovaně na původních nebo i modifikovaných datech.

R je známé svou obecností (jde v něm řešit téměř jakýkoli problém), množstvím funkcí (R pokrývá téměř každé odvětví statistiky, ekonometrie a *data science* vůbec) a schopností vytvářet prvotřídní grafy v publikační kvalitě. Cenou za tuto obecnost a šíři je však rozsah a jistá složitost tohoto jazyka. O R se často říká, že má "steep learning curve". V této knize začínáme jazyk R vysvětlovat "od podlahy". To znamená, že několik prvních kapitol strávíme poměrně nudnými a složitými, ale zároveň velmi důležitými základy, a k zábavnějším věcem se dostaneme až po zvládnutí těchto základů. Nechceme však, aby vás tyto základy odradily. Proto jsme na samý začátek knihy zařadili tuto kapitolu.

Tato kapitola má dva cíle. Prvním je ukázat vám, že přes všechnu svoji složitost je R správná volba pro každého, kdo to myslí s analýzou dat vážně. Druhým cílem je ukázat vám pár příkladů toho, co budete po přečtení této knihy umět, abyste se měli na co těšit. Stejně jako celá kniha, i tato kapitola je vytvořená přímo v jazyce R, přesněji v jeho nadstavbě R Markdown, se kterým se seznámíte v kapitole **??**.

1.1 Proč právě R

Existuje mnoho dobrých důvodů, proč k analýze dat používat R. To ukazuje i fakt, že R je nejpoužívanějším nástrojem v *data science*, jak ukazuje např. studie, kterou v roce 2015 provedl *Rexer Analytics* (studie je dostupná na https://goo.gl/UWcMCF). Téměř 80 % respondentů této studie uvedlo, že ve své analytické práci používají R, z toho téměř polovina jako hlavní analytický nástroj, viz obrázek 1.1. Studie také ukazuje, že používání R je na vzestupu, viz obrázek 1.2. R používají jak výzkumníci na univerzitách, tak i ve velkých komerčních firmách, mimo jiné v Microsoftu, Facebooku, Googlu, Twitteru, Fordu, Uberu, John Deere, Firefoxu, The New York Times, The Human Rights Data Analysis Group a dalších firmách (podrobnosti, viz např. https://goo.gl/oE4hjo a https://goo.gl/4d2ez2; více se o rozšíření R můžete dočíst také na https://goo.gl/zKpMco).

Velké rozšíření užívání R mezi profesionály znamená, že R má velkou komunitu často velmi pokročilých uživatelů. Díky tomu je dostupné velké množství dobrých materiálů, ať už se jedná o knihy, blogy, návody na webu, video tutoriály nebo kurzy na Courseře. Zároveň to znamená, že vždy najdete někoho, kdo je ochotný vám poradit.

Dalším důsledkem existence velké a rostoucí komunity uživatelů je i to, že R obsahuje velké množství metod pro řešení téměř všech problémů v různých oblastech statistiky, ekonometrie a data science. Většina těchto metod je distribuována formou balíků, které jsou uloženy na jednom z centrálních repositářů. Růst počtu balíků na hlavním repositáři balíků pro R ukazuje obrázek 1.3. Všechny toto nástroje jsou přitom přítomné v jednom softwaru, takže už nemusíte používat jeden software na přípravu dat, jiný na jejich vizualizaci, jiný na ekonometrii, další na *machine learning* atd.

R navíc obsahuje velmi mocný, ale relativně jednoduchý programovací jazyk, takže svoji analýzu můžete nejen provést interaktivně, ale i uložit si ji do skriptu a svůj výpočet kdykoli zopakovat stisknutím jednoho tlačítka, a to jak na původních, tak na nových nebo aktualizovaných datech. Díky tomu po vás může kdokoli vaši analýzu zopakovat, což je jedna ze zásad *reproducible research*. Přitom můžete smíchat výpočet a text a vytvořit "živé dokumenty" (jak ostatně vznikla tato kniha). Svoji práci můžete dále zautomatizovat tak,

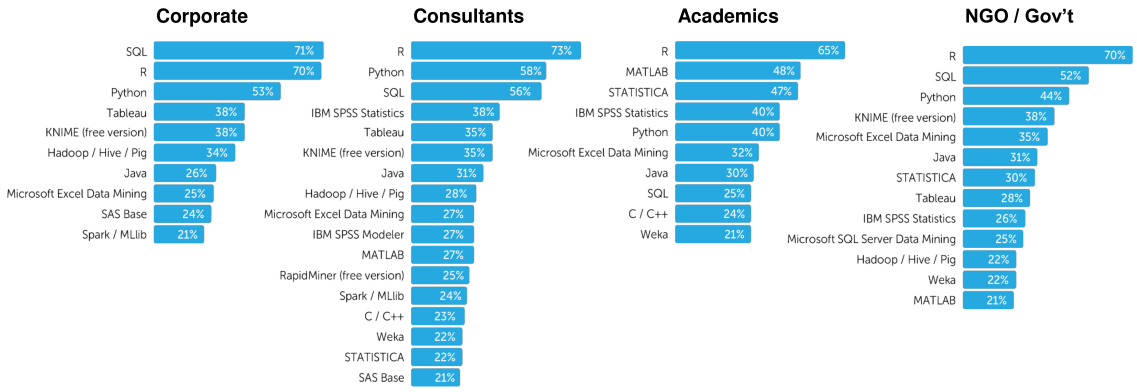


Figure 1.1: Analytické nástroje nejvíce používané respondnety Rexer Analytics Survey v roce 2017; každý respondent mohl zaškrtnout více nástrojů Zdroj: Rexer Analytics: A Decade of Surveying Analytic Professionals: 2017 Survey Highlights.

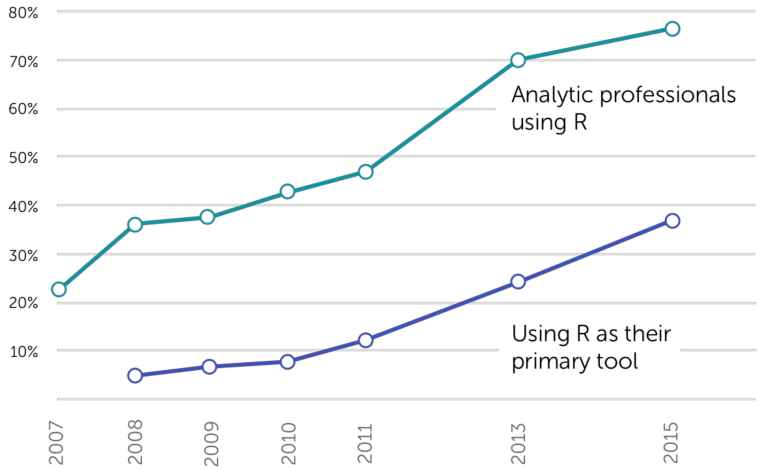


Figure 1.2: Růst využití R v čase respondnety Rexer Analytics Survey v roce 2017. Zdroj: Rexer Analytics: A Decade of Surveying Analytic Professionals: 2017 Survey Highlights.

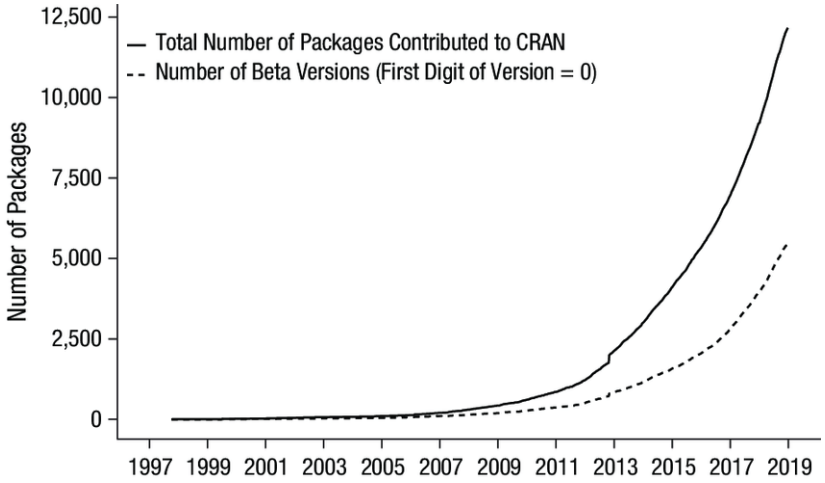


Figure 1.3: Počet balíků na CRANu ukazuje strmý růst objemu analytických nástrojů v R. Zdroj: Sacha Epskamp: "Reproducibility and Replicability in a Fast-Paced Methodological World", Advances in Methods and Practices in Psychological Science, 2019.

že si na často opakované úkoly vytvoříte vlastní funkci nebo skript. A pokud vám nějaká metoda v R chybí, můžete ji sami naprogramovat a případně i sdílet s ostatními ve formě nového balíku. (Ačkoli je R poněkud specializovaný programovací jazyk, RedMonk jej ve své analýze popularity programovacích jazyků umístil v roce 2018 R na 12 místo mezi *všemi* programovacími jazyky, viz obrázek 1.4.)

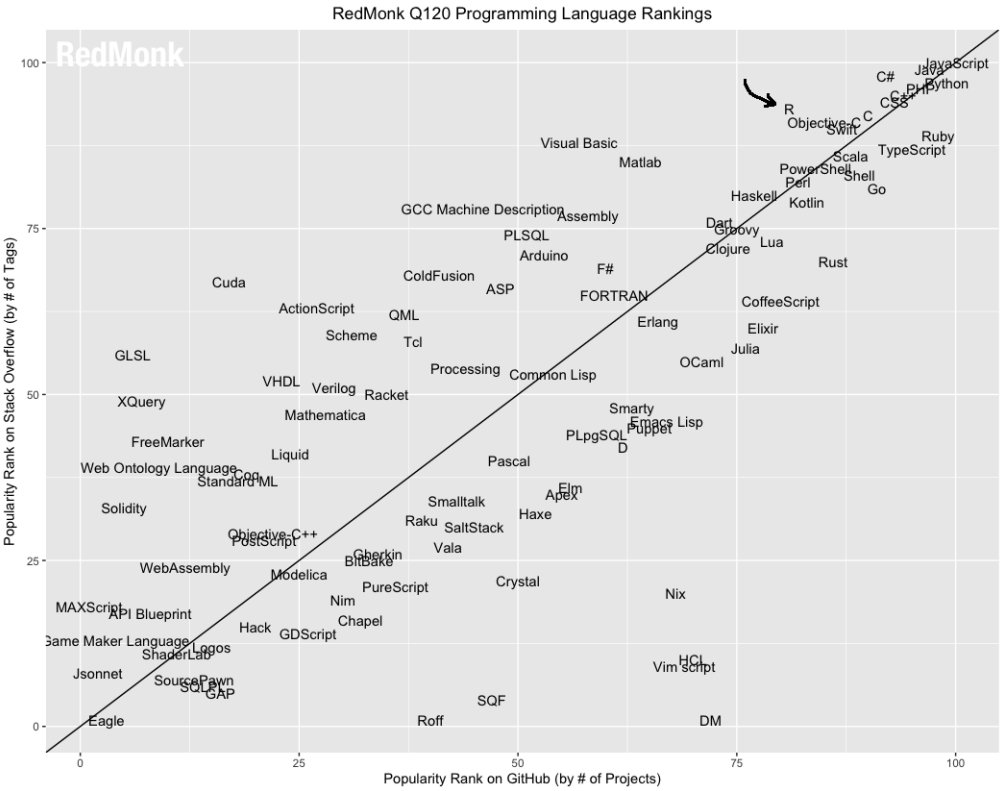


Figure 1.4: Oblíbenost programovacích jazyků měřená počtem projektů na vodorovné ose počtem tagů na serveru StackOverflow na svislé ose. Zdroj: RedMonk, 2020.

Poslední, ale nikoli nejméně významnou výhodou, je to, že R je volně šiřitelný software dostupný pro všechny hlavní operační systémy. To znamená, že jej budete mít k dispozici vždy a všude – a navíc zdarma.

Kromě výhod má R samozřejmě i slabé stránky. Jednou z nich je to, že R je jazyk specializovaný na analýzu dat, nikoli obecný programovací jazyk jako je např. Python, C++ nebo Java. V některých situacích se tedy může stát, že budete potřebovat i nějaký další nástroj. Pro vlastní analýzu dat si však s R naprosto vystačíte.

Často uváděným nedostatkem R je jeho rychlost. Tvrdívá se, že R je poměrně pomalý jazyk. Striktně vzato je to pravda (i když se jeho výkon neustále zlepšuje), ve většině případů to však nevadí, protože funkce, které by v R běžely pomalu, jsou dávno implementované v C++ nebo případně Fortranu a běží rychlostí těchto jazyků, tj. téměř rychlostí strojového kódu.

R také dokáže pracovat pouze s daty, která má uložená v operační paměti počítače. To je však zřídkakdy výrazné omezení. Např. tabulka s milionem pozorování o 50 proměnných bude mít typicky něco kolem 400 MB, takže se do operační paměti běžného počítače pohodlně vejde. R navíc dokáže velmi dobře spolupracovat s databázemi, a tak omezení pamětí počítač hravě obejít.

Překvapivě asi největší nevýhodou R je tak to, že se jedná o poměrně starý jazyk, který prošel dlouhým vývojem, přičemž se na jeho vývoji se podílely tisíce dobrovolníků bez výraznější centrální koordinace. To kromě dobrých vlastností zmíněných výše působí i některé problémy. Názvy funkcí a jejich syntaxe jsou někdy nekonzistenční. Každou věc je možné udělat mnoha různými způsoby a k řešení mnoha problémů existuje několik různých balíků funkcí, takže někdy strávíte nějaký čas rozhodováním, kterou cestu zvolit. Navíc je chování některých funkcí v některých speciálních situacích nečekané, protože to kdysi někomu přišlo jako dobrý nápad. Naštěstí se v poslední době většina těchto problémů řeší vznikem balíků ze skupiny **tidyverse**. V této knize budeme většinu věcí, kde existuje **tidyverse** alternativa řešit právě funkcemi definovanými v těchto balících.

1.2 Ochutnávka práce s daty

V tomto oddíle začneme se slíbenou ochutnávkou toho, na co se můžete v R těšit. Začneme práci s daty. Nevadí, že nebudete rozumět všemu, co se v této kapitole děje. Věnujte však pozornost tomu, jak jednoduše můžete dostat celkem složitý výsledek.

Jak se v R zachází s datovými soubory, si ukážeme na příkladu data setu *diamonds*. Nejdříve načteme potřebné balíky pomocí funkce `library()` a následně i vlastní data.

```
library(dplyr)
library(purrr)
library(ggplot2)
library(stargazer)
data("diamonds")
```

Nejprve se na data podíváme. Data set obsahuje údaje o 53 940 diamantech. Pro každý diamant (řádek tabulky) data set uvádí váhu diamantu v karátech, kvalitu jeho řezu (cit), jeho barvu (chlor), rozměry a cenu (proce). R inteligentně vypíše jen několik prvních řádků tabulky.

```
diamonds
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # ... with 53,930 more rows
```

R umožňuje s daty manipulovat velmi elegantně. Například se můžeme podívat na cenu a váhu u nejlépe zbarvených kamenů. Následující kód nejprve z data setu vyfiltruje kameny se správnou barvou, a pak vybere zvolené sloupce:

```
diamonds %>%
    filter(color == "D") %>%
    select(color, price, carat)
```

```
## # A tibble: 6,775 x 3
##    color price carat
##    <ord> <int> <dbl>
##  1 D       357  0.23
##  2 D       402  0.23
##  3 D       403  0.26
##  4 D       403  0.26
##  5 D       403  0.26
##  6 D       404  0.22
##  7 D       552  0.3
##  8 D       552  0.3
```

```
##  9 D       552  0.3
## 10 D       553  0.24
## # ... with 6,765 more rows
```

Data lze i jednoduše řadit. Například je možné se podívat na ty nejdražší kameny. Následující kód setřídí kameny sestupně podle ceny (tj. od nejvyšší po nejnižší cenu):

```
diamonds %>%
    arrange(desc(price))
```

```
## # A tibble: 53,940 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  2.29 Premium   I     VS2      60.8    60 18823  8.5   8.47  5.16
##  2  2    Very Good G     SI1      63.5    56 18818  7.9   7.97  5.04
##  3  1.51 Ideal     G     IF       61.7    55 18806  7.37  7.41  4.56
##  4  2.07 Ideal     G     SI2      62.5    55 18804  8.2   8.13  5.11
##  5  2    Very Good H     SI1      62.8    57 18803  7.95  8     5.01
##  6  2.29 Premium   I     SI1      61.8    59 18797  8.52  8.45  5.24
##  7  2.04 Premium   H     SI1      58.1    60 18795  8.37  8.28  4.84
##  8  2    Premium   I     VS1      60.8    59 18795  8.13  8.02  4.91
##  9  1.71 Premium   F     VS2      62.3    59 18791  7.57  7.53  4.7
## 10  2.15 Ideal     G     SI2      62.6    54 18791  8.29  8.35  5.21
## # ... with 53,930 more rows
```

Jednotlivé operace lze snadno kombinovat do větších celků. Řekněme, že nás např. zajímá, jaká cena nejdražšího kamene pro každou barvu. Nejdříve rozdělíme tabulku do skupin podle barvy kamene, pak každou skupinu setřídíme podle ceny od nejvyšší po nejnižší, a pak vybereme 1. (tj. nejdražší) kámen v každé skupině.

```
diamonds %>%
    group_by(color) %>%
    arrange(desc(price)) %>%
    slice(1L)
```

```
## # A tibble: 7 x 10
## # Groups:   color [7]
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  2.19 Ideal     D     SI2      61.8    57 18693  8.23  8.49  5.17
## 2  2.02 Very Good E     SI1      59.8    59 18731  8.11  8.2   4.88
## 3  1.71 Premium   F     VS2      62.3    59 18791  7.57  7.53  4.7
## 4  2    Very Good G     SI1      63.5    56 18818  7.9   7.97  5.04
## 5  2    Very Good H     SI1      62.8    57 18803  7.95  8     5.01
## 6  2.29 Premium   I     VS2      60.8    60 18823  8.5   8.47  5.16
## 7  3.01 Premium   J     SI2      60.7    59 18710  9.35  9.22  5.64
```

Data lze i jednoduše agregovat. Můžeme např. snadno spočítat průměrnou cenu pro každou barvu kamenů. Opět kameny rozdělíme do skupin podle barvy a pro každou skupinu vypočítáme průměrnou cenu kamenů ve skupině:

```
diamonds %>%
    group_by(color) %>%
    summarise(average_price = mean(price, na.rm = TRUE))
```

```
## # A tibble: 7 x 2
##   color average_price
##   <ord>         <dbl>
## 1 D             3170.
## 2 E             3077.
## 3 F             3725.
## 4 G             3999.
## 5 H             4487.
## 6 I             5092.
## 7 J             5324.
```

Kritéria je možné i kombinovat. Následující tabulka obsahuje průměrnou cenu kamene pro kombinaci barvy a řezu:

```
diamonds %>%
    group_by(color, cut) %>%
    summarise(average_price = mean(price, na.rm = TRUE))
```

```
## # A tibble: 35 x 3
## # Groups:   color [7]
##    color cut       average_price
##    <ord> <ord>             <dbl>
##  1 D     Fair              4291.
##  2 D     Good              3405.
##  3 D     Very Good         3470.
##  4 D     Premium           3631.
##  5 D     Ideal             2629.
##  6 E     Fair              3682.
##  7 E     Good              3424.
##  8 E     Very Good         3215.
##  9 E     Premium           3539.
## 10 E     Ideal             2598.
## # ... with 25 more rows
```

1.3 Ochutnávka vizualizace dat

Další část ochutnávky schopností jazyka R se týká vizualizace dat, tj. tvorby grafů. R je schopné vytvářet komplexní grafy v publikační kvalitě – a přitom velmi elegantně a snadno. Všechny grafy v tomto oddíle byly vytvořeny přímo v R nebyly nijak ručně upravovány. Do webové i PDF verze knihy je upravilo samo R.

I nadále budeme pracovat s tabulkou *diamonds*. Nejdříve ze všeho by nás mohlo zajímat, jak časté jsou různé ceny diamantů. K tomuto účelu se obvykle používá histogram, viz obrázek 1.5, vytvořený následujícím kódem:

```
diamonds %>%
    ggplot(aes(price)) +
    geom_histogram() +
    theme_bw()
```

Výsledek není překvapivý – čím vyšší cena, tím méně kamenů se za ni prodává. Zajímavější otázka však je, zda se ceny liší podle kvality řezu kamene. Zde nám histogram nepomůže. Existuje však celá řada jiných možností. První z nich je tzv. boxplot, viz obrázek 1.6. Výška každé krabice ukazuje mezikvartilovou vzdálenost, tlustá vodorovná čára mediánovou cenu a jednotlivé tečky odlehlá pozorování. Histogram byl vytvořen takto:

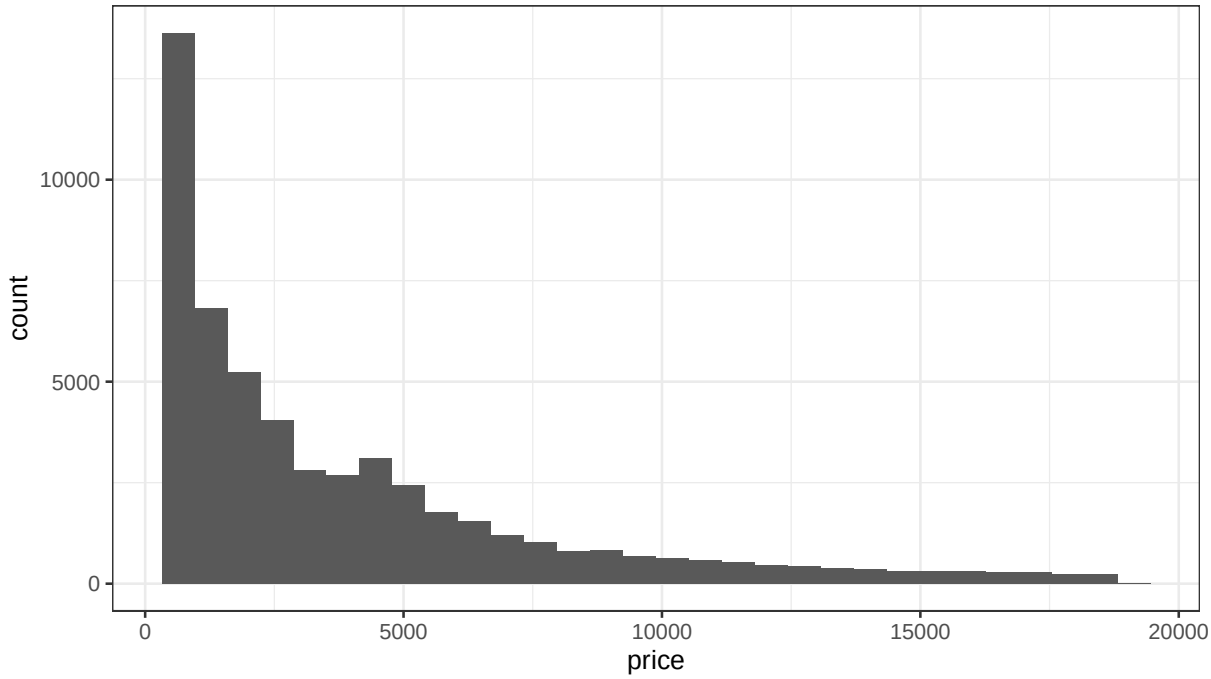


Figure 1.5: Histogram rozdělení cen diamantů

```
diamonds %>%
    ggplot(aes(cut, price)) +
    geom_boxplot() +
    theme_bw()
```

Výsledek je překvapivý: zdaleka neplatí, že dokonalejší řez znamená vyšší ceny. Možná je to však tím, že boxploty neukazují celý tvar statistického rozdělení, ale jen pár vybraných charakteristik tohoto rozdělení. Celé rozdělení můžeme zobrazit např. pomocí odhadu jádrové hustoty. Abychom mohli pozorovat rozdíly mezi jednotlivými typy řezů, částečně grafy hustot zprůhledníme, viz obrázek 1.7 vytvořený takto:

```
diamonds %>%
    ggplot(aes(price, fill = cut)) +
    geom_density(alpha = 0.35) +
    theme_bw()
```

Nejzajímavější otázky se však vždy týkají vztahů mezi veličinami. Řekněme, že nás například zajímá, jak souvisí cena kamene s jeho váhou. Takový vztah dokáže pěkně zobrazit *scatter plot*, viz obrázek 1.8. Vlastní trend vztahu zobrazíme snadno přidáním regresní přímky. V tomto grafu navíc nebudeme líní a změníme nápisy v grafu do češtiny:

```
diamonds %>%
    ggplot(aes(carat, price)) +
    geom_point(alpha = 0.05) +
    geom_smooth(method = lm) +
    xlab("váha") + ylab("cena") +
    ylim(0, 20000) +
    theme_bw()
```

Mohlo by nás ovšem zajímat, zda je vztah mezi cenou a váhou kamene stejný bez ohledu na barvu kamene. Abychom toho dosáhli, musíme do grafu přidat další rozměr. Na první pohled se zdá, že budeme potřebovat

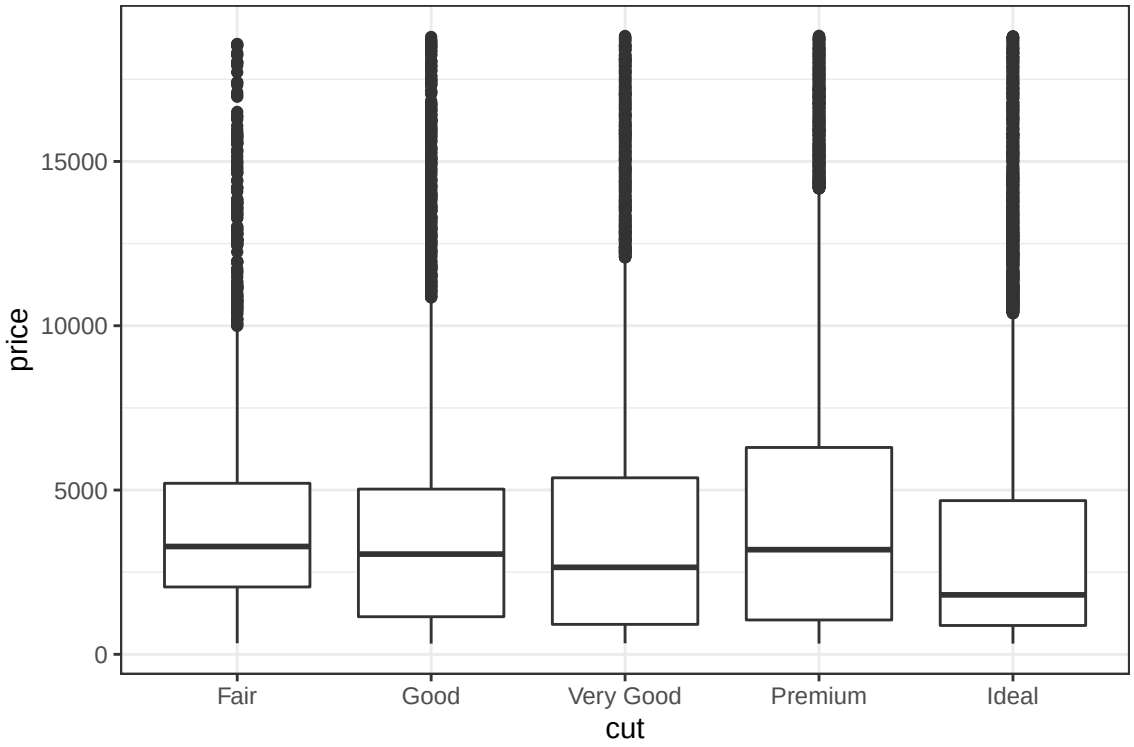


Figure 1.6: Porovnání rozdělení cen podle typu řezu pomocí boxplotů

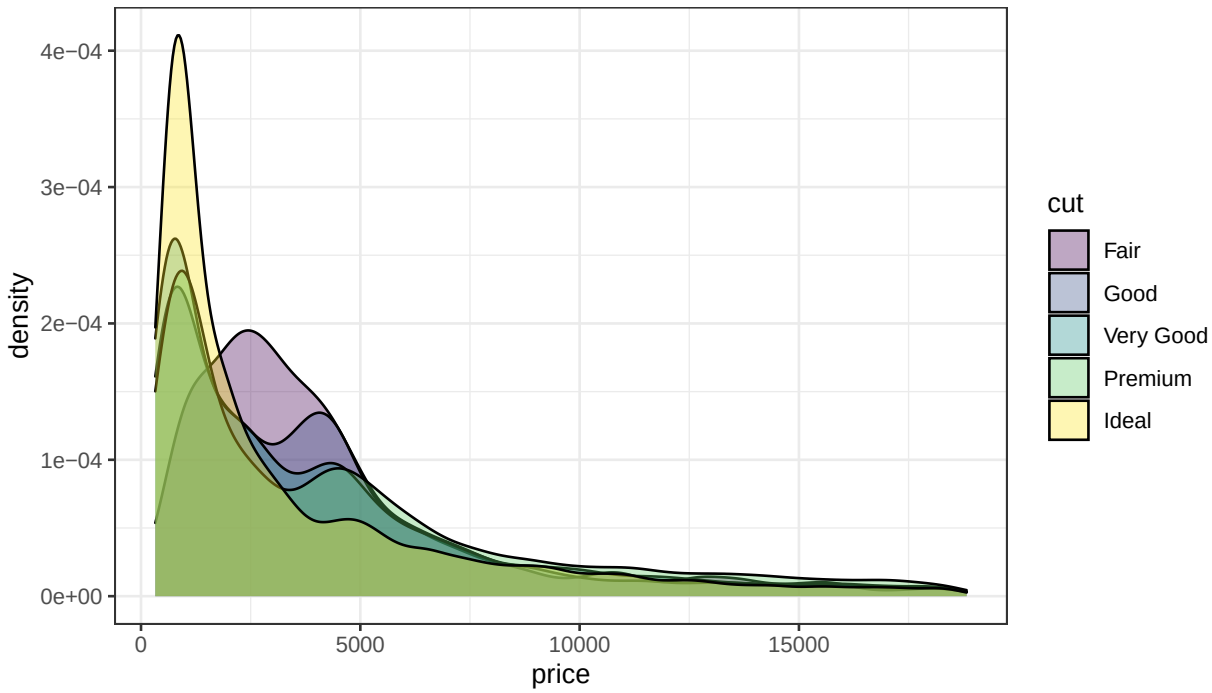


Figure 1.7: Porovnání rozdělení cen podle typu řezu pomocí odhadů jádrové hustoty

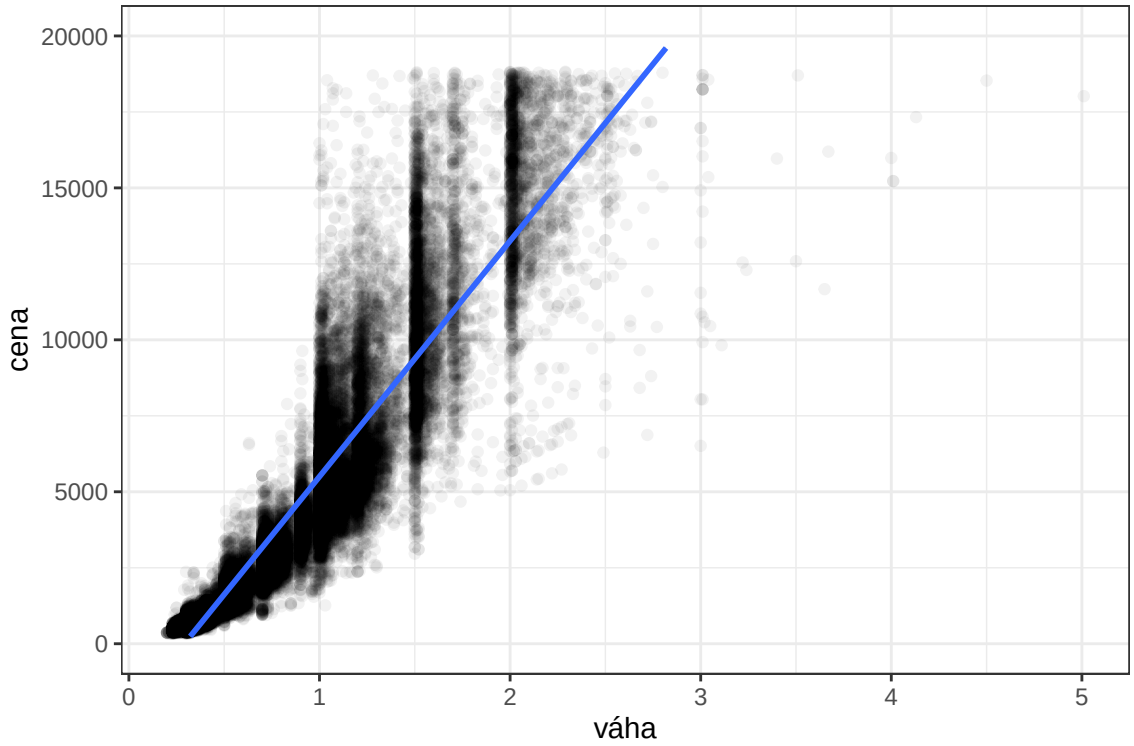


Figure 1.8: Vztah mezi váhou kamenů a jejich cenou

třírozměrný graf. Takové grafy jsou však velmi nepřehledné. R má v zásobě něco lepšího: umožní nám přidáním jednoho řádku do předchozího kódu rozdělit kameny do dílčích grafů, viz obrázek 1.9:

```
diamonds %>%
    ggplot(aes(carat, price)) +
    geom_point(alpha = 0.05) +
    geom_smooth(method = lm) +
    facet_wrap(~ color) +
    xlab("váha") + ylab("cena") +
    ylim(0, 20000) +
    theme_bw()
```

1.4 Ochutnávka regresní analýzy

Nejdůležitějším nástrojem analýzy dat je beze sporu regrese. R obsahuje nástroje pro velmi pokročilé regresní techniky, zde se však podíváme na jednoduchou lineární regresi. Pokusíme se vysvětlit, jak závisí cena diamantů na jejich charakteristikách, jako jsou váha, typ řezu, barva a velikost. Začneme tím, že odhadneme jednodušší model, který je popsaný rovnicí:

$$\text{price} = \beta_0 + \beta_1\ \text{carat} + \beta_2\ \text{color} + \beta_3\ \text{cut} + \beta_4\ \text{table} + \varepsilon.$$

Proměnné `cut` a `color` jsou kategoriální. R pro ně automaticky připraví potřebné umělé proměnné. Nejdříve však tyto faktory převedeme na ne-ordinální, protože tak bude mít odhad modelu jednodušší interpretaci:

```
diamonds <- mutate(diamonds,
                   color = factor(color, ordered = FALSE),
                   cut = factor(cut, ordered = FALSE))
```

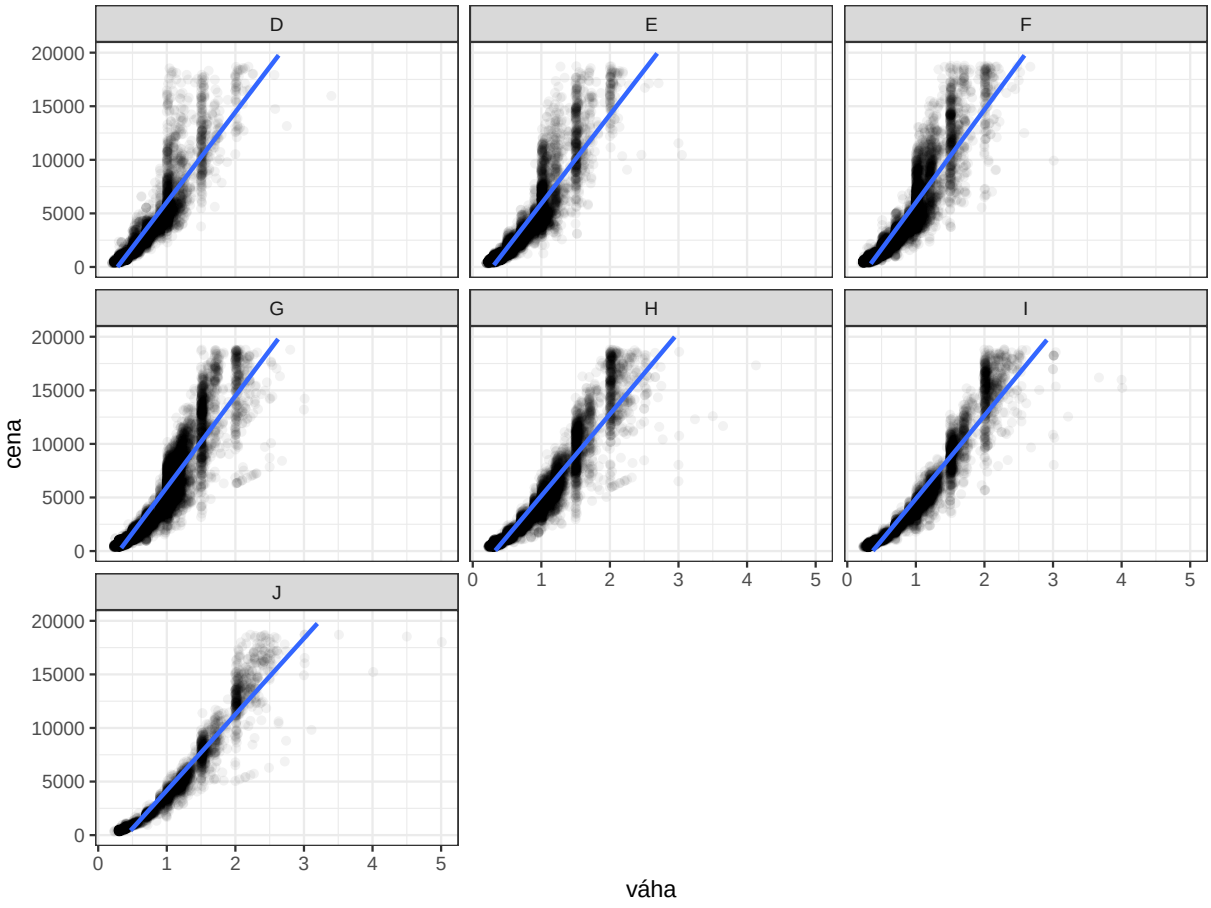


Figure 1.9: Vztah mezi váhou kamenů a jejich cenou při kontrole o barvu kamenů

Vlastní model odhadneme takto:

```
model <- price ~ carat + color + cut + table
em <- lm(model, diamonds)
summary(em)  # vypíše výsledek regrese
```

```
##
## Call:
## lm(formula = model, data = diamonds)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -17351.4   -751.5    -84.3    544.7  12226.7
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -2471.895    201.814 -12.248  < 2e-16 ***
## carat         8192.894     13.962 586.784  < 2e-16 ***
## colorE         -89.573     22.625  -3.959 7.53e-05 ***
## colorF         -72.834     22.772  -3.198  0.00138 **
## colorG        -106.719     22.070  -4.836 1.33e-06 ***
## colorH        -734.724     23.702 -30.999  < 2e-16 ***
## colorI       -1077.256     26.574 -40.537  < 2e-16 ***
## colorJ       -1909.932     32.863 -58.118  < 2e-16 ***
```

```
## cutGood       1120.648     41.223  27.185  < 2e-16 ***
## cutVery Good  1495.866     38.273  39.084  < 2e-16 ***
## cutPremium    1437.307     37.774  38.050  < 2e-16 ***
## cutIdeal      1742.944     38.589  45.167  < 2e-16 ***
## table          -21.951      3.366  -6.521 7.05e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1432 on 53927 degrees of freedom
## Multiple R-squared:  0.8713, Adjusted R-squared:  0.8712
## F-statistic: 3.041e+04 on 12 and 53927 DF,  p-value: < 2.2e-16
```

Často potřebujeme porovnat odhady různých specifikací modelu. Například by mohlo být zajímavé nahradit proměnnou `table` jiným měřítkem velikosti kamene. Zde odhadneme původní model a čtyři jeho alternativy:

```
model <- list(
    model,
    model %>% update(. ~ . - table + depth),
    model %>% update(. ~ . - table + x),
    model %>% update(. ~ . - table + y),
    model %>% update(. ~ . - table + z)
)
em <- map(model, ~ lm(., data = diamonds))
```

Výsledky porovnáme v přehledné tabulce s názvem “Výsledky odhadů různých specifikací modelů”:

```
stargazer(em, type = pandoc.output.format(), df = FALSE,
          notes.append = FALSE, notes = "", header = FALSE,
          no.space = TRUE,
          title = "Výsledky odhadů různých specifikací modelu")
```

Table 1.1: Výsledky odhadů různých specifikací modelu

	Dependent variable:				
	price				
	(1)	(2)	(3)	(4)	(5)
carat	8,192.894***	8,183.774***	11,041.860***	9,492.338***	9,725.135***
	(13.962)	(13.889)	(58.130)	(42.709)	(43.207)
colorE	−89.573***	−91.842***	−97.617***	−92.161***	−93.096***
	(22.625)	(22.621)	(22.115)	(22.416)	(22.342)
colorF	−72.834***	−72.405***	−52.582**	−61.845***	−62.069***
	(22.772)	(22.767)	(22.262)	(22.564)	(22.488)
colorG	−106.719***	−100.884***	−107.555***	−104.873***	−101.694***
	(22.070)	(22.064)	(21.568)	(21.862)	(21.789)
colorH	−734.724***	−727.123***	−761.310***	−744.271***	−740.949***
	(23.702)	(23.703)	(23.172)	(23.484)	(23.404)
colorI	−1,077.256***	−1,070.565***	−1,138.112***	−1,104.141***	−1,100.604***
	(26.574)	(26.577)	(26.004)	(26.344)	(26.250)
colorJ	−1,909.932***	−1,902.818***	−1,992.011***	−1,945.948***	−1,939.625***
	(32.863)	(32.865)	(32.165)	(32.581)	(32.463)
cutGood	1,120.648***	1,067.103***	1,184.977***	1,195.907***	1,058.334***
	(41.223)	(41.910)	(40.300)	(40.889)	(40.738)
cutVery Good	1,495.866***	1,438.418***	1,576.936***	1,596.985***	1,428.780***
	(38.273)	(39.451)	(37.283)	(37.851)	(37.721)
cutPremium	1,437.307***	1,342.996***	1,545.595***	1,509.262***	1,316.495***
	(37.774)	(39.854)	(36.970)	(37.474)	(37.444)
cutIdeal	1,742.944***	1,724.554***	1,873.955***	1,880.359***	1,706.301***
	(38.589)	(38.768)	(36.460)	(37.001)	(36.909)
table	−21.951***				
	(3.366)				
depth		−35.979***			
		(4.602)			
x			−1,232.425***		
			(24.372)		
y				−566.150***	
				(17.492)	
z					−1,082.736***
					(28.780)
Constant	−2,471.895***	−1,458.123***	965.254***	−1,622.037***	−1,055.518***
	(201.814)	(297.291)	(101.796)	(77.707)	(82.654)
Observations	53,940	53,940	53,940	53,940	53,940
R^2	0.871	0.871	0.877	0.874	0.874
Adjusted R^2	0.871	0.871	0.877	0.874	0.874
Residual Std. Error	1,431.632	1,431.386	1,399.404	1,418.485	1,413.764
F Statistic	30,410.660***	30,422.700***	32,036.900***	31,060.700***	31,298.560***

Note:

Part I

Základy

2 Základy práce s R a s RStudiem

Dříve než se pustíme do studia R, potřebujete jisté základy. V této kapitole se naučíte

- jak nainstalovat R,
- jak nainstalovat, zkonfigurovat a používat RStudio,
- jak načíst, nainstalovat a aktualizovat balíky funkcí a dat,
- jak číst dokumentaci a kde najít pomoc,
- jak používat konzolu a
- jak psát a spouštět skripty.

2.1 Instalace R

Analyzovat a vizualizovat data v R se nenaučíte čtením knihy: sami si musíte hrát s R a experimentovat se skutečnými nebo simulovanými daty. K tomu budete potřebovat mít R nainstalované na svém počítači. V tomto oddíle stručně popíšeme, jak R instalovat ve Windows a v Linuxu. Instalace je v obou případech nenáročná; v Linuxu vám poskytneme několik rad, jak optimalizovat výkon R.

Vlastní návod k instalaci i instalační soubory najdete na https://cran.r-project.org/. Instalace ve Windows je jednoduchá. Mimo jiné ji ukazuje tento tutoriál: http://youtu.be/Ohnk9hcxf9M. Určité problémy mohou vzniknout, pokud máte v cestě ke svému domovskému adresáři mezery a písmena s háčky a čárkami. (Těm je však lepší se vždy vyhnout.) Pokud by standardní cesty kvůli mezerám, háčkům a čárkám nefungovaly, je možné nastavit jiné cesty, viz dále.

V mnoha distribucích Linuxu je R obsaženo přímo ve standardních repositářích, a to včetně jednotlivých přidaných balíků. Doporučuji se těmto balíků *vyhnout*. Rozumnější je postupovat podle návodu na výše uvedené stránce: přidat si CRAN do repositářů a nainstalovat pouze jádro R a ty balíky, které CRAN nabízí v binární podobě. Ostatní balíky si nainstalujete přímo v R. Tak budete mít vždy aktuální verze.

R používá k maticovým výpočtům standardní numerické knihovny BLAS a LAPACK. Existuje několik verzí těchto knihoven, které se od sebe velmi liší výkonem. Mezi nejlepší patří OpenBLAS, AtlasBLAS a Intel MKL (pouze pro procesory značky Intel). V Linuxu se implicitně používá nepříliš efektivní verze těchto knihoven, proto doporučujeme nainstalovat a zvolit některou efektivnější verzi. Výrazně tak zrychlíte mnoho svých výpočtů. Než si zvolíte jednu z těchto knihoven, podívejte se na aktuální výsletky testů výkonosti na internetu. (V současnosti je asi nejlepší obecnou volbou OpenBlas.)[1]

V Ubuntu naistalujete jednotlivé knihovny takto:

```
# instalace OpenBLAS
sudo apt install libopenblas-base
# instalace ATLAS
sudo apt install libatlas3-base liblapack3
# intalace Intel MKL
sudo apt install intel-mkl-full
```

[1] Při volbě alternativních knihoven BLAS a LAPACK však buďte opatrní. Minimálně v Ubuntu 20.04 jsou knihovny OpenBlas a Intel MKL pokažené – někdy skončí chybou, někdy vrací náhodný výsledek. V této verzi je tedy nejlepší použít Atlas. Pro jiné verze googlete.

Poslední nainstalovaný BLAS by se měl automaticky použít. Později jej můžete zkonfigovat takto:

```
# volba BLAS
sudo update-alternatives --config libblas.so.3-x86_64-linux-gnu
# volba LAPACK
sudo update-alternatives --config liblapack.so.3-x86_64-linux-gnu
```

Funkce `sessionInfo()` v R vypíše, kromě jiných informací, i to, který BLAS a LAPACK vaše R používá. V ostatních distribucích Linuxu budete zřejmě postupovat analogicky.

Volba efektivnějších verzí BLAS a LAPACK ve Windows je podstatně obtížnější a může zahrnovat kompilaci daných knihoven i vlastního R. Pokud si ji chcete vyzkoušet, hledejte na Internetu nejnovější návod.

2.2 RStudio

R funguje jako program na příkazovém řádku (ve Windows má k dispozici jednoduché grafické rozhraní). Pro vážnou práci s ním je však vhodné použít nějaké vývojové prostředí (IDE). Nejlepší vývojové prostředí pro R je v současné době RStudio, které je šířené zdarma pod licencí AGPL v3 pro Windows, Linux i Mac. RStudio ke svému běhu potřebuje Javu (doporučuji oficiální Javu od Oracle, ne její svobodné ekvivalenty). Jak RStudio vypadá, ukazuje obrázek 2.1.

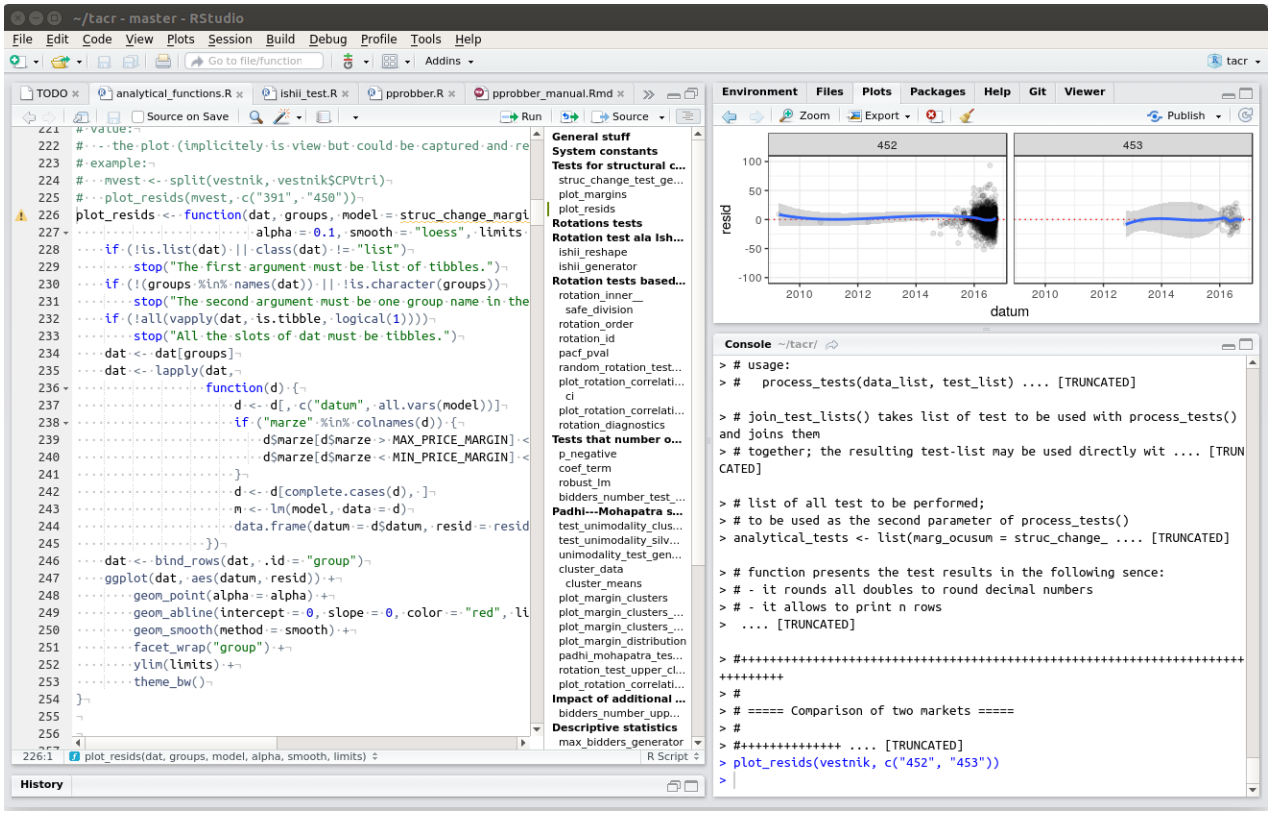


Figure 2.1: Vývojové prostředí RStudio.

RStudio je zdarma ke stažení na www.rstudio.com, konkrétně na https://rstudio.com/products/rstudio/download/. Pro Windows je dispozici instalační soubor, pro Linux standardní balíky `.deb` a `.rpm`. Po instalaci byste měli jednou za čas zkontrolovat, zda máte nejnovější verzi RStudia, a to v menu RStudia `Help→Check for Updates`.

RStudio se skládá z obecného menu a čtyř panelů. Každý panel může obsahovat několik záložek. Můžete si zkonfigurovat, kde bude který panel, jak bude veliký a které záložky budou ve kterém panelu. Mezi hlavní záložky RStudia patří:

- **Konzola** (Console) je určena pro interaktivní práci s R. Kód, který do ní zapíšete, R okamžitě vyhodnotí a výsledky vypíše do konzoly nebo zobrazí jako graf.
- **Editor** (Source) – slouží k psaní skriptů. Kód, který do něj napíšete, můžete uložit a opakovaně spouštět a ladit. V editoru můžete současně editovat libovolné množství souborů různých typů (R script, R markdown, textové dokumenty a mnohé další).
- **Přehled prostředí R** (Environment) zobrazuje všechny objekty (data, funkce apod.), které aktuálně žijí ve zvoleném prostředí v R (implicitně v globálním prostředí). Současně ukazuje i to, kolik paměti zabírá současné sezení R. Přehled prostředí také umožňuje importovat některé typy dat a mazat dříve vytvořené objekty.
- **Soubory** (Files) zobrazují soubory a adresáře, které jsou v aktuálním projektu nebo adresáři a umožňuje s nimi dělat základní operace (mazat je, přejmenovávat apod.).
- **Grafy** (Plots) zobrazují grafy, které jste v R vykreslili. Tato záložka se otevře teprve ve chvíli, kdy nějaký graf vytvoříte.
- **Balíky** (Packages) zobrazují seznam instalovaných balíků. Zároveň umožňují i balíky načítat, instalovat, aktualizovat a odstraňovat.
- **Nápověda** (Help) zobrazuje dokumentaci k funkcím, datům a balíkům.
- **Historie** (History) zobrazuje kód, který jste v minulosti spustili v konzoli. Umožňuje jej také uložit a přesunout do konzoly a do editoru.

Další typy záložek se objeví v případě, že budete dělat něco pokročilého, např. budete používat `make` nebo správu verzí.

Ikona *Workspace Panes* (vypadá jako čtyři malá okénka) umožňuje jednotlivé panely a záložky dočasně zvětšit přes celou obrazovku (po ťuknutí zobrazí klávesové zkratky).

V pravém horním rohu RStudia je přepínač projektů. **Projekty** umožňují elegantně oddělit různé projekty, na kterých současně pracujete. Každý projekt má svůj vlastní adresář, vlastní proces R atd. Víc o projektech najdete na https://goo.gl/CxyHVS.

Doporučuji, abyste si před vlastní prací RStudio zkonfigurovali. Z menu RStudia vyvolejte `Tools→Global Options...` a nastavte, jak se má RStudio chovat. Vysvětlení jednotlivých položek najdete např. na stránce https://support.rstudio.com/hc/en-us/articles/200549016-Customizing-RStudio. Doporučuji zejména následující nastavení (moje nastavení ukazují obrázky 2.2 a 2.3):

- V záložce `General` doporučuji vypnout `Restore .RData into workspace` a nastavit `Save workspace to .RData on exit` na `Never`. V opačném případě se vám na začátku sezení nahrají do paměti výsledky výpočtů z předchozího sezení. Na první pohled to vypadá to jako dobrý nápad a úspora času, ale ve skutečnosti je to zdroj chyb, které se špatně hledají. Měli byste si zvyknout, že veškeré výpočty máte uložené v podobě skriptu a můžete je tedy kdykoli znovu provést.
- V záložce `Code→Editing` si zapněte `Insert spaces for tab` a `Tab width` nastavte aspoň na 4.
- V záložce `Code→Display` zapněte vše (snad kromě `Highlight selected line`) a `Margin column` nastavte na hodnotu kolem 80 nebo 90.
- V záložce `Code→Saving` zapněte vše. Zvažte, zda nenastavit kódování na UTF-8, což je rozumný standard pro výměnu textů.
- V záložce `Code→Completetion` zapněte vše snad kromě `Show help tooltip on cursor idle`.
- V záložce `Code→Diagnostics` zapněte vše. Tak vám RStudio bude při psaní kódu v editoru zobrazovat diagnostické rady.
- V záložce `Sweave` nastavte `Weave Rnw files using` na `knitr` a `Typeset LaTeX into PDF using` na `pdflatex`

Pokud jste hračičkové, můžete si stáhnout a nainstalovat do systému nějaký font, který podporuje ligatury technických symbolů, a v menu `Tools→Global Options...→Appearance` si jej nastavit jako `Editor font`. Pak se vám budou některé symboly složené z více znaků jako je např. `<-`, `|>`, `>=`, `==`, `!=` apod. zobrazovat jako jeden znak (v kódu budou samozřejmě stále oba). Vhodný font jen např. `JetBrains Mono` (https://www.jetbrains.com/lp/mono/) nebo `FiraCode` (https://github.com/tonsky/FiraCode).

Kromě menu můžete RStudio ovládat i pomocí klávesových zkratek. Seznam klávesových zkratek se zobrazí po volbě menu `Tools→Keyboard shortcuts help` nebo po stisku `Alt+Shift+K`. Úplný seznam klávesových

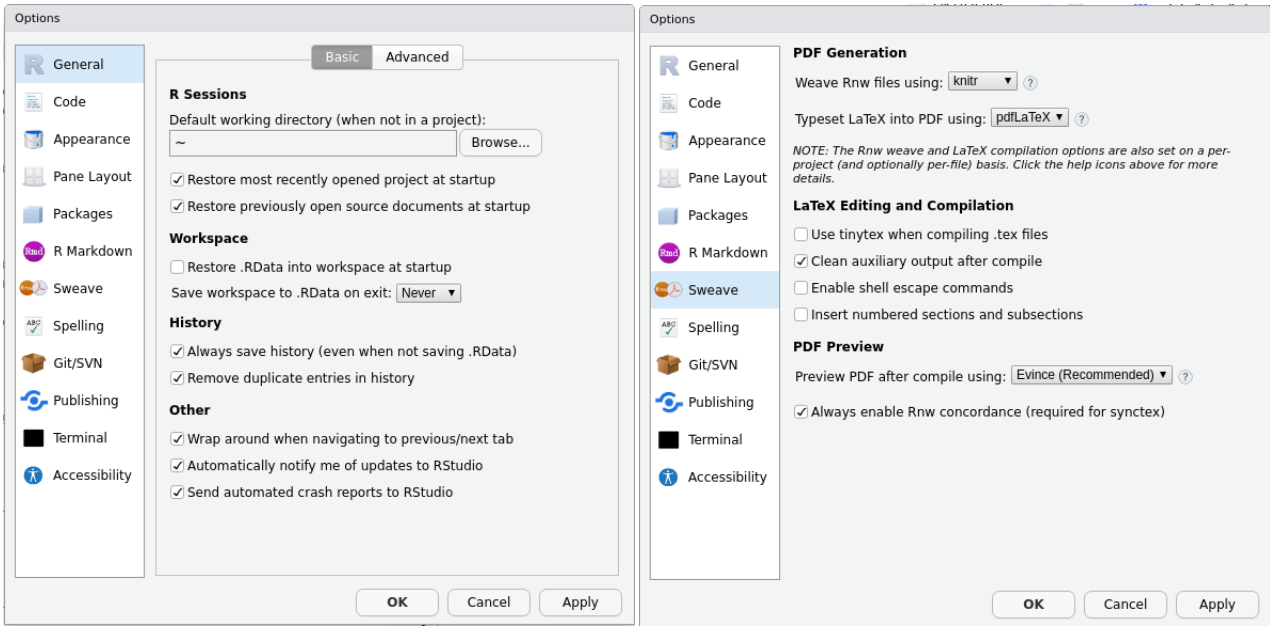


Figure 2.2: Nastavení obecných částí RStudia.

zkratek najdete i na https://goo.gl/aPSel6. Ve verzi 1.4 přidalo RStudio i populární "palety". Spustíte je klávesovou zkratkou `Ctrl+Shift+P`. Pomocí palety můžete nejen rychle provádět operace, ke kterým byste jinak museli jít do menu, ale i nastavovat různé parametry prostředí RStudia. Dokumentaci k této vlastnosti najdete na stránce https://blog.rstudio.com/2020/10/14/rstudio-v1-4-preview-command-palette/.

R lze ukončit funkcí `q()`. Pokud běží v RStudiu, ukončíte jej jednoduše buď v menu `File→Quit Session...` nebo křížkem okna. R standardně při ukončení uloží všechny objekty v paměti (data, uživatelem definované funkce apod.) do souboru a při opětovném spuštění je opět načte (netýká se načtených balíků – ty je třeba načíst pokaždé znovu). Někdy se to hodí, ale často je to zdrojem chyb, které se špatně hledají. Doporučuji tuto funkci v nastavení zakázat, viz výše.

Schopnosti RStudia je možné rozšířit pomocí "doplňků" (*addins*). Doplňky se instalují jako běžné balíky R (viz následující oddíly) a po instalaci jsou dostupné v roletce `Addins` v paletě nástrojů RStudia. Pokud si nainstalujete balík **addinslist**, přibude vám v menu doplňků volba `Browse RStudio addins`, která vám zobrazí dostupné doplňky, jejich popisy a umožní je jednoduše instalovat.

Různé návody k používání RStudia najdete zde na adrese https://goo.gl/ik2Yb9. Cheatsheet pro používání RStudia získáte v menu `Help` → `Cheatsheets` → `RStudio IDE Cheat Sheet`.

2.3 Balíky

Všechny funkce, datové struktury, data i vše ostatní je v R organizováno do balíků. Základní balíky (`base`, `methods`, `datasets`, `utils` apod.) jsou přítomny v každé instalaci R a načtou se automaticky při jeho spuštění. Ostatní balíky je potřeba nainstalovat a před použitím načíst do paměti "ručně". Seznam balíků, které máte aktuálně nainstalované, můžete v RStudiu zobrazit v záložce `Packages`. Balíky, které jsou aktuálně načtené do paměti, mají v seznamu zapnuté zaškrtávátko.

Pokud máte nějaký balík nainstalovaný, můžete jej začít používat. R však samo o sobě o objektech uložených v balících neví. Před jejich použitím je tedy třeba načíst balík do paměti (ve skutečnosti se načtou jen jména objektů v balíku do cesty, ve které R jednotlivé objekty hledá). K tomu slouží funkce `library()`:

```
library(dplyr)    # v závorce je jméno funkce
```

Data a funkce z balíku je možné využít i bez jeho načtení pomocí operátoru `::`, kterým se oddělí jméno balíku a jméno funkce. To se hodí zejména ve dvou situacích: 1) Při načtení nového balíku se někdy stane, že objekty

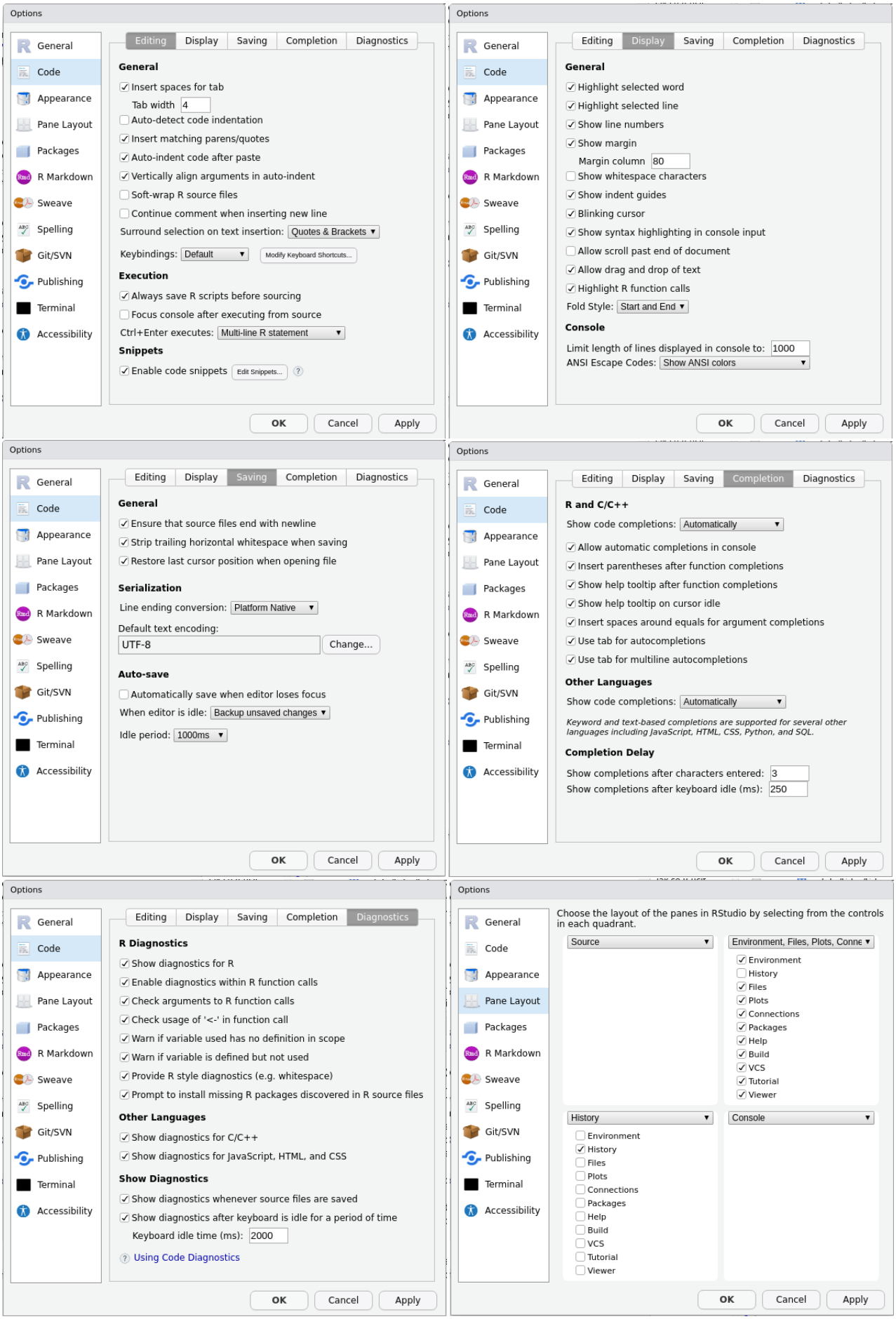


Figure 2.3: Nastavení editoru RStudia.

z nově načteného balíku překryjí data a funkce z balíku, který jste načetli dříve. Dvojtečkový operátor vám umožní použít i tyto "překryté" objekty. 2) Někdy chcete použít jen jednu funkci a nechcete načítat celý balík. Pokud bychom tedy nenačetli balík **dplyr**, stále bychom mohli použít jednu z jeho funkcí takto:

```
dplyr::anti_join()  # jméno_balíku::jméno_funkce()
```

Pokud ovšem máte balík načtený pomocí funkce `library()`, můžete funkci zavolat jen jejím jménem bez jména balíku:

```
anti_join()  # jméno funkce z načteného balíku
```

Pokud chcete použít jakýkoli balík, musíte jen nejdříve nainstalovat. Instalace balíků v R je snadná, protože jejich valná většina je k dispozici v centralizovaných repositářích. Hlavním repositářem je CRAN (https://cran.r-project.org/). V RStudiu nainstalujete balíky z CRANu tak, že v záložce `Packages` kliknete na tlačítko `Install`. (Při první instalaci je třeba nastavit adresu zrcadla CRANu, ze kterého se budou balíky stahovat. Doporučuji použít zrcadlo `Global CDN RStudio`.) K ruční instalaci balíků slouží funkce `install.packages()`. Balíky se neustále vyvíjejí (většinou zlepšují) a obvykle je dobré mít instalovány poslední verze. K aktualizaci balíků nainstalovaných z CRANu slouží v RStudiu v záložce `Packages` klikátko `Update`.

Kromě CRANu existuje i několik dalších repositářů. Nejvýznamnějším z nich je GitHub. GitHub (https://github.com/) obsahuje vývojové verze balíků a nové balíky, které se dosud nedostaly na CRAN. Návod, jak instalovat balíky z GitHubu, najdete na https://goo.gl/ttEz9J.

Implicitně se balíky instalují do uživatelského adresáře. V tomto adresáři se vytvoří vždy nový podadresář pro každou novou verzi R. To je proto, že při větší aktualizaci R už nemusejí původní balíky fungovat a je třeba je znovu stáhnout nebo zkompilovat.

Pokud máte v cestě k do uživatelského adresáře umístěny znaky s diakritiku (např. proto, že jméno vašeho uživatelského je např. účtu `Jiřík`), nebudou věci fungovat správně. Zejména vám nepůjdou spustit skripty (ani pomocí tlačítka `Source` v RStudiu) a nepůjdou vám instalovat a načítat balíky. Problém se skripty vyřešíte snadno: vytvoříte někde adresář na práci a tam budete skladovat své skripty. Problém s balíky vyřešíte tak, že si vytvoříte alternativní knihovnu. Knihovna je adresář, kde R instaluje a hledá balíky. Postup je následující:

1. Vytvořte si adresář, kam můžete zapisovat a který neobsahuje v cestě divné znaky, ve Windows např. adresář `C:/Rlibs`. Následně si ověřte, že je nastaven pro čtení i zápis.
2. Ve svém domovském adresáři vytvořte soubor `.Rprofile` a zapište do něj `.libPaths("C:/Rlibs")`. Všimněte si, že místo zpětných lomítek používáme v cestě obyčejná lomítka (jinak musíte zpětná lomítka zdvojit, tj. psát `\\`). Pokud soubor neexistuje, stačí v R spustit následující řádek:

   ```
   writeLines('.libPaths("C:/Rlibs")', con = "~/.Rprofile")
   ```

3. Po restartu RStudia se budou balíky hledat v tomto adresáři. Při instalaci balíků klikněte na pulldown menu "Install to Library:" a vyberte tento adresář. RStudio by si jej mělo po první instalaci balíku do tohoto adresáře pamatovat jako defaultní volbu

V Linuxu se balíky při instalaci kompilují pro váš operační systém, zatímco ve Windows se stahují už zkompilované. Pokud chcete mít možnost kompilovat balíky ze zdrojového kódu i ve Windows, musíte nainstalovat `RTools`. Stáhnout si je můžete zde: https://cran.r-project.org/bin/windows/Rtools/. Po jejich instalaci je třeba přidat je do vyhledávací cesty. Návod je opět na této stránce stránce. Nejjednodušší způsob, jak to udělat, je zpsat cestu do souborou `.Renviron` ve vašem domovském adresáři. Pokud tento soubor neexistuje, stačí v R spustit řádek:

```
writeLines('PATH="${RTOOLS40_HOME}\\usr\\bin;${PATH}"', con = "~/.Renviron")
```

Pokud jste si jistí, že nechcete balíky kompilovat, pak RTools v zásadě nepotřebujete. R však bude v takovém případě vypisovat následující varování:

```
WARNING: Rtools is required to build R packages but is not currently installed.
Please download and install the appropriate version of Rtools before proceeding:
http://cran.rstudio.com/bin/windows/Rtools/
```

V Linuxu jsou nástroje pro kompilaci balíků přítomné automaticky, takže žádné další kroky k instalaci `RTools` nejsou potřeba.

2.4 Nápověda

Nikdo si nemůže pamatovat všechno – a pamatovat si detaily použití jednotlivých funkcí je absurdní. Proto má R velmi dobrý systém nápovědy a dokumentace. RStudio tento systém ještě dále vylepšuje. Je velmi důležité, abyste se naučili dokumentaci k funkcím číst.

Nápovědu ke zvolené funkci můžete získat jedním ze tří způsobů. V konzoli napíšete jméno funkce (např. funkce `mean()`) bez závorek za otazníkem nebo jako argument funkce `help()`

```
?mean        # za otazníkem je jméno funkce
help("mean") # v uvozovkách je jméno funkce
```

nebo po napsání jména funkce do konzoly nebo editoru zmáčknete v RStudiu klávesu `F1`. Všechny tyto cesty vedou ke stejnému výsledku: RStudio zobrazí stránku nápovědy k dané funkci.

Jedna stránka nápovědy může dokumentovat několik různých funkcí, které mají něco společného. Stránky dokumentace mají v R standardní strukturu. Je potřeba, abyste se s ní seznámili. Stránka dokumentace má následující strukturu:

- **Název funkce nebo tématu a balíku**, např. `mean {base}`. To znamená, že strana dokumentuje funkci `mean()` definovanou v balíku `base`.
- **Jméno stránky dokumentace**, zde "Arithmetic Mean".
- **Popis, co funkce dělají** (Description).
- **Popis syntaxe**, jak se funkce používají (Usage). Zde se uvádí zejména to, jaké má funkce parametry a jaké mají tyto parametry implicitní hodnoty (pokud nějaké mají) a. Ve funkci `mean()` je prvním parametrem `x`, který žádnou implicitní hodnotu nemá. Naproti tomu parametry `trim` a `na.rm` implicitní hodnoty mají. Pokud jejich hodnoty nezadáte, použijí se tyto implicitní hodnoty, tj. např. `trim` bude mít hodnotu 0. Všimněte si, že jednotlivé parametry jsou oddělené čárkou `(,)`.
- **Vysvětlení parametrů** (Arguments). V této části se vysvětluje, co parametr představuje a v jaké datové struktuře má být parametr uložený.
- **Hodnota funkce** (Value) je oddíl, kde se vysvětluje, jaké hodnoty funkce vrací, co znamenají a v jaké datové struktuře je hodnota funkce uložená.
- **Odkazy na literaturu** (References).
- **Odkazy na jiné funkce, balíky nebo data** (See Also) uvádějí seznam funkcí, které nějak souvisejí s funkcemi, jejichž dokumentaci právě čtete.
- **Příklady použití funkcí** (Examples) ukazují, jak funkci použít. Často pomáhají pochopit, jak funkce funguje a jak ji použít.

Pokud dokumentaci otevřete v RStudiu, budou všechny odkazy klikací.

V R nemají dokumentaci jen jednotlivé funkce a datasety, ale i celé balíky. Dokumentaci k balíkům včetně seznamu funkcí, které jsou v ní obsažené, je možné získat dvěma způsoby. Buď v konzoli zavoláte funkci `help()` s parametrem `package`

```
help(package = "dplyr")  # v uvozovkách je jméno balíku
```

nebo v RStudiu v záložce Packages kliknete na jméno zvoleného balíku. Dokumentace k balíkům ukazuje, co vše daný balík nabízí. Obsahuje zejména soubor `DESCRIPTION`, který vysvětluje, kdo balík napsal, k čemu

slouží a na jakých jiných balících závisí. Často obsahuje i odkaz na webovou stránku, kde se o daném balíku můžete dozvědět více. Dále může dokumentace balíků obsahovat seznam vinět a seznam funkcí a dat, které vám balík poskytuje.

Viněty jsou texty, které se nezaměřují na jednotlivé funkce, nýbrž ukazují, jak se balík používá jako celek, nebo vysvětlují nějaký princip, na kterém balík stojí. Seznam vinět přítomných v daném balíku je možné zobrazit v dokumentaci balíku kliknutím na "User guides, package vignettes and other documentation" nebo v konzoli pomocí funkce `vignette()`:

```
vignette(package = "dplyr") # v uvozovkách je jméno balíku
```

Jednotlivou vinětu můžete zobrazit zobrazit kliknutím na její jméno v seznamu v RStudiu, nebo můžete v konzoli zadat jméno viněty do funkce `vignette()`:

```
vignette("introduction", package = "dplyr") # první parametr je jméno viněty
```

Mnoho funkcí a balíků má k dispozici i demonstrační kód. Tyto kódy ukazují možnosti použití balíku nebo funkce nebo jejich obecné schopnosti. Tento kód můžete spustit takto:

```
demo("graphics") # parametr funkce je téma / jméno demonstrace
demo("bench-set", package = "dplyr") # pokud není balík načtený
```

Pokud chcete zjistit, jaké demonstrace obsahuje nějaký balík, zadejte

```
demo(package = "dplyr") # v uvozovkách je jméno balíku
```

a RStudio otevře záložku se jmény demonstrací přítomných v daném balíku. Pak vyvoláte demonstraci obvyklým způsobem.

2.5 Kde najít pomoc

Někdy dokumentace R nestačí. To se děje typicky ve chvíli, když nevíte, v jaké funkci a v jakém balíku je implementovaná nějaká metoda, kterou potřebujete, nebo ve chvíli, když dostanete chybovou hlášku, které nerozumíte. Když se do takové situace dostanete, zeptejte se Googlu.

Řekněme například, že potřebujete zjistit, jak neparametricky porovnat shodu střední hodnoty, ale nevíte, kterou funkci použít; navíc si ani nemůžete vzpomenout, jaký statistický test provést. Stačí, když Googlu položíte tento (nesprávný) dotaz: "r non-parametric t test". Uvidíte, že dostanete několik odkazů, které budou naprosto dokonale relevantní.

Někdy jste však v situaci, kdy si opravdu nevíte rady a potřebujete položit dotaz živému člověku. K tomu slouží celá řada fór. Z nich nejdůležitější je StackOverflow (https://stackoverflow.com/).

2.6 Konzola

R umožňuje práci ve dvou režimech: interaktivní práci a spouštění skriptů. K interaktivní práci s R slouží konzola (Console). Všechny výrazy, které sem napíšete za prompt (zobák, ">"), se okamžitě vyhodnotí a výsledky se vypíší zpět do konzole, tj. "na obrazovku". Konzola tak slouží jako kalkulačka. Zkuste do ní zapsat následující výrazy a každý z nich "odeslat" klávesou `Enter`. V této knize je výstup kódu v textu uvozen dvěma křížky (`##`).

```
2.3 + 3 * 4      # výraz se okamžitě vyhodnotí
```

```
## [1] 14.3
```

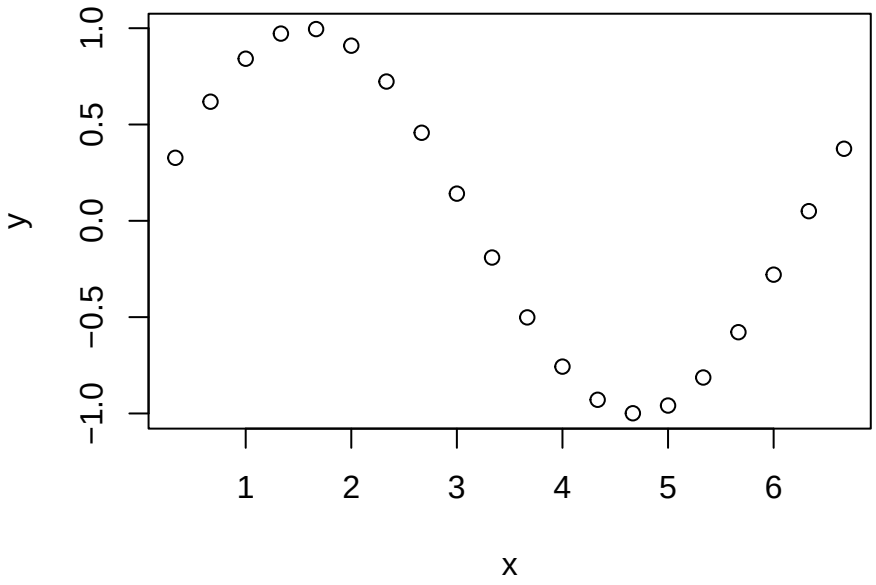
```
x <- (1:20) / 3  # přiřazení hodnot do proměnné x (nic nevypíše)
print(x)         # vypsání hodnot proměnné x
```

```
##  [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.0000000 2.3333333
##  [8] 2.6666667 3.0000000 3.3333333 3.6666667 4.0000000 4.3333333 4.6666667
## [15] 5.0000000 5.3333333 5.6666667 6.0000000 6.3333333 6.6666667
```

```
x                # totéž, co print(x)
```

```
##  [1] 0.3333333 0.6666667 1.0000000 1.3333333 1.6666667 2.0000000 2.3333333
##  [8] 2.6666667 3.0000000 3.3333333 3.6666667 4.0000000 4.3333333 4.6666667
## [15] 5.0000000 5.3333333 5.6666667 6.0000000 6.3333333 6.6666667
```

```
y <- sin(x)      # vyhodnocení funkce a přiřazení do proměnné y  (nic nevypíše)
plot(x, y)       # vykreslení hodnot
```




```
rm(x, y)         # vymazání proměnných x a y
# vše za symbolem křížku až do konce řádku je komentář -- R to ignoruje
```

Všimněte si, že k přiřazení hodnoty do proměnné se v R používá značka `<-` a k oddělení desetinných míst čísla se používá desetinná tečka, zatímco čárka slouží k oddělení jednotlivých parametrů funkce. Pokud tedy napíšete

```
median(2,5)
```

```
## [1] 2
```

funkce nespočítá medián z hodnoty 2.5, nýbrž medián z hodnoty 2; číslo 5 se funkci předá jako její druhý parametr, což je zde `na.rm`. Obecně platí, že první hodnota zadaná do funkce představuje její první parametr, druhá hodnota druhý parametr atd. Parametry, které mají implicitní hodnoty, můžete vynechat. Víc se o těchto věcech dozvíte v oddíle 8.3.

V RStudiu má konzola dvě užitečné klávesové zkratky: šipky nahoru a dolů umožňují procházet historií minulých výrazů. Pokud tedy zadáte a spustíte výraz, můžete jej spustit znovu tak, že zmáčknete šipku nahoru a `Enter`. Ke složitějšímu vyhledávání slouží `Ctrl`–šipky. Ty prochází seznam historie tak, že uvažují jen ty výrazy, které začínají stejně jako to, co jste právě napsali na prompt konzoly.

2.7 Skripty

Konzola slouží k experimentování a ladění kódu. Ke skutečné práci však slouží skripty, které vám umožní spouštět kód opakovaně a při tom jej měnit a ladit. Za výsledek své práce byste nikdy neměli považovat výsledky interaktivního hraní v konzoli, ať už ve formě spočítaných výsledků nebo vykreslených grafů, ale právě funkční skript. Pokud se ke své práci po čase vrátíte, nedokážete z "výsledků" nijak zjistit, jak jste k nim dospěli (a jestli jste v průběhu neudělali chybu), ani něco v již proběhlém výpočtu změnit. Naproti tomu skript jasně říká, co jste udělali, umožňuje, abyste svůj výpočet upravili a samozřejmě umožňuje i znovu získat jakékoli výsledky, ke kterým jste dospěli. Měli byste si tedy zvyknout pracovat primárně v editoru a konzoli používat jen k testování dílčích kusů kódu.

R skript je obyčejný textový soubor, do kterého napíšete R-kové výrazy jeden za druhý – každý nový výraz na nový řádek. Když pak skript spustíte, tyto řádky se provedou úplně stejně, jako byste je napsali přímo do konzoly. Jedinou výjimkou je část řádku za znakem křížku (`#`), která se považuje za komentář – R tuto část řádku ignoruje.

Skripty je zvykem ukládat do souborů s koncovkou `.R`. Soubor se skriptem můžete vytvořit v jakémkoli textovém editoru, který k textu nepřidává žádné značky, tj. např. ne v MS Wordu. RStudio však poskytuje velmi dobrý editor, který umí barevně zvýraznit syntaxi, odhalit některé chyby, napovědět vám, jak se funkce jmenuje a jaké má parametry atd. Nový skript vytvoříte v RStudiu klávesovou zkratkou `Ctrl-Shift-N` nebo v menu `File→New File→R Script`.

Kvůli ladění chyb i kvůli čitelnosti kódu je dobré skripty pěkně formátovat. Doporučuji dodržovat některý z následujících stylů:

- styl Hadleyho Wickhama http://adv-r.had.co.nz/Style.html nebo
- styl Googlu https://google.github.io/styleguide/Rguide.xml.

RStudio vás dokáže upozornit na špatný styl, pokud si tuto volbu zapnete, a umí i částečně váš skript přeformátovat do pěknějšího (v menu `Code→Reformat code`); pomáhá také možnost automaticky odsadit řádky kódu (v menu `Code→Reindent lines`) a pěkně zarovnat komentáře (v menu `Code→Reflow comments`).

Jednou napsaný skript můžete spouštět znovu a znovu. Skript je možné spustit třemi způsoby. V RStudiu k tomu slouží klikátko `Source` v pravém horním rohu editoru a klávesová zkratka `Ctrl-Shift-S`. Skript jde samozřejmě spustit i z konzoly nebo jiného skriptu pomocí funkce `source()`:

```
# jméno souboru do uvozovek
source("jmeno_skriptu_a_cesta_k_němu")
```

Funkce `source()` má mnoho dalších parametrů, viz dokumentace. Užitečný je zejména logický parametr `echo`, který ovlivňuje, zda se při spuštění skriptu vypisují do konzoly výrazy, které se právě vyhodnocují. Klikátko `Source` v RStudiu má podobnou volbu.

Někdy nechceme spustit celý skript naráz, ale chcete spouštět jen jednotlivé řádky kódu. Aktuální řádek nebo skupinu vybraných řádků spustíte klávesovou zkratkou `Ctrl+Enter`. Další možnosti spuštění skupin řádků a jejich klávesové zkratky najdete v menu `Code`.

I když skript může obsahovat všechny platné výrazy jazyka R, některé z nich není vhodné do skriptu umisťovat. Mezi takové výrazy patří zejména funkce, které mění vnější prostředí R. Ve skriptu zejména nikdy neinstalujte balíky pomocí funkce `install.packages()` a raději ani neměňte pracovní adresář pomocí funkce `setwd()`. Je bezohledné, pokud s někým sdílíte svůj skript a v něm voláte takové funkce, protože tím měníte nastavení cizího počítače.

Naproti tomu nesmíte zapomenout ve skriptu načíst balíky, které používáte, pomocí funkce `library()`. Je rozumné, abyste balíky načetli vždy na začátku skriptu. Díky domu snadno dohledáte, které balíky skutečně voláte (a které tedy musíte mít nainstalované).

Pokud jste nastavili RStudio tak, jak jsme uvedli v oddíle 2.2, pak vám RStudio bude poskytovat velmi užitečné diagnostické rady pomocí ikonek zobrazených vlevo od čísel řádků kódu. Pokud na ikonku "najedete myší", zobrazí se diagnostická rada. Pro větší přehlednost RStudio vlnovkou podtrhne vyhodnocovaný kus kódu.

Červená barva signalizuje chybu, žlutá varování a modrá upozornění na porušení stylistických konvencí, které sice nezabrání běhu vašeho kódu, ale znesnadňují jeho čtení.

Ukažme si použití skriptu na příkladů převzatém z knihy Radové a Dvořáka: *Finanční matematika pro každého*, Grada, 1993, s. 92–95. Neděste se, že zatím nebudete rozumět všemu, co kód dělá – postupně se všechno potřebné dozvíte v této knize. Předkládejme, že musíme splácet dluh ve výši $D = 40\,000$ pomocí $n = 6$ stejných ročních splátek, tj. anuit. Úroková sazba je $i = 12\,\%$. Kniha nám říká, že výši anuity A, úrokové platby U a úmoru M (anuita je úroková platba plus úmor) můžeme spočítat podle následujících vztahů, kde t je čas:

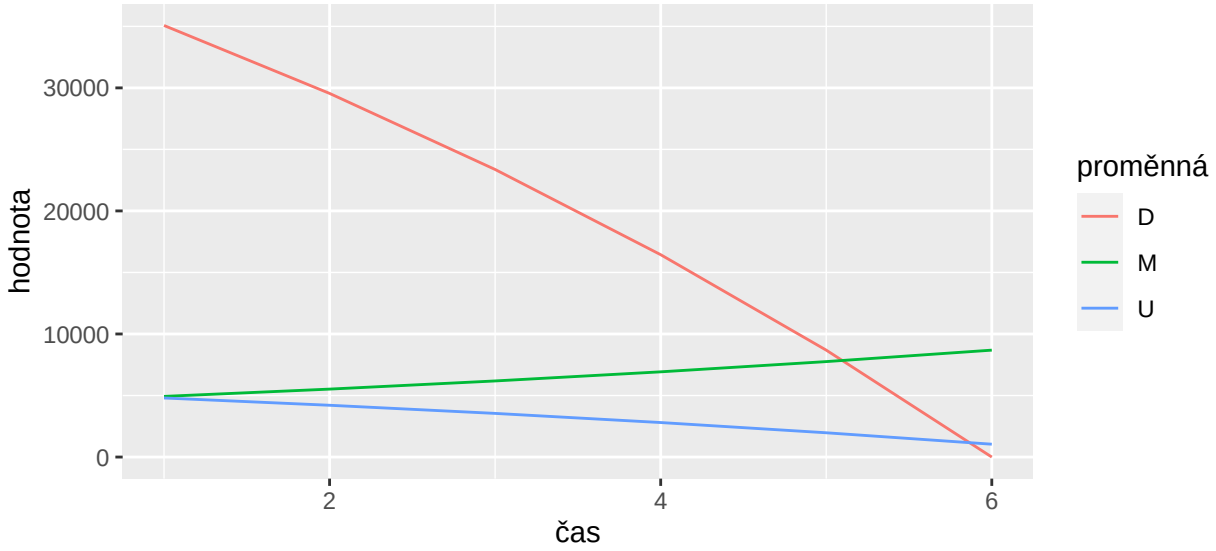
$$v = 1/(1+i), \tag{2.1}$$
$$a = Di/(1-v^n), \tag{2.2}$$
$$U_{t+1} = a(1-v^{n-t}), \tag{2.3}$$
$$M_{t+1} = a(1-v^{n-t}), \tag{2.4}$$
$$D_t = D - \sum_{i<t} M_i. \tag{2.5}$$

Spočítat výši anuity a vývoj úrokové platby, úmoru a zbývajícího dluhu znamená jen přepsat matematické výrazy do R kódu. Zkuste nejprve následující kód zapsat přímo do konzoly. Zjistíte, že to není právě pohodlné. Pokud uděláte chybu, nemusíte přesně vědět, odkud začít: pokud např. na řádku `Dz <- D - cumsum(M)` napíšete omylem `D` místo `Dz`, přepíšete si původní hodnotu. A pokud se nakonec rozhodnete změnit výši dluhu např. na 1 milion, budete muset všechny řádky spustit ručně znovu. Pokud však kód napíšete do skriptu, můžete kdykoli cokoli opravit, a to včetně výše dluhu, úrokové sazby nebo počtu období (vyzkoušejte si to).

```
# vyčisti pracovní prostředí
rm(list = ls())
# načti potřebné balíky
library(ggplot2)
library(tidyr)
# zadej konstanty
D <- 40000  # výše dluhu
i <- 12 / 100  # úroková sazba
n <- 6  # počet období
# vlastní výpočet
t <- 1:n  # vektor času
v <- 1 / (1 + i)  # diskontní faktor
a <- D * i / (1 - v ^ n)  # anuita
U <- a * (1 - v ^ (n - t + 1))  # vektor úrokových plateb v čase t
M <- a * v ^ (n - t + 1)  # vektor úmorů v čase t
Dz <- D - cumsum(M)  # vektor zbývajícího dluhu v čase t
df <- round(data.frame(t = t, U = U, M = M, D = Dz), 2)
# výpis výsledku v tabulce
print(df)
```

```
##   t       U       M        D
## 1 1 4800.00 4929.03 35070.97
## 2 2 4208.52 5520.51 29550.46
## 3 3 3546.06 6182.97 23367.49
## 4 4 2804.10 6924.93 16442.55
## 5 5 1973.11 7755.92  8686.63
## 6 6 1042.40 8686.63     0.00
```

```
# vykreslení vývoje veličin do grafu
ggplot(gather(df, proměnná, hodnota, -t),
       aes(x = t, y = hodnota, color = proměnná)) +
    geom_line() +
    xlab("čas")
```



Všimněte si, že ve skriptu načítáme všechny potřebné balíky, ale neinstalujeme je!

2.8 Jak se R učit

Nakonec několik rad, jak se učit s R pracovat.

1. Nesnažte se zapamatovat všechny detaily volání každé funkce – to snadno najdete v dokumentaci. Snažte se spíše pochopit princip, jak věci fungují.
2. Naučte se číst dokumentaci funkcí – a skutečně ji čtěte. Když narazíte na neznámou funkci, přečtěte si pozorně dokumentaci a vyzkoušejte příklady na jejím konci.
3. Hrajte si a zkoušejte věci. Vymyslete si vlastní problém a zkuste jej vyřešit.
4. Ke kódu přistupujte "experimentálně". Pokud nevíte, jak něco funguje, vytvořte si hypotézu a vymyslete experiment, jak ji ověřit. Vždy přemýšlejte, jak a proč něco funguje.
5. Zkoušejte příklady, na které narazíte. Kód z příkladů raději opište než kopírujte `Ctrl-C Ctrl-V`. Když kód opíšete, něco si tím zapamatujete. Než kód spustíte, odhadněte, co udělá, a proč. Pak svůj odhad porovnejte se skutečností.
6. Až se naučíte základy, zkuste číst zdrojový kód cizích funkcí, které používáte. Tak se naučíte hodně o tom, jak pracovat (i o tom, co raději nedělat).
7. Nepropadejte panice! Zvládnete to.
8. Když všechno selže, je tu Google a StackOverflow.

3 Proměnné

Málo kdy pracujeme s daty přímo jako s čísly. V případě velkých dat by to ani nebylo možné. Místo toho svá data uložíme do proměnných a dále pracujeme s těmito proměnnými. Kromě usnadnění práce to má ještě jednu výhodu: naše skripty jsou tak obecnější. Můžeme např. napsat analytický kód s pomocí dat z pilotního průzkumu, a pak jej spustit znovu ve chvíli, kdy doběhl celý sběr dat, s novými daty. Vyměníme jen datový soubor a jinak nemusíme svůj kód vůbec měnit.

V této kapitole se naučíte

- co jsou proměnné a k čemu slouží
- jaká jména proměnných jsou povolená (a jak tuto restrikci obejít)
- jak přiřadit do proměnné hodnotu
- jak vypsat obsah proměnné
- jak proměnnou smazat
- jak zjistit, jaká metadata proměnná obsahuje, a jak je změnit

3.1 K čemu slouží proměnné

Koncept proměnné znáte z matematiky. Tam je proměnná "krycí název" pro nějakou hodnotu, např. pro číslo 5. Krása proměnných spočívá v tom, že umožňují počítat zcela obecně. Pokud např. označíme délku trasy v kilometrech písmenem s a rychlost jízdy v kilometrech za hodinu písmenem v, pak víme, že vzdálenost s ujedeme za s/v hodin. Tento výsledek platí bez ohledu na to, jak daleko a jak rychle jedeme. Jak už naznačuje název proměnná, můžeme hodnotu proměnné měnit, a výraz s/v vyhodnotit pokaždé znovu. Pro jednoduchý výpočet, který jsme právě uvažovali, nemusí být takové zobecnění příliš užitečné, ale při složitých výpočtech nad velkými daty je zavedení proměnných velká pomoc. V počítači plní proměnné stejnou úlohu jako v matematice: uchovávají nějakou hodnotu. Můžeme počítači říct, co má s touto hodnotou dělat, bez toho, abychom přesně věděli, jaká tato hodnota je.

I když to technicky není přesné, můžeme si proměnnou představit jako krabičku, do které se vloží nějaká hodnota. Každá proměnná může v jednom okamžiku obsahovat vždy jen jednu hodnotu. Pokud do proměnné uložíme novou hodnotu, stará hodnota se ztratí. Slovo "hodnota" je však třeba brát volně: "hodnota" může být stejně dobře jedno číslo jako složitá struktura složená z čísel, textů a jiných objektů. (Technicky přesnější je říct, že proměnná jméno, které se odkazuje na nějakou oblast v paměti počítače, kde jsou uložena naše data. Aby počítač věděl, jak s daným kusem paměti zacházet, musí vědět, jakého typu jsou uložená data, např. zda se jedná o celá čísla nebo o text, a v jaké datové struktuře jsou data uložena, např. ve vektoru.) R je volně typovaný jazyk. To znamená, že typ proměnné nemusíte nijak deklarovat a R jej samo odhadne podle vložené hodnoty. Do proměnné můžete v jedné chvíli uložit číslo a později toto číslo nahradit třeba kusem textu. R si s tím poradí. (Proměnná bude prostě nejdříve odkazovat na jedno místo v paměti, a pak na jiné. Nepoužívanou oblast paměti R samo uvolní.)

3.2 Jména proměnných

Jména proměnných musí splňovat určité vlastnosti. Jméno proměnné se může skládat jen z písmen, číslic, teček a podtržítek a musí začínat písmenem nebo tečkou, za kterou nenásleduje číslice. Jména `a`, `a2`, `myNumber`, `my_number`, nebo `.my.way` jsou přípustná; jména jako `2way`, `.2way` nebo `my-number` nejsou povolená. Stejně tak nejsou povolená rezervovaná slova: `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`,

`TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_` a `NA_character_`. Ve jménech proměnných záleží na velikosti písmenem, takže `x` a `X` jsou dvě různé proměnné.

Jméno proměnné by ideálně mělo být stručné a mělo by výstižně popisovat, jakou hodnotu proměnná obsahuje. Pokud se jméno skládá z více slov, slova se historicky oddělovala tečkami (např. `pv.of.bond`). V současné době se to nedoporučuje, protože tečky se používají i k oddělení generické a specifické části jmen objektových metod, viz kapitola 9. Místo toho se doporučuje používat podtržítka (`pv_of_bond`) nebo případně tzv. "Camel Case" standard (`pvOfBond`), který však v R není příliš obvyklý.

Pokud si nejste jistí, jak jméno proměnné sestavit, může vám pomoci funkce `make.names()`, která převede zadaný řetězec na syntakticky platné jméno proměnné, ovšem bohužel s tečkami:

```
make.names("pv of bond")
```

```
## [1] "pv.of.bond"
```

Někdy je potřeba pracovat s proměnnou, jejíž jméno není v R povoleno. (Taková situace nejčastěji vznikne při importu dat z jiného softwaru.) Proměnnou s nelegálním jménem můžete použít, pokud její jméno uzavřete mezi dva zpětné apostrofy ("backticks"). Jméno proměnné nemůže obsahovat (mimo jiné) mezeru. Pomocí zpětných apostrofů však můžete pracovat i s proměnnou, jejíž jméno mezeru obsahuje:

```
`ahoj lidičky!` <- 5
2 * `ahoj lidičky!`
```

```
## [1] 10
```

I když je možné používat i "nedovolená" jména proměnných, výrazně to nedoporučujeme.

3.3 Přiřazení hodnoty do proměnné

Hodnoty se do proměnných přiřazují pomocí "šipky" `<-`, kde šipka vždy ukazuje ke jménu proměnné, zatímco na druhé straně je výraz, který má R vyhodnotit. V RStudiu lze šipku vložit pomocí klávesové zkratky `Alt-`. (Funguje i šipka `->` otočená opačným směrem. Tuto kuriozitu však používá jen p. Mikula a výrazně vám nedoporučuji ji používat, protože při čtení kódu zjistíte, co se s daným výrazem stane, až na konci možná velmi dlouhé konstrukce.) Někteří lidé používají k přiřazení do proměnné i symbol rovnítka (`=`). To nedoporučuji. Rovnítko v některých kontextech funguje jako synonymum šipky, zatímco v jiných ne (tam rovnítko znamená něco jiného). Použití rovnítka k přiřazení je tak matoucí. Detaily najdete na https://goo.gl/Tnu8Q5.

Příklad:

```
x <- (2 + 3) * 4  # x má nyní hodnotu 20
x  # jak se vzápětí přesvědčíme
```

```
## [1] 20
```

3.4 Vypsání hodnoty proměnné do konzoly

Při přiřazení hodnoty do proměnné se výsledek nevypíše. Pokud jej chcete vypsat, musíte o to R požádat. To můžete udělat třemi způsoby: 1) explicitně vypsat obsah proměnné pomocí funkce `print()`, 2) implicitně vypsat obsah proměnné tak, že napíšete její jméno do konzoly (R volá implicitně funkci `print()` za vás) nebo 3) tak, že celý výraz přiřazení zabalíte do závorek.

```r
x <- "This is some text."  # hodnota se přiřadí, nic se nevypíše
x         # implicitní vypsání hodnoty proměnné x
```

```
## [1] "This is some text."
```

```r
print(x)  # explicitní vypsání hodnoty proměnné x
```

```
## [1] "This is some text."
```

```r
(y <- 2)  # výraz se vyhodnotí a hodnota implicitně vypíše
```

```
## [1] 2
```

Implicitní forma vypsání obsahu proměnné je vhodná pro interaktivní práci v konzoli, nemusí však fungovat uvnitř funkcí a skriptů, protože ve skutečnosti jen žádáte R o vrácení hodnoty proměnné. Podle kontextu může být vrácená proměnná využita různě. V konzoli se využije tak, že konzola na hodnotu zavolá funkci `print()`. Uvnitř funkce však může být vrácená hodnota použitá funkcí jinak. Uvnitř funkcí a podobných struktur je tedy třeba obsah proměnné vypsat explicitně pomocí funkce `print()`.

To, jak R vypíše obsah proměnné, se může lišit od skutečného obsahu dané proměnné. R totiž pro různé objekty volá různé varianty funkce `print()` přizpůsobené těmto objektům a může vypsat více nebo méně informací, než je jich v dané proměnné obsaženo. To, jak vypisuje data pro základní datové typy a struktury se dozvíte v následujících dvou kapitolách.

Někdy proměnná obsahuje mnoho hodnot (např. dlouhý vektor, tabulku s mnoha řádku apod.) a my ji nechceme vypsat celou, nýbrž z ní chceme získat jen nějakou ukázku, typicky několik prvních nebo posledních hodnot. Několik prvních hodnot vrací funkce `head()`, posledních hodnot funkce `tail()`. Obě vrací implicitně 6 hodnot (prvků vektoru, řádků matice apod.); tento počet lze změnit nastavením parametru `n`:

```r
x <- matrix(1:1200, ncol = 3)  # vytvoří matici se 400 řádky
head(x)                        # vypíše 6 prvních řádků matice
```

```
##      [,1] [,2] [,3]
## [1,]    1  401  801
## [2,]    2  402  802
## [3,]    3  403  803
## [4,]    4  404  804
## [5,]    5  405  805
## [6,]    6  406  806
```

```r
head(x, n = 3)                 # vypíše 3 první řádky matice
```

```
##      [,1] [,2] [,3]
## [1,]    1  401  801
## [2,]    2  402  802
## [3,]    3  403  803
```

```r
tail(x)                        # vypíše 6 posledních řádků matice
```

```
##        [,1] [,2] [,3]
## [395,]  395  795 1195
## [396,]  396  796 1196
## [397,]  397  797 1197
## [398,]  398  798 1198
## [399,]  399  799 1199
## [400,]  400  800 1200
```

Pokud pracujete v RStudiu, můžete použít k zobrazení obsahu proměnné i funkci `View()`. Tato funkce otevře novou záložku a zobrazí obsah proměnné. Způsob zobrazení závisí na datové struktuře. Atomické vektory, matice a tabulky zobrazí ve formě interaktivní tabulky, která umožňuje hodnoty třídit a filtrovat. Seznamy a objekty postavené nad seznamy se zobrazí podobně, jako je vypisuje funkce `str()`. Funkci `View()` je možné vyvolat i pomocí myši tak, že v záložce `Environment` kliknete na ikonku tabulky (tabulární pohled) nebo trojúhelníku vedle jména proměnné. Pozor: seznam proměnných musí být v režimu "List".

3.5 Atributy (metadata)

Proměnné v R obsahují kromě vlastních hodnot také metadata (informace o datech). V R se metadata nazývají atributy.

Funkce `attributes()` vypíše seznam všech atributů dané proměnné.

```
x <- c(a = 1, b = 2, c = 3)  # vektor s pojmenovanými prvky
x
```

```
## a b c
## 1 2 3
```

```
attributes(x)
```

```
## $names
## [1] "a" "b" "c"
```

```
X <- matrix(1:12, nrow = 3)  # matice má počet řádků a sloupců
X
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
attributes(X)
```

```
## $dim
## [1] 3 4
```

Atributy proměnných mohou zahrnovat třídu objektu, dimenze proměnných (počet řádků, sloupců a případně dalších rozměrů objektu), jména řádků, sloupců, jednotlivých prvků vektorů a případně další informace.

Hodnotu jednoho atributu je možné získat funkcí `attr()`; tuto funkci je zároveň možné použít i ke změně hodnoty atributu (ve skutečnosti se volá jiná funkce, syntaxe však vypadá stejně):

```
attr(x, "names")
```

```
## [1] "a" "b" "c"
```

```
attr(x, "names") <- c("Ahoj", "Bum", "Cak")
attr(x, "names")
```

```
## [1] "Ahoj" "Bum"  "Cak"
```

```
x
```

```
## Ahoj  Bum  Cak
##    1    2    3
```

Pokud se zeptáte na hodnotu atributu, který není v proměnné přítomen, funkce `attr()` vrací hodnotu `NULL`:

```
attr(x, "coriandr")
```

```
## NULL
```

Atribut zrušíte tak, že do něj přiřadíte hodnotu `NULL`.

```
attr(x, "names") <- NULL
x
```

```
## [1] 1 2 3
```

3.6 Smazání proměnné

Ke smazání proměnné z aktuálního prostředí slouží funkce `rm()`:

```
rm(x)  # smaže proměnnou x
rm(x, y, z)  # smaže proměnné x, y a z
rm(list = ls())  # smaže všechny proměnné z aktuálního prostředí
```

V RStudiu můžete proměnné mazat i myší. V záložce `Environment` k tomu slouží ikonka koštěte. Pokud máte proměnné zobrazené v režimu `Grid`, můžete vybrat, které proměnné budou smazány. V režimu `List` budou smazány všechny proměnné.

Někdy je však potřeba vyčistit paměť R důkladněji. Smazání všech proměnných totiž stále zanechá v R mnoho změn: načtené balíky, změněné cesty apod. Nejdůkladnější vyčištění pracovního prostředí přestavuje restart R. V RStudiu je to možné udělat v menu `Session→Restart R` nebo pomocí klávesové zkratky `Ctrl-Shift-F10`. Pokud si chcete ověřit, že váš skript běží spolehlivě, vždy byste jej měli vyzkoušet v čistém prostředí R, tj. po jeho restartu.

3.7 Aplikace: výpočet dojezdové vzdálenosti

Jako příklad práce s proměnnými vytvoříme jednoduchý skript, který spočítá, za jak dlouho dorazíme do cíle. Nejdříve si vyčistíme pracovní prostředí smazáním všech dříve vytvořených proměnných (což je velmi užitečný zvyk). Následně vytvoříme proměnné vzdálenost a rychlost, pak vypočteme dobu jízdy a vypíšeme ji. Náš skript bude tedy vypadat takto:

```
rm(list = ls())
vzdalenost <- 1079  # vzdálenost z Brna do Dubrovniku v km
rychlost <- 90  # očekávaná průměrná rychlost v km / h
doba_jizdy <- vzdalenost / rychlost
print(doba_jizdy)
```

Pokud jste svůj skript napsali do nového okna editoru v RStudiu, spustíte jej jednoduše stisknutím tlačítka `Source`. Výsledkem by mělo být, že pojedete necelých 12 hodin. To byste mohli mnohem jednodušeji zjistit i s pomocí obyčejné kalkulačky. Skript vám však umožní zjistit i to, co se stane, pokud pojedete rychleji nebo pomaleji: stačí jen změnit rychlost na třetím řádku a skript spustit znovu. Podobně můžete změnit i vzdálenost na druhém řádku a zjistit, jak dlouho pojedete do jiné destinace. (V kapitole 8 se naučíte proces, který vám umožní snadno měnit parametry výpočtu jako jsou rychlost a vzdálenost ještě více automatizovat pomocí tvorby vlastních funkcí.)

4 Základní datové typy

Každá hodnota uložená v proměnné má nějaký datový typ. Abyste porozuměli chování R, musíte se s vlastnostmi a možnostmi jednotlivých datových typů seznámit.

V této kapitole se naučíte

- jaké jsou základní datové typy používané v datové analýze
- jaký je význam speciálních hodnot `NA`, `NaN`, `Inf` a `NULL`
- jak poznáte, jakého typu je která hodnota
- jak je možné převádět data z jednoho typu na druhý
- jak používat základní aritmetické a logické operátory

4.1 Základní datové typy

Data se skládají z jednotlivých hodnot. Každá taková hodnota má určitý datový typ. V R existuje mnoho základních typů dat. Pro datovou analýzu se však hodí především následující čtyři datové typy:

- **logical** může obsahovat jen dvě logické hodnoty: `TRUE` ("pravda") a `FALSE` ("nepravda"). Tyto hodnoty je možné zkrátit na `T` a `F`, ale výrazně se to nedoporučuje, protože zkrácená jména `T` a `F` je možné předefinovat, po čemž by původně funkční kód dělal nepředvídatelné věci.
- **integer** může obsahovat kladná i záporná celá čísla. Pokud je chcete zadat, musíte za číslo napsal `L`, tj. např. `1L`.
- **double** může obsahovat kladná i záporná reálná čísla. Když zadáte v konzoli nebo skriptu číslo bez příznaku `L`, bude v paměti reprezentované jako double, i když to bude shodou okolností celé číslo. V konzoli však R vypíše celá čísla inteligentně bez desetinných míst, i když jsou uložená v proměnné typu *double*.
- **character** může obsahovat libovolný řetězec (text). V R se řetězce zadávají mezi dvěma uvozovkami nebo apostrofy. Uvozovky a apostrofy nelze kombinovat, tj. nejde jeden řetězec uvést uvozovkou a ukončit apostrofem. To umožňuje zadat apostrofy nebo uvozovky jako součást řetězce tak, že je např. řetězec uvozen i ukončen uvozovkami a uvnitř řetězce je použit apostrof. Jinou možností, jak zadat uvozovky, apostrofy a jiné zvláštní znaky, je "escapovat" je, tj. napsat je pomocí zpětného lomítka `\` a vybraného znaku (např. `\"` znamená uvozovku, `\n` konec řádku apod.).

```
x1 <- TRUE  # logická hodnota
x1
```

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

```
x2 <- 1L  # celé číslo
x2
```

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

```
x3 <- 1  # reálné číslo
x3
```

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

```
x4 <- 'Josef řekl: "Miluji R!"'  # řetězec
x4
```

```
## [1] "Josef řekl: \"Miluji R!\""
```

Při zadávání reálných čísel se desetinná místa oddělují *tečkou*, ne čárkou.

```
1.3
```

```
## [1] 1.3
```

Čárky oddělují parametry ve funkcích – jinde způsobí chybovou hlášku!

```
1,3
```

```
## Error: <text>:1:2: unexpected ','
## 1: 1,
##      ^
```

Reálná čísla jde zadat i pomocí tzv. "vědecké notace", kde číslo před `e` je mantisa, číslo za `e` dekadický exponent:

```
1.3e3   # 1.3 krát 10 na třetí, tj. 1 300
```

```
## [1] 1300
```

```
2.7e-5  # 2.7 krát 10 na minus pátou, tj. 0.000027
```

```
## [1] 2.7e-05
```

R někdy reálná čísla takto samo vypisuje. Pokud chcete ovlivnit, jak bude reálné číslo vypsané, můžete použít funkci `format()` (víc parametrů viz nápověda funkce):

```
format(2.7e-5, scientific = FALSE)
```

```
## [1] "0.000027"
```

Mezi proměnnou typu *integer* a *double* je několik rozdílů. Hlavní z nich se týká přesnosti: typ *integer* sice umí zahrnout jen celá čísla, reprezentuje je však naprosto přesně. Naproti tomu typ *double* umí zahrnout i desetinná čísla (a také velmi velká čísla), reprezentuje je však pouze přibližně, takže v následujícím výpočtu vznikne chyba, i když velmi malá:

```
sqrt(2) ^ 2 - 2  # odmocnina 2 umocněná na druhou minus 2
```

```
## [1] 4.440892e-16
```

4.2 Testování datového typu

R umožňuje otestovat, jaký datový typ má zvolená proměnná, pomocí funkcí `is.X()`, kde `X` je daný datový typ. Tyto funkce vrací `TRUE`, pokud je daná proměnná daného datového typu. Existuje i funkce `is.numeric()`, která vrací hodnotu `TRUE` v případě, že proměnná je číselná, ať už celočíselná nebo reálná.

Funkce `typeof()` vrací datový typ proměnné jako řetězec (např. `"logical"`). Podobná, ale zdaleka ne stejná, je funkce `class()`, která vrací třídu objektu z hlediska objektově orientovaného programování. Pro atomické vektory však vrací typ proměnných.

```
typeof(x1)
```

```
## [1] "logical"
```

```
is.logical(x1)
```

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

```
is.numeric(x1)
```

```
## [1] FALSE
```

```
typeof(x2)
```

```
## [1] "integer"
```

```
is.integer(x2)
```

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

```
is.numeric(x2)
```

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

```
typeof(x3)
```

```
## [1] "double"
```

```
is.integer(x3)
```

```
## [1] FALSE
```

```
is.double(x3)
```

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

```
is.numeric(x3)
```

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

```
typeof(x4)
```

```
## [1] "character"
```

```
is.character(x4)
```

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

Podobné, ale poněkud přísněji se chovající testy nabízí balík **purrr**, např.:

```
purrr::is_integer(x1)
```

```
## [1] FALSE
```

4.3 Chybějící a "divné" hodnoty

V některých případech může proměnná obsahovat příznak, že její hodnota chybí nebo je chybná. R k tomu má tři speciální hodnoty:

- `NA` je chybějící hodnota ("not available"). Existuje pro všechny základní datové typy.
- `NaN` znamená, že numerická hodnota není číslo, ale je chybná ("not a number"). Tato hodnota existuje jen pro typ *double*.
- `Inf` označuje nekonečnou hodnotu; podobně `-Inf` zápornou nekonečnou hodnotu. Obě hodnoty také existují jen pro typ *double*.

Nekonečná hodnota vznikne např. při dělení nenulového čísla nulou:

```
1 / 0
```

```
## [1] Inf
```

Chybná hodnota vznikne při různých nepovolených matematických operacích, které však nevedou na nekonečno, např. při dělení nuly nulou:

```
0 / 0
```

```
## [1] NaN
```

Chybějící hodnoty `NA` se obvykle používají při zadávání hodnot v konzoli nebo ve skriptu a při ukládání čísel do souboru, aby se označilo, která hodnota chybí. Mohou však také být výsledkem výpočtu, když danou operaci není možné provést, ale nejedná se při tom o neplatné počítání s čísly, které by vedlo na `Inf` nebo `NaN`.

Existují testy, které testují, zda je hodnota proměnné `NA`, `NaN` nebo `Inf`, a které vracejí jako výsledek logickou hodnotu testu `TRUE` nebo `FALSE`: `is.na()`, `is.nan()`, `is.finite()` a `is.infinite()`. Funkce

`is.infinite()` vrací `TRUE` jak pro `Inf`, tak i pro `-Inf`; funkce `is.finite()` naopak. Pozor: funkce `is.na()` vrací `TRUE` jak pro `NA`, tak i pro `NaN`.

Stejně jako ostatní hodnoty v R, tak i chybějící hodnoty mají svůj typ. V celočíselné proměnné je tak `NA` ve skutečnosti reprezentované jako `NA_integer_`, zatímco v reálné proměnné jako `NA_real_` apod. Pokud byste vypsali obsah těchto dvou proměnných na obrazovku, uvidíte `NA`; pokud byste použili k otestování shody jejich obsahu funkci `identical()` (viz dále), zjistíte, že hodnoty nejsou stejné:

```
x1 <- c(1L, NA)[2]    # vezme se druhá hodnota celočíselného vektoru
x2 <- c(1, NA)[2]     # vezme se druhá hodnota reálného vektoru
x1
```

```
## [1] NA
```

```
x2
```

```
## [1] NA
```

```
identical(x1, x2)
```

```
## [1] FALSE
```

Velmi speciální hodnotou je `NULL`. `NULL` je speciální objekt (má vlastní datový typ) a zároveň rezervované slovo, které R vrací v situaci, kdy nějaká hodnota není definovaná nebo nějaký vektor neexistuje. `NULL` se často chová jako vektor nulové délky. K otestování, zda je hodnota objektu `NULL` slouží funkce `is.null()`.

4.4 Převody mezi datovými typy

V případě, že R potřebuje nějakým způsobem sladit dva základní datové typy (např. je spojit do jednoho atomického vektoru), provede R jejich automatickou konverzi a převede jednodušší typ na obecnější typ. Převod probíhá od logických proměnných k celočíselným (`TRUE` se převede na 1 a `FALSE` na 0), od celočíselných k reálným a od nich k řetězcům. Při automatické konverzi záleží na pořadí:

```
# funkce c() spojí hodnoty v závorkách do vektoru a převede je na společný typ
c(TRUE, 1L, 1, "1")
```

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

```
c(c(TRUE, 1L), 1, "1")
```

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

```
c(c(TRUE, 1L, 1), "1")
```

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

Automatické konverze lze někdy využít k zajímavým trikům. Pokud např. chceme sečíst počet případů, ve kterých platí nějaká podmínka, jde použít na logický vektor numerickou funkci pro součet hodnot prvků vektoru `sum()` a využít automatickou konverzi:

```
x <- c(1, 2, 3, 7, 19, 31)  # vytvoří vektor daných čísel
# kolik hodnot x je větší než 10?
sum(x > 10)
```

```
## [1] 2
```

Výraz `sum(x > 10)` se vyhodnotí postupně: nejdříve se vyhodnotí výraz `x > 10`, jehož výsledkem je logický vektor, kde je každé číslo větší než 10 nahrazeno `TRUE` a každé číslo menší rovno 10 nahrazeno `FALSE`. Ve druhém kroku R automaticky nahradí každé `TRUE` jedničkou a každé `FALSE` nulou. Ve třetím kroku sečte vektor jedniček a nul.

V některých situacích je třeba provést konverzi ručně. K tomu slouží funkce `as.X()`, kde `X` je jméno datového typu.

```
as.character(TRUE)
```

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

```
as.integer(TRUE)
```

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

```
as.logical(c(-1, 0, 0.1, 1, 2, 5))  # nula se převede na FALSE, ostatní čísla na TRUE
```

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

Překvapivě je možné převést i řetězce na čísla, pokud dané řetězce obsahují číselné hodnoty, nebo na logické hodnoty, pokud je obsahují. Tyto operace však nemusejí být bezpečné.

```
as.integer("11")
```

```
## [1] 11
```

```
as.double("11.111")
```

```
## [1] 11.111
```

```
as.logical("TRUE")
```

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

Někdy R neví, jak nějaký objekt převést. Pak je výsledkem hodnota `NA` a R vydá varování:

```
x <- "ahoj"
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA
```

Poznámka: V R je zvykem, že objekty má testovací funkci `is.X` a konverzní funkci `as.X`. Neplatí to sice vždy, ale valná většina objektů tyto funkce má.

4.5 Základní aritmetické operace

Základní aritmetické operace jsou sčítání (+), odčítání (-), násobení (*), dělení (/) a umocňování (^). K celočíselnému dělení slouží symbol %/%, zbytek po dělení vrací %%:

```
1 + 2
```

```
## [1] 3
```

```
3 * 4
```

```
## [1] 12
```

```
12 / 4
```

```
## [1] 3
```

```
2 ^ 3
```

```
## [1] 8
```

```
9 %/% 2
```

```
## [1] 4
```

```
9 %% 2
```

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

Operátory mají normální prioritu, na jakou jsme zvyklý z matematiky, tj. součin má přednost před sčítáním apod.:

```
1 + 2 * 3  # 7, nikoli 9
```

```
## [1] 7
```

Pokud potřebujeme změnit pořadí vyhodnocování výrazů, slouží k tomu stejně jako v matematice obyčejné kulaté závorky:

```
(1 + 2) * 3  # 9, nikoli 7
```

```
## [1] 9
```

4.6 Srovnání čísel

Ke srovnání aritmetických hodnot slouží následující operátory: porovnání shody celých čísel (==), různosti celých čísel (!=), větší (<), větší rovno (<=), menší (>) a menší rovno (>=). Výsledkem srovnání je logická hodnota `TRUE` nebo `FALSE`:

```
1L == 3L
```

```
## [1] FALSE
```

```
2L == 2L
```

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

```
1L != 3L
```

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

```
1 < 3
```

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

```
1 > 3
```

```
## [1] FALSE
```

Srovnat stejnost nebo různost dvou reálných čísel není pomocí operátorů `==` a `!=` možné. R sice srovnání provede, ale to nemusí mít žádný smysl, jak ukazuje následující příklad:

```
x1 <- 0.5 - 0.3
x2 <- 0.3 - 0.1
x1 == x2                          # na většině počítačů FALSE
```

```
## [1] FALSE
```

Důvod je ten, že necelá čísla, která v desítkové soustavě vypadají nezáludně, není vždy možné vyjádřit dobře ve dvojkové soustavě a výsledek se proto zaokrouhluje. Proto přestože je výsledek předchozích operací v desítkové soustavě stejný (1/10), ve dvojkové soustavě dopadne jinak.

Ke srovnání dvou reálných čísel slouží následující fráze:

```
identical(all.equal(x1, x2), TRUE) # TRUE všude
```

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

Funkce `all.equal()` vrací logickou hodnotu `TRUE`, pokud jsou všechny prvky dvou vektorů stejné; jinak vrací komentář k velikosti rozdílů. Ovšem "jsou stejné" je v této funkci chápáno volně: dvě reálná čísla jsou stejná, pokud se neliší více než o několik násobků strojové přesnosti počítače. Funkce `identical()` vrací logickou hodnotu `TRUE`, pokud jsou dva objekty identické; jinak vrací `FALSE`. Dohromady vrátí fráze hodnotu `TRUE` jen v případě, kdy jsou obě téměř stejná (až na chybu, která zřejmě vznikla kvůli tomu, jak jsou reálná čísla v počítači uložená).

Balík **dplyr** nabízí ke srovnání hodnot typu *double* příjemnou funkci `near()`

```
dplyr::near(x1, x2)  # TRUE všude
```

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

4.7 Základní logické operace

Základní logické operace zahrnují logický součin ("a zároveň", `&`), logický součet ("nebo", `|`) a negaci ("opak", `!`). Kromě toho samozřejmě fungují i závorky. Význam jednotlivých operací ukazuje tabulka 4.1. Všimněte si, že `!(V1 & V2)` = `!V1 | !V2` a `!(V1 | V2)` = `!V1 & !V2`.

Table 4.1: Význam základních logických operací.

V1	V2	V1 & V2	V1 | V2	!V1	!(V1 & V2)	!(V1 | V2)
TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE
FALSE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE

Logický součin a součet existují v R ve dvou formách: jednoduché a "zkratující". Ta druhá operátory zdvojuje, takže místo `&` se použije `&&` a místo `|` se použije `||`. Jednoduchá forma se používá při vyhodnocování operátorů na logických vektorech:

```
c(TRUE, TRUE, FALSE, FALSE) & c(TRUE, FALSE, TRUE, FALSE)
```

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

```
c(TRUE, TRUE, FALSE, FALSE) | c(TRUE, FALSE, TRUE, FALSE)
```

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

Zkratující forma se používá v podmínkách, viz oddíl 7.1. V tomto případě se vyhodnocování výrazu zastaví ve chvíli, kdy je výsledek jednoznačně známý, tj. např. ve výrazu

```
FALSE && !(TRUE || FALSE)
```

```
## [1] FALSE
```

vyhodnocování skončí hned prvním `FALSE`, protože po jeho vyhodnocení výsledek jasný. Pokud byste použili zkratující formu na vektory, výsledek se bude týkat první položky vektoru:

```
c(TRUE, FALSE) && c(TRUE, TRUE)
```

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

Vektorová funkce `all()` vrátí `TRUE`, pokud jsou všechny prvky vektoru `TRUE`; jinak vrátí `FALSE`. Vektorová funkce `any()` vrátí `TRUE`, pokud je aspoň jedna hodnota `TRUE`; jinak vrátí `FALSE`. (Jedná se tedy o logický součin a součet přes všechny prvky vektoru.) Funkce `all.equal()` a `identical()` byly představeny výše.

```
all(c(TRUE, TRUE, TRUE))
```

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

```
all(c(TRUE, TRUE, FALSE))
```

```
## [1] FALSE
```

```
any(c(TRUE, TRUE, FALSE))
```

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

```
any(c(FALSE, FALSE, FALSE))
```

```
## [1] FALSE
```

5 Základní datové struktury

Data, která budeme zkoumat, většinou netvoří izolované hodnoty (jednotlivá izolovaná čísla), nýbrž větší množství hodnot, které mají nějaký vztah. Výsledkem ekonomického experimentu může být např. datový soubor, který obsahuje pro každý subjekt experimentu identifikační číslo daného subjektu, jeho treatment, identifikaci skupiny, do které patřil a seznam akcí, které zahrál, a výplat, kterých dosáhl. Takový datový soubor můžeme zorganizovat jako tabulku, ve které řádky odpovídají jednotlivých subjektům a sloupce jednotlivým proměnným. Pokud jsou všechny proměnné číselné, je tabulka matice. Alternativně můžeme uspořádat každou proměnnou zvlášť jako jednotlivé vektory. V každém případě však potřebujeme k uchování hodnot získaných z experimentu určitý typ datové struktury.

Základní datové struktury, které nám R nabízí, lze roztřídit podle dvou charakteristik: 1) podle jejich dimensionality na jednorozměrné, dvourozměrné a vícerozměrné objekty a 2) podle homogenity použitých datových typů na homogenní a heterogenní struktury. Jednotlivé kombinace uvádí tabulka 5.1. Homogenní struktury mají všechny položky stejného typu, např. celá čísla. Mezi homogenní struktury patří zejména atomické vektory a matice. Heterogenní struktury mohou mít jednotlivé položky různých typů, takže mohou najednou obsahovat např. reálná čísla i řetězce. Mezi nejdůležitější heterogenní struktury patří seznamy a různé typy tabulek, jako jsou tabulky tříd *data.frame* a *tibble*. Jednorozměrné datové struktury mají jen jeden rozměr, délku. Sem patří zejména atomické vektory a seznamy (seznamy jsou neatomické vektory). Dvourozměrné struktury mají dva rozměry, takže tvoří tabulku. Nejdůležitější dvourozměrné struktury jsou homogenní matice a nehomogenní tabulky.

Table 5.1: Význam základních logických operací.

dimenze	homogenní	heterogenní
1	atomický vektor	seznam
2	matice	tabulka
více	pole	

V této kapitole se

- seznámíte se základními datovými strukturami: atomickými vektory, atomickými maticemi, seznamy a tabulkami třídy *data.frame* a *tibble*
- naučíte převádět data z jedné datové struktury na jinou
- naučíte se získávat podmnožiny (*subsety*) těchto struktur
- dozvíte něco o tom, jaké struktury volit

5.1 Atomické vektory

Nejzákladnější datovou strukturou je v R atomický vektor. R nemá datovou strukturu pro skalár (jedinou logickou hodnotu, jediné číslo, znak nebo řetězec) – každý skalár je ve skutečnosti atomický vektor s jediným prvkem. Atomický vektor je vektor hodnot, jehož všechny prvky mají stejný typ (např. celé číslo). Atomické vektory se vytvářejí funkcí `c()` (od “concatenate”), která “slepí” jednotlivé hodnoty dohromady, přičemž provede automatickou konverzi (pokud je potřeba).

```r
x <- c(1, 2, 3, 17)
print(x)
```

```
## [1]  1  2  3 17
```

Pomocí funkce `c()` je možné "slepit" i celé vektory:

```r
x1 <- c(1, 2, 3)
x2 <- c(4, 5, 6, NA)
x <- c(x1, x2)
print(x)
```

```
## [1]  1  2  3  4  5  6 NA
```

Všechny prvky atomického vektoru musejí mít stejný typ. Pokud tedy při tvorbě atomického vektoru smícháte proměnné různých typů, dojde k automatické konverzi:

```r
c(TRUE, 1, "ahoj")  # výsledek je vektor tří řetězců
```

```
## [1] "TRUE" "1"    "ahoj"
```

Jednotlivé prvky atomických vektorů mohou mít jména. Jména jsou uložena v atributu `names`. Je možné je zadat čtyřmi různými způsoby: přímo ve funkci `c()`, pomocí funkce `attr()`, pomocí speciální funkce `names()` nebo funkce `setNames()`:

```r
x <- c(a = 1, "good parameter" = 7, c = 17)
x
```

```
##              a good parameter              c
##              1              7             17
```

```r
attr(x, "names") <- c("A", "Good Parameter", "C")
x
```

```
##              A Good Parameter              C
##              1              7             17
```

```r
names(x) <- c("aa", "bb", "cc")
names(x)
```

```
## [1] "aa" "bb" "cc"
```

```r
x
```

```
## aa bb cc
##  1  7 17
```

```
setNames(x, c("A", "B", "C"))
```

```
##  A  B  C
##  1  7 17
```

Pokud mají jednotlivé prvky vektorů přiřazená jména, vypisují se na obrazovku nad vlastní hodnoty.

Délku vektoru je možné zjistit pomocí funkce `length()`:

```
length(x)
```

```
## [1] 3
```

Pozor: atomický vektor může mít i nulovou délku, pokud neobsahuje žádné prvky. (Podobně i další datové struktury mohou mít nulové rozměry, např. nulový počet řádků apod.) Prázdný vektor vznikne často tak, že z existujícího vektoru vyberete hodnoty pomocí podmínky, kterou žádný prvek vektoru nesplní (jak se vybírá část datové struktury uvidíte později):

```
x <- 1:9
length(x)
```

```
## [1] 9
```

```
y <- x[x > 10]  # vybereme prvky větší než 10, viz dále
y
```

```
## integer(0)
```

```
length(y)
```

```
## [1] 0
```

Prázdný vektor je možné vytvořit pomocí konstruktorových funkcí `logical()`, `integer()`, `numeric()`, `character()` apod., které mají jediný parametr, počet prvků. Pokud je zadaný počet prvků nulový, funkce vrátí prázdný vektor. Pokud je počet prvků kladný, vznikne vektor se zadanou délkou. To je užitečné např. v situaci, kdy si chcete dopředu připravit vektor určité délky, a později jej naplnit hodnotami. (Funkce `numeric()` má poněkud nekonzistentní název, protože vytváří vektor typu *double*.)

```
z <- numeric(0)  # parametr je délka vektoru
z
```

```
## numeric(0)
```

```
z <- numeric(10)  # vektor 10 hodnot
z
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

Některé vektory obsahují předvídatelné sekvence čísel. Pro vytváření takových vektorů existuje operátor dvojtečka (`:`) a speciální funkce `seq()` a `rep()` a jejich specializované varianty:

```
1:10   # vektor celých čísel 1 až 10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
10:1   # vektor celých čísel sestupně 10 až 1
```

```
##  [1] 10  9  8  7  6  5  4  3  2  1
```

```
# sekvence od ... do
seq(from = 1, to = 10, by = 3)  # s daným krokem
```

```
## [1]  1  4  7 10
```

```
seq(from = 1, to = 10, length.out = 4)  # s danou délkou výsledku
```

```
## [1]  1  4  7 10
```

```
seq_along(c(1, 3, 17, 31))  # celá čísla od 1 do délky zadaného vektoru
```

```
## [1] 1 2 3 4
```

```
seq_len(7)  # celá čísla od 1 nahoru se zadanou nezápornou délkou
```

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

```
# opakování hodnot ve vektoru
rep(c(1, 3), times = 5)  # celý vektor 5 krát
```

```
##  [1] 1 3 1 3 1 3 1 3 1 3
```

```
rep_len(c(1, 3), length.out = 5)  # celý vektor do délky 5
```

```
## [1] 1 3 1 3 1
```

```
rep(c(1, 3), each = 3)  # každý prvek 3 krát
```

```
## [1] 1 1 1 3 3 3
```

```
rep(1:6, each = 2, times = 3)
```

```
##  [1] 1 1 2 2 3 3 4 4 5 5 6 6 1 1 2 2 3 3 4 4 5 5 6 6 1 1 2 2 3 3 4 4 5 5 6 6
```

Složitější varianty použití funkce `rep()` viz dokumentace.

Pozor! Konstrukce vektorů pomocí operátoru dvojtečka je někdy nebezpečná. Řekněme, že chcete provést nějakou operaci pro každý prvek vektoru `x` a že to chcete udělat pomocí cyklu `for`, viz oddíl 7.2.1. Při psaní cyklů se často prochází hodnoty pomocného vektoru `k = 1:length(x)`. Pokud má vektor `x` kladnou délku, je vše v pořádku. Pokud však vektor `x` neobsahuje žádné hodnoty, pak má nulovou délku. Čekali bychom, že pomocný vektor `k` bude mít také nulovou délku, takže cyklus neproběhne ani jednou. To však není pravda. Vektor `k` je v tomto případě zkonstruován jako `1:0`, má tedy hodnotu `c(1, 0)` a délku 2! Taková věc je zdrojem špatně dohledatelných chyb. Je lepší použít `k = seq_along(x)`. Ještě lepší je cyklům se vyhýbat. R k tomu má velmi užitečné funkce typu `map()`, viz kapitola 10.

Na obrazovku se vektory vypisují tak, že se jejich jednotlivé hodnoty skládají vedle sebe do řádku. Pokud se všechny hodnoty na řádek nevejdou, začne se vypisovat na dalším řádku. Pokud nejsou jednotlivé prvky vektoru pojmenované, pak je každý řádek uvozen číslem v hranatých závorkách. To je index prvního prvku vektoru na daném řádku. Protože jsou prvky vektorů číslované přirozenými čísly (tj. první prvek má index 1), bude první řádek začínat `[1]`. Pokud další řádek začíná např. `[31]`, znamená to, že první číslo na daném řádku je 31. prvek daného vektoru atd.

```
1:100
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

Pokud jsou prvky vektoru pojmenované, pak R nevypisuje na začátek řádku indexy prvního prvku, ale vypisuje nad jednotlivé prvky jejich jména:

```
setNames(1:26, letters)
```

```
##  a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  r  s  t  u  v  w  x  y  z
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
```

Poznámka: I když se vektory vypisují na obrazovku po řádcích, neznamená to že jsou v R vektory řádkové – nejsou ani řádkové, ani sloupcové (jako je to implicitně třeba v Matlabu). Hodnoty se vypisují po řádcích prostě pro úsporu místa.

R je vektorizovaný jazyk. To znamená, že veškeré aritmetické a logické operace a většina funkcí, která pracuje s vektory, provádí danou operaci na celých vektorech po prvcích. Pokud např. vynásobíte dva vektory x a y, výsledkem bude vektor, jehož prvky budou násobky odpovídajících prvků obou vektorů, takže $z_i = x_i \cdot y_i$:

```
x <- 1:6
y <- 2:7
x * y
```

```
## [1]  2  6 12 20 30 42
```

```
x ^ 2
```

```
## [1]  1  4  9 16 25 36
```

Pozor: pokud se délka vektorů liší, pak R automaticky "recykluje" kratší vektor, tj. opakuje jeho hodnoty znovu a znovu. Při tom vydá R varování jen v případě, že délka delšího vektoru není celočíselným násobkem délky kratšího vektoru:

```
x <- 1:2
y <- 1:6
x + y
```

```
## [1] 2 4 4 6 6 8
```

```
x * y
```

```
## [1]  1  4  3  8  5 12
```

```
y <- 1:7
x + y
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
## [1] 2 4 4 6 6 8 8
```

Otestovat, zda je proměnná atomický vektor, není úplně snadné. R nabízí dvě potřebné funkce, z nichž však žádná sama o sobě nestačí. Funkce `is.atomic()` vrátí hodnotu `TRUE`, pokud je daná proměnná atomická (tj. všechny její prvky mají stejný datový typ). Tuto hodnotu však nevrací jen pro atomické vektory, ale i pro atomické matice, viz oddíl 5.2. Funkce `is.vector()` vrací logickou hodnotu `TRUE`, pokud je proměnná vektor; vektory však v R nemusejí být nutně atomické; funkce proto vrací `TRUE` i pro neatomické vektory (seznamy), viz oddíl 5.3. Zda proměnná obsahuje atomický vektor proto musíme otestovat spojením těchto funkcí:

```
is.atomic(x)  # x je atomická, tj.\ homogenní proměnná
```

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

```
is.vector(x)  # x je vektor
```

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

```
is.atomic(x) & is.vector(x)  # x je atomický vektor
```

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

Někdy je potřeba získat z datové struktury jen vybrané prvky: z vektoru jednotlivé prvky, z matice vybrané řádky nebo sloupce apod. K tomu slouží subsetování. Subsetování lze použít nejen k získání vybraných prvků z datové struktury, ale také k jejich nahrazení nebo doplnění. K základnímu subsetování slouží hranaté závorky (`[]`). V nich se určí indexy prvků, které je třeba vybrat. Prvky mohou být vybrány třemi způsoby: pomocí svých indexů, pomocí svých jmen a nebo pomocí logických hodnot. Ukážeme si to nejprve na atomických vektorech.

1. Výběr pomocí číselných indexů. Prvky atomických vektorů jsou číslované přirozenými čísly $1, \ldots, N$, kde N je délka vektoru (tj. první prvek vektoru má index 1, nikoli 0). Při výběru prvků pomocí indexů se vyberou prvky s danými indexy (pokud jsou indexy kladné), nebo se vynechají prvky s danými indexy (pokud jsou indexy záporné). Indexování pomocí kladných a záporných čísel nelze míchat. Index 0 se tiše ignoruje.

```
# vektor letters obsahuje 26 malých písmen anglické abecedy
x <- letters[1:12]  # prvních dvanáct písmen abecedy
x
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

```
x[1]  # první prvek
```

```
## [1] "a"
```

```
x[3]  # třetí prvek
```

```
## [1] "c"
```

```
x[length(x)]  # poslední prvek
```

```
## [1] "l"
```

```
x[3:6]  # třetí až šestý prvek včetně
```

```
## [1] "c" "d" "e" "f"
```

```
x[c(2, 3, 7)]  # druhý, třetí a sedmý prvek
```

```
## [1] "b" "c" "g"
```

```
x[c(-1, -3)]  # vynechají se první a třetí prvek
```

```
##  [1] "b" "d" "e" "f" "g" "h" "i" "j" "k" "l"
```

2. Výběr pomocí jmen prvků. Pokud mají prvky vektoru jména, je možné vybírat pomocí vektoru jejich jmen (zde samozřejmě nejde vynechávat pomocí znaménka minus, protože R nemá záporné řetězce):

```
x <- c(c = 1, b = 2, a = 3)
x
```

```
## c b a
## 1 2 3
```

```
x["a"]  # prvek s názvem a, tj. zde poslední prvek
```

```
## a
## 3
```

```
x[c("b", "c")]  # prvky s názvy b a c
```

```
## b c
## 2 1
```

3. Výběr pomocí logických hodnot. R vybere prvky, které jsou indexovány logickou hodnotou `TRUE` a vynechá ostatní. Pozor: pokud je logický vektor kratší než subsetovaný vektor, pak se recykluje!

```
x <- 1:12
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
x[c(TRUE, TRUE, FALSE)]
```

```
## [1]  1  2  4  5  7  8 10 11
```

Výběr pomocí logických hodnot je užitečný zejména v situaci, kdy chceme vybrat prvky, které splňují nějakou podmínku:

```
x[x > 3 & x < 11]  # vybere prvky, které jsou větší než tři a menší než 11
```

```
## [1]  4  5  6  7  8  9 10
```

```
x[x < 3 | x > 11]  # vybere prvky, které jsou menší než tři nebo větší než 11
```

```
## [1]  1  2 12
```

Subsetování lze využít k nahrazení prvků jednoduše tak, že se do výběru uloží nová hodnota, která nahradí starou:

```
x <- c(1:3, NA, 5:7)
x
```

```
## [1]  1  2  3 NA  5  6  7
```

```
x[7] <- Inf  # nahrazení poslední hodnoty nekonečnem
x
```

```
## [1]   1   2   3  NA   5   6 Inf
```

```
x[is.na(x)] <- 0  # nahrazení všech hodnot NA nulou
x
```

```
## [1]   1   2   3   0   5   6 Inf
```

```
x[length(x) + 1] <- 8  # přidání nové hodnoty za konec vektoru
x
```

```
## [1]   1   2   3   0   5   6 Inf   8
```

Pozor: postupné rozšiřování datových struktur vždy o několik málo prvků je výpočetně velmi neefektivní, protože R musí (téměř) pokaždé alokovat nové místo v paměti, do něj zkopírovat staré hodnoty a na konec přidat nový prvek. Mnohem efektivnější je naráz alokovat velký blok paměti, do něj postupně uložit hodnoty a blok na konci případně zkrátit:

```
x <- numeric(1e6)  # alokace prázdného vektoru o milonu pruků
x[1] <- 1          # přidání pruků (další řádky vynechány)
n <- 7654          # skutečný počet vložených pruků
x <- x[1:n]        # zkrácení vektoru na potřebnou délku
```

Ekvivalentně je vhodné postupovat v případě všech homogenních datových struktur.

Pozor: `numeric()` vytvoří vektor samých nul. Možná je lepší použít `rep(NA_real_, 1e6)`, které vytvoří reálný vektor hodnot `NA`. Většina lidí však používá funkce `numeric()`, `character()` apod.

V R je možné výběry ze všech datových struktur řetězit – následující výběr vybírá z výsledku předchozího výběru:

```
v <- 1:10
v[6:10]     # 6. až 10. prvek v
```

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

```
v[6:10][2]  # druhý prvek z výběru 6. až 10. pruku v
```

```
## [1] 7
```

Do zřetězeného výběru je obvykle možné dosazovat hodnoty:

```
v[6:10][2] <- NA
v
```

```
##  [1]  1  2  3  4  5  6 NA  8  9 10
```

5.2 Atomické matice

Atomická matice je matice (tj. dvourozměrná tabulka), jejíž všechny prvky mají stejný datový typ (např. celé číslo). Pro datovou analýzu nejsou matice příliš důležité, někdy se však hodí pro rychlou maticovou algebru a také některé funkce vracejí nebo očekávají jako vstup matice.

Nejjednodušší způsob, jak vytvořit atomickou matici je pomocí funkce `matrix()`. Prvním parametrem je vektor, který obsahuje data. Další parametry určují počet řádků a počet sloupců matice, způsob, jak budou data do matice skládaná (zda podle řádků či sloupců; implicitně se data do matic skládají po sloupcích) a pojmenování dimenzí matice. Není potřeba zadávat všechny parametry, pokud R dokáže odhadnout hodnotu jednoho parametru z hodnot ostatních parametrů. R např. umí z délky zadaného vektoru a zadaného počtu řádků odhadnout počet sloupců.

```
matrix(1:12, nrow = 3)  # matice se třemi řádky a čtyřmi sloupci, po sloupcích
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
matrix(1:12, ncol = 4, byrow = TRUE)  # stejný rozměr, data po řádcích
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

Při tvorbě matic R recykluje data. To znamená, že pokud je zadaných hodnot méně, než vyžadují rozměry matice, R začne číst datový vektor znovu od začátku. To může být zdrojem nepříjemných chyb. Naštěstí R vypíše varování, ovšem pouze v případě, že počet prvků matice není celočíselným násobkem délky zadaného vektoru.

```
matrix(1:9, nrow = 3, ncol = 4)
```

```
## Warning in matrix(1:9, nrow = 3, ncol = 4): data length [9] is not a sub-
## multiple or multiple of the number of columns [4]
```

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

Otestovat, zda je objekt matice, je možné pomocí funkce `is.matrix()`; převést data na matici je možné pomocí konverzní funkce `as.matrix()`.

Zjistit rozměry matice je možné pomocí následujících funkcí: `nrow()` vrátí počet řádků matice, `ncol()` vrátí počet sloupců matice, `dim()` vrací vektor s počtem řádků a sloupců matice a `length()` vrací počet prvků matice. (Pro vektory vrací funkce `nrow()`, `ncol()` a `dim()` hodnotu `NULL`.)

```
m <- matrix(1:12, nrow = 3)
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
nrow(m)
```

```
## [1] 3
```

```
ncol(m)
```

```
## [1] 4
```

```
dim(m)
```

```
## [1] 3 4
```

```
length(m)
```

```
## [1] 12
```

Matice a podobné objekty je možné skládat pomocí funkcí `rbind()` a `cbind()`. První (`rbind()` od "row bind") spojuje matice po řádcích (tj. skládá je pod sebe), druhá (`cbind()` od "column bind") po sloupcích (tj. skládá je vedle sebe):

```
A <- matrix(1:12, nrow = 3)
B <- matrix(101:112, nrow = 3)
rbind(A, B)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
## [4,]  101  104  107  110
## [5,]  102  105  108  111
## [6,]  103  106  109  112
```

```
cbind(A, B)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    4    7   10  101  104  107  110
## [2,]    2    5    8   11  102  105  108  111
## [3,]    3    6    9   12  103  106  109  112
```

Matice mohou mít následující atributy: `dim` je celočíselný vektor rozměrů (viz výše), jména řádků (čte i nastavuje se funkcí `rownames()`), jména sloupců (čte i nastavuje se funkcí `colnames()`) a jména dimenzí včetně jmen řádků a sloupců (čte i nastavuje se funkcí `dimnames()`):

```
rownames(A) <- c("a", "b", "c")
colnames(A) <- c("alpha", "beta", "gamma", "delta")
A
```

```
##   alpha beta gamma delta
## a     1    4     7    10
## b     2    5     8    11
## c     3    6     9    12
```

```
dimnames(A) <- list(id = c("A", "B", "C"), variables = c("Alpha", "Beta", "Gamma", "Delta"))
A
```

```
##    variables
## id  Alpha Beta Gamma Delta
##   A     1    4     7    10
##   B     2    5     8    11
##   C     3    6     9    12
```

```
attributes(A)
```

```
## $dim
## [1] 3 4
##
## $dimnames
## $dimnames$id
## [1] "A" "B" "C"
##
## $dimnames$variables
## [1] "Alpha" "Beta"  "Gamma" "Delta"
```

Atomická matice je implementována jako atomický vektor, který má přiřazený atribut `dim`, který je celočíselný vektor délky dva:

```
M <- 1:12
M
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
dim(M) <- c(3, 4)
M
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
is.matrix(M)
```

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

Podobně lze zrušením atributu `dim` převést matici zpět na vektor (matice se vektorizuje po sloupcích). Stejného výsledku jde dosáhnout pomocí funkce `as.vector()`:

```
as.vector(M)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
dim(M) <- NULL
M
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Protože matice je atomický vektor, který má přiřazený atribut `dim`, funkce `is.atomic()` vrací pro matici hodnotu `TRUE`; funkce `is.vector()` však samozřejmě vrací `FALSE`, protože testuje, zda je v proměnné uložen vektor.

```
M <- matrix(1:12, nrow = 3)
is.atomic(M)
```

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

```
is.vector(M)
```

```
## [1] FALSE
```

5.2.1 Maticová aritmetika

Obyčejné symboly násobení (`*`), dělení (`/`) a umocňování (`^`) pracují "po prvcích". Při násobení se např. vynásobí odpovídající prvky matice. Matice tedy musejí mít stejné rozměry (stejný počet řádků a sloupců).

```
A * B
```

```
##    variables
## id  Alpha Beta Gamma Delta
##   A   101  416   749  1100
##   B   204  525   864  1221
##   C   309  636   981  1344
```

```
A ^ 2
```

```
##    variables
## id  Alpha Beta Gamma Delta
##   A     1   16    49   100
##   B     4   25    64   121
##   C     9   36    81   144
```

Pro skutečné maticové násobení se používá operátor `%*%`. Inverzní matici vrací funkce `solve()` (obecně tato funkce řeší soustavy lineárních rovnic). K transponování matice slouží funkce `t()`. Hlavní diagonálu matice vrací funkce `diag()`.

Speciální matice:

```
diag(1, nrow = 3, ncol = 3)  # jednotková matice
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
matrix(0, nrow = 3, ncol = 3)  # nulová matice (využívá recyklace)
```

```
##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

Příklad inverze:

```
M <- matrix(c(1:8, 0), nrow = 3)
invM <- solve(M)
E <- diag(1, nrow = nrow(M), ncol = ncol(M))
all.equal(M %*% invM, E)
```

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

```
all.equal(invM %*% M, E)
```

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

5.2.2 Subsetování matic

Subsetování matic je podobné jako u atomických vektorů s jedním rozdílem: protože má matice řádky a sloupce, je třeba subsetovat pomocí dvou indexů. První index vybírá řádky, druhý sloupce:

```
M <- matrix(1:12, nrow = 3)
M
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```
M[2, 3]  # prvek ve druhém řádku a třetím sloupci
```

```
## [1] 8
```

```
M[1:2, c(1,4)]  # prvky na prvních dvou řádcích a v prvním a čtvrtém sloupci
```

```
##      [,1] [,2]
## [1,]    1   10
## [2,]    2   11
```

```
M[-1, -1]  # matice bez prvního řádku a sloupce
```

```
##      [,1] [,2] [,3]
## [1,]    5    8   11
## [2,]    6    9   12
```

Pokud je jeden z indexů prázdný, vybírá celý řádek nebo sloupec:

```
M[1:2, ]  # celé první dva řádky
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
```

```
M[, c(1, 3)]  # první a třetí sloupec
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    2    8
## [3,]    3    9
```

```
M[M[, 1] >= 2, ]  # všechny řádky, ve kterých je prvek v prvním sloupci >= 2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2    5    8   11
## [2,]    3    6    9   12
```

```
M[M[, 1] >= 2, M[1, ] < 6]  # submatice
```

```
##      [,1] [,2]
## [1,]    2    5
## [2,]    3    6
```

```
M[ , ]  # celá matice M
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

Pokud se matice indexuje jen jedním indexem, R ji tiše převede na jeden vektor (spojí sloupce matice za sebe) a vybere prvky z takto vzniklého vektoru:

```
M[4]  # vrací 1. prvek ve 2. sloupci, protože je to 4. prvek vektoru
```

```
## [1] 4
```

```
M[c(1, 4:7)]
```

```
## [1] 1 4 5 6 7
```

Subsetování může nejen vybírat hodnoty, ale také měnit jejich pořadí. Zadáním indexů můžeme např. otočit pořadí sloupců matice:

```
M <- matrix(c(8, 5, 7, 2, 3, 11, 6, 12, 1, 4, 9, 10), nrow = 3)
M
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    8    2    6    4
## [2,]    5    3   12    9
## [3,]    7   11    1   10
```

```
M[, 4:1]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    4    6    2    8
## [2,]    9   12    3    5
## [3,]   10    1   11    7
```

Stejným způsobem můžeme dosáhnout i zajímavějších efektů: např. seřadit řádky nebo sloupce podle hodnot vybraného vektoru nebo je náhodně permutovat. Funkce `order()` vrací indexy uspořádané podle velikosti původního vektoru. Funkce např. umožňuje setřídit hodnoty všech sloupců matice podle jednoho sloupce:

```
# indexy pruků 1. sloupce matice M seřazené podle velikosti pruků,
# tj. na 1. místě je 2. prvek původního vektoru (5), pak 3. prvek (7) atd.
order(M[, 1])
```

```
## [1] 2 3 1
```

```
M[order(M[, 1]), ]  # řádky matice seřazené podle prvního sloupce
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    5    3   12    9
## [2,]    7   11    1   10
## [3,]    8    2    6    4
```

Funkce `sample()` náhodně permutuje zadaná čísla. Lze jí tak mimo jiné využít k náhodné permutaci sloupců matice:

```
o <- sample(ncol(M))  # čísla 1:ncol(M) v náhodném pořadí
o
```

```
## [1] 3 4 2 1
```

```
M[, o]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    6    4    2    8
## [2,]   12    9    3    5
## [3,]    1   10   11    7
```

Subsetování se vždy snaží snížit rozměry matice – pokud počet řádků nebo sloupců klesne na 1, matice se změní ve vektor. Pokud tomu chceme zabránit, je třeba přidat parametr `drop = FALSE`. (Nemělo by vám být divné, že je možné hranatým závorkám přidávat parametry – jako vše v R je i použití hranatých závorek volání funkce – a funkce mohou mít parametry.)

```
M[, 1]
```

```
## [1] 8 5 7
```

```
M[, 1, drop = FALSE]
```

```
##      [,1]
## [1,]    8
## [2,]    5
## [3,]    7
```

5.3 Neatomické vektory (seznamy)

Neatomické vektory (častěji nazývané seznamy) jsou vektory, jejichž jednotlivé prvky mohou mít různé datové typy, třeba i jiné seznamy. Seznamy jsou někdy užitečné, protože umožňují v jedné proměnné skladovat různé typy dat. Jejich hlavní význam však spočívá v tom, že se používají jako základ pro tvorbu většiny objektů v systému S3, viz kapitola 9.

Seznamy se vytvářejí pomocí funkce `list()`:

```
l <- list(1L, 11, 1:3, "ahoj", list(1, 1:3, "ahoj"))
l
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 11
##
## [[3]]
## [1] 1 2 3
##
## [[4]]
## [1] "ahoj"
##
## [[5]]
## [[5]][[1]]
## [1] 1
##
## [[5]][[2]]
## [1] 1 2 3
##
## [[5]][[3]]
## [1] "ahoj"
```

Stejně jako atomické vektory, i seznamy mohou mít atribut `names`, tj. jednotlivé prvky seznamu mohou mít svá jména. Ta se přiřazují stejně jako v případě atomických vektorů:

```
l <- list(a = 1, b = "ahoj", c = 1:3, d = list(1:3, "ahoj"))
names(l)
```

```
## [1] "a" "b" "c" "d"
```

```
l
```

```
## $a
## [1] 1
```

```
##
## $b
## [1] "ahoj"
##
## $c
## [1] 1 2 3
##
## $d
## $d[[1]]
## [1] 1 2 3
##
## $d[[2]]
## [1] "ahoj"
```

Délku seznamu zjistíme pomocí funkce `length()`:

```
length(l)
```

```
## [1] 4
```

K otestování, zda je proměnná seznam, slouží funkce `is.list()`. Funkce `is.vector()` vrací hodnotu `TRUE` jak pro atomické vektory, tak i pro seznamy.

```
is.list(l)
```

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

```
is.vector(l)
```

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

Strukturu seznamu je možné přehledně zobrazit pomocí funkce `str()`. Podobný výsledek dostanete v RStudiu tak, že v záložce `Environment` kliknete na trojúhelníček v kolečku vedle jména seznamu. Musíte však být v módu List.

```
str(l)
```

```
## List of 4
##  $ a: num 1
##  $ b: chr "ahoj"
##  $ c: int [1:3] 1 2 3
##  $ d:List of 2
##   ..$ : int [1:3] 1 2 3
##   ..$ : chr "ahoj"
```

Subsetování seznamů je poněkud složitější, než je tomu u atomických proměnných. Subsetování pomocí hranatých závorek zachovává mód proměnné. To znamená, že použití hranatých závorek na seznam vrací opět seznam. Pokud chceme získat přímo prvek uložený v seznamu, musíme použít dvojité hranaté závorky (`[[]]`). Podobnou funkci plní i operátor dolar (`$`).

Subsetování seznamu pomocí hranatých závorek vrací prvky opět zabalené do seznamu:

```
l <- list(a = 1, b = 1:3, c = "ahoj")
l
```

```
## $a
## [1] 1
##
## $b
## [1] 1 2 3
##
## $c
## [1] "ahoj"
```

```
l[1]
```

```
## $a
## [1] 1
```

```
is.list(l[1])
```

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

```
l[1:2]
```

```
## $a
## [1] 1
##
## $b
## [1] 1 2 3
```

Pokud chceme získat vlastní prvek seznamu, musíme použít dvojité hranaté závorky. Dvojité hranaté závorky "vybalí" daný prvek ze seznamu ven:

```
l[[2]]
```

```
## [1] 1 2 3
```

```
is.list(l[[2]])
```

```
## [1] FALSE
```

```
is.numeric(l[[2]])
```

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

Syntaxe dvojitých hranatých závorek je poněkud nečekaná. Pokud je argumentem vektor, nevrací dvojité hranaté závorky vektor hodnot (to ani nejde, protože výsledkem by musel být opět seznam), nýbrž se vektor přeloží na rekurentní volání dvojitých hranatých závorek:

```
l[[2]][[3]]  # třetí prvek vektoru, který je druhým prvkem seznamu
```

```
## [1] 3
```

```
l[[2:3]]     # totéž
```

```
## [1] 3
```

```
# protože druhým prvkem seznamu je zde atomický vektor, mohou být druhé závorky jednoduché:
l[[2]][3]
```

```
## [1] 3
```

Pokud jsou prvky seznamu pojmenované, nabízí R zkratku ke dvojitým hranatým závorkám: operátor dolar (`$`): `l[["b"]]` je totéž jako `l$b`:

```
l[["b"]]  # prvek se jménem b
```

```
## [1] 1 2 3
```

```
l$b       # totéž (uvozovky se zde neuvádějí)
```

```
## [1] 1 2 3
```

Použití dolaru od dvojitých závorek v jednom ohledu liší: pokud máme jméno prvku, který chceme získat uloženo v proměnné, je třeba použít hranaté závorky – operátor dolar zde nelze použít:

```
element <- "c"
# element není v uvozovkách, protože vybíráme hodnotu, která je v něm uložená:
l[[element]]
```

```
## [1] "ahoj"
```

Pokud indexujeme jménem prvek seznamu, který v seznamu chybí, dostaneme hodnotu `NULL`. Pokud jej však indexujeme číselným indexem, dostaneme chybu:

```
l[["d"]]
```

```
## NULL
```

```
l$d
```

```
## NULL
```

```
l[[4]]  # chybný řádek
```

```
## Error in l[[4]]: subscript out of bounds
```

Seznamy umožňují používat i jen části jmen prvků, pokud jsou určeny jednoznačně (tomu se říká "partial matching"). S dolarem partial matching zapnutý vždy; s dvojitými hranatými závorkami jen v případě, že o to požádáte parametrem `exact = FALSE`.

```
l <- list(prvni_prvek = 1, druhy_prevk = 2)
l$p
```

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

```
l[["p"]]
```

```
## NULL
```

```
l[["p", exact = FALSE]]
```

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

Doporučuji partial matching nikdy nevyužívat – může být zdrojem špatně dohledatelných chyb!

5.4 Tabulky třídy *data.frame*

Pro analýzu dat jsou nejdůležitější datovou strukturou tabulky. Každý sloupec tabulky může obsahovat proměnné jiného typu, v rámci sloupce však musí být typ stejný. Tím se tabulky liší od atomických matic, které musí mít všechny prvky stejného typu. Obvyklé použití tabulek je takové, že řádky tabulky představují jednotlivá pozorování a sloupce jednotlivé proměnné.

R má několik implementací tabulek. Základní třída tabulek, které se budeme věnovat v tomto oddíle, se nazývá *data.frame*. Technicky je implementovaná jako seznamy atomických vektorů o stejné délce, které jsou spojené vedle sebe. V příštím oddíle se podíváme na poněkud příjemnější variantu tabulek třídy *tibble*.

Tabulky třídy *data.frame* se tvoří pomocí funkce `data.frame()`. Řekněme, že chceme zaznamenat údaje o subjektech, které se zúčastnily nějakého experimentu. Pro každý subjekt pozorujeme jeho identifikační číslo id, výšku a váhu. Pokud máme čtyři subjekty, můžeme vytvořit tabulku např. takto:

```
experiment <- data.frame(id = c(1, 2, 3, 41),
                         vyska = c(158, 174, 167, 203),
                         vaha = c(51, 110, 68, 97))
experiment
```

```
##   id vyska vaha
## 1  1   158   51
## 2  2   174  110
## 3  3   167   68
## 4 41   203   97
```

Při zadávání vektorů do tabulek můžeme zadat jejich jména, která pak R vypíše. R samo přidá jména řádků (automaticky jim dá přirozená čísla od 1 do počtu proměnných).

I při konstrukci tabulek R recykluje proměnné, pokud není zadáno dost hodnot. Pokud jsou všechny naše subjekty muži, stačí zadat tuto hodnotu jen jednou – R ji zrecykluje.

```
experiment <- data.frame(id = c(1, 2, 3, 41),
                         gender = "muž",
                         vyska = c(158, 174, 167, 203),
                         vaha = c(51, 110, 68, 97),
                         zdravy = c(TRUE, TRUE, FALSE, TRUE))
experiment
```

```
##   id gender vyska vaha zdravy
## 1  1    muž   158   51   TRUE
## 2  2    muž   174  110   TRUE
## 3  3    muž   167   68  FALSE
## 4 41    muž   203   97   TRUE
```

Poznámka: Starší verze R při zadání dat do tabulek pomocí funkce `data.frame()` automaticky převedly všechny řetězce na faktory, viz oddíl 6.1. Této konverzi šlo zabránit nastavením parametru `stringsAsFactors = FALSE`. Od verze R 4.0 má parametr `stringsAsFactors` hodnotu `FALSE` implicitně. Pokud tedy chceme v novějším R převést řetězce na faktory, musíme nastavit parametr `stringsAsFactors` na hodnotu `TRUE` nebo je převést pomocí funkce `factor()`, což je zřejmě rozumnější, viz oddíl 6.1.

Někdy se hodí vytvořit tabulku, která obsahuje všechny možné kombinace hodnot nějakého vektoru. K tomu slouží funkce `expand.grid()`:

```
expand.grid(x = 1:3, y = factor(c("male", "female")), z = c(TRUE, FALSE))
```

```
##    x      y     z
## 1  1   male  TRUE
## 2  2   male  TRUE
## 3  3   male  TRUE
## 4  1 female  TRUE
## 5  2 female  TRUE
## 6  3 female  TRUE
## 7  1   male FALSE
## 8  2   male FALSE
## 9  3   male FALSE
## 10 1 female FALSE
## 11 2 female FALSE
## 12 3 female FALSE
```

Počet řádků tabulky zjistíme pomocí funkce `nrow()`, počet sloupců funkcí `ncol()` nebo `length()`; funguje i funkce `dim()`:

```
nrow(experiment)    # počet řádků
```

```
## [1] 4
```

```
ncol(experiment)    # počet sloupců
```

```
## [1] 5
```

```
length(experiment)  # počet sloupců
```

```
## [1] 5
```

```
dim(experiment)     # vektor počtu řádků a sloupců
```

```
## [1] 4 5
```

Tabulky mají standardně tři atributy: `class` (jméno třídy – data set je totiž objekt), `names` obsahuje jména sloupců (tj. jednotlivých proměnných) a `row.names` obsahuje jména jednotlivých řádků (tj. pozorování, implicitně mají hodnoty 1, 2 atd.).

```
attributes(experiment)
```

```
## $names
## [1] "id"     "gender" "vyska"  "vaha"   "zdravy"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4
```

Jména řádků můžete zjistit i změnit pomocí funkcí `rownames()` a `row.names()`, jména sloupců pomocí funkcí `colnames()` nebo `names()`:

```
colnames(experiment) <- c("id", "sex", "height", "weight", "healthy")
experiment
```

```
##   id sex height weight healthy
## 1  1 muž    158     51    TRUE
## 2  2 muž    174    110    TRUE
## 3  3 muž    167     68   FALSE
## 4 41 muž    203     97    TRUE
```

Jména řádků vypadají na první pohled jako dobrý způsob, jak uložit nějakou identifikaci pozorování, např. id subjektu v experimentu. Nedělejte to! Veškeré informace o pozorování ukládejte přímo do tabulky. Je to filosoficky správnější a i praktičtější: některé funkce, které se používají ke zpracování tabulek, jména řádků odstraní. Stejně tak některé formáty, do kterých se data ukládají, jména řádků nepodporují, takže byste přišli o důležité informace. Naproti tomu jména sloupců (tj. proměnných) jsou bezpečná.

Pozor: R vám dovolí změnit i třídu objektu tím, že přepíšete atribut `class` (buď pomocí funkce `attr()` nebo funkce `class()`). Pak se však budou pro daný objekt volat jiné funkce (metody) a výsledek může být podivný. Nedělejte to, pokud nevíte, co děláte.

Někdy je užitečné moci převést tabulku na matici a matici na tabulku. K převodu tabulky na matici slouží funkce `as.matrix()` a `data.matrix()`. První převede všechny sloupce tabulky automatickou konverzí na stejný typ, a pak na matici. Druhá převede logické hodnoty a faktory na celá čísla, řetězce na faktory a ty na celá čísla. Pokud jsou po této konverzi všechny sloupce typu *integer*, má výsledná matice tento typ, jinak má typ *double*. Při automatické konverzi můžeme skončit s řetězci, což nemusí být žádoucí; s explicitní konverzí na reálná čísla můžeme řetězce ztratit a faktory mohou být zavádějící (faktory se převedou na čísla jako při konverzi na celé číslo).

```
as.matrix(experiment)    # použije automatickou konverzi na stejný typ
```

```
##      id   sex   height weight healthy
## [1,] " 1" "muž" "158"  " 51"  "TRUE"
## [2,] " 2" "muž" "174"  "110"  "TRUE"
## [3,] " 3" "muž" "167"  " 68"  "FALSE"
## [4,] "41" "muž" "203"  " 97"  "TRUE"
```

```
data.matrix(experiment)  # použije explicitní konverzi na reálná čísla
```

```
##      id sex height weight healthy
## [1,]  1   1    158     51       1
## [2,]  2   1    174    110       1
## [3,]  3   1    167     68       0
## [4,] 41   1    203     97       1
```

Matici lze převést na tabulku pomocí funkcí `as.data.frame()` i `data.frame()`. Pokud má matice pojmenované sloupce, jejich jména jsou v tabulce zachována; v opačném případě je R samo pojmenuje `V1`, `V2` atd nebo `X1`, `X2` atd.

```
M <- matrix(1:12, nrow = 3)
as.data.frame(M)
```

```
##   V1 V2 V3 V4
## 1  1  4  7 10
## 2  2  5  8 11
## 3  3  6  9 12
```

```
data.frame(M)
```

```
##   X1 X2 X3 X4
## 1  1  4  7 10
## 2  2  5  8 11
## 3  3  6  9 12
```

```
colnames(M) <- c("a", "b", "c", "d")
as.data.frame(M)
```

```
##   a b c  d
## 1 1 4 7 10
## 2 2 5 8 11
## 3 3 6 9 12
```

```
data.frame(M)
```

```
##   a b c  d
## 1 1 4 7 10
## 2 2 5 8 11
## 3 3 6 9 12
```

5.4.1 Subsetování tabulek třídy *data.frame*

Tabulky jsou "kříženec" mezi seznamy a maticemi, takže je na ně možné je subsetovat jako matice i jako seznamy. Pokud použijete jeden index, pak je indexujete jako seznamy, pokud dva indexy, pak je indexujete jako matice. V prvním případě tedy `[` vrátí tabulku, ve druhém může vrátit tabulku (pokud se vybere více sloupců), nebo vektor (pokud se vybere jen jeden sloupec). Dolar i dvojité hranaté závorky vrací jeden sloupec, tj. vektor:

```
d <- data.frame(x = 1:7,
                y = c(3, 1, NA, 7, 5, 12, NA))
d
```

```
##   x  y
## 1 1  3
## 2 2  1
## 3 3 NA
## 4 4  7
## 5 5  5
## 6 6 12
## 7 7 NA
```

```
d$x      # vektor x
```

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

```
d[["x"]] # totéž
```

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

```
d[[1]]   # totéž
```

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

```
d["x"]   # tabulka s jediným sloupcem
```

```
##   x
## 1 1
## 2 2
## 3 3
## 4 4
## 5 5
## 6 6
## 7 7
```

```
d[1]     # opět tabulka s jediným sloupcem
```

```
##   x
## 1 1
## 2 2
## 3 3
## 4 4
## 5 5
## 6 6
## 7 7
```

```
d[1:2, "x"]              # vektor prvních dvou hodnot z vektoru x
```

```
## [1] 1 2
```

```
d[1:2, 1]                # totéž
```

```
## [1] 1 2
```

```
d[1:2, 1, drop = FALSE]  # tabulka složená z prvních dvou hodnot vektoru x
```

```
##   x
## 1 1
## 2 2
```

```
d[1:2, 1:2]              # tabulka složená z prvních dvou řádků
```

```
##   x y
## 1 1 3
## 2 2 1
```

```
d[1:2, c("x", "y")]      # tabulka složená z prvních dvou řádků
```

```
##   x y
## 1 1 3
## 2 2 1
```

```
d[1:2, ]                 # tabulka složená z prvních dvou řádků
```

```
##   x y
## 1 1 3
## 2 2 1
```

Samozřejmě je možné použít i indexování pomocí logických hodnot:

```
d[d[, "y"] < 7, ]  # výběr řádků, kde je hodnota y menší než 7
```

```
##       x  y
## 1     1  3
## 2     2  1
## NA   NA NA
## 5     5  5
## NA.1 NA NA
```

```
d[d$y < 7 , ]      # totéž
```

```
##       x  y
## 1     1  3
## 2     2  1
## NA   NA NA
## 5     5  5
## NA.1 NA NA
```

Výběr zachová i řádky, kde je hodnota *y* `NA`. To jde vyřešit např. takto:

```
# vybíráme pouze prvky, kde y zároveň není NA a zároveň je menší než 7 nebo
d[!is.na(d$y) & d$y < 7, ]
```

```
##   x y
## 1 1 3
## 2 2 1
## 5 5 5
```

K vyřazení neúplných hodnot z tabulky a podobných struktur slouží funkce `complete.cases()`. V případě tabulky vrací vektor logických hodnot, který je `TRUE` pro každý řádek tabulky, který má všechny hodnoty známé, a `FALSE` jinak.

```
complete.cases(d)
```

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

```
d[complete.cases(d), ]
```

```
##   x  y
## 1 1  3
## 2 2  1
## 4 4  7
## 5 5  5
## 6 6 12
```

Pro složitější výběry z tabulek existuje funkce `subset()`. Té však nebudeme věnovat pozornost, protože se později naučíte mnohem příjemnější a rychlejší funkce implementované v balíku **dplyr**, viz kapitola 16.

Do existující tabulky přidáte novou proměnnou (nový sloupec) tak, že do nové proměnné přidáte hodnoty vektoru:

```
d$z <- letters[1:nrow(d)]
d
```

```
##   x  y z
## 1 1  3 a
## 2 2  1 b
## 3 3 NA c
## 4 4  7 d
## 5 5  5 e
## 6 6 12 f
## 7 7 NA g
```

Nová proměnná se přidá jako poslední sloupec.

Pokud do existující proměnné přiřadíte hodnotu `NULL`, vyřadíte tím proměnnou z tabulky:

```
d$z <- NULL
d
```

```
##   x  y
## 1 1  3
## 2 2  1
## 3 3 NA
## 4 4  7
## 5 5  5
## 6 6 12
## 7 7 NA
```

Jiná možnost, jak vynechat proměnnou nebo změnit jejich pořadí, je využít subsetování sloupců tabulky.

Subsetování jde použít i ke změně pořadí řádků nebo sloupců. Řekněme, že chceme řádky tabulky `d` seřadit podle proměnné "y" a zároveň vyměnit pořadí sloupců. To můžeme udělat např. takto:

```
d[order(d$y), 2:1]
```

```
##    y x
## 2  1 2
## 1  3 1
## 5  5 5
## 4  7 4
## 6 12 6
## 3 NA 3
## 7 NA 7
```

Hodnoty `NA` skončí implicitně na konci (jde změnit ve funkci `order()`).

5.5 Tabulky třídy *tibble*

Kromě tabulek třídy *data.frame* existuje v R ještě několik dalších typů tabulek. Nejpohodlnější z nich je třída *tibble*, která je součástí **tidyverse**, skupiny balíků určených pro datovou analýzu, kterými se budeme zabývat v pozdějších částech tohoto textu. Tato třída tabulek má některé velmi příjemné vlastnosti, pro které je programování s tabulkami třídy *tibble* pohodlnější než programování s tabulkami třídy *data.frame*.

Tabulku třídy *tibble* vytvoříte pomocí funkce `tibble()`:

```
library(tibble)
ds <- tibble(x = 1:1e6, y = 2 * x, zed = x / 3 + 1.5 * y - 7)
ds
```

```
## # A tibble: 1,000,000 x 3
##        x     y    zed
##    <int> <dbl>  <dbl>
##  1     1     2 -3.67
##  2     2     4 -0.333
##  3     3     6  3
##  4     4     8  6.33
##  5     5    10  9.67
```

```
##  6     6    12 13
##  7     7    14 16.3
##  8     8    16 19.7
##  9     9    18 23
## 10    10    20 26.3
## # ... with 999,990 more rows
```

Pokud potřebujete vytvořit tabulku třídy *tibble* ručně, existuje i příjemná funkce `tribble()`, která umožňuje zadávat data po řádcích:

```
tribble(
    ~name, ~weight, ~height,
    "Adam", 68, 193,
    "Bětka", 55, 163,
    "Cyril", 103, 159
)
```

```
## # A tibble: 3 x 3
##   name  weight height
##   <chr>  <dbl>  <dbl>
## 1 Adam      68    193
## 2 Bětka     55    163
## 3 Cyril    103    159
```

Ke konverzi jiných tříd na *tibble* slouží funkce `as_tibble()`.

Vytvoření *tibble* se od vytvoření *data.frame* v několika ohledech liší: *tibble*

1. nemění "nepovolená" jména sloupců na povolená nahrazením divných znaků tečkami,
2. vyhodnocuje své argumenty postupně, takže můžete později zadaný argument použít při tvorbě dříve zadaného argumentu (jako v příkladu výše),
3. podporuje jen omezenou recyklaci: všechny zadané vektory musejí mít buď stejnou délku, nebo délku 1,
4. nepoužívá jména řádků (která jsou ostatně nebezpečná k uchovávání dat) a
5. převod na *tibble* pomocí `as_tibble()` je rychlejší než převod na *data.frame* pomocí `as.data.frame()`.

Liší se také to, jak *tibble* vypisuje svou hodnotu do konzole. Oproti *data.frame* zobrazí *tibble* navíc rozměr tabulky a typ jednotlivých proměnných. Naopak vypíše jen prvních deset řádků a jen takový počet sloupců, které se vejdou na obrazovku. Počet vypsaných řádků je možné ovlivnit ve funkci `print()` pomocí parametru `n`; počet sloupců pomocí parametru `width`, kde `width` je maximální počet znaků, které může *tibble* při tisku použít:

```
print(ds, n = 5)
```

```
## # A tibble: 1,000,000 x 3
##       x     y    zed
##   <int> <dbl>  <dbl>
## 1     1     2 -3.67
## 2     2     4 -0.333
## 3     3     6  3
## 4     4     8  6.33
## 5     5    10  9.67
## # ... with 999,995 more rows
```

Hlavní rozdíl mezi *tibble* a *data.frame* se však týká subsetování: *tibble* má konzistentnější chování a (na rozdíl od *data.frame*) vrací vždy stejnou datovou strukturu, což je výhodné zejména při programování. *tibble* nikdy nezahazuje zbytečné rozměry a ani je nepřidává. To znamená, že `[]` vždy vrací *tibble*, zatímco `[[]]` a `$` vždy vrací vektor. Navíc *tibble* nikdy nepodporuje partial matching:

```
ds <- ds[1:6, ]  # omezíme ds na prvních 6 řádků
ds[, 1]
```

```
## # A tibble: 6 x 1
##       x
##   <int>
## 1     1
## 2     2
## 3     3
## 4     4
## 5     5
## 6     6
```

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

```
## [1] 1 2 3 4 5 6
```

```
ds$z
```

```
## Warning: Unknown or uninitialised column: `z`.
```

```
## NULL
```

Tato příjemná konzistence má však i svá nebezpečí. Technicky je *tibble* (stejně jako *data.frame*) seznamem vektorů. Přestože jsme zatím vždy uvažovali atomické vektory, ve skutečnosti mohou být sloupci *tibble* i seznamy. To znamená, že následují kód ošklivě selže, protože do proměnné vloží celou tabulku:

```
ds <- tibble(a = 1:3, b = 11:13)
ds$c <- ds[, "a"]  # vybere se tibble s jedním sloupcem a vloží do sloupce b
ds
```

```
## # A tibble: 3 x 3
##       a     b   c$a
##   <int> <int> <int>
## 1     1    11     1
## 2     2    12     2
## 3     3    13     3
```

```
ds$c
```

```
## # A tibble: 3 x 1
##       a
##   <int>
## 1     1
## 2     2
## 3     3
```

```
class(ds$c)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

Správný postup je opět použít operátor, který z dat vybere jeden sloupec:

```
ds$d <- ds$a
ds$e <- ds[["a"]]
ds
```

```
## # A tibble: 3 x 5
##       a     b   c$a     d     e
##   <int> <int> <int> <int> <int>
## 1     1    11     1     1     1
## 2     2    12     2     2     2
## 3     3    13     3     3     3
```

```
class(ds$d)
```

```
## [1] "integer"
```

```
class(ds$e)
```

```
## [1] "integer"
```

Možnost využívající dvojité hranaté závorky se hodí zejména v případě, kdy máte jméno sloupce uložené v nějaké proměnné, takže je nemůžete zapsat přímo:

```
name <- "a"
ds$f <- ds[[name]]
ds
```

```
## # A tibble: 3 x 6
##       a     b   c$a     d     e     f
##   <int> <int> <int> <int> <int> <int>
## 1     1    11     1     1     1     1
## 2     2    12     2     2     2     2
## 3     3    13     3     3     3     3
```

Někdy se nehodí pracovat s *tibble* (např. proto, že některé funkce očekávají jiný výsledek subsetování). V takovém případě můžete *tibble* převést na *data.frame* pomocí konverzní funkce `as.data.frame()`:

```
as.data.frame(ds)
```

```
## Warning in format.data.frame(if (omit) x[seq_len(n0), , drop = FALSE] else x, :
## corrupt data frame: columns will be truncated or padded with NAs
```

```
##   a  b                 c d e f
## 1 1 11 # A tibble: 3 x 1 1 1 1
## 2 2 12                 a 2 2 2
## 3 3 13             <int> 3 3 3
```

5.6 Operátor trubka (|> a %>%)

Při datové analýze se často hodí používat operátor "trubka". R má dnes dvě základní verze tohoto operátoru: operátor `|>`, který je součástí základního R (od verze 4.1) a operátor `%>%` definovaný v balíku **magrittr** (načtou jej však i některé další balíky, jako např. **dplyr**). RStudio zavádí pro "trubku" speciální klávesovou zkratku `Ctrl-Shift-M`. V menu `Tools→Global Options...→Code→Editing` si můžete vybrat, zda se použije nativní verze, nebo verze z balíku **magrittr**.

Trubka umožňuje zapsat výraz se složitě zanořenými funkcemi poněkud čitelnějším způsobem. Řekněme, že chceme na proměnnou `x` aplikovat funkci `f()` a na výsledek funkci `g()`. Standardním způsobem bychom to zapsali takto:

```
g(f(x))
```

Operátor trubka umožní výraz přepsat do čitelnější podoby:

```
library(magrittr)
x |> f() |> g()  # nebo
x %>% f() %>% g()
```

Tato syntaxe lépe vystihuje to, co chceme provést. Doslova říká: vezmi proměnnou `x` a vlož ji jako první argument do funkce `f()`. Výsledek vlož jako první argument do funkce `g()`. Výrazy `x |> f()` a `x %>% f()` jsou tedy ekvivalentní výrazu `f(x)`. Funkce `f()` a `g()` samozřejmě mohou obsahovat i další parametry. Ty se zařadí za proměnnou `x`, takže `x |> mean(na.rm = TRUE)` je ekvivalentní výrazu `mean(x, na.rm = TRUE)`; stejně tak výraz s `%>%`.

Mnoho funkcí, se kterými budeme pracovat, je přizpůsobeno používání trubek, takže data do nich vstupují jako první argument. Některé funkce však berou data na jiné pozici. V takovém případě není možné použít základní operátor `|>`, který dosadí výraz na levé straně vždy na první pozici funkce na pravé straně. Verze `%>%` z balíku **magrittr** umožňuje toto omezení obejít pomocí speciální proměnné tečka (`.`). Do té trubka vloží vstupní data. Díky tomu je možné zapsat výraz `f(y, x)` jako `x %>% f(y, .)` a `f(y, z = x)` jako `x %>% f(y, z = .)`.

Pozor: Pokud je speciální proměnná tečka použita ve funkci přímo, pak trubka nevloží proměnnou `x` na první místo ve funkci (jak jsme to viděli v předchozích případech). Pokud je však tečka použita jako součást nějakého výrazu, pak trubka proměnnou `x` na první místo ve funkci vloží! To znamená, že `x %>% f(y = nrow(.), z = ncol(.))` je ekvivalentní s `f(x, y = nrow(x), z = ncol(x))`. Pokud tomu chcete zabránit, musíte funkci `f()` uzavřít do bloku. Takže `f(y = nrow(x), z = ncol(x))` musíme zapsat pomocí trubky jako `x %>% {f(y = nrow(.), z = ncol(.))}`.[1]

Podívejme se na typický příklad použití trubek. Řekněme, že máme tabulku `df` (zde si ji nasimulujeme) a chceme pro každou unikátní hodnotu identifikátoru `x` spočítat průměrné hodnoty proměnných `y` a `z`. To můžeme udělat např. takto:

```
library(tibble)
library(dplyr)
df <- tribble(
    ~x, ~y, ~z,
     1,  1,  1,
     1,  2,  3,
     2,  3,  5,
     2,  4,  6,
     3,  5,  7
)
df |> group_by(x) |> summarize(my = mean(y), mz = mean(z))
```

[1] Nativní trubka `|>` je méně flexibilní, je však rychlejší. Ve většině případů na tom nezáleží, protože volání trubek trvá v řádu nano až mikrosekund. Pokud však opakovaně simulujete velké množství vzorků dat, může být i takový drobný rozdíl užitečný.

```
## # A tibble: 3 x 3
##       x    my    mz
##   <dbl> <dbl> <dbl>
## 1     1   1.5   2
## 2     2   3.5   5.5
## 3     3   5     7
```

Výraz s použitím trubky %>% dá stejný výsledek:

```
df %>% group_by(x) %>% summarize(my = mean(y), mz = mean(z))
```

Detaily fungování tohoto kódu si vysvětlíme v kapitole 16. Nyní nás zajímá jen fungování trubek: proměnná `df` se nejdříve vloží do funkce `group_by()` a vyhodnotí se jako `group_by(df, x)`. Výsledek této operace se pak vloží na první místo do funkce `summarize()` a vyhodnotí se jako `summarize(., my = mean(y), mz = mean(z))`, kde tečka označuje výsledek předchozího výpočtu, tj. funkce `group_by()`.

Balík **magrittr** definuje i některé další operátory. Seznámit se s nimi můžete na webu balíku http://magrittr.tidyverse.org/ a v kapitole "Pipes" ve Wickham and Grolemund (2017) dostupné na http://r4ds.had.co.nz/pipes.html. Doporučuji však omezit se více méně jen na základní operátory |> a %>%.

5.7 Poznámka k doplňování hodnot do tabulek

Stejně jako v případě ostatních základních datových struktur v R, je i v případě tabulek možné přiřadit novou hodnotu do výběru. To se hodí např. v situaci, kdy potřebujete nějakou hodnotu opravit nebo aktualizovat. Často se také hodí ukládat vypočtené nebo stahované hodnoty do tabulky. V tomto posledním případě je však potřeba jisté opatrnosti, protože ukládání do výběru tabulky je výpočetně extrémně nákladná operace (kvůli nákladům na vyhledání dané hodnoty). V případě malých tabulek to není problém, v případě tabulek o milionech řádků to už však může být problematické. Ukažme si to na příkladu (nenechte se vyvést z míry tím, že nebudete rozumět všem detailům; většině porozumíte později v tomto textu).

Nejdříve vytvoříme dva vektory o délce 10 milionů hodnot, první celočíselný a druhý reálný. Vektory naplníme chybějícími hodnotami `NA`. Vytvoříme i dvě tabulky, které naplníme těmito vektory. První tabulka bude třídy *data.frame*, druhá třídy *tibble*. Náš test spočívá v tom, že do každého vektoru budeme chtít vložit na stejnou pozici jedničku. Funkce `microbenchmark()` ze stejnojmenného balíku nám změří, jak dlouho bude tato operace trvat.

```
library(microbenchmark)
library(dplyr)
library(tibble)

x <- rep(NA_integer_, 1e7)
y <- rep(NA_real_, 1e7)
df <- data.frame(x = x, y = y)
dt <- tibble(x = x, y = y)

performance <- microbenchmark(
    "vektory" = {x[1000] <- 1L; y[1000] <- 1},
    "data.frame 1" = {df[1000, "x"] <- 1L; df[1000, "y"] <- 1},
    "data.frame 2" = {df$x[1000] <- 1L; df$y[1000] <- 1},
    "data.frame 3" = df[1000, ] <- data.frame(x = 1L, y = 1),
    "tibble 1" = {dt[1000, "x"] <- 1L; dt[1000, "y"] <- 1},
    "tibble 2" = {dt$x[1000] <- 1L; dt$y[1000] <- 1},
    "tibble 3" = dt[1000, ] <- tibble(x = 1L, y = 1),
    unit = "ms"
) %>%
    summary() %>% select(expr, min, mean, median, max)
```

Table 5.2: Čas potřebný na vložení dvou hodnot do vektorů a tabulek v milisekundách.

expr	min	mean	median	max
vektory	0.00098	0.5402052	0.0073015	53.27643
data.frame 1	50.16850	65.2382566	52.7233060	139.45085
data.frame 2	33.15991	63.0271840	52.1328995	137.33941
data.frame 3	50.89538	67.8227680	54.0518705	135.06075
tibble 1	50.75956	64.0785199	52.9026900	137.56557
tibble 2	50.59952	68.1711059	52.9680190	137.13886
tibble 3	51.27623	66.2451496	53.5862770	139.26486

Výsledky jsou dramaticky odlišné, jak ukazuje tabulka 5.2. Vložení dvou hodnot do tabulek trvá (na mém poměrně výkonném počítači) v průměru několik desítek milisekund, zatímco vložení do vektoru trvá na stejném počítači tisíciny milisekund (pokud se díváme na mediány). Praktický výsledek je zřejmý: pokud vkládáte do tabulky jednotlivé hodnoty, nevadí to, pokud je tabulka malá nebo vkládáte jen velmi málo hodnot. V opačném případě se výrazně vyplatí pracovat s vektory. Jednou jsem potřeboval zjistit vzdálenosti vzdušnou čarou mezi asi 5 000 čerpacími stanicemi. Bylo tedy potřeba spočítat a uložit asi 12 milionů vzdáleností (vzdálenost vzdušnou čarou je symetrická). Vlastní výpočet vzdálenosti je extrémně rychlý. Přesto však celý výpočet neskončil ani za čtyři dny – spočítalo se asi jen 1.5 milionu hodnot. Na vině bylo to, že jsem doplňoval hodnoty přímo do tabulky. Když jsem kód přepsal tak, aby pracoval s vektory, které jsem spojil do tabulky až nakonec, výpočet proběhl zhruba za půl hodiny.

Všimněte si také toho, že jsem vektory a tabulky celé předalokoval. Kdybychom chtěli datové struktury zvětšovat postupně přidáváním dalších a dalších hodnot nebo řádků, byl by výpočet ještě pomalejší, protože by R muselo neustále alokovat nové bloky paměti a do nich nejdříve překopírovat stará data, a teprve potom přidat nové hodnoty. To by se opakovalo při každém zvětšení vektoru nebo tabulky.

Pokud se chcete dozvědět víc o efektivnosti kódu v R, doporučuji zejména Wickham (2014), kap. Performance a Profiling. Užitečné je také Burns (2011).

5.8 Volba datové struktury

Vstupní data pro jakoukoli analýzu budou mít ve většině případů formát tabulky. Naproti tomu si strukturu dat, která vzniknou transformacemi původních dat, můžete zvolit sami. Tuto strukturu byste si měli dopředu pořádně rozmyslet. Vhodně zvolená struktura vám umožní s daty pracovat jednoduše; špatně zvolená struktura může v následné práci dělat problémy.

Ve většině případů doporučuji používat pro uschování jakýchkoli dat tabulky (nejlépe třídy *tibble*). Oproti jiným strukturám mají několik výhod: 1) snadno se ukládají, čtou a převádí do jiného software, 2) snadno se z nich dělají výběry, 3) snadno se transformují a 4) snadno se vizualizují, ať už jako tabulky nebo v grafech. R nabízí mnoho balíků pro transformace tabulek (zejména **tidyr** a **dplyr**) a pro jejich vizualizaci (zejména **ggplot2**); o těchto balících bude řeč později. Práce s ostatními datovými strukturami je mnohem méně standardizovaná, takže si víc kódu budete muset napsat sami.

Použití jiné datové struktury k úschově dat má smysl pouze ve speciálních situacích: 1) pokud jinou strukturu (typicky vektory nebo matice) vyžadují použité funkce jako své vstupy, 2) když potřebujete rychlou maticovou aritmetiku a 3) když jsou složitější objekty výsledkem výpočtů (např. modelové objekty v ekonometrii). Pro dočasnou úschovu nehomogenních dat se hodí i seznamy, které typicky vznikají iteracemi nad vektory. Přesto vám výrazně doporučuji, abyste si pokaždé, když budete chtít zvolit jinou datovou strukturu než tabulku, tuto volbu raději několikrát promysleli.

6 Speciální datové typy

Kromě základních datových typů existují v R i další datové typy, které jsou implementované jako třídy objektů (o nich se více dozvíte v kapitole 9). Z nich nejdůležitější jsou faktory a třídy pro uchovávání datumů a času. Všechny tyto typy jsou implementovány jako rozšíření datových typů integer nebo double.

V této kapitole se naučíte

- vytvářet, používat a měnit faktory a
- základy práce s datumy a časem

6.1 Faktory

Faktory slouží k uchovávání kategoriálních proměnných. Kategoriální proměnné jsou proměnné, kterou mohou nabývat jen určitých předem stanovených úrovní. Tyto úrovně mohou být buď uspořádané (pak se jedná o ordinální proměnné), nebo neuspořádané. Příkladem ordinální kategoriální proměnné je kvalita služeb zjišťovaná v dotazníku. Ta může nabývat např. hodnot "velmi špatná", "špatná", "průměrná", "dobrá" a "výborná". Jiné úrovně nejsou (v rámci kódování dotazníku) možné. Přitom platí, že hodnoty jsou uspořádané od nejhorší po nejlepší, takže vždy můžeme dvě úrovně porovnat a říci, která je lepší. Příkladem neordinální kategoriální proměnné je např. pohlaví, které může nabývat hodnot "žena" nebo "muž". Na rozdíl od předchozího případu zde není jasné pořadí, ve kterém by měly být hodnoty uspořádány.

Kategoriální proměnné je možné kódovat např. jako celá čísla: např. muž bude 0 a žena 1, nejhorší kvalita služby bude 0, druhá nejhorší 1 atd. To však není dobrý nápad hned z několika důvodů: 1) je obtížné pamatovat si, co která hodnota znamená, 2) R nebude vědět, jak s takovými proměnnými zacházet a bude je považovat za kardinální veličiny (tj. bude např. kvalitu služby "průměrnou" kódovanou jako 2 považovat za dvakrát lepší než kvalitu "špatnou" kódovanou jako 1, přestože jediné, co víme, je, že "průměrná" kvalita je lepší než "špatná", ale už ne o kolik nebo kolikrát) a 3) R nebude schopné hlídat, zda není zadána nesmyslná úroveň proměnné (např. kvalita služby 7). Faktory řeší všechny tyto problémy: jednotlivým hodnotám dávají "nálepky", které ukazují na jejich význam, a zároveň říkají R, že se jedná o kategoriální proměnnou. Ve statistické analýze pak zachází s faktory správně; v ekonometrické analýze např. automaticky vytvoří pro jednotlivé úrovně faktoru potřebné umělé proměnné. R také zná platné úrovně faktorů a hlídá, zda je zadaná úroveň platná.

Faktory se tvoří pomocí funkce `factor()`. Ta vyžaduje nutně pouze vektor řetězců, který obsahuje hodnoty, které se mají na faktor převést:

```
factor(c("žena", "muž", "muž", "žena"))
```

```
## [1] žena muž  muž  žena
## Levels: muž žena
```

V tomto případě R odhaduje úrovně faktoru z dat, což není příliš bezpečné, protože v konkrétním datovém vzorku může některá platná úroveň faktoru chybět a některá zadaná úroveň nemusí být platná (např. kvůli chybě zapisovatele do dotazníků). Obecně je bezpečnější říct funkci `factor()` i to, jakých hodnot může faktor nabývat (pomocí parametru `levels`). To umožní zadat i hodnoty, které nyní zadaný vektor neobsahuje, ale obsahovat by je mohl, a také určit pořadí faktorů, které R jinak řadí podle abecedy. (Pořadí úrovní je

důležité zejména pro ordinální faktory. Ovšem i u neordinálních faktorů při některých statistických metodách určuje, která hodnota bude brána jako referenční úroveň.) Parametr `labels` navíc umožňuje úrovně faktorů překódovat:

```
factor(c("male", "female", "female", "male", "female"),  # hodnoty vektoru
       levels = c("female", "male", "asexual"),           # možné úrovně
       labels = c("žena", "muž", "asexuální"))             # co se bude vypisovat
```

```
## [1] muž  žena žena muž  žena
## Levels: žena muž asexuální
```

Všimněte si, že při vypsání faktor vypisuje nejen hodnoty vektoru, nýbrž i seznam hodnot, kterých mohou nabývat.

Pokud se pokusíte uložit do faktoru hodnotu, která neodpovídá zadaným úrovním, R danou hodnotu tiše nahradí hodnotou `NA`:

```
factor(c("male", "female", "female", "male", "beaver"),  # hodnoty vektoru
       levels = c("female", "male", "asexual"),           # možné úrovně
       labels = c("žena", "muž", "asexuální"))             # co se bude vypisovat
```

```
## [1] muž  žena žena muž  <NA>
## Levels: žena muž asexuální
```

To je velmi šikovné, protože to umožňuje hlídat, zda jsou všechny hodnoty zadané správně. Řekněme, že svá data stahujete z nějakého serveru, který s vámi nespolupracuje a může kdykoli bez varování změnit kódování některých kategoriálních hodnot. Jedna možnost, jak to zjistit, je převádět je z řetězců na faktory s pevně zadanými hodnotami úrovní. Pokud se mezi hodnotami faktoru objeví `NA`, znamená to, že server změnil kódování dané proměnné.

Implicitně jsou všechny faktory ne-ordinální. Pokud chcete R říci, že faktor je ordinální, přidáte parametr `ordered = TRUE`. Pak na faktory funguje porovnání větší a menší:

```
quality <- factor(c("poor", "satisfactory", "excelent"),
                  levels = c("poor", "satisfactory", "excelent"),
                  ordered = TRUE)
quality
```

```
## [1] poor         satisfactory excelent
## Levels: poor < satisfactory < excelent
```

```
quality[1] < quality[3]  # quality[i] je i-tý prvek vektoru
```

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

Funkce `is.ordered()` vrací logickou hodnotu `TRUE`, pokud je faktor ordinální.

Pozor! Pokud opravdu dobře nevíte, co děláte, používejte raději ne-ordinální faktory. Ordinální faktory se ve formulích (tj. např. v ekonometrické analýze, viz kapitola 18) chovají jinak, než byste možná čekali, viz https://goo.gl/HY3uNf a https://goo.gl/F9Shll, oddíl 11.1.1.

Hodnoty úrovní můžete získat pomocí funkce `levels()`; jejich počet pomocí funkce `nlevels()`. Funkce `levels()` umožňuje i měnit hodnoty úrovní faktoru:

```
f <- factor(c("female", "male", "female"))
f
```

```
## [1] female male   female
## Levels: female male
```

```
levels(f) <- c("a", "b", "c")
f
```

```
## [1] a b a
## Levels: a b c
```

6.1.1 Tvorba faktoru ze spojité proměnné

Někdy je užitečné rozdělit spojitou škálu do diskrétních hodnot, takže spojitou proměnnou změníte na ordinální faktor. Např. můžeme chtít rozdělit studenty do tří kategorií podle jejich studijního průměru: na excelentní žáky (do průměru 1.2 včetně), běžné žáky (od průměru 1.2 do 2.5 včetně) a ostatní. K tomu slouží funkce `cut()`:

```
grades <- c(1.05, 3.31, 2.57, 1.75, 2.15)  # studijní průměry
students <- cut(grades, breaks = c(0, 1.2, 2.5, Inf), right = TRUE)
students
```

```
## [1] (0,1.2]   (2.5,Inf] (2.5,Inf] (1.2,2.5] (1.2,2.5]
## Levels: (0,1.2] (1.2,2.5] (2.5,Inf]
```

```
levels(students) <- c("excelent", "normal", "rest")
students
```

```
## [1] excelent rest     rest     normal   normal
## Levels: excelent normal rest
```

Změnu názvů úrovní je možné nastavit přímo ve funkci `cut()` parametrem `labels`. Detaily použití funkce `cut()` najdete v dokumentaci funkce.

Jako mnoho jiných funkcí ve standardní výbavě R má i funkce `cut()` mnoho (na první pohled) složitých parametrů. Balík **ggplot2** nabízí tři specializovanější funkce, které udělají stejnou práci jako `cut()`, ale jejich použití je jednodušší: `cut_width()` nařeže původní proměnnou na úrovně s danou šířkou, `cut_interval()` na daný počet úrovní se stejnou šířkou a `cut_number()` na daný počet úrovní se stejným počtem pozorování.

6.1.2 Převod faktorů na čísla

Faktory jsou užitečné, ale i zrádné. Technicky jsou implementované jako vektor celých čísel, který má navíc pojmenované úrovně:

```
unclass(factor(c("žena", "muž", "muž", "žena")))
```

```
## [1] 2 1 1 2
## attr(,"levels")
## [1] "muž"  "žena"
```

Při automatické konverzi se tedy může stát, že se faktor převede na celá čísla – svých úrovní. Většinou to nevadí, existují však zrádné situace, kdy název úrovně faktoru je složen z číslic. Nejdříve si to ukážeme na poněkud legračním příkladu:

```
f <- factor(c("747", "737", "777", "747"))  # vektor typů vašich letadel Boeing
f
```

```
## [1] 747 737 777 747
## Levels: 737 747 777
```

```
as.integer(f)  # chcete dostat zpět typy letadel, ale ouha! dostanete čísla úrovní!
```

```
## [1] 2 1 3 2
```

```
as.integer(as.character(f))  # je potřebná dvojí konverze!
```

```
## [1] 747 737 777 747
```

Realističtější situace vznikne, když faktory odpovídají nařezané spojité proměnné. Zde si ji vytvoříme, ale typicky už je taková proměnná obsažena v datech:

```
x <- rnorm(100)  # sto čísel náhodně vybraných z normovaného norm. rozdělení
h <- cut(x,  # faktor nařezaný od -10 po +10 po dvou, jméno binu je střed intervalu
         breaks = seq(from = -10, to = 10, by = 2),
         labels = seq(from = -9, to = 9, by = 2))
h
```

```
##   [1] 1  1  -1 -1 -1 1  1  -1 1  -1 1  1  1  1  -1 1  1  1  1  -1 -1 1  -1 1  -1
##  [26] 3  -1 1  1  1  1  1  1  1  1  1  -1 1  -1 1  3  -1 -1 1  -1 -1 1  -1 -1 1
##  [51] 1  -1 -1 3  -1 -1 1  -1 -1 1  1  1  1  -1 -1 1  -1 -1 1  -3 -1 1  1  1  -1
##  [76] -1 -1 1  1  1  1  -1 1  -1 -1 1  1  -1 1  -1 -1 -1 -1 1  1  1  -1 1  1  1
## Levels: -9 -7 -5 -3 -1 1 3 5 7 9
```

Řekněme, že chceme spočítat průměrnou hodnotu pozorování a zkusíme to udělat takto:

```
mean(as.numeric(h))
```

```
## [1] 5.59
```

V našem případě bychom čekali výsledek někde kolem nuly, takže náš výsledek je očividně nesprávný. V případě skutečných dat s neznámým rozdělením si však nemusíte všimnout ničeho podezřelého. Chybný výsledek vznikl tak, že R prvně převedlo funkcí `as.numeric()` faktory na jejich úrovně označeného 1, 2 atd. a z nich následně spočítalo průměr. To, co jsme chtěli dostat, bylo ve skutečnosti toto:

```
mean(as.numeric(as.character(h)))
```

```
## [1] 0.18
```

```
mean(as.numeric(levels(h)[h]))  # totéž jinými slovy
```

```
## [1] 0.18
```

6.1.3 Úpravy faktorů

Většinou pracujeme s faktory tak, jak jsou. Někdy je však potřebujeme nějakým způsobem transformovat: vyřadit nepoužité úrovně, spojit různé faktory, přejmenovat jejich úrovně apod. Některé z těchto úprav jsou překvapivě netriviální. Naštěstí existuje balík, který tyto úpravy zjednodušuje. Zde se podíváme jen na vybrané základní situace. Pro náročnější situace doporučujeme se podívat na dokumentaci balíku **forcats**.

Obvykle nejpotřebnější úpravou faktorů je vyřazení nepotřebných úrovní. Pokud totiž z faktoru vybereme jen některá pozorování, zůstanou ve faktoru zachovány všechny úrovně, i ty, kterým ve vektoru už neodpovídá žádné pozorování:

```
h[1:5]
```

```
## [1] 1  1  -1 -1 -1
## Levels: -9 -7 -5 -3 -1 1 3 5 7 9
```

Pokud chceme nepoužité úrovně vypustit, musíme operátoru hranatá závorka dát parametr `drop=TRUE`:

```
h[1:5, drop = TRUE]
```

```
## [1] 1  1  -1 -1 -1
## Levels: -1 1
```

Druhou častou úpravou je spojování dvou vektorů faktorů. Od verze 4.1 funguje spojování faktorů přesně tak, jak byste očekávali:

```
f1 <- factor(c("a", "b", "c"))
f2 <- factor(c("x", "y", "z"))
c(f1, f2)
```

```
## [1] a b c x y z
## Levels: a b c x y z
```

Ve starších verzích to nebylo tak jednoduché, protože funkce `c()` převedla faktory na celá čísla (jejich úrovně) a nálepky úrovní zahodila. Pokud se navíc úrovně faktorů lišily, tento fakt byl zcela zapomenut. Spojení faktorů pak bylo třeba provést oklikou: před spojením faktory převést na jejich úrovně (řetězce), ty spojit, a výsledek převést opět na faktor:

```
factor(c(as.character(f1), as.character(f2)))
```

```
## [1] a b c x y z
## Levels: a b c x y z
```

```
factor(c(levels(f1)[f1], levels(f2)[f2]))  # totéž jinými slovy
```

```
## [1] a b c x y z
## Levels: a b c x y z
```

V moderním R to však už naštěstí není potřeba. Faktory jsou díky tomu podstateně bezpečnější.

Poslední obvyklou transformací faktorů je změna pořadí jejich úrovní. K tomu existují v principu dva důvody: Zaprvé, někdy potřebujeme nastavit referenční úroveň faktoru. Např. v ekonometrii R samo vytvoří pro jednotlivé úrovně faktoru potřebné umělé proměnné. Přitom samozřejmě jednu úroveň (referenční úroveň nebo také kontrast) vynechá. R automaticky vynechává první úroveň. Pokud jsme nezadali jména úrovní explicitně, pak R vynechá tu úroveň, která je první v abecedě. To však často není to, co chceme. Změnu referenční úrovně provedeme snadno pomocí funkce `relevel()`. Jejím první parametrem je faktor, druhým je nová referenční úroveň (všimněte si pořadí úrovní na konci výpisu faktoru):

```
relevel(h, "1")
```

```
##   [1] 1  1  -1 -1 -1 1  1  -1 1  -1 1  1  1  1  -1 1  1  1  1  -1 -1 1  -1 1  -1
##  [26] 3  -1 1  1  1  1  1  1  1  1  1  -1 1  -1 1  3  -1 -1 1  -1 -1 1  -1 -1 1
##  [51] 1  -1 -1 3  -1 -1 1  -1 -1 1  1  1  1  -1 -1 1  -1 -1 1  -3 -1 1  1  1  -1
##  [76] -1 -1 1  1  1  1  -1 1  -1 -1 1  1  -1 1  -1 -1 -1 -1 1  1  1  -1 1  1  1
## Levels: 1 -9 -7 -5 -3 -1 3 5 7 9
```

Zadruhé, někdy chceme seřadit úrovně faktorů podle nějaké charakteristiky dat – např. proto, že pořadí úrovní ovlivňuje pořadí, ve kterém jsou data vykreslena v grafu. Toto je překvapivě netriviální problém. Můžete použít bud základní funkci `reorder()`, která je velmi mocná, ale má poněkud komplikované ovládání, nebo přívětivější funkce z balíku **forcats**. Návod na jejich použití najdete v kapitole o balíku **forcats* v knize Wickham and Grolemund (2017) dostupné také na http://r4ds.had.co.nz/factors.html nebo v referenční příručce balíku, která je dostupná na http://forcats.tidyverse.org/.

6.1.4 Poznámka k faktorům

Historicky se faktory používaly velmi často, protože šetřily paměť počítače. Místo vektoru řetězců, kde se hodnoty často opakovaly, zbyl krátký vektor unikátních hodnot řetězců (úrovně `levels`) a paměťově úspornější vektor celých čísel, který říkal, která úroveň se právě používá. Dnes to však už není pravda: R skladuje řetězce v jednom velkém skladu, každý z nich pouze jednou a vektory řetězců jsou implementovány jako odkazy do tohoto skladu. Prakticky to znamená, že převedením řetězců na faktory se žádná paměť neušetří.

Nicméně, z těchto historických důvodů se funkce ze základních balíků R často snaží řetězce převádět na faktory. Příkladem takových funkcí jsou např. funkce `data.frame()` a `read.csv()`, které historicky převáděly řetězce na faktory. Tomu šlo zabránit nastavením parametru `stringsAsFactors` na hodnotu `FALSE`. R od verze 4.0 však tyto konverze v těchto funkcích neprovádí. Pokud tedy chcete převést řetězec na faktor, musíte buď nastavit explicitně parametr `stringsAsFactors` na hodnotu `TRUE`, nebo převést proměnnou na faktor ručně pomocí funkce `factor()`, což je mnohem rozumnější.

6.2 Datum a čas

Datum a čas lze v R ukládat do vektorů několika různých tříd. Z nich jsou nejdůležitější jsou třídy *Date* pro reprezentaci celých dnů bez času a *POSIXct* pro reprezentaci dne i hodiny. Datové typy *Date* a *POSIXct* nepatří mezi základní datové typy v R, ale jedná se o třídy objektů odvozené od tříd *integer* a *double*. První ukládá datum tak, že celé číslo reprezentuje počet dnů od 1. ledna 1970, druhá reprezentuje čas jako počet sekund, které od tohoto dne uběhl. Při výpisu však obě funkce zobrazují datum uživatelsky přátelským způsobem.

R nabízí pro práci s datumy a časem několik užitečných funkcí, které však mohou být v určitých situacích poněkud těžkopádné. Uživatelsky přívětivější funkce nabízí balík **lubridate**. Zde si ukážeme některé základní operace s datumy a časem jak pomocí základních funkcí, tak pomocí funkcí z balíku **lubridate**. Proto jej musíme nejprve načíst:

```
library(lubridate)
```

6.2.1 Zadávání datumů a času

Datum nejčastěji vytvoříme konverzí z řetězce. K tomu slouží funkce `as.Date()`, která předpokládá, že řetězec obsahuje datum uložené ve formátu obvyklém v anglofonním světě, kdy na prvním místě stojí rok, pak měsíc a nakonec den a jednotlivé položky jsou oddělené pomlčkou (tak R také datumy vypisuje):

```
as.Date("2016-11-25")  # 25. listopadu 2016
```

```
## [1] "2016-11-25"
```

Pokud je datum zadáno v jiném formátu, musíte tento formát popsat v parametru `format`. Formát se zadává pomocí formátovacích řetězců, které jsou popsané v dokumentaci funkce `strptime()`. Základní parametry uvádí tabulka 6.1. Řekněme, že máme data zadaná ve formátu číslo dne, zkratka měsíce a rok, kde jednotlivé položky nejsou nijak oddělené. Pak bude mít řetězec formátu tvar "%d%b%Y": "%d" znamená, že na prvním místě je den v měsíci, "%b" říká, že na druhém místě je zkrácený název měsíce a "%Y" že na posledním místě je rok zadaný čtyřmi číslicemi.

```
as.Date(c("1led1960", "2led1960", "31bře1960", "30čec1960"), format = "%d%b%Y")
```

```
## [1] "1960-01-01" "1960-01-02" "1960-03-31" "1960-07-30"
```

Table 6.1: Vybrané formátové značky pro zadávání datumu a času

kód	význam
%d	číslo dne v měsíci
%m	číslo měsíce v roce
%b	zkrácené jméno měsíce
%B	plné jméno měsíce
%y	rok zadaný dvěma číslicemi
%Y	rok zadaný čtyřmi číslicemi
%H	číslo hodiny (0–23)
%M	číslo minuty (0–59)
%S	číslo sekundy (0–61 pro přestupné vteřiny)

Pokud by jednotlivé položky datumu byly nějak oddělené, zadaly by se tyto oddělovače do formátu. Datum ve formátu "1. červen 1974" by mělo formátový řetězec "%d. %B %Y":

```
as.Date(c("1. červen 1974", "17. listopad 1989"), format = "%d. %B %Y")
```

```
## [1] NA NA
```

Jména měsíců a dnů v týdnu standardně se berou z lokalizace vašeho operačního systému. Pokud vám tato lokalizace nevyhovuje (potřebujete jména v jiném jazyce) nebo musíte svůj kód přenášet mezi více počítači, můžete lokalizaci manuálně přepnout (víc detailů najdete v dokumentaci k `locales`: `?locales`):

```
# nastav locales, aby jména dnů byla anglicky
# (zde se tento řádek neprovede, aby byl zbytek textu česky)
Sys.setlocale("LC_TIME", "C")
```

Při načítání datumů je možné nastavit i časovou zónu pomocí parametru `tz`. Jinak se automaticky vezme časová zóna z vašeho operačního sytému (v mém případě nyní "CEST", tj. středoevropský letní čas).

Pokud máte dobrodružnější povahu a jednotlivé části datumu jsou zadané pomocí čísel dnů, měsíců a let, můžete použít pohodlnější funkce z balíku **lubridate**. Těchto funkcí je celá řada a jmenují se na první pohled krypticky `ymd()`, `ydm()`, `mdy()`, `myd()`, `dmy()` a `dym()`. Jednotlivá písmena určují, v jakém pořadí jsou uvedeny jednotlivé složky datumu: `y` znamená rok, `m` měsíc a `d` den, takže funkce `ymd()` předpokládá, že v datumu je nejdříve uveden rok, pak měsíc a nakonec den. Všechny tyto funkce jsou chytré, takže si poradí s nejrůznějšími oddělovači. Pokud funkce na některém prvku vektoru selže, vydá varování. (Všimněte si, jak funkce naloží s rokem zadaným jen dvěma číslicemi.)

```
dmy(c("1.6.1974", "17. 11. 1989", "1-1-2001", "1.3.99", "1.3.39", "bžů"))
```

```
## Warning: 1 failed to parse.
```

```
## [1] "1974-06-01" "1989-11-17" "2001-01-01" "1999-03-01" "2039-03-01"
## [6] NA
```

```
ymd(20140531)  # funguje i celé číslo, pokud je jednoznačné
```

```
## [1] "2014-05-31"
```

Datum je možné složit i z jednotlivých komponent pomocí funkce `make_date()`:

```
make_date(year = 2011, month = 7, day = 5)
```

```
## [1] "2011-07-05"
```

K zadání dne i hodiny slouží funkce `as.POSIXct()`, která opět implicitně předpokládá datum a čas v anglofonním formátu. Formát je opět možné změnit zadáním parametru `format`.

```
as.POSIXct("1974-06-01 7:30")
```

```
## [1] "1974-06-01 07:30:00 CET"
```

Balík **lubridate** opět nabízí uživatelsky přívětivější (ale méně striktní, a tedy bezpečné) funkce `ymd_hms()`, `ymd_hm()`, `ymd_h()`, `dmy_hms()`, `dmy_hm()`, `dmy_h()`, `mdy_hms()`, `mdy_hm()`, `mdy_h()`, `ydm_hms()`, `ydm_hm()` a `ydm_h()`, kde písmena `d`, `m` a `y` mají stejný význam jako výše a `h` znamená hodiny, `m` minuty a `s` sekundy:

```
ymd_hm("1974-06-01 7:30")
```

```
## [1] "1974-06-01 07:30:00 UTC"
```

```
dmy_hm("1. 6. 1974 7.30")
```

```
## [1] "1974-06-01 07:30:00 UTC"
```

I datum a čas je možné složit z jednotlivých komponent pomocí funkce

```
make_datetime(year = 2017, month = 1:12, day = 1)
```

```
##  [1] "2017-01-01 UTC" "2017-02-01 UTC" "2017-03-01 UTC" "2017-04-01 UTC"
##  [5] "2017-05-01 UTC" "2017-06-01 UTC" "2017-07-01 UTC" "2017-08-01 UTC"
##  [9] "2017-09-01 UTC" "2017-10-01 UTC" "2017-11-01 UTC" "2017-12-01 UTC"
```

6.2.2 Jednotlivé komponenty datumu a času

Často je potřeba z datumu nebo času získat příslušnou hodnotu dne, měsíce nebo hodiny. K je možné použít funkce `weekdays()`, `months()`, `days()` a `quarters()` ze základního balíku. Funkce `weekdays()` a `months()` vrací jména dnů a měsíců v *locace* počítače; mohou vrátit i zkrácenou verzi jména, pokud je parametr `abbreviate` nastaven na hodnotu `TRUE`.

```
d <- as.Date("2016-11-05")
weekdays(d)  # den v týdnu
```

```
## [1] "Sobota"
```

```
months(d)    # měsíc v roce
```

```
## [1] "listopadu"
```

```
quarters(d)  # čtvrtletí
```

```
## [1] "Q4"
```

Balík **lubridate** opět definuje další funkce pro práci s jednotlivými komponentami datumu a času: `year()` vrací rok, `month()` vrací měsíc, `mday()` vrací číslo dne v měsíci, `yday()` číslo dne v roce, `wday()` číslo dne v týdnu, `hour()` hodinu, `minute()` minutu a `second()` sekundu. Funkce `month()` a `wday()` mají navíc parametry `labels` a `abbr`. Pokud je `label` nastaveno na hodnotu `TRUE`, pak funkce vrací místo čísla jméno. Pokud je navíc `abbr` nastaveno na `TRUE`, pak je jméno zkráceno. Tyto funkce však vracejí vždy anglická jména dnů a měsíců, navíc kódovaná jako faktor. Také prvním dnem týdne je neděle (s číslem 1). Funkce `tz()` vrací časovou zónu.

```
year(d)
```

```
## [1] 2016
```

```
month(d)
```

```
## [1] 11
```

```
month(d, label = TRUE)
```

```
## [1] lis
## 12 Levels: led < úno < bře < dub < kvě < čen < čec < srp < zář < ... < pro
```

```
wday(d)
```

```
## [1] 7
```

```
wday(d, label = TRUE)
```

```
## [1] So
## Levels: Ne < Po < Út < St < Čt < Pá < So
```

```
tz(d)
```

```
## [1] "UTC"
```

Tyto funkce z balíku **lubridate** je možné použít i k úpravě datumu a času:

```
d <- ymd("2000-01-01")
day(d) <- 31
month(d) <- 12
d
```

```
## [1] "2000-12-31"
```

Pokud je potřeba upravit více prvků data naráz, je možné použít funkci `update()`:

```
update(d, year = 1989, month = 11, mday = 17)
```

```
## [1] "1989-11-17"
```

6.2.3 Operace s datem a časem

Pro třídy *Date* a *POSIXct* fungují některé aritmetické a logické operace, jako je sčítání, odečítání a diference, násobení, dělení a porovnávání:

```
d <- as.Date(c("2016-01-10", "2016-03-11"))
d
```

```
## [1] "2016-01-10" "2016-03-11"
```

```
d[2] - d[1]               # kolik dnů je mezi druhým a prvním dnem ve vektoru?
```

```
## Time difference of 61 days
```

```
diff(d)                   # funguje i funkce pro diferenci
```

```
## Time difference of 61 days
```

```
as.numeric(d[2] - d[1])  # převod na celé číslo
```

```
## [1] 61
```

```
d > "2016-02-05"         # které dny ve vektoru jsou po 5. únoru 2016?
```

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

Většina aritmetických operací však pracuje na rozdílech mezi dvěma datumy, což je proměnná třídy *difftime*. Práce s touto třídou není úplně intuitivní, protože datumy a čas jsou plné různých nepravidelností (počet dnů se v jednotlivých měsících liší, přestupné roky mají více dnů a některé minuty mají 61 sekund). Proto se zde blíže podíváme jen na dvě operace: zjištění časové vzdálenosti dvou bodů a tvorbu vektoru datumů s konstantním rozestupem.

Pokud nás zajímá časová vzdálenost mezi dvěma daty, obyčejná diference vypíše rozdíl v automaticky zvolených jednotkách. Užitečnější proto může být funkce `difftime()`, která vrací vzdálenost mezi dvěma dny v jednotkách zadaných uživatelem pomocí parametru `units`, který může nabývat hodnot `"auto"`, `"secs"`, `"mins"`, `"hours"`, `"days"` nebo `"weeks"`:

```
difftime(d[2], d[1], units = "weeks")
```

```
## Time difference of 8.714286 weeks
```

Balík **lubridate** definuje velké množství dalších funkcí datumovou aritmetiku a práci s časovými intervaly. Zde se jim však nebudeme věnovat. Zájemci se mohou podívat do Wickham and Grolemund (2017), kap. 13 dostupné na http://r4ds.had.co.nz/dates-and-times.html, na web balíku **lubridate** na http://lubridate.tidyverse.org/ nebo do původního článku o tomto balíku, Grolemund and Wickham (2011), který je dostupný ke stažení na stránkách časopisu i balíku **lubridate**.

6.2.4 Tvorba ekvidistantních časových vektorů

Často potřebujeme vytvořit vektor s datumy nebo časem s ekvidistantním odstupem. V některých speciálních případech to můžeme udělat pomocí funkcí pro práci s jednotlivými složkami datumu. Řekněme například, že potřebujeme vektor datumů pro začátek každého měsíce v roce. To uděláme snadno např. takto:

```
d <- ymd("2017-01-01")
month(d) <- 1:12
d
```

```
##  [1] "2017-01-01" "2017-02-01" "2017-03-01" "2017-04-01" "2017-05-01"
##  [6] "2017-06-01" "2017-07-01" "2017-08-01" "2017-09-01" "2017-10-01"
## [11] "2017-11-01" "2017-12-01"
```

Vytvořit jiné vektory je však obtížnější. Například vektor posledních dnů v měsíci takto jednoduše vytvořit nejde, protože různé měsíce mají různý počet dnů. Náš algoritmus také nevytvořil data s ekvidistantním odstupem, a to právě proto, že různé měsíce jsou různě dlouhé.

K vytvoření vektorů se skutečně stejným rozestupem mezi jednotlivými daty je možné použít funkci `seq()`. Stejně jako v případě číselných vektorů je možné nastavit počátek a konec období a počet hodnot, nebo počátek období a odstup mezi položkami. Ten se nastaví pomocí parametru `by`. Tento parametr může mít hodnotu "day", "week", "month", "quarter" nebo "year" případně násobenou celým číslem; ke jménu délky periody je také možné přidat koncové "s". Jednodenní odstup je pak "day", dvoudenní "2days" apod.

```
seq(from = ymd("2015-1-1"), to = ymd("2015-12-31"), length.out = 15)
```

```
##  [1] "2015-01-01" "2015-01-27" "2015-02-22" "2015-03-20" "2015-04-15"
##  [6] "2015-05-11" "2015-06-06" "2015-07-02" "2015-07-28" "2015-08-23"
## [11] "2015-09-18" "2015-10-14" "2015-11-09" "2015-12-05" "2015-12-31"
```

```
seq(ymd("2015-1-1"), by = "2 days", length.out = 10)
```

```
##  [1] "2015-01-01" "2015-01-03" "2015-01-05" "2015-01-07" "2015-01-09"
##  [6] "2015-01-11" "2015-01-13" "2015-01-15" "2015-01-17" "2015-01-19"
```

Kromě toho je odstup možné nastavit i na jakoukoli hodnotu třídy *difftime*, kterou vrací funkce `difftime()`, nebo na kteroukoli jednotku, kterou funkce `difftime()` dokáže zpracovat. Před jednotku je možné přidat celé číslo jako násobek této jednotky:

```
odstup <- ymd("2017-06-05") - ymd("2017-06-01")  # odstup 4 dny
seq(ymd("2017-06-01"), by = odstup, length.out = 6)
```

```
## [1] "2017-06-01" "2017-06-05" "2017-06-09" "2017-06-13" "2017-06-17"
## [6] "2017-06-21"
```

6.2.5 Systémový čas

Aktuální (systémový) čas lze zjistit funkcí `Sys.time()`, která vrací objekt třídy *POSIXct*:

```
Sys.time()
```

```
## [1] "2021-09-10 16:03:57 CEST"
```

Balík **lubridate** navíc přidává funkce `today()`, která vrací dnešní datum jako objekt třídy *Date*, a `now()`, která vrací okamžité datum a čas jako objekt třídy *POSIXct*:

```
today()  # den, kdy byla tato kniha naposledy zkompilována
```

```
## [1] "2021-09-10"
```

```
now()  # a nyní včetně času
```

```
## [1] "2021-09-10 16:03:57 CEST"
```

7 Řídící struktury

Zatím jsme předpokládali, že R provádí kód skriptu řádek po řádku. Někdy je však potřeba běh kódu různě modifikovat: některé řádky provést pouze, když je splněná určitá podmínka; jiné řádky provádět opakovaně; a někdy vypsat varování nebo zastavit běh skriptu s chybovým hlášením. Těmito problémy se budeme nyní zabývat.

V této kapitole se naučíte

- větvit kód pomocí podmínek
- opakovat kód pomocí cyklů
- vypisovat varování a zastavit kód chybovým hlášením
- že je možné chybová hlášení odchytit a vyřešit

7.1 Větvení kódu

Normálně R zpracovává skript tak, že vyhodnocuje řádek po řádku. Někdy to však nechceme. Některé řádky můžeme např. chtít provést pouze v případě, že je splněná nějaká podmínka. K tomu v R slouží podmínka `if`. Jediným argumentem `if` uvedeným v závorkách je podmínka, tj. výraz, který se musí vyhodnotit na logický vektor délky 1. Pokud je tento logický výraz splněný (má hodnotu `TRUE`), pak R vyhodnotí následující výraz. Pokud má logický výraz hodnotu `FALSE`, pak R následující výraz přeskočí. Ukažme si to na příkladu:

```
x <- 1
y <- 2
if (x == 1)
    print("O.K.: x je jedna!")
```

```
## [1] "O.K.: x je jedna!"
```

```
if (y == 1)
    print("O.K.: y je jedna!")
```

V tomto příkladě se první podmínka vyhodnotila jako pravdivá (`x` má opravdu hodnotu 1), takže R provedlo následující výraz a vypsalo "O.K.: x je jedna!". Naproti tomu druhá podmínka se vyhodnotila jako nepravdivá, takže druhý tiskový řádek R přeskočilo a nic nevypsalo. Všimněte si také, že k porovnání dvou hodnot se používá `==`.

Podmínka `if` se v R vztahuje vždy jen na jeden následující výraz. Pokud se má při splnění podmínky provést více než jeden řádek kódu, je třeba tyto řádky seskupit pomocí složených závorek (výrazy zabalené do složených závorek tvoří *blok* kódu):

```
if (x == 1) {
    a <- 5
    print("O.K.: x je jedna!")
}
```

```
## [1] "O.K.: x je jedna!"
```

```
a
```

```
## [1] 5
```

Nyní při splnění podmínky R nejdříve vloží do proměnné a hodnotu 5, a pak vypíše "O.K.: x je jedna!". Pokud by podmínka splněná nebyla, R by přeskočila oba řádky uvedené ve složených závorkách.

Někdy chceme, aby se část kódu provedla, pokud podmínka platí, zatímco jiná část kódu, pokud podmínka neplatí. K tomu slouží klauzule `else`. V následujícím kódu se vypíše "O.K.: y je jedna!", pokud je y rovno jedné, a "O.K.: y není jedna!", pokud se y od jedné liší.

```
if (y == 1) {
    print("O.K.: y je jedna!")
} else {
    print("O.K.: y není jedna!")
}
```

```
## [1] "O.K.: y není jedna!"
```

Pozor: `else` musí být na stejném řádku, jako končící složená závorka nebo kód, který se provádí při splnění podmínky.

Podmínky lze libovolně řetězit, jak ukazuje následující příklad. Nejdříve se porovná x s jedničkou. Pokud se x rovná jedné, pak se vypíše "jedna" a kód pokračuje za poslední složenou závorkou. Pokud se x od jedničky liší, prozkoumá se, zda je větší než jedna. Pokud je, vypíše se "větší", pokud není, vypíše se "menší". (Pozor, tento kód nepočítá s tím, že by x mohlo mít i hodnotu `NA` nebo `NaN` – vyzkoušejte si to.)

```
x <- 1
if (x == 1) {
    print("jedna")
} else if (x > 1) {
    print("větší")
} else {
    print("menší")
}
```

```
## [1] "jedna"
```

Je také možné vnořit jednu podmínku do jiné, jak ukazuje následující příklad:

```
if (x > 0) {
    if (x > 10) {
        print("x je opravdu velké kladné.")
    } else {
        print("x je malé kladné")
    }
}
```

```
## [1] "x je malé kladné"
```

Podmínky byste však neměli zanořovat příliš, protože by váš kód byl nepřehledný. (Všimněte si také, jak jsou v kódu odsazené řádky. R to nijak nevyžaduje. Je to však velmi užitečná konvence, která výrazně zvyšuje čitelnost kódu.)

Vlastní podmínka je vždy logická hodnota, přesněji logický vektor délky 1, který má hodnotu buď `TRUE` nebo `FALSE`, tj. ne `NA`. Pokud potřebujete složitější podmínku, musíte použít "zkratující" verze logických operátorů, viz oddíl 4.7. Pokud se má nějaký kus kódu provést pouze v případě, že platí `x` a zároveň `y` nebo `z`, zapíšeme to takto:

```
if (x && (y || z)) {
    ...
}
```

Nápovědu k `if` (a dalším řídícím strukturám) získáte jedním z následujících způsobů:

```
?`if`
help("if")
```

7.2 Opakování kódu

V některých situacích potřebujeme, aby se nějaký kus kódu provedl vícekrát. K tomu slouží cykly. Existuje několik typů cyklů, které se liší podle toho, kolikrát se má daný kus kódu zopakovat.

7.2.1 Cyklus se známým počtem opakování

Nejjednodušší je cyklus `for`, který použijeme, když chceme nějaký kus kódu provést x-krát, kde x je libovolné číslo známé před započetím cyklu. Cykly se používají nejčastěji k rekurzivním výpočtům. Řekněme např., že potřebujete nasimulovat data o náhodné procházce. Počáteční hodnota $y_1 = 0$, další hodnoty jsou konstruované rekurzivně jako $y_t = y_{t-1} + \epsilon_t$ pro $t > 1$, kde ϵ_t je náhodná chyba vygenerovaná ze standardizovaného normálního rozdělení. Vektor tisíce pozorování je pak možné nasimulovat např. takto (výsledek ukazuje obrázek 7.1):

```
N <- 1000
y <- numeric(N)
y[1] <- 0  # zbytečné, ale pro přehlednost lepší
for (t in 2:N)  # vlastní cyklus
    y[t] <- y[t - 1] + rnorm(1)  # kód, který se opakuje 999x
plot(y, type = "l")  # vykreslení výsledku
```

Náš kód funguje takto: Nejdříve ze všeho vytvoříme celý vektor `y`. Sice bychom mohli začít vektorem délky 1 a postupně jej prodlužovat, to by však nutilo R neustále doalokovávat další paměť a výrazně by to zpomalilo výsledek. Proto je vždy lepší si dopředu předalokovat celou potřebnou paměť tím, že vytvoříme celé datové struktury ve velikosti, jakou bude mít výsledek výpočtu.

Vlastní cyklus začíná klíčovým slovem `for`. Proměnná `t` je *počítadlo* cyklu a postupně nabývá hodnot z vektoru uvedeného za klíčovým slovem `in`. Protože chceme, aby `t` postupně nabývalo hodnot 2 až 1 000, mohlo by se zdát jednodušší napsat `t in 2:1000`. To by však nebyl dobrý nápad. Nyní sice chceme simulovat právě tisíc pozorování, ale v budoucnu se možná rozhodneme jinak. Bezpečnější tak určitě je napsat `t in 2:N`, protože pak můžeme počet pozorování změnit na jediném řádku, a vše bude korektně fungovat. Cyklus `for` proběhne v našem případě tak, že $N - 1$ krát spustí řádek `y[t] <- y[t - 1] + rnorm(1)`, přičemž počítadlo `t` bude postupně nabývat hodnot 2, 3, 4 atd., až se naplní celý vektor `y`. (Výsledek simulace záleží na generátoru pseudonáhodných čísel, takže bude pokaždé jiný.)

Stejně jako podmínka `if` i cyklus `for` provádí vždy jen jeden výraz. Pokud chceme, aby provedl více výrazů, pak musíme tyto výrazy uzavřít do složených závorek, tj. vytvořit z nich blok, jak ukazuje následující příklad:

Figure 7.1: Výsledek simulace náhodné procházky pomocí cyklu.

```
N <- 1000
y <- numeric(N)
z <- numeric(N)
for (t in 2:1000) {
    y[t] <- y[t - 1] + rnorm(1)
    z[t] <- sum(y[1:t])
}
```

Opět je vhodné kód výrazně odsadit, abychom zlepšili jeho čitelnost.

Někdy potřebujeme iterovat nad nějakým vektorem `y`. Na první pohled by se mohlo zdát, že počítadlo by mělo nabývat hodnot `t in 1:lenght(y)`. To však není dobrý nápad. Někdy se totiž může stát, že vektor `y` bude mít nulovou délku. Pak chceme, aby cyklus vůbec neproběhl. Pokud však počítadlo nastavíme výše uvedeným způsobem, cyklus překvapivě proběhne (a pravděpodobně skončí chybou). Důvod je ten, že dvojtečka má v cyklu svůj normální výraz konstruktoru vektorů. Pokud je `lenght(y)` rovno nule, pak má cyklus iterovat přes prvky vektoru `1:0`, což je vektor o délce 2 a hodnotách `c(1, 0)`. Správně tedy musí být počítadlo nastavené pomocí funkce `seq_along()` jako `t in seq_along(y)`.

Iterovat nad prvky nějakého vektoru `x` můžeme třemi různými způsoby: 1) můžeme iterovat nad indexy prvků, jako jsme to dělali v našem příkladu (použijeme např. `for (k in seq_along(x))`), 2) můžeme iterovat přímo nad hodnotami daného vektoru (použijeme `for (k in x)`) nebo 3) můžeme iterovat nad jmény prvků vektoru (použijeme `for (k in names(x))`). První způsob se používá nejčastěji, ale i další varianty jsou někdy užitečné.

Existují situace, kdy se bez cyklu `for` neobejdeme. V R se však tento cyklus používá mnohem méně než v jiných jazycích. Důvody jsou dva: Zaprvé, R je vektorizovaný jazyk se spoustou vektorizovaných funkcí, takže mnoho operací, které je v jiných jazycích nutné psát pomocí cyklu, v R vyřeší vektorizace. Dokonce i náš první příklad cyklu je vlastně zbytečný a simulaci je možné provést takto:

```
N <- 1000
e <- c(0, rnorm(N - 1))  # simulace náhodné složky
y <- cumsum(e)  # kumulativní součet
```

Za druhé, pro iteraci nad vektory má R jiný velmi silný nástroj, a to funkce typy `map()`, se kterými se seznámíte v kapitole 10.

Dokumentaci k cyklu `for` najdete pomocí `help("for")`.

7.2.2 Cykly s neznámým počtem opakování

V některých situacích nevíme, kolikrát bude potřeba kus kódu opakovat. Pro tyto účely slouží dva standardní typy cyklů: cyklus `while` opakuje kus kódu, dokud je splněná nějaká podmínka; naproti tomu cyklus `repeat` opakuje kus kódu donekonečna s možností cyklus přerušit, pokud je nějaká podmínka splněná. Rozdíl mezi cykly spočívá v tom, kdy se podmínka vyhodnotí: v cyklu `while` se podmínka vyhodnocuje na začátku, takže cyklus nemusí proběhnout ani jednou; naproti tomu v cyklu `repeat` se podmínka vyhodnocuje v principu až na konci, takže cyklus vždy proběhne aspoň jednou. Tyto cykly se při datové analýze nepoužívají příliš často. Zato jsou velmi užitečné v simulacích, optimalizacích a podobných úlohách. Zde se podíváme pouze na jednodušší a častěji používaný cyklus `while`. Více se o obou cyklech můžete dozvědět z dokumentace (`help("while")`).

Použití cyklu `while` si ukážeme na následujícím příkladu: Předpokládejme, že chceme zjistit, kolikrát musíme hodit kostkou, než padne šestka. To můžeme provést např. následujícím kódem:

```
pocet <- 0
kostka <- 0
while (kostka != 6) {
    pocet <- pocet + 1
    kostka <- sample(6, size = 1)
}
print(pocet)
```

```
## [1] 10
```

Skript funguje takto: nejdříve si vytvoříme proměnnou `pocet`, do které budeme shromažďovat uplynulý počet hodů. Dále vytvoříme proměnnou `kostka`, do které uložíme hod kostkou. Vlastní cyklus začíná klíčovým slovem `while`. V závorce za ním je podmínka, tj. výraz, který se musí vyhodnotit na logický vektor délky 1. Pokud je podmínka splněná (logický výraz se vyhodnotí na `TRUE`), vyhodnotí se výraz, který následuje. Protože chceme vyhodnotit dva výrazy, musíme je pomocí složených závorek uzavřít do bloku.

Při prvním průchodu cyklu je `kostka` rovna nule, proto se cyklus provede: počítadlo `pocet` se zvýší o 1 a "hodíme kostkou" (funkce `sample()` v našem případě vygeneruje náhodné celé číslo od 1 do 6). Pokud "padlo" jiné číslo než 6, cyklus proběhne znovu (počítadlo se zvýší o další 1 a znovu se hodí kostkou). To se opakuje, dokud je podmínka splněná (tj. `kostka` je různá od 6). Jakmile se `kostka` rovná šesti, cyklus už neproběhne a R přejde na vypsání hodnoty `pocet`. (Výsledek simulace záleží na generátoru pseudonáhodných čísel, takže bude pokaždé jiný.)

Podívejme se na jiný stylizovaný příklad. Řekněme, že chceme zjistit, pro jaký vstup z určitého intervalu nabývá nějaká funkce určité hodnoty. Aby byla úloha jednoduchá, budeme předpokládat, že funkce je monotónně rostoucí. V takovém případě můžeme použít primitivní algoritmus půlení intervalů. Jako funkci budeme v našem příkladu pro jednoduchost uvažovat přirozený logaritmus a budeme hledat takovou hodnotu x z intervalu $[0, 10]$, pro kterou je $log(x) = 1$:

```
hodnota <- 1
funkce <- log
dint <- 0
hint <- 10
tolerance <- 1e-10
chyba <- Inf
while (abs(chyba) > tolerance) {
    vysledek <- (dint + hint) / 2
```

```
    pokusna_hodnota <- funkce(vysledek)
    if (pokusna_hodnota < hodnota)
        dint <- vysledek
    if (pokusna_hodnota > hodnota)
        hint <- vysledek
    chyba <- hodnota - pokusna_hodnota
}
vysledek
```

```
## [1] 2.718282
```

Nejprve jsme zadali hledanou hodnotu (`hodnota`), použitou funkci (`funkce`), horní a dolní mez prohledávaného intervalu (`dint` a `hint`) a toleranci (`tolerance`), se kterou má algoritmus pracovat. Zadali jsme i počáteční velikost chyby (`chyba`). Vlastní výpočet funguje takto: pokud je absolutní hodnota chyby větší než zadaná tolerance, provedeme úpravu mezí, a to tak, že 1. najdeme hodnotu uprostřed intervalu, 2. vyhodnotíme hodnotu funkce v tomto bodě, 3. pokud je výsledek nižší než požadovaná hodnota, posuneme dolní mez na úroveň středu intervalu; v opačném případě takto posuneme horní mez, 4. spočítáme velikost chyby. Cyklus upravuje meze tak dlouho, dokud není chyba menší než zadaná tolerance. Nakonec vypíšeme hledanou hodnotu, která zůstala v proměnné `vysledek`. Výsledek si můžeme snadno ověřit "zkouškou"; můžeme se také podívat, že chyba je opravdu menší než zadaná tolerance (protože pro přirozený logaritmus máme k dispozici inverzní funkci):

```
log(vysledek)
```

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

```
exp(1) - vysledek
```

```
## [1] -1.064779e-10
```

Algoritmus je velmi rychlý, ale samozřejmě není příliš obecný: funguje jen pro monotonně rostoucí funkce. (R má naštěstí celou řadu funkcí, které dokážou numericky optimalizovat zadanou funkci.)

7.3 Zastavení kódu a varování

Někdy je potřeba výpočet zastavit a vydat chybovou hlášku. Typicky to chcete udělat, když váš kód dostal špatný vstup. Řekněme např., že stahujete data z nějakého serveru a víte, že některá proměnná může mít jen určité úrovně. Převedete ji tedy na faktor a zkontrolujete, že žádná hodnota ve faktoru není `NA`, což by signalizovalo, že server změnil kódování úrovní. Pokud tedy ve vektoru najdete `NA`, chcete kód ukončit chybou, abyste zjistili, že musíte situaci řešit.

K zastavení běhu skriptu slouží funkce `stop()`: zastaví běh skriptu a jako chybovou hlášku vypíše svůj argument. Následující kód zastaví běh skriptu a vypíše chybovou hlášku, pokud `v` obsahuje řetězec:

```
v <- "ahoj"
if (is.character(v))
    stop("v je řetězec!")
```

```
## Error in eval(expr, envir, enclos): v je řetězec!
```

Jednodušší variantou předchozího kódu je použití funkce `stopifnot()`. Ta zastaví kód, pokud se zadaný výraz nevyhodnotí na `TRUE`. V tom případě se vypíše jako chybová hláška, že daný výraz není `TRUE`. Předchozí podmínku pak můžeme zapsat přibližně takto (všimněte si znaku `!`, který neguje zadaný výraz):

```
stopifnot(!is.character(v))
```

```
## Error: !is.character(v) is not TRUE
```

Někdy problém není tak velký, že bychom chtěli běh skriptu zastavit. Chceme však upozornit uživatele (nejčastěji sami sebe), že někde nastal nějaký problém. R umí posílat dva typy signálů: zprávy (messages) a varování (warnings). Zprávy je možné do konzoly vypsat pomocí funkce `message()`, varování pomocí funkce `warning()`. Obě tyto funkce vypíší do konzoly svůj argument:

```
if (!is.list(v))
    warning("Pozor: v není seznam!")
```

```
## Warning: Pozor: v není seznam!
```

Do konzoly je samozřejmě možné vypisovat i pomocí funkcí `print()`, `cat()` apod. Zprávy o běhu kódu však vypisujte raději pomocí `message()` a `warning()`: v RStudiu jsou barevně odlišené a je možné je snadno potlačit pomocí funkcí `suppressMessages()` a `suppressWarnings()`, pokud nejsou žádoucí, což v případě `print()` a spol. nejde.

7.4 Odchycení chyb

Někdy potřebujeme napsat skript, který musí být rezistentní vůči chybám – jeho běh nesmí skončit ani v případě, že chyba nastane. Typickým případem je situace, kdy stahujeme nějaká data ze sítě, např. z API Googlu, ale naše připojení k internetu je poruchové. Pak funkce, která stahuje data skončí chybou. My však nechceme, aby skript zhavaroval. Místo toho chceme chybu odchytit a nějak zpracovat, např. chvíli počkat, a pak se pokusit stáhnout data znovu.

Stejně jako většina pokročilých programovacích jazyků, i R má nástroje pro "zpracování výjimek". V R je zastupují funkce `try()` a `tryCatch()`. Pokud je budete někdy potřebovat použít, podívejte se do jejich dokumentace.

7.5 Aplikace: simulace hry Hadi a žebříky

Existuje hra, kterou malé děti milují a jejich rodiče nenávidí: *Hadi a žebříky* (v originále *Snakes and ladders*). Pravidla hry jsou jednoduchá: každý hráč má právě jednu figurku, kterou postaví na první pole hry. Hráči se pak po řadě střídají v tom, kdo hází kostkou. Hráč, který právě hodil kostkou, posune svoji figurku o tolik polí, kolik mu na kostce padlo. Vítězem je ten hráč, který se jako první dostane na poslední, 100. pole. Hra je ozvláštněna speciálními poli: kdo šlápne na pole paty žebříku, ten se posune o daný počet polí dopředu ("vyjede po žebříku"), kdo však šlápne na hlavu hadovi, vrátí se o daný počet polí zpět ("sjede po hadovi"). Herní plán ukazuje obrázek 7.2

Už asi tušíte, proč děti hru milují: vše je dáno jen náhodou, jejíž zvraty jsou kvůli umístění žebříků a hadů často veliké, takže poslední hráč se může pár tahy vyšvihnout do začátku pelotonu, zatímco dosud první hráč může snadno spadnout na konec. Navíc hra trvá opravdu dlouho. Asi je také jasné, proč rodiče hru naprosto nenávidí: vlastně se nejedná o hru! Vše je dáno náhodou, chybí jakákoli interakce mezi hráči a hra může trvat neskutečně dlouho (nekonečně dlouho, zdá se znuděnému rodiči). Člověče nezlob se je proti této hře posledním výkřikem moderních strategických her.

Naším cílem bude nasimulovat deset tisíc krát průběh hry a zjistit, jaké je rozdělení počtu hodů potřebných k ukončení hry. Pro jednoduchost si problém zjednodušíme tak, že budeme uvažovat jen jednoho hráče. Dále budeme předpokládat, že k vítězství ve hře není potřeba se přesně trefit na 100. pole, ale že stačí se dostat na

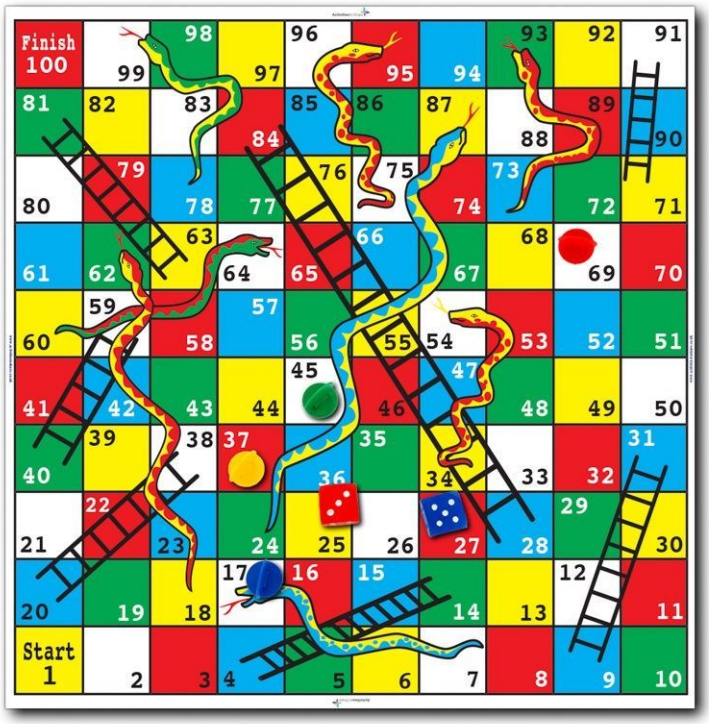


Figure 7.2: Koberec s deskovou hrou Snakes and ladders nabízený serverem *Activities for elderly people with dementia and Alzheimer's*, http://www.activitiestoshare.co.uk/snakes-and-ladders-floor-mat.

ně nebo za ně. Následně použijeme data ze simulace s jedním hráčem k odhadu počtu tahů, které potřebují k ukončení hry dva nebo tři hráči. Pro jednoduchost budeme předpokládat, že hra končí ve chvíli, kdy jeden z hráčů vyhrál, tj. že se nedohrává až do konce. (Nápověda: hráči nijak neinteragují.)

Začneme tím, že si první úkol rozebereme. Budeme potřebovat dvě proměnné: Zaprvé, proměnnou `panacek`, ve které budeme mít uložené pole, na kterém panáček stojí. Na začátku bude mít tato proměnná hodnotu 1. Zadruhé, budeme potřebovat proměnnou `hody`, kam si budeme zaznamenávat, kolik hodů už proběhlo. Počáteční hodnota této proměnné je samozřejmě 0.

Každá jednotlivá hra bude spočívat v opakování tří kroků: 1. Hodíme kostkou a posuneme panáčka o tolik kroků, o kolik bude potřeba. 2. Až se panáček posune, zkontrolujeme, na jakém poli figurka stojí: pokud je to žebřík, posuneme jej nahoru, pokud had, posuneme jej dolů, jinak jej necháme na místě. 3. Zvýšíme hodnotu počítadla hodů o 1. Tyto tři kroky budeme opakovat tak dlouho, dokud panáček nebude stát na poli 100 nebo za ním. Na konci si zapamatujeme, kolik hodů bylo k ukončení hry potřeba. Kód pro jednu hru tak bude vypadat nějak takto:

```
panacek <- 1L
hod <- 0L
while (panacek < 100L) {
    hod <- hod + 1L
    panacek <- panacek + sample(6, size = 1)
    panacek <- pole[panacek]
}
```

Funkce `sample(6, size = 1)` na předposledním řádku cyklu vrátí náhodné číslo z oboru 1, 2, . . . , 6 (tj. simuluje hod kostkou). Otázkou je, jak vyřešíme hady a žebříky. To je možné udělat celou řadů způsobů. Můžeme např. pro každého hada a žebřík napsat podmínku typu

```
if (panacek == 4)
    panacek <- 14
```

která posune panáčka ze 4. na 14. pole (1. žebřík). Já preferuji poněkud globálnější přístup, který ukazuje poslední řádek cyklu: vytvoříme vektor `pole`, který pro každé herní políčko řekne, kam se má panáček posunout. Pokud dané pole neobsahuje ani hada, ani žebřík, bude obsahovat svůj vlastní index. Proměnnou pole vytvoříme např. takto:

```
# inicializace hracího pole
pole <- 1:105
# -- žebříky
pole[4] <- 14L
pole[9] <- 31L
pole[20] <- 38L
pole[28] <- 84L
pole[40] <- 59L
pole[63] <- 81L
pole[71] <- 91L
# -- hadi
pole[17] <- 7L
pole[54] <- 34L
pole[62] <- 18L
pole[64] <- 60L
pole[87] <- 24L
pole[93] <- 73L
pole[95] <- 75L
pole[98] <- 78L
```

Nejdříve inicializujeme `pole` tak, aby platilo `pole[i] = i`, tj. aby hráč, který na toto pole šlápne, na něm i zůstal. Pak přidáme žebříky a hady. Všimněte si, že pole nemá 100 prvků, ale 105. To je proto, že panáček se může díky hodu kostkou dostat za 100. `pole`. Pokud bychom je neinicializovali, skript by skončil chybou. (Náš přístup k poli je poněkud nedbalý – pokud by se hrací pole zvětšilo, rostly by nároky na paměť počítače. Hra *Hadi a žebříky* se však, doufejme, nikdy nezvětší.)

Nyní tedy umíme zahrát jednu hru. Zbývá nám ji ještě deset tisíc krát zopakovat. To provedeme takto:

```
# počet simulací
N <- 1e4

# ... sem se vloží inicializace hracího pole...

# alokace vektoru výsledků
vysledek <- rep(NA, N)

# vlastní simulace
for (k in seq_len(N)) {
    # ... sem se vloží kód pro jednu hru
    vysledek[k] <- hod
}
```

Nyní nám stačí celý kód spustit a vypsat popisné statistiky pomocí funkce `summary()` a případně vykreslit histogram rozdělení počtu hodů pomocí funkce `hist()` (v kapitole 17 se naučíte kreslit hezčí grafy). Celý kód tedy vypadá takto:

Table 7.1: Souhrnné statistiky počtu hodů potřebných k ukončení hry *Hadi a žebříky*, pokud hraje jeden hráč.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
7	25	39	48.19	61	303

```
# počet simulací
N <- 1e4

# inicializace hracího pole
pole <- 1:105
# -- žebříky
pole[4] <- 14L
pole[9] <- 31L
pole[20] <- 38L
pole[28] <- 84L
pole[40] <- 59L
pole[63] <- 81L
pole[71] <- 91L
# -- hadi
pole[17] <- 7L
pole[54] <- 34L
pole[62] <- 18L
pole[64] <- 60L
pole[87] <- 24L
pole[93] <- 73L
pole[95] <- 75L
pole[98] <- 78L

# alokace vektoru výsledků
vysledek <- rep(NA, N)

# vlastní simulace
for (k in seq_len(N)) {
    panacek <- 1L
    hod <- 0L
    while (panacek < 100L) {
        hod <- hod + 1L
        panacek <- panacek + sample(6, size = 1)
        panacek <- pole[panacek]
    }
    vysledek[k] <- hod
}
```

```
# shrnutí výsledků
summary(vysledek)
hist(vysledek)
```

Výsledky simulace ukazuje tabulka 7.1, rozdělení počtu hodů potřebných k ukončení hry ukazuje obrázek 7.3. Průměrný počet hodů, potřebný k dokončení při v jednom hráči, je 48.19; ve čtvrtině her však nebude stačit ani 61 hodů.

Naším druhým úkolem je zjistit, kolik hodů kostkou by bylo potřeba, kdyby hru hrálo $M > 1$ hráčů. Na první pohled by se mohlo zdát, že potřebujeme celou naši simulaci přepsat tak, aby v rámci každé dílčí hry hrálo M hráčů. To však vůbec není potřeba, a to díky tomu, že hráči ve hře nijak neinteragují. Pokud tedy hrají tři hráči, je to stejné, jako by nezávisle na sobě hráli tři hráči. Hra skončí, když kterýkoli z nich dojde

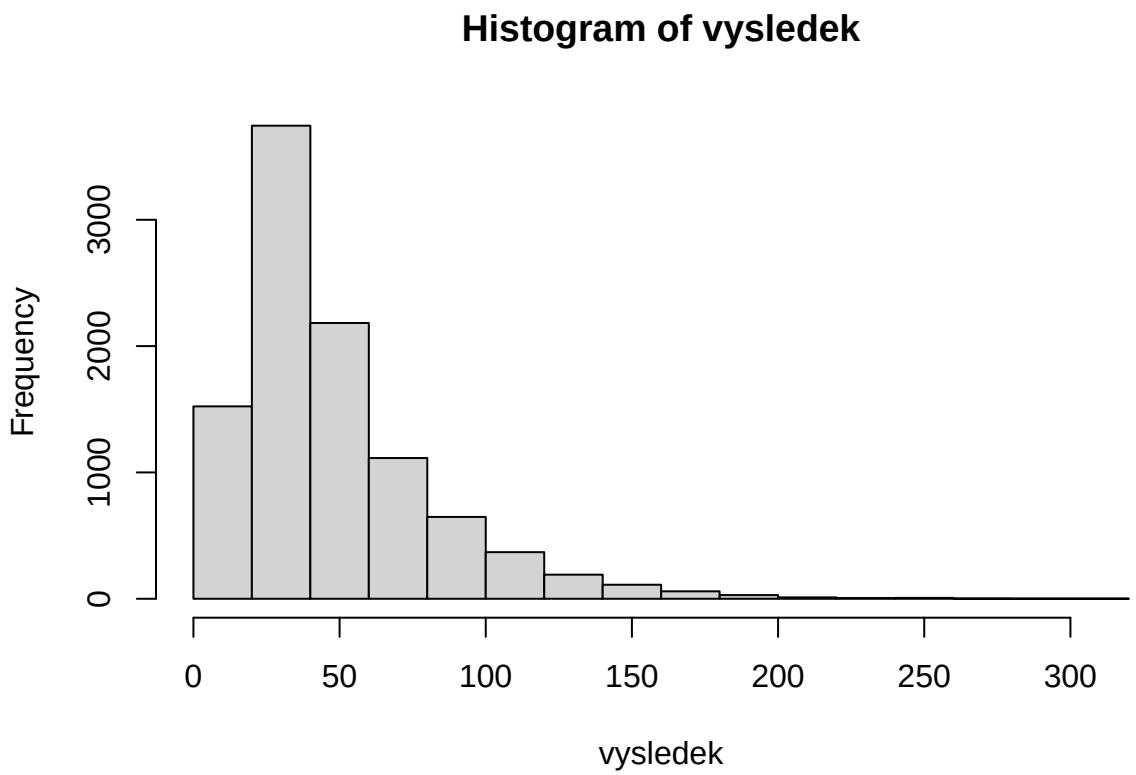


Figure 7.3: Rozdělení počtu hodů potřebných k ukončení hry *Hadi a žebříky*, pokud hraje jeden hráč.

na 100. políčko. Kolik hodů k tomu potřebuje, to máme uložené v proměnné `vysledek`. Přibližně správný odhad tedy můžeme získat tak, že z vektoru výsledek náhodně vybereme tři hodnoty (s opakováním) a z nich vezmeme nejmenší číslo (stanovili jsme si, že hra končí, když vyhrál první hráč). Tak zjistíme, kolikrát by musel hodit vítězný hráč. Jeho počet hodů musíme samozřejmě vynásobit počtem hráčů, protože ve skutečnosti každý hráč musel hodit tolikrát.

Pokud jsme estéti, můžeme ještě provést jistou korekci pro posledního hráče. Řekněme, že vítězný hráč hodil právě L-krát. Pokud hrají tři hráči, pak máme tři možnosti: 1) Vítězný hráč začínal hru; pak je celkový počet hodů $(L-1)\times 3+1$. 2) Vítězný hráč házel jako druhý; pak je celkový počet hodů $(L-1)\times 3+2$. A konečně 3) vítězný hráč házel jako poslední; pak je celkový počet hodů $3L$. Každá z těchto možností se stala právě s pravděpodobností $1/3$. Pokud tedy hrají tři hráči, musíme od jednoduchého výsledku získaného jako trojnásobek počtu hodů vítězného hráče odečíst s pravděpodobností $1/3$ dvojku, s pravděpodobností $1/3$ jedničku a s pravděpodobností $1/3$ nulu. Obecně musíme odečíst $(M-1)/2$.

Tímto postupem zjistíme, jak dlouho by hrálo M hráčů v jedné konkrétní hře. Výsledkem je tedy opět náhodné číslo. Simulaci opět potřebujeme zopakovat (řekněme 10 000 krát), abychom dostali rozdělení počtu hodů.

Celou simulaci provedeme snadno takto:

```
# počet hráčů
hracu <- 3

# alokace vektorů výsledků
vysledek2 <- rep(NA, N)

# korekce počtu tahů (hráč, který hru ukončil nemusel hrát jako poslední)
korekce <- (hracu - 1) / 2

# vlastní simulace (bootstrap)
for (k in seq_len(N)) {
```

Table 7.2: Souhrnné statistiky počtu hodů potřebných k ukončení hry *Hadi a žebříky*, pokud hrají tři hráči.

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
20	50	68	75.5786	92	374

```
    vyber <- sample(N, size = hracu, replace = TRUE)
    vysledek2[k] <- hracu * min(vysledek[vyber]) - korekce
}
```

Nejdříve do proměnné `hracu` uložíme počet hráčů, v našem případě 3. Následně si předalokujeme vektor pro uložení výsledků simulace a spočítáme správnou korekci. Vlastní simulaci zopakujeme N-krát (10 000 krát). Při každé jednotlivé simulaci vybereme pomocí funkce `sample()` tři náhodná čísla z rozsahu $1, 2, \ldots, N$ s opakováním. Tato čísla použijeme jako indexy k výběru tří náhodných délek hry z vektoru `vysledek` a s jejich pomocí spočítáme střední dobu délky hry tří hráčů. Nakonec se podíváme na souhrnné statistiky pomocí funkce `summary()` a na histogram pomocí funkce `hist()`.

Tabulka 7.2 ukazuje základní statistiku pro tři hráče. Průměrný počet nutných hodů při třech hráčích vzrostl z 48.19 na 75.5786; ve čtvrtině her však nebude stačit ani 92 hodů. Obrázek 7.4 ukazuje srovnání distribuce počtu hodů při jednom a při třech hráčích. Takové obrázky se naučíte kreslit v kapitole 17.

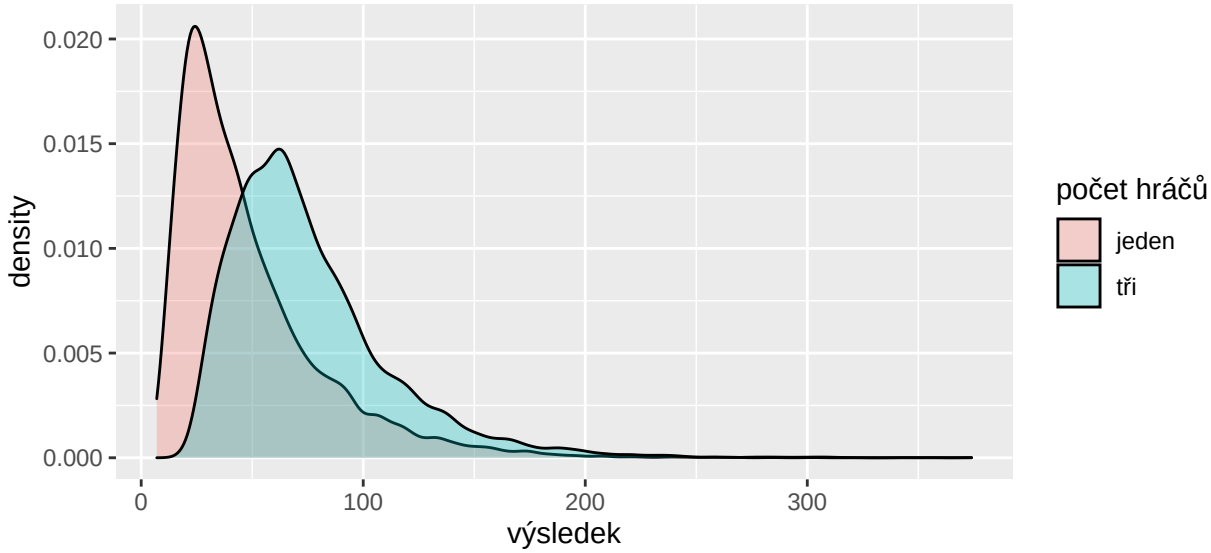


Figure 7.4: Rozdělení počtu hodů potřebných k ukončení hry *Hadi a žebříky*, pokud hrají tři hráči.

Náš odhad distribuce počtu hodů nebude při opakování 10 000 krát úplně přesný a při každém spuštění vrátí poněkud odlišné výsledky. Zvýšení přesnosti můžete dosáhnout zvýšením počtu simulací, např. na milion. Náš kód je možné ještě zobecnit a doladit tím, že je přepíšeme pomocí funkcí (viz kapitola 8) a funkce `map_dbl()` (viz kapitola 10).

8 Funkce

Funkce jsou základní stavební kámen v R: všechno, co se v R děje, je volání funkce. Proto se potřebujete naučit, jak přesně funkce používat. Je také velmi užitečné se naučit vytvářet vlastní funkce, protože vám to umožní výrazně zjednodušit a zautomatizovat mnoho výpočtů.

V této kapitole se naučíte

- k čemu funkce slouží,
- jak je vytvářet,
- jak je používat a
- jak se hledají proměnné ve funkcích použité

8.1 Funkce a jejich užití

Pojem funkce znáte z matematiky. Volně řečeno, funkce je nějaký předpis, který bere vstupy, nějak je využije a vrátí nějaký výstup. Funkce může např. vzít vektor pozorování, spočítat z nich průměr a ten vrátit jako svůj výstup. Tento význam mají funkce i v R. Technicky přesněji je funkce zapouzdřený kus kódu s jasným rozhraním (interface). Uživatel nemusí vědět nic o tom, co se děje uvnitř funkce – stačí mu vědět, jak funkci zavolat (jaké má jméno a jaké bere vstupní parametry) a jaké vrací hodnoty. Jako příklad si představte funkci `mean()`, kterou už znáte. Abyste ji mohli použít, nepotřebujete vědět nic o tom, jak je funkce naprogramovaná. Stačí, když znáte její jméno, seznam parametrů, které bere, a rozumíte tomu, jakou hodnotu vrací.

Existuje několik důvodů, proč vytváříme a používáme funkce:

- Některé řádky kódu spouštíme znovu a znovu. Pokaždé dané řádky kopírovat je hloupé: váš výsledný kód by byl zbytečně dlouhý a nepřehledný. Navíc při kopírování vznikne snadno chyba, a to zejména v situaci, kdy v kódu potřebujete změnit jména proměnných. Rozkopírovaný kód se také špatně udržuje: pokud chcete něco změnit, musíte to změnit na mnoha místech a doufat, že jste na žádné nezapomněli. Při změnách také snadno něco pokazíte. Pokud jste naproti tomu svůj kód uložili do funkce, stačí, když jej změníte na jednom jediném místě.
- Je dobré oddělit kód a jeho interface. Pak můžeme vlastní kód snadno změnit. Pokud se nezměnil interface, je volání kódu stále stejné, takže nemusíme měnit nic jiného.
- Je snazší předat uživateli funkci než kód. Uživatel (včetně vašeho budoucího já) nemusí tušit, co se děje uvnitř; stačí mu, když ví, jak funkci spustit.
- Kód je modulárnější, úspornější a lépe se čte a udržuje. Pamatujte, že váš skript musí být schopen přečíst a porozumět mu nejen počítač, ale také lidé (minimálně vy sami v budoucnosti).

Ukažme si to na příkladu. Řekněme, že chceme normovat několik vektorů tak, že každé hodnotě ve vektoru přiřadíme odpovídající hodnotu v kumulativní distribuční funkci. Srovnejte následují dvě možnosti: kód, který vznikl rozkopírováním:

```
e1 <- ecdf(x1)
x1n <- e1(x1)
e2 <- ecdf(x2)
x2n <- e2(x2)
```

```
e3 <- ecdf(x3)
x3n <- e3(x3)
e4 <- ecdf(x4)
x4n <- e4(x4)
```

s kódem pomocí funkce:

```
normuj <- function(x) {
    e <- ecdf(x)
    e(x)
}
x1n <- normuj(x1)
x2n <- normuj(x2)
x3n <- normuj(x3)
x4n <- normuj(x4)
```

Kód napsaný s pomocí nové funkce je přehlednější a stručnější: čím více vektorů chcete normovat, tím více řádků kódu ušetříte. Je také menší šance, že při kopírování něco pokazíte (např. zapomenete na jeden řádek, kde se volá funkce `ecdf()` nebo v posledním řádku zapomenete změnit `x1` na `x4`). Kód napsaný s pomocí funkce se také lépe udržuje. Představte si, že se rozhodnete svá data normovat jinak, např. tak, že spočítáte jejich z-skóre. V prvním případě budete muset přepsat celý kód. Ve druhém stačí změnit jen tělo funkce `normuj()`. Kód napsaný pomocí funkce navíc šetří operační paměť počítače, protože nezaplevelil pracovní prostředí R čtyřmi mezivýpočty (funkcemi `e1()` až `e4()`).

Pokud to zobecníme, svůj kód byste měli přepsat jako funkci v jednom z několika následujících případů:

1. Pokud nějaký kus kódu rozkopírováváte, měli byste z něj raději vytvořit funkci. (V programování obecně platí zásada, že byste se neměli nikdy opakovat, takže každá konkrétní hodnota nebo operace by měla být v kódu definovaná jen na jednom jediném místě, kde je snadné ji konzistentně změnit.)
2. Pokud potřebujete, abyste mohli nějaké funkci předat svůj kód jako parametr, pak jej také potřebujete zabalit do funkce (touto situací se budeme zabývat v kapitole 10).
3. Pokud je váš kód velmi dlouhý (nevejde se na obrazovku), pak byste měli zvážit jeho rozdělení na kusy. Zde se nabízí dvě možnosti: (a) rozdělit kód na kusy vizuálně pomocí komentářů a hlaviček (RStudio umožňuje vložit do kódu komentářové hlavičky klávesovou zkratkou `Ctrl-Shift-R`; zobrazuje je pak vlevo dole v okně editoru a případně v osnově v pravém pruhu vedle editoru, což zapnete ikonkou osnovy); nebo (b) rozdělit svůj kód na funkce. (Stejně tak, pokud vytvoříte funkci, jejíž kód je delší než jedna obrazovka, měli byste zvážit ji rozdělit do více kratších funkcí, protože tím výrazně zvýšíte čitelnost svého kódu.)

Pokud jste na pochybách, zda vytvořit funkci, raději ji vytvořte. Většina začátečníků chybuje spíše tím, že vytváří složitý dlouhý monolitický kód, než že by jej příliš členila do jednotlivých funkcí.

8.2 Tvorba funkce

V R je funkce objekt jako jakýkoli jiný. To znamená, že vytvořenou funkci jde přiřadit do proměnné, funkci jde předat jako parametr jiné funkci (viz např. kapitola 10), funkci můžeme vytvořit i uvnitř jiné funkce (vznikne tzv. *nested function*) a jedna funkce může vracet jinou funkci jako svou hodnotu (takto vráceným funkcím se říká "uzávěry", *closures*; příkladem takové uzávěry jsou funkce `e()`, které vrací funkce `ecdf()` použitá výše). Funkce v R nemusejí být *čisté funkce*, ale mohou mít i vedlejší účinky (*side effects*). Příkladem takové funkce je např. funkce `print()` – místo, aby vracela nějakou hodnotu, vypíše svůj parametr nějakým způsobem do konzoly. Je jednodušší a bezpečnější psát čisté funkce bez vedlejších účinků. V této kapitole se podíváme pouze na základy psaní funkcí v R. Mnohem podrobnější informace najdete ve Wickham (2014), kap. 6, která je dostupná i na http://adv-r.had.co.nz/Functions.html.

Funkci tvoří v R tři části:

- interface funkce, tj. parametry, které funkce bere;
- tělo funkce, tj. kód funkce a
- prostředí (*environment*) funkce.

Všimněte si, že jméno funkce není její součástí. Funkce je v R objekt, který může být k nějakému jménu přiřazen pomocí klasického přiřazovacího operátoru šipka (`<-`), ale také nemusí. Pokud funkci takto přiřadíme do proměnné, vznikne normální pojmenovaná funkce, kterou můžeme volat jménem této proměnné. Pokud funkci do proměnné neuložíme, vznikne *anonymní funkce*.[1] Použití anonymních funkcí uvidíte v kapitole 10.

Pojmenovanou funkci tedy vytvoříme tak, že výsledek vrácený funkcí `function()` uložíme do proměnné. Touto proměnnou bude jméno nové funkce. Toto jméno musí samozřejmě splňovat všechny nároky na platné názvy proměnných v R, viz oddíl 3.2. Navíc byste je měli zvolit tak, aby bylo stručné, jednoznačné a jasně vystihlo, co funkce dělá. Zároveň by nemělo překrýt jméno jiné funkce nebo proměnné. Hadley Wickham doporučuje, že by jméno funkce mělo být sloveso nebo případně podstatné jméno, pokud jasně popisuje dobře definovaný výsledek (jako např. jméno funkce `mean()`). V kulatých závorkách za `function` uveďte čárkami oddělený seznam parametrů funkce, tj. vstupních nebo také nezávislých proměnných. Také jména parametrů funkce byste měli zvolit tak, aby se dobře pamatovala a aby jasně vystihla, jakou hodnotu mají obsahovat. Také byste se měli snažit dodržet obvyklé konvence v R, např. pojmenovat parametr pro vyřazení hodnot `NA` z výpočtu jménem `na.rm` apod. Vstupní parametry mohou v principu obsahovat data nebo různé volby (např. jak zacházet s hodnotami `NA`). V R je zvykem dát datové proměnné na první místo a volby uvést až za ně. Tělo funkce je výraz uvedený za koncovou kulatou závorkou za `function`.

Novou funkci tedy vytvoříme takto:

```
jmeno_funkce <- function(parametry_funkce_oddělené_čárkami)
    výraz_který_funkce_vyhodnocuje
```

Pokud tělo funkce obsahuje víc než jeden výraz, je třeba tyto výrazy sbalit do bloku tím, že je uzavřeme do složených závorek:

```
jmeno_funkce <- function(parametry_funkce_oddělené_čárkami) {
    výraz 1
    výraz 2
    ...
    výraz n
}
```

Funkce nemusí mít žádné parametry ani žádný kód. Pokud má funkce nějaký kód, pak vrací poslední vyhodnocený výraz jako svoji hodnotu. Pokud je třeba vrátit hodnotu někde jinde než jako poslední výraz v těle funkce, použijeme funkci `return()` – ta vrátí hodnotu funkce a zároveň ukončí její běh. To se používá typicky v situaci, kdy pro některé hodnoty vstupů má funkce hned na začátku vrátit nějaký výsledek (např. `NA`), místo toho, aby něco složitě počítala. (Pokud funkce vrací výsledek zabalený do funkce `invisible()`, funkce sice vrací hodnotu, ale při použití v konzole výsledek nevypíše.)

Řekněme, že chceme vytvořit funkci, která vezme dvě čísla a vrátí jejich násobek. Tuto funkci vytvoříme takto:

```
vynasob <- function(x, y)
    x * y
```

Snadno si ověříme, že výsledkem je opravdu funkci a že funguje tak, jak má:

```
class(vynasob)
```

[1] R od verze 4.1 nabízí i zkrácenou syntaxi pro tvorbu anonymních funkcí: `\(x) {...}`, kde výraz ve složených závorkách `{...}` nahrazuje tělo funkce. I takto vytvořenou funkce je samozřejmě možné uložit do proměnné, a tak ji pojmenovat: `f <- \(x) sin(x)`.

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

```
vynasob(3, 4)
```

```
## [1] 12
```

```
vynasob(7, 8)
```

```
## [1] 56
```

Pokud funkci plánujete používat hodně a dlouho, měli byste ji zabezpečit proti špatným vstupům. To můžeme udělat dvěma způsoby: buď chceme, aby funkce zhavarovala, nebo aby vypsala varování a vrátila `NA`. Druhý případ je jednou ze situací, kdy použijeme funkci `return()`. Řekněme, že chceme, aby naše funkce `vynasob()` vypsala varování a vrátila `NA` v případě, že vstupní proměnné `x` a `y` nebudou číselné vektory o délce jedna. To můžeme udělat např. takto:

```
vynasob <- function(x, y) {
    if (!is.vector(x) || !is.vector(y) || !is.numeric(x) || !is.numeric(y) ||
        length(x) != 1 || length(y) != 1) {
        warning("Obě vstupní proměnné musejí být číselný skalár.\n")
        return(NA)
    }
    x * y
}
vynasob(2, 3)
```

```
## [1] 6
```

```
vynasob("a", 3)
```

```
## Warning in vynasob("a", 3): Obě vstupní proměnné musejí být číselný skalár.
```

```
## [1] NA
```

Všimněte si, že funkce `return()` vrátí hodnotu `NA` a ukončí běh funkce, takže pokud je podmínka splněná, na vyhodnocení výrazu `x * y` vůbec nedojde.

Pokud chcete funkci ukončit chybovým hlášením, když jsou vstupy špatné, můžete použít funkce `stop()` a `stopifnot()`, viz oddíl @ref(). Naši funkci bychom v tom případě mohli přepsat např. do tvaru:

```
vynasob <- function(x, y) {
    stopifnot(is.vector(x) && is.vector(y) && is.numeric(x) && is.numeric(y) &&
        length(x) == 1 && length(y) == 1)
    x * y
}
vynasob(2, 3)
```

```
## [1] 6
```

```
vynasob("a", 3)
```

```
## Error in vynasob("a", 3): is.vector(x) && is.vector(y) && is.numeric(x) && is.numeric(y) &&  ....
```

Parametry funkce mohou mít implicitní hodnoty – pokud není hodnota parametru zadána, vezme se jeho implicitní hodnota. Jako implicitní hodnota se volí nejčastější hodnota nebo nejbezpečnější hodnota. Většina funkcí v R, které mají parametr `na.rm`, mají tento parametr např. implicitně nastavené na `na.rm = FALSE`, protože je bezpečnější implicitně žádné hodnoty nevyřazovat. Implicitní hodnota se nastavuje přímo v seznamu parametrů v kulatých závorkách za voláním `function()`. V následující funkci má proměnná `y` přiřazenou implicitní hodnotu:

```
vynasob2 <- function(x, y = 2)
    x * y
vynasob2(3, 4)
```

```
## [1] 12
```

```
vynasob2(3)  # y = 2, implicitni hodnota
```

```
## [1] 6
```

Parametry funkcí jsou vyhodnocovány líně (*lazy evaluation*). To znamená, že se jejich hodnota vyhodnotí až ve chvíli, kdy jsou opravdu použité. Pokud tedy není parametr ve funkci vůbec použit, R nevyhlásí chybu, když hodnotu parametru nezadáte.

```
f <- function(x, y)
    3 * x
f(2, 4)
```

```
## [1] 6
```

```
f(2)  # R nevyhlásí chybu, protože y není reálně použito
```

```
## [1] 6
```

Líné vyhodnocování parametrů umožňuje mimo jiné i zadat implicitní hodnoty parametrů, které jsou spočítané z jiných parametrů. Můžeme např. vytvořit funkci `f()`, kde implicitní hodnota parametru `y` bude záviset na zadané hodnotě parametru `x`:

```
f <- function(x, y = 3 * x + 1)
    c(x, y)
f(2)  # y je implicitně 3 * 2 + 1
```

```
## [1] 2 7
```

```
f(2, 3)  # y je explicitně zadáno jako 3
```

```
## [1] 2 3
```

Veškeré proměnné definované uvnitř funkce jsou lokální, tj. platí pouze uvnitř funkce a neovlivní nic mimo funkci. Při ukončení běhu funkce zaniknou (leda byste vytvořili uzávěru, viz dále). To platí i pro parametry funkce – pokud se do nich uvnitř funkce pokusíte uložit nějakou hodnotu, R tiše vytvoří lokální proměnnou se stejným jménem, které zastíní vlastní parametr – jeho hodnota nebude ovlivněna. Ukažme si to na příkladu:

```
a <- 3
b <- 7
f <- function(x, y) {
    a <- 5
    x <- 2 * x
    a + x + y
}
f(b, 3)  # vrací 5 + 2 * 7 + 3 = 22
```

```
## [1] 22
```

```
a  # hodnota a se mimo funkci nezměnila
```

```
## [1] 3
```

```
b  # ani hodnota b se mimo funkci nezměnila
```

```
## [1] 7
```

Vždy byste se měli snažit o to, aby každá vaše funkce byla čitelná a srozumitelná nejen stroji, ale i člověku. (Možná ji v budoucnu budete muset mírně změnit. Pokud jí nebudete rozumět, budete ji muset napsat zbrusu znova.) K tomu vám výrazně pomůže odsazení kódu (doporučuji používat odsazení o 4 mezery, ne v R obvyklé 2 mezery) a obecně i dodržování jednotného stylu kódování, viz oddíl 2.7.

Pokud hodláte svoje funkce používat dlouhodobě, měli byste je opatřit komentářem dvou typů. První typ komentářů kopíruje nápovědu k funkcím definovaných v balících. Před definici funkce doporučuji si vždy poznačit co funkce dělá, jaké hodnoty mají mít její parametry (co tyto hodnoty znamenají a v jaké datové struktuře a typu mají být uložené) a jakou hodnotu funkce vrací (co výsledek znamená a v jaké datové struktuře a typu je uložen). Neuškodí také, když si napíšete příklad použití funkce. Až se k ní později vrátíte, nebudete muset složitě přemýšlet a zkoušet, jak funkci použít. Komentář pro funkci `vynásob()` by mohl vypadat např. takto:

```
# funkce vynasob() vynásobí dvě čísla
# vstupy:
#   - x ... (numeric) první číslo, které má být vynásobeno
#   - y ... (numeric) druhé číslo, které má být vynásobeno
# výstup:
#   - (numeric) součin čísel x a y
# příklad použití:
#   vynasob(2, 3)
vynasob <- function(x, y)
    x * y
```

Druhý typ komentářů patří do vlastního těla funkce. Zde je dobré si poznamenat, jak funkce funguje. Není až tak důležité vysvětlovat co který kus kód dělá a jak to dělá, protože to by mělo být zřejmé z vlastního kódu (pokud to jasné není, měli byste kód přepsat tak, aby to jasné bylo). Spíše je potřeba si poznamenat, proč to dělá. Pokud váš kód také nepracuje obecně s každým možným typem vstupů, je dobré si poznamenat předpoklady, za kterých bude funkce pracovat správně.

Po napsání byste měli každou funkci vyzkoušet s různými vstupy. U platných vstupů byste měli hlídat, že funkce vrací hodnoty, jaké vracet má. U neplatných vstupů by funkce měla co nejdříve "zemřít" a vrátit přitom nějakou smysluplnou chybovou hlášku. (Je velmi nepříjemné, pokud se chybná hodnota dlouho táhne výpočtem, protože pak může být těžké zjistit, kde se co pokazilo. Proto se snažte dodržovat princip "die early".) Pokud víte něco o testování chyb v softwaru obecně, můžete se podívat na balík **testthat**, který umožňuje efektivně psát jednotkové testy (*unit tests*). Dokumentaci balíku najdete na viz http://r-pkgs.had.co.nz/tests.html.

8.3 Volání funkce

Funkce se vždy volá se závorkami, i kdyby neměla žádné parametry. Pokud zavoláme funkci bez závorek, vypíše se kód funkce (její tělo):

```
hello_world <- function()
    print("Ahoj, světe!")
hello_world()
```

```
## [1] "Ahoj, světe!"
```

```
hello_world
```

```
## function()
##     print("Ahoj, světe!")
```

Parametry mohou být funkci předány třemi způsoby:

- jménem, např. `f(a = 1, b = 2)` – v tomto případě nezáleží na pořadí parametrů,
- pozicí, např. ve funkci `f(a, b)` znamená volání `f(1, 2)`, že $a = 1$ a $b = 2$, nebo
- pokud má parametr implicitní hodnotu, je možné jej vynechat – R vezme místo parametru implicitní hodnotu.

Při zadání parametru jménem R umožňuje jméno parametru zkrátit, pokud je zkratka jednoznačná (tomu se říká *partial matching*). Ve funkci `f(number, notalk)` je např. možné první parametr zkrátit na `num` i `nu`, ale ne na `n`, protože `n` není jednoznačné a R by nevědělo, zda `n` znamená `number`, nebo `notalk`. Zkracování parametrů zjednodušuje interaktivní práci; při psaní skriptů se však výrazně nedoporučuje, protože autor funkce by později mohl přidat další parametr a zkrácené jméno už by nemuselo být jednoznačné.

Předávání parametrů těmito třemi způsoby je možné libovolně míchat, tj. volat některé parametry pozicí, jiné jménem a další (s implicitní hodnotou) vynechat. V takovém případě postupuje R takto:

1. vezme parametry volané plným jménem (*exact matching*) a přiřadí jim hodnoty,
2. vezme parametry volané zkráceným jménem (*partial matching*) a přiřadí jim hodnoty a
3. vezme parametry pozičně.

Ukážeme si to na příkladu funkce `mean()`. Tato funkce má podle dokumentace pro číselné vektory použití `mean(x, trim = 0, na.rm = FALSE, ...)`. Následující výrazy jsou tedy ekvivalentní:

```
mean(x, na.rm = TRUE)
mean(x, trim = 0, na.rm = TRUE)
mean(x = x, trim = 0, na.rm = TRUE)
mean(x, 0, TRUE)
```

Výraz `mean(x, TRUE)` už však ekvivalentní není a skončil by chybou.

Pokud nechcete mít v kódu zmatek, doporučuji abyste pozicí volali pouze několik první nejdůležitějších parametrů. Všechny ostatní parametry volejte plným jménem. parametry volané jménem uveďte až za parametry volané pozicí. Ve funkci `mean()` tedy pozičně předejte jen průměrovaný vektor a ostatní parametry volejte jménem, např.:

```
mean(x, na.rm = TRUE)
```

8.4 Speciální parametr ...

Kromě explicitně vyjmenovaných parametrů může funkce v R obsahovat i speciální parametr tři tečky (`...`). Tento parametr absorbuje libovolný počet parametrů na dané pozici; všechny parametry uvedené za ním musejí být tedy volány plným jménem.

Parametr `...` se používá zejména ve dvou situacích: 1) když počet parametrů není dopředu znám a 2) když chceme některé parametry předat z naší funkce další funkci, kterou naše funkce volá. Příkladem funkce, která bere libovolný počet vstupních parametrů, je funkce `paste()`, která umožňuje spojit libovolný počet řetězců: buď spojuje vektory po prvcích, nebo kolabuje vektor do jednoho řetězce. Počet parametrů není znám dopředu, proto funkce paste používá parametr `...`:

```
paste("Ahoj,", "lidi.", "Jak se máte?")
```

```
## [1] "Ahoj, lidi. Jak se máte?"
```

```
paste
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## .Internal(paste(list(...), sep, collapse, recycle0))
## <bytecode: 0x55910063db98>
## <environment: namespace:base>
```

Pokud chcete napsat vlastní funkci s libovolným počtem parametrů, můžete použít následující trik: parametr `...` vložíme do seznamu, s jehož prvky pak už pracujeme obvyklým způsobem. Jako příklad použití `...` vytvoříme funkci, která vezme libovolný počet atomických vektorů a vrátí součet počtu prvků všech těchto vektorů:

```
count_all_members <- function(...) {
    param <- list(...)
    sum(purrr::map_int(param, length))
}
count_all_members(1)
```

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

```
count_all_members(1, 1:10, 1:100, 1:1000)
```

```
## [1] 1111
```

(Co přesně dělá funkce `map_int()` z balíku **purrr*, vysvětlíme v kapitole 10.)

Příkladem funkce, která předává svoje parametry dál, je funkce `print()`, která dokáže vypsat do konzoly obsah mnoha různých objektů – pro každý objekt volá speciální metodu, tj. funkci přizpůsobenou tomuto objektu. Proto je třeba mít možnost funkci `print()` předat libovolné parametry, které funkce `print()` předá dál zvolené metodě:

```
print
```

```
## function (x, ...)
## UseMethod("print")
## <bytecode: 0x55910115f308>
## <environment: namespace:base>
```

(O objektech a metodách se dozvíte více v kapitole 9.)

8.5 Scoping rules a uzávěry

Když R hledá hodnotu nějaké proměnné, prochází série různých prostředí (*environment*). Pokud např. zadáte výpis proměnné `x` v konzoli, začíná R hledat `x` nejdříve v globální prostředí (tj. základním pracovním prostředí R). Když ji tam nenajde, pokračuje do mateřského prostředí (*parental environment*) globálního prostředí, což je typicky poslední načtený balík. Pokud ji nenajde ani tam, pokračuje do dalšího mateřského prostředí atd. Pokud ji nikde nenajde, vypíše R chybu.

Situace ve funkcích je zajímavější. R používá tzv. *lexical scoping*, což znamená, že vyhledávání začíná v prostředí, ve kterém byla funkce *definovaná*. Pokud je funkce zadefinovaná v globálním prostředí, vše funguje stejně jako v předchozím případě: pokud se nějaká proměnná nenajde v prostředí funkce, bude se dál hledat v globálním prostředí:

```
a <- 7
nasob_a <- function(x)
    a * x
nasob_a(5)  # neskončí chybou, ale vrátí 35
```

```
## [1] 35
```

Náš kód neskončí chybou, protože když R nenašlo proměnnou `a` v těle funkce ani mezi jejími parametry, začne ji hledat v globálním prostředí, kde ji (v našem případě) najde. Volání funkce `nasob_a()` by skončilo chybou jen v případě, že by R nenašlo proměnnou `a` ani v globálním prostředí, ani v žádném balíku načteném do paměti. Použití proměnné, která není definovaná ve funkci, ale v globálním prostředí (jedná se o tzv. *globální proměnnou*), umožňuje dělat zajímavé triky; zároveň se však takový kód špatně ladí. Snadno se totiž může stát, že jste proměnnou prostě jen zapomněli definovat – pokud daná proměnná žije v globálním prostředí, dostanete divné chování. Pokud však máte v RStudiu zapnutou diagnostiku, RStudio označí proměnné, které nejsou v daném rozsahu (*scope*) definované žlutou vlnovkou a na začátek řádku přidá výstražný trojúhelníček. (Víc o diagnostice v RStudiu najdete na https://goo.gl/ovWFpE.)

Hodnoty globálních proměnných hledá R dynamicky (*dynamic lookup*). To znamená, že začne proměnnou hledat v místě, kde byla funkce definovaná, ale použije její aktuální hodnotu, jak ukazuje následující případ:

```
a <- 7
nasob_a <- function(x)
    a * x
nasob_a(5)  # a je nyní 7
```

```
## [1] 35
```

```
a <- 5
nasob_a(5)  # a je nyní 5
```

```
## [1] 25
```

Pokud se ve funkci pokusíte něco přiřadit do globální proměnné, R automaticky vytvoří lokální proměnnou stejného jména, která zamaskuje globální proměnnou. Pomocí operátoru šipka (`<-`) tedy není možné přiřazovat hodnoty mimo vlastní tělo funkce:

```
a <- 5
nasob_a <- function(x) {
    a <- 7
    a * x  # hodnota a je ve funkci lokálně 7
}
nasob_a(5)
```

```
## [1] 35
```

```
a  # hodnota a je mimo tělo funkce stále 5
```

```
## [1] 5
```

Pokud je funkce `g()` definovaná uvnitř jiné funkce `f()`, pak vyhledávání začíná v prostředí vnitřní funkce `g()`, a pak v prostředí vnější funkce `f()`. To je možné (mimo jiné) využít k tvorbě tzv. *function factories*. Ukažme si to na velmi konvenčním příkladu:

```
n <- 17
make_power <- function(n) {
    g <- function(x)
        x ^ n
    g
}
square <- make_power(2)
cube <- make_power(3)
square(2)  # ve funkci square() je n lokálně rovno 2
```

```
## [1] 4
```

```
cube(2)  # ve funkci cube() je n lokálně rovno 3
```

```
## [1] 8
```

```
n
```

```
## [1] 17
```

Funkce `make_power()` vrací funkci jedné proměnné. Tyto funkce však obsahují kromě formálního parametru `x` také volný parametr `n`. Když R hledá jeho hodnotu, začne nejdříve v prostředí, ve kterém byly funkce `square()` a `cube()` definovány. Tato prostředí je prostředí funkce `make_power()` ve chvíli, kdy byla tato funkce spuštěna. Funkce `square()` byla definována v prostředí, ve kterém bylo `n = 2`, a toto prostředí si s sebou táhne. Stejně tak funkce `cube()` si s sebou táhne prostředí, ve kterém je `n = 3`. (Normálně volací prostředí funkce ve chvíli ukončení jejího běhu zaniká a každá funkce je vždy spuštěna z čistého stavu.)

Funkce `square()` a `cube()` jsou tzv. "uzávěry" (*closures*). Existují situace, kdy je vytváření uzávěr užitečné. Jindy však uzávěry vzniknou, aniž byste si to výslovně přáli. To pak může být zdrojem překvapivých chyb. Pokud nevíte, co děláte, raději nedefinujte žádné funkce v těle jiných funkcí a nepoužívejte příliš globální proměnné.

Typická situace, kde se něco pokazí kvůli nepochopení toho, jak R hledá proměnné, nastává např. v následující situaci: Vytvoříme funkci `f()`, jejíž hodnota závisí částečně na explicitně zadaném parametru `n` a částečně na implicitním (globálním) parametru `z`. Funkci `f()` pak chceme využít ve funkci `g()`, která nastaví hodnotu `z` podle svých potřeb. Jak ale ukazuje následující příklad, nebude to fungovat, protože `z` použité ve funkci `f()` se nehledá ve funkci `g()`, nýbrž v globálním prostředí, kde byla funkce `f()` definovaná:

```
z <- 3
f <- function(n) n * z
g <- function(a) {
    z <- 7  # neúspěšná snaha nastavit z
    f(a)  # f() hledá z v prostředí, kde byla definovaná, ne kde je spuštěná
}
g(5)  # vrátí 15, ne 35, protože z je 3, ne 7!
```

```
## [1] 15
```

Víc se o funkcích a uzávěrách můžete dočíst v 6. a 10. kapitole Wickham (2014), dostupných na http://adv-r.had.co.nz/Functions.html a http://adv-r.had.co.nz/Functional-programming.html.

8.6 Seznam užitečných funkcí

R má implementováno velký objem velmi užitečných funkcí. Kapitola Vocabulary v knize Wickham (2014) uvádí seznam některých velmi užitečných funkcí. Velmi doporučujeme, abyste si tento slovník prošli a naučili se z něj co nejvíce funkcí. Jejich význam a použití najdete v dokumentaci. Celá kapitola je dostupná na http://adv-r.had.co.nz/Vocabulary.html.

9 Objekty

R je ve své podstatě objektově orientovaný jazyk: vše, co žije v prostředí R, jsou objekty. R je však velmi zvláštní objektově orientovaný jazyk. Na rozdíl od jiných objektových jazyků nejsou objekty (v užším smyslu slova) součástí definice vlastního jazyka, ale jsou vytvořeny v rámci tohoto jazyka. V důsledku toho (a dlouhého vývoje R a jeho předchůdce, jazyka S) dnes v R existuje několik různých systémů objektů, z nichž nejvýznamnější je systém S3, který je nejpoužívanější a zároveň nejjednodušší. Zde se zaměříme právě na tento systém. (Vysvětlení ostatních systémů OOP v R najdete v Wickham (2014) v kapitole 7, která je dostupná na http://adv-r.had.co.nz/OO-essentials.html.)

V této kapitole se naučíte

- základní terminologii,
- vytvářet nové objekty, metody a generické funkce a
- používat existující objekty.

Cílem zde mimořádně není to, abyste vytvářeli vlastní objekty – to není při běžné datové analýze většinou potřeba; cílem je zajistit, že budete mít povšechné povědomí o tom, jak se s objekty v R zachází. K tomu je užitečné vědět, jak jsou objekty a metody zhruba implementované.

9.1 Základní pojmy objektově orientovaného programování (pro laiky)

Základním pojmem OOP je *objekt*. Objekt je datová struktura s jasně definovanými metodami, jak s těmito daty zacházet. Každý objekt má určitou *třídu*. Třída (*class*) popisuje strukturu dat objektu, vztah k ostatním třídám objektů (v rámci dědičnosti) a to, jaké funkce (metody) se volají při práci s objektem. Dědičnost umožňuje vytvářet mezi objekty hierarchii předků a potomků, kde potomci dědí data a metody od svých předků, mohou je však modifikovat a přidávat k nim. Dědičnost vytváří hierarchii: potomci dědí vlastnosti a chování předků. Pokud potomek nemá implementovanou nějakou funkci, pak se použije funkce jeho předka.

Např. objekt třídy *člověk* může obsahovat datové položky jako jméno, příjmení, výšku, váhu apod. Pro třídu *člověk* mohou být definovány nějaké funkce (metody), např. funkce, která spočítá BMI. Objekt třídy *manažer* může být logicky svázán s objektem třídy *člověk* – obsahuje stejná data (může obsahovat i další, např. seznam podřízených). Stejně tak může využívat metody se stejnými jmény, které však mohou být implementovány odlišně. Logickou vazbu tříd *člověk* a *manažer* popisuje dědičnost: třída *manažer* je potomek třídy *člověk*.

9.2 Systém S3

Systém S3 používá poměrně neobvyklý přístup k objektovému programování, tzv. *generic function OOP*. Tento přístup je jednoduchý (až primitivní), ale funguje velmi dobře a používá jej drtivá většina balíků v R. V normálních OOP jazycích patří objektu nebo třídě jak data, tak metody (funkce). V R však objektu patří jen data a jméno třídy; metody patří tzv. generické funkci. Nejjednodušší způsob, jak pochopit strukturu objektu, je nějaký objekt vytvořit.

S3 nemá formální definici tříd. Objekt se vytvoří tak, že se nějaké základní datové struktuře (většinou seznamu) nastaví atribut `class` na hodnotu jména třídy:

```
foo <- list()         # vytvoří prázdný seznam
class(foo) <- "foo"   # přiřadí mu třídu foo
```

Alternativně to jde provést naráz pomocí funkce `structure()`, která nastavuje danému objektu atributy:

```
foo <- structure(list(), class = "foo")
```

Nyní je proměnná `foo` objekt třídy *foo*:

```
class(foo)
```

```
## [1] "foo"
```

R nemá žádný mechanismus, který by kontroloval, zda je datová struktura objektu správná. Již vytvořenému objektu je možné přidat nebo ubrat datové sloty (technicky obvykle prvky seznamu) nebo změnit jeho třídu. Někdy se to hodí; nikdy to však nedělejte, pokud opravdu dobře nevíte, co děláte, jinak vznikne velmi těžko předvídatelné chování a těžko dohledatelné chyby.

Aby se zajistilo, že je struktura objektu správná, je vhodné vytvořit *konstruktor*, tj. funkci, která vytváří objekt. (Funkce jako `numeric()` jsou také konstruktory.) Ukažme si to na příkladu. Budeme chtít mít objekty třídy *human*, které budou obsahovat datové položky `name`, `height` a `weight`. Nejdříve vytvoříme konstruktor, a pak jeden objekt:

```
human <- function(name, height, weight)  # konstruktor
    structure(list(name = name, height = height, weight = weight),
              class = "human")
adam <- human("Adam", 173, 63)  # tvorba objektu pomocí konstruktoru human()
```

Dědičnost se zajistí tak, že atribut `class` je vektorem jmen několik tříd – zleva doprava od potomků k předkům. Budeme např. chtít mít objekty třídy *manager*, které jsou potomky třídy *human*. Opět pro ně vytvoříme konstruktor a jeden objekt:

```
manager <- function(name, height, weight, rank)  # konstruktor
    structure(list(name = name, height = height, weight = weight, rank = rank),
              class = c("manager", "human"))
eva <- manager("Eve", 169, 52, "CEO")  # objekt třídy manager
```

Eva nyní dědí vlastnosti člověka:

```
class(eva)
```

```
## [1] "manager" "human"
```

```
inherits(eva, "manager")
```

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

```
inherits(eva, "human")
```

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

Protože je většina objektů v S3 postavená pomocí seznamů, můžete zjistit strukturu objektu pomocí funkce `str()`. Hodnotu jednotlivých slotů objektů typu S3 můžete získat pomocí operátoru `$`:

```
str(eva)
```

```
## List of 4
##  $ name  : chr "Eve"
##  $ height: num 169
##  $ weight: num 52
##  $ rank  : chr "CEO"
##  - attr(*, "class")= chr [1:2] "manager" "human"
```

```
eva$rank
```

```
## [1] "CEO"
```

Hlavní pointa systému S3 spočívá v tom, že stejně pojmenovaná funkce volá pro každou třídu objektu jinou metodu (jinak implementovanou funkci) přesně uzpůsobenou dané třídě dat. Funkcím, které to dokážou, se říká *generické funkce*. Příkladem generické funkce je funkce `print()`, která tiskne různé informace v závislosti na tom, jaká je třída objektu, kterou chceme vytisknout. Třída *human* zatím nemá definovanou žádnou metodu pro generickou funkci `print()`, proto zavolá metodu pro seznam:

```
print(adam)
```

```
## $name
## [1] "Adam"
##
## $height
## [1] 173
##
## $weight
## [1] 63
##
## attr(,"class")
## [1] "human"
```

Když vytvoříte novou třídu objektu, můžete vytvořit i novou metodu ke generické funkci tak, že vytvoříte funkci, jejíž jméno má tvar `jmeno_genericke_funkce.jmeno_tridy`, tj. jméno generické funkce oddělené tečkou od názvu třídy objektu. Metodu pro tisk objektů třídy *human* vytvoříme např. takto:

```
print.human <- function(x)
    cat("*** Human ***",
        paste("Name:", x$name),
        paste("Height:", x$height),
        paste("Weight:", x$weight),
        sep = "\n")
print(adam)
```

```
## *** Human ***
## Name: Adam
## Height: 173
## Weight: 63
```

Protože třída *manager* je potomkem třídy *human*, dědí její chování. To znamená, že pokud tato třída nemá definovanou svou metodu `print()`, pak zavolá metodu svého předka, třídy *human*:

```
eva  # implicitně se volá funkce print
```

```
## *** Human ***
## Name: Eve
## Height: 169
## Weight: 52
```

Novou generickou funkci vytvoříme tak, že vytvoříme funkci, která bude obsahovat jediný řádek `UseMethod("xxx")`, kde `xxx` je jméno nové generické funkce. Řekněme, že chceme vytvořit novou generickou funkci, která vrátí pro objekty třídy *human* jejich body mass index. Samozřejmě musíme vytvořit i příslušné metody. Řekněme, že budeme chtít mít různou metodu `bmi()` pro objekt třídy *human* a objekt třídy *manager*:

```
bmi <- function(x)          # generická funkce
    UseMethod("bmi")
bmi.human <- function(x)    # metoda pro třídu human
    x$weight / (x$height / 100) ^ 2
bmi.manager <- function(x)  # metoda pro třídu manager
    "classified"
```

Nyní můžeme zjistit, jaký mají Adam a Eva BMI (bohužel je informace o BMI manažerů tajná):

```
bmi(adam)
```

```
## [1] 21.04982
```

```
bmi(eva)
```

```
## [1] "classified"
```

Generická funkce může mít i implicitní metodu, která se použije v případě, že pro daný typ není žádná metoda k dispozici. Tato metoda se jmenuje `default`:

```
bmi.default <- function(x)
    "Unknown class"
bmi(1)
```

```
## [1] "Unknown class"
```

(Bez definování implicitní metody by předchozí řádek skončil chybou.)

9.3 Práce s objekty

Ve většině případů nebudete vytvářet vlastní objekty – stačí, když budete schopní zacházet s objekty, které vrátí funkce, které budete používat.

Nejdříve ze všeho potřebujete poznat, zda je nějaká proměnná objekt v systému S3. Jsou tři možnosti, jak to můžete udělat: 1) můžete se podívat, zda je proměnná objekt pomocí funkce `is.object()`, a že se nejedná o objekt v systému S4, 2) můžete použít funkci `otype()` z balíku **pryr** nebo 3) se samozřejmě můžete podívat do dokumentace:

```
is.object(eva) & !isS4(eva)
```

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

```
pryr::otype(eva)  # funkce z balíku pryr
```

```
## [1] "S3"
```

Často potřebujete zjistit, jak se jmenují jednotlivé položky objektu. Pokud je objekt v systému S3 postaven nad seznamem, což je nejčastější případ, můžete to udělat pomocí funkce `str()`. Když znáte jméno dané položky, můžete získat její obsah pomocí operátoru `$`:

```
str(eva)
```

```
## List of 4
##  $ name  : chr "Eve"
##  $ height: num 169
##  $ weight: num 52
##  $ rank  : chr "CEO"
##  - attr(*, "class")= chr [1:2] "manager" "human"
```

```
eva$height
```

```
## [1] 169
```

Někdy se hodí zjistit, jaké metody má daná třída k dispozici. K tomu slouží funkce `methods()`. Můžete ji použít dvě různými způsoby: 1) ke zjištění metod, které patří dané generické funkci, a 2) ke zjištění metod, které jsou k dispozici pro danou třídu:

```
head(methods("print"))  # metody generické funkce print() -- jen prvních 6 metod
```

```
## [1] "print.acf"     "print.AES"     "print.anova"   "print.aov"
## [5] "print.aovlist" "print.ar"
```

```
methods(class = "human")  # metody dostupné pro třídu human
```

```
## [1] bmi   print
## see '?methods' for accessing help and source code
```

Jména konkrétních metod lze zjistit také pomocí funkce `apropos()`, když se požádá, aby hledala regulární výraz pro cokoli, co končí ".třída_objektu" (s regulárními výrazy se seznámíte v kapitole 13):

```
apropos(".*\\.human")
```

```
## [1] "bmi.human"   "print.human"
```

Důležité je také najít dokumentaci k dané metodě. Z dokumentace ke generické funkci se totiž nemusíte se dozvědět vše, co potřebujete. Pokud chcete nápovědu k tomu, jak se generická funkce chová pro daný objekt, hledejte dokumentaci k jeho metodě (bohužel ne všechny balíky dokumentují všechny metody, které implementují):

```
?print.summary.lm  # dokumentace k metodě pro tisk summary(m), kde m je objekt vráceny z\ lineární r
```

10 Iterace nad prvky atomických vektorů a seznamů

Často je potřeba provést nějakou operaci s každým prvkem atomického vektoru, seznamu apod. Většina programovacích jazyků k tomuto účelu používá cyklus `for` (viz oddíl 7.2.1). R však nabízí dvě lepší řešení. Zaprvé, mnohé funkce jsou v R vektorizované, tj. pracují s vektorem po prvcích a vrátí vektor stejné délky, kde každý prvek je výsledkem aplikace dané funkce na odpovídající prvek původního vektoru. Příkladem takové funkce je např. funkce `log()`. Druhou možností je použití funkce `map()` a jí podobných z balíku **purrr**. Tyto funkce berou jako svůj vstup nejen data, ale také jinou (nevektorizovanou) funkci a aplikují ji na každý prvek dat. Základní R sice poskytuje podobné funkce jako balík **purrr**, my se však zaměříme na funkce z balíku **purrr**, a to z několika důvodů: 1) jejich ovládání je jednodušší, 2) je kompatibilní s dalšími funkcemi ve skupině balíků **tidyverse**, 3) funkce usnadňují řešení problémů a ladění kódu a 4) umožňují snadnou paralelizaci pomocí balíku **furrr**.

V této kapitole se naučíte

- aplikovat nevektorizované funkce na prvky vektorů
- aplikovat funkce na sloupce tabulek
- zjednodušovat výsledků na atomické vektory
- filtrovat prvky vektorů
- redukovat prvky vektorů
- paralelizovat svůj výpočet

Před vlastní prací musíme načíst balík **purrr**:

```
library(purrr)
```

10.1 Základní iterace nad prvky vektorů pomocí `map()`

Nejčastějším případem je, že potřebujeme pomocí nějaké funkce transformovat každý prvek vektoru (atomického nebo seznamu). Základní funkce pro iterace nad vektory je funkce `map(.x, .f, ...)`. Její první parametr je vektor `.x` (atomický nebo seznam) a druhý parametr je funkce `.f`. Funkce `map()` spustí funkci `.f` na každý prvek vektoru `.x` a výsledky poskládá do seznamu stejné délky, jako má vektor `.x`. Funkci `map()` si tedy můžete představit jako výrobní linku, kde nad pásovým dopravníkem pracuje robot. Dopravník nejdříve robotovi doručí první prvek `.x`, tj. `.x[[1]]`. Robot na doručeném prvku provede funkci `.f()`, tj. vyhodnotí `.f(x[[1]])` a výsledek uloží zpátky na dopravníkový pás, tj. jako první prvek nově vytvářeného seznamu. Pak se dopravník posune, robot spustí funkci `.f()` na `.x[[2]]` atd., dokud linka nezpracuje všechny prvky vektoru `.x`. Fungování linky ukazuje obrázek 10.1.

Všimněte si dvou věcí: 1) Funkci `.f()` předáváme funkci `map()` jako proměnnou, tj. bez kulatých závorek. 2) Funkce `.f()` od `map()` vždy dostane jeden prvek vektoru `.x` jako svůj první parametr. Ukážeme si to nejdříve na jednoduchém příkladu. Máme seznam `v`, který obsahuje různě dlouhé atomické vektory. Chceme zjistit délku jednotlivých vektorů v seznamu:

```
v <- list(1, 1:2, 1:3, 1:4, 1:5)  # vytvoří seznam vektorů
map(v, length)  # zjistí délku jednotlivých vektorů v seznamu v
```

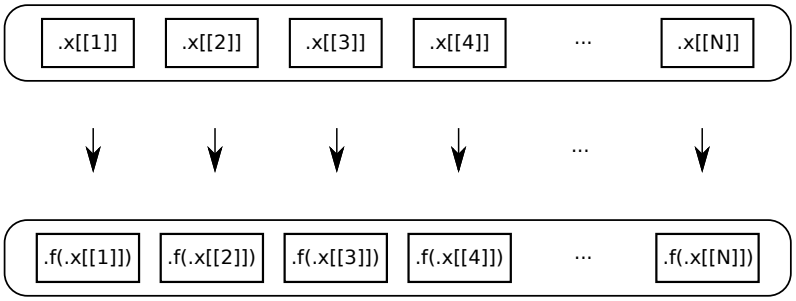


Figure 10.1: Funkce `map(.x, .f)` aplikuje funkci `.f()` na každý prvek vektoru `.x` a vrací seznam stejné délky.

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 4
##
## [[5]]
## [1] 5
```

V dalším příkladu chceme zjistit, jaký datový typ mají jednotlivé sloupce tabulky. Využijeme toho, že tabulky tříd *data.frame* i *tibble* jsou technicky implementované jako seznamy sloupců a že funkce `map()` s nimi takto zachází:

```
df <- tibble::tibble(x = 1:6,  # celá čísla
                     y = c(rnorm(1:5), NA),  # reálná čísla
                     z = c(NA, letters[1:4], NA)  # řetězce
)
df
```

```
## # A tibble: 6 x 3
##       x      y z
##   <int>  <dbl> <chr>
## 1     1 -0.818 <NA>
## 2     2 -0.721 a
## 3     3  0.243 b
## 4     4  0.192 c
## 5     5 -0.293 d
## 6     6 NA     <NA>
```

```
map(df, class)  # funkce class() je použita postupně na všechny sloupce df
```

```
## $x
## [1] "integer"
##
## $y
```

```
## [1] "numeric"
##
## $z
## [1] "character"
```

Výše uvedené příklady ukazují, jak pomocí funkce `map()` aplikovat na jednotlivé prvky vektoru nějakou existující funkci. Často však chceme na prvky vektoru aplikovat nějaký vlastní výpočet. Pokud jej chceme provést jen na jednom vektoru, nestojí za to vytvářet pojmenovanou funkci, která by zaplevelila naše pracovní prostředí. V takovém případě můžeme použít anonymní funkci, jak ukazuje následující příklad. V něm chceme zjistit, kolik každý sloupec tabulky obsahuje hodnot `NA`. Druhým parametrem funkce `map()` je anonymní funkce, tj. funkce, kterou jsme před tím neuložili do žádné proměnné. Naše anonymní funkce musí brát jako svůj první parametr jeden prvek vektoru, nad kterým se iteruje:

```
map(df, function(x) sum(is.na(x)))  # x nabývá vždy hodnotu jednoho sloupce z df
```

```
## $x
## [1] 0
##
## $y
## [1] 1
##
## $z
## [1] 2
```

Od verze 4.1 je samozřejmě možné využít i zkrácené syntaxe pro tvorbu anonymních funkcí – se stejným výsledkem:

```
map(df, \(x) sum(is.na(x)))  # x nabývá vždy hodnotu jednoho sloupce z df
```

Protože tvorba anonymních funkcí může být protivná, nabízí funkce `map()` "zkratku": místo funkce zadat pravostrannou formuli, kterou `map()` automaticky převede na funkci. Syntaxe této formule je jednoduchá. Začíná symbolem `~` (vlnkou) za kterou následuje výraz, který se má vyhodnotit. Hodnotu prvku vektoru `.x`, který funkce právě vyhodnocuje, zadáme jako `.` nebo `.x`. Předchozí výpočet je tedy možné zadat i takto:

```
map(df, ~ sum(is.na(.)))
```

nebo

```
map(df, ~ sum(is.na(.x)))
```

```
## $x
## [1] 0
##
## $y
## [1] 1
##
## $z
## [1] 2
```

Funkce `map()` umožňuje využít i další zajímavou zkratku, která je užitečná v případě, že chceme ze seznamu extrahovat prvky, které se nachází na určité pozici nebo mají určité jméno. V takovém případě zadáme místo funkce `.f` vektor celých čísel, řetězců nebo jejich kombinaci. Pokud zadáme celé číslo, `map()` vrátí z každého

prvku vektoru `.x` jeho prvek na této pozici; pokud zadáme řetězec, pak jeho prvek s tímto jménem. Pokud zadáme víc pozic, `map()` postupuje rekurzivně. Ukažme si to na příkladu. Vytvoříme datovou strukturu, která obsahuje vybrané informace o hráčích nějaké hry. Všimněte si, že vnější seznam obsahuje dílčí seznamy, které mají stejné pojmenované položky:

```
dungeon <- list(
  list(id = 11, name = "Karel", items = list("sword", "key")),
  list(id = 12, name = "Emma", items = list("mirror", "potion", "dagger"))
)
```

Nejdříve chceme získat seznam id jednotlivých hráčů. Protože každý vnitřní seznam má `id` na prvním místě, můžeme jejich seznam získat takto:

```
map(dungeon, 1)
```

```
## [[1]]
## [1] 11
##
## [[2]]
## [1] 12
```

Pokud chceme získat seznam jmen hráčů, můžeme jej získat buď pomocí pozice (jména jsou ve vnitřních seznamech na druhém místě) výrazem `map(dungeon, 2)`. Můžeme je však extrahovat i jménem:

```
map(dungeon, "name")
```

```
## [[1]]
## [1] "Karel"
##
## [[2]]
## [1] "Emma"
```

Podobně můžeme získat i seznam artefaktů, které má každý hráč k dispozici (výsledek je seznamem seznamů):

```
map(dungeon, "items")
```

```
## [[1]]
## [[1]][[1]]
## [1] "sword"
##
## [[1]][[2]]
## [1] "key"
##
##
## [[2]]
## [[2]][[1]]
## [1] "mirror"
##
## [[2]][[2]]
## [1] "potion"
##
## [[2]][[3]]
## [1] "dagger"
```

Jednotlivé artefakty pak můžeme získat zadáním více pozic, které použijí rekurzivně:

```
map(dungeon, c(3, 1))
```

```
## [[1]]
## [1] "sword"
##
## [[2]]
## [1] "mirror"
```

```
map(dungeon, list("items", 1))
```

```
## [[1]]
## [1] "sword"
##
## [[2]]
## [1] "mirror"
```

Oba předchozí výrazy vrátili seznam artefaktů, které mají oba hráči na první pozici mezi svými položkami. Všimněte si, že pokud jsme chtěli adresovat `items` jménem, museli jsme jméno a pozici spojit pomocí seznamu, protože funkce `c()` by převedla obě pozice na řetězce.

Pokud požádáme o prvky, které neexistují, dostaneme výsledek `NULL`:

```
map(dungeon, list("items", 3))
```

```
## [[1]]
## NULL
##
## [[2]]
## [1] "dagger"
```

```
map(dungeon, "aloha")
```

```
## [[1]]
## NULL
##
## [[2]]
## NULL
```

Pokud chceme, aby `map()` v takovém případě vrátil jinou hodnotu, můžeme ji nastavit pomocí parametru `.default`:

```
map(dungeon, list("items", 3), .default = NA_character_)
```

```
## [[1]]
## [1] NA
##
## [[2]]
## [1] "dagger"
```

Funkci, kterou spouštíme nad jednotlivými prvky vektoru, můžeme předat i další parametry. Ty se zadají na pozici `...`, tedy jako třetí a případně další parametry ve funkci `map()`, jak ukazuje následující příklad. V něm chceme vytvořit několik vektorů gaussovských náhodných čísel o různých délkách a průměru 10 a směrodatné odchylce také 10. K tvorbě náhodných vektorů s normálním rozložením použijeme funkci `rnorm()`:

```
map(1:5, rnorm, mean = 10, sd = 10)
```

```
## [[1]]
## [1] 12.21627
##
## [[2]]
## [1]  2.781486 10.890383
##
## [[3]]
## [1]  9.521791 18.697109 25.804600
##
## [[4]]
## [1] 5.763117 6.014887 4.757158 8.880556
##
## [[5]]
## [1] 18.083642 20.962579 17.809818  3.659877 22.737522
```

Funkce `map()` v našem příkladu iteruje nad vektorem `1:5`, jehož jednotlivé prvky se předají funkci `rnorm()` jako její první parametr, tj. jako délka vytvářeného vektoru. Střední hodnotu a směrodatnou odchylku vytvářených náhodných vektorů jsme ve funkci `rnorm()` nastavili pomocí pojmenovaných parametrů `mean` a `sd`. Funkce `map()` tedy postupně vyhodnocovala výrazy `rnorm(1, mean = 10, sd = 10)`, `rnorm(2, mean = 10, sd = 10)` atd.

Jak jsme viděli, funkce `map()` vrací svůj výsledek jako seznam. To je často užitečné, protože seznamy umožňují v jedné datové struktuře skladovat hodnoty různých délek, datových typů a struktur. Někdy však funkce `.f()` vrací na našich datech vždy atomický vektor stejného datového typu a délky 1. V tom případě můžeme chtít výsledek zjednodušit na atomický vektor. K tomu slouží funkce `map_lgl()`, `map_int()`, `map_dbl()` a `map_chr()`, které mají stejné vstupní parametry jako funkce `map()`, ale svůj výsledek zjednoduší na logický, celočíselný nebo reálný vektor nebo vektor řetězců.

Ukážeme si to nejdříve na příkladu, který jsme viděli výše: Chceme zjistit počet chybějících hodnot v jednotlivých sloupcích tabulky, výsledek však chceme mít uložený v atomickém vektoru (názvy hodnot odpovídají jménům sloupců tabulky):

```
map_int(df, ~ sum(is.na(.)))
```

```
## x y z
## 0 1 2
```

Pokud by výsledek nebylo možné zjednodušit na daný datový typ, volání funkce by skončilo chybou, jak ukazuje následující příklad:

```
map_int(1:5, log)
```

```
## Error: Can't coerce element 1 from a double to a integer
```

Podívejme se nyní na komplexnější ukázku použití funkce `map()` převzatou z její dokumentace. Řekněme, že chceme zjistit, jak silně závisí spotřeba aut na jejich váze – ve smyslu, kolik procent rozptylu spotřeby vysvětlí váha aut, a to zvlášť pro každý počet válců. K tomu použijeme standardní dataset `mtcars` (třída

data.frame). Rozdělíme jej pomocí funkce `split(x, f, drop = FALSE, ...)`. Funkce `split()` bere na vstupu atomický vektor, seznam nebo tabulku `x` a vektor `f`, který určí rozdělení `x` do skupin. Funkce vrací seznam, jehož jednotlivé prvky obsahují části proměnné `x` – každý prvek obsahuje část `x`, pro kterou má `f` stejnou hodnotu. Následující kód tedy rozdělí tabulku `mtcars` na tři tabulky uložené v jednom pojmenovaném seznamu `cars`. Každá z těchto dílčích tabulek bude obsahovat jen pozorování se stejným počtem válců: první tabulka 4, druhá 6 a třetí 8 válců:

```
cars <- split(mtcars, mtcars$cyl)
```

Od verze 4.1 může být `f` i pravostranná formule, která za vlnkou obsahuje jméno proměnné ze zvoleného datasetu, podle jejíchž hodnot se má dataset rozdělit; o formulích najdete více v oddíle 18.2. Stejnou operaci je pak možné zapsat kompaktněji takto:

```
cars <- split(mtcars, ~cyl)
```

Na každé této tabulce zvlášť odhadneme lineární model, který vysvětluje spotřebu (`mpg`, počet mil, který vůz ujede na galon paliva) pomocí váhy vozidla (`wt`) a úrovňové konstanty. Vlastní odhad provede funkce `lm()`. Její první parametr je formule `mpg ~ wt`, která popisuje odhadovanou rovnici (zde $\text{mpg}_i = b_0 + b_1 \text{wt}_i + \epsilon_i$). Druhý parametr jsou data, na který se má model odhadnout. Pomocí funkce `map()` aplikujeme funkci `lm()` na každý prvek seznamu `cars`, tj. na tři tabulky, které jsme vytvořili v předchozím kroku:

```
estimates <- map(cars, ~ lm(mpg ~ wt, data = .))
```

Výsledkem je seznam, který obsahuje opět tři prvky: objekty, které popisují odhad modelu na třech částech rozdělené tabulky. Nás z nich zajímá jediná hodnota: koeficient determinace R^2, který říká, kolik procent rozptylu vysvětlované veličiny model vysvětlil. Ten získáme jako slot `r.squared` objektu, který vrací funkce `summary()`. Výpočet opět musíme spustit pro všechny prvky seznamu `estimates`:

```
s <- map(estimates, summary)
map_dbl(s, "r.squared")
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

Vidíme, že sama váha vysvětluje velkou část spotřeby. Celý výpočet můžeme zapsat kompaktněji pomocí operátoru `|>` nebo `%>%`, který předá výsledek předchozího výpočtu (svou levou stranu) jako první parametr do funkce v následném výpočtu (na své pravé straně), viz oddíl 5.6:

```
mtcars %>%
  split(~ cyl) %>%
  map(~ lm(mpg ~ wt, data = .x)) %>%
  map(summary) %>%
  map_dbl("r.squared")
```

```
##         4         6         8
## 0.5086326 0.4645102 0.4229655
```

Více se o ekonometrii dozvíte v kapitole 18.

Funkce `map_dfc(x, f)` a `map_dfr(x, f)` fungují podobně jako `map()`, ale výsledek transformují na tabulku po sloupcích a řádcích respektive. Detaily najdete v dokumentaci. Zde si ukážeme jen několik příkladů. Řekněme, že máme pojmenovaný seznam několika atomických vektorů. Chceme je nějak transformovat (např. spočítat jejich druhou mocninu) a výsledky složit vedle sebe jako sloupce tabulky:

Podobně můžeme rozdělit tabulku pomocí funkce `split()` na části, které uložíme do seznamu. Potom, co na každé části tabulky provedeme nějaké operace, můžeme výsledek opět spojit pomocí `map_df()`:

Nakonec si ukážeme poněkud složitější příklad. Řekněme, že máme seznam atomických vektorů `s` (zde si jej nasimulujeme) a pro každý vektor v seznamu chceme zjistit základní popisné statistiky. Pro jeden vektor nám je spočítá funkce `summary()`. My však chceme tyto statistiky spočítat pro každý jednotlivý vektor a uložit je do tabulky, kde bude každý řádek odpovídat jednomu původnímu vektoru a sloupec jedné statistice. To můžeme udělat takto:

```
s <- replicate(5, rnorm(100), simplify = FALSE)
map_dfr(s, ~ set_names(as.list(summary(.)), nm = names(summary(.))))
```

```
## # A tibble: 5 x 6
##    Min. `1st Qu.`  Median    Mean `3rd Qu.`  Max.
##   <dbl>     <dbl>   <dbl>   <dbl>     <dbl> <dbl>
## 1 -1.97    -0.810 -0.264  -0.120      0.519  3.09
## 2 -2.16    -0.794  0.0483  0.0114     0.811  2.07
## 3 -1.89    -0.705  0.0242  0.0731     0.894  2.35
## 4 -1.91    -0.584  0.0694  0.119      0.871  3.06
## 5 -2.73    -0.708 -0.0993 -0.0777     0.581  2.54
```

Výraz `as.list(summary(.)` spočítá statistiky pro právě zpracovávaný prvek a převede je na seznam. Funkce `set_names()` pojmenuje prvky seznamu podle jednotlivých prvků objektu, který vrací funkce `summary()`. Funkce `map_dfr()` tak pro každý vektor v seznamu `s` získá pojmenovaný seznam statistik. Ten převede na tabulku a jednotlivé tabulky spojí.

Někdy nechceme transformovat všechny, ale jen vybrané prvky nějakého vektoru. K tomu slouží funkce `map_if(.x, .p, .f, ..., .else = NULL)` a `map_at(.x, .at, .f, ...)`. Tyto funkce fungují podobně jako `map()`, liší se však tím, na které prvky se použije funkce `.f`: `map()` ji použije všechny prvky vektoru `.x`, `map_at()` pouze na vyjmenované prvky a `map_if()` pouze na prvky, které splňují nějakou podmínku. Funkce `map_if()` má stejné parametry jako `map()` plus dva navíc: `.p` je *predikátová funkce* (tj. funkce, která vrací logické hodnoty), která určí, které prvky se budou transformovat; funkce `.f` se použije na ty prvky `.x`, kde `.p()` vrací `TRUE`. Pokud je zadaná i funkce `.else`, pak se použije na ty prvky `.x`, kde `.p()` vrací `FALSE`. Pokud parametr `.else` nezadáme, pak se tyto prvky ponechají beze změny.

Ukažme si to na příkladu. Řekněme, že máme seznam, který obsahuje vektory čísel a řetězců. Numerické vektory chceme standardizovat tak, že od nich odečteme jejich střední hodnotu, zatímco řetězce chceme ponechat beze změny. To můžeme snadno udělat takto:

```
v <- list(1:5, rnorm(5), LETTERS[1:5])  # tvorba seznamu
v
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1]  1.0060321 -0.7565377 -0.1138234  1.0701556 -0.3078905
##
## [[3]]
## [1] "A" "B" "C" "D" "E"
```

```
map_if(v, is.numeric, ~ . - mean(.))
```

```
## [[1]]
## [1] -2 -1  0  1  2
```

```
##
## [[2]]
## [1]  0.8264449 -0.9361249 -0.2934107  0.8905684 -0.4874777
##
## [[3]]
## [1] "A" "B" "C" "D" "E"
```

Pokud bychom chtěli v jednom kroku zároveň odečíst střední hodnotu od numerických vektorů a řetězce převést na malá písmena, mohli bychom to provést takto:

```
map_if(v, is.numeric, ~ . - mean(.), .else = stringr::str_to_lower)
```

```
## [[1]]
## [1] -2 -1  0  1  2
##
## [[2]]
## [1]  0.8264449 -0.9361249 -0.2934107  0.8905684 -0.4874777
##
## [[3]]
## [1] "a" "b" "c" "d" "e"
```

Protože tabulky jsou v R implementované jako seznam sloupců, můžeme totéž provést i s tabulkou:

```
df  # původní tabulka
```

```
## # A tibble: 6 x 3
##       x      y z
##   <int>  <dbl> <chr>
## 1     1 -0.818 <NA>
## 2     2 -0.721 a
## 3     3  0.243 b
## 4     4  0.192 c
## 5     5 -0.293 d
## 6     6 NA     <NA>
```

```
map_if(df, is.numeric, ~ . - mean(., na.rm = TRUE))
```

```
## $x
## [1] -2.5 -1.5 -0.5  0.5  1.5  2.5
##
## $y
## [1] -0.53869258 -0.44117688  0.52198411  0.47161292 -0.01372757          NA
##
## $z
## [1] NA  "a" "b" "c" "d" NA
```

Funkce `map_at()` funguje podobně, ale které prvky se mají transformovat, musíme zadat jejich jménem nebo pozicí. Stejně jako u výběru prvků vektoru je možné používat kladné i záporné vektory pozic: kladné určují, které se mají transformovat, záporné říkají, které prvky se nemají transformovat. Odečíst střední hodnoty od numerických vektorů tedy můžeme i takto:

```
map_at(df, 1:2, ~ . - mean(., na.rm = TRUE))
```

```
## $x
## [1] -2.5 -1.5 -0.5  0.5  1.5  2.5
##
## $y
## [1] -0.53869258 -0.44117688  0.52198411  0.47161292 -0.01372757          NA
##
## $z
## [1] NA  "a" "b" "c" "d" NA
```

```
map_at(df, -3, ~ . - mean(., na.rm = TRUE))  # totéž
map_at(df, c("x", "y"), ~ . - mean(., na.rm = TRUE))  # totéž
```

Funkce `map()` a její odvozeniny vracejí vždy konkrétní objekt: `map()` vrací seznam, `map_lgl()` vrací logický vektor apod. Pokud potřebujeme, aby funkce vrátila stejnou datovou strukturu, jako dostala na vstupu, můžete místo nich použít funkci `modify(.x, .f, ...)` a její odvozeniny, `modify_if(.x, .p, .f, ..., .else = NULL)`, `modify_at(.x, .at, .f, ...)`. Tyto funkce se používají stejně jako odpovídající funkce z rodiny `map()`. Srovnejte rozdíl:

```
map(1:3, ~ . + 2L)
```

```
## [[1]]
## [1] 3
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 5
```

```
modify(1:3, ~ . + 2L)
```

```
## [1] 3 4 5
```

```
map_if(df, is.numeric, ~ . ^ 2) %>% str()
```

```
## List of 3
##  $ x: num [1:6] 1 4 9 16 25 36
##  $ y: num [1:6] 0.6693 0.5192 0.0588 0.0369 0.0859 ...
##  $ z: chr [1:6] NA "a" "b" "c" ...
```

```
modify_if(df, is.numeric, ~ . ^ 2) %>% str()
```

```
## tibble [6 x 3] (S3: tbl_df/tbl/data.frame)
##  $ x: num [1:6] 1 4 9 16 25 36
##  $ y: num [1:6] 0.6693 0.5192 0.0588 0.0369 0.0859 ...
##  $ z: chr [1:6] NA "a" "b" "c" ...
```

Funkce `map()` aplikovaná na celočíselný vektor vrací seznam, zatímco funkce `modify()` vrací celočíselný vektor. (Zkuste, co se stane, pokud se pokusíte místo `2L` přičíst 2.) Podobně funkce `map_if()` aplikovaná na tabulku vrací seznam, zatímco funkce `modify_if()` vrací tabulku stejné třídy, jako je její vstup.

Funkce typu `modify()` se tedy hodí např. pro transformace vybraných sloupců tabulek. Řekněme, že máme tabulku s příjmy a výdaji z hyperinflační ekonomiky. Pokud by došlo k měnové reformě, potřebujeme u všech numerických sloupců škrtnout tři nuly. To můžeme provést např. takto:

```
df <- tibble::tibble(names = c("Adam", "Bětka", "Cyril"),
                     income = c(1.3, 1.5, 1.7) * 1e6,
                     rent = c(500, 450, 580) * 1e3,
                     loan = c(0, 250, 390) * 1e9)
df <- modify_if(df, is.numeric, ~ . / 1000)
df
```

```
## # A tibble: 3 x 4
##   names income  rent      loan
##   <chr>  <dbl> <dbl>     <dbl>
## 1 Adam    1300   500         0
## 2 Bětka   1500   450 250000000
## 3 Cyril   1700   580 390000000
```

V kapitole 16 se však naučíte jiné funkce specializované pro práci s tabulkami z balíku **dplyr**.

Někdy nechceme při iterování nad prvky vektoru vracet hodnoty, nýbrž provést nějaké jiné operace, které iterovaná funkce provádí jako své vedlejší účinky. K tomuto účelu slouží funkce `walk(x, f, ...)`. Její syntaxe je stejná jako u funkce `map()`. Řekněme, že chceme vypsat všechny prvky nějakého vektoru. To můžeme pomocí funkce `walk()` udělat např. takto:

```
v <- list(1, "a", 3)
walk(v, print)
```

```
## [1] 1
## [1] "a"
## [1] 3
```

Funkce `walk()` tiše vrací vektor `v`, takže je možné i zařadit i do proudu "trubek" (viz oddíl 5.6):

```
v %>% walk(print) %>% map_int(length)
```

```
## [1] 1
## [1] "a"
## [1] 3
```

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

To je užitečné např. při ladění dlouhého výrazu s mnoha "trubkami", protože do proudu můžeme zařadit výpis nebo vykreslení výsledků bez toho, abychom museli "potrubí" narušit.

10.2 Iterace nad více vektory současně

Někdy potřebujeme iterovat nad více vektory současně. Můžeme např. chtít vytvořit seznam vektorů tisíce gaussovských náhodných čísel, kde každý vektor bude mít jinou střední hodnotu a směrodatnou odchylku. Pomocí funkce `map()` bychom to mohli udělat např. takto:

```
m <- 0:5  # požadované střední hodnoty
std <- 1:6  # požadované směrodatné odchylky
z <- map(seq_along(m), ~ rnorm(1000, mean = m[.], sd = std[.]))
str(z)  # struktura výsledného seznamu
```

```
## List of 6
##  $ : num [1:1000] 0.675 -0.678 0.242 0.508 -0.061 ...
##  $ : num [1:1000] -2.0989 0.3351 -0.0143 1.436 -0.2521 ...
##  $ : num [1:1000] 2.68 1.06 4.06 5.78 -2.69 ...
##  $ : num [1:1000] 3.42 6.06 2.6 -1.6 5.27 ...
##  $ : num [1:1000] 2.45 4.84 -5.94 2.52 -4.8 ...
##  $ : num [1:1000] -2.02 11.35 11.49 14.15 8.25 ...
```

```
z %>% map_dbl(mean)  # střední hodnoty jednotlivých vektorů v seznamu
```

```
## [1] -0.00374628  1.03684235  2.07678403  2.97687652  4.07215524  4.77451595
```

Balík **purrr** však pro tyto účely nabízí příjemnější funkce. Pro iterace nad dvěma vektory zavádí funkci `map2(.x, .y, .f, ...)` a odpovídající zjednodušující funkce `map2_lgl()`, `map2_int()` atd. Všechny tyto funkce berou vektory `.x` a `.y`, nad kterými mají iterovat, jako své první dva parametry. Třetí parametr je jméno iterované funkce (musí brát aspoň dva parametry). Případné další parametry jsou předány funkci `.f()` jako její třetí a další parametr. Postup výpočtu ukazuje obrázek 10.2.

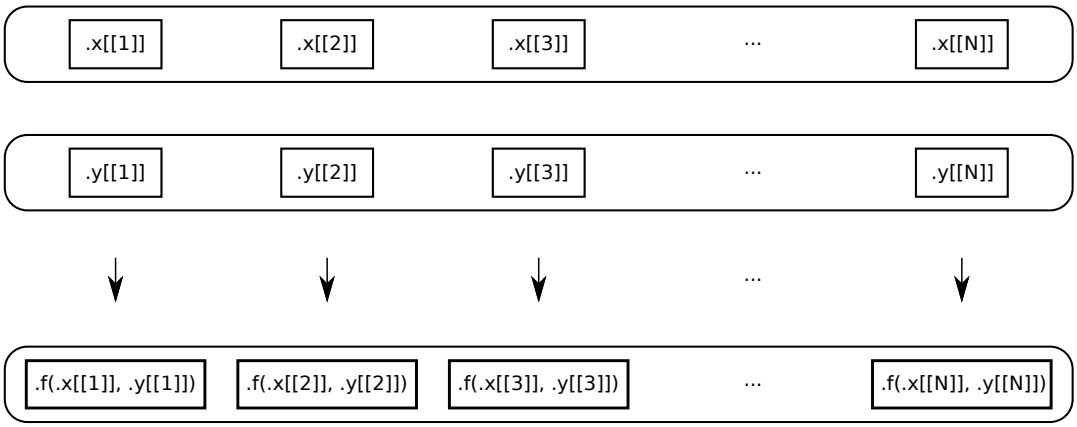


Figure 10.2: Funkce `map2(.x, .y, .f)` aplikuje funkci `.f()` na odpovídající prvky vektorů `.x` a `.y` a vrací seznam stejné délky.

Pokud chceme stejně jako výše vytvořit pět náhodných gaussovsky rozdělených vektorů se středními hodnotami 0, 1 atd. a směrodatnými odchylkami 1, 2 atd., můžeme je sestavit takto (všimněte si, že další parametry, zde délka vytvářených vektorů, musejí být uvedeny až za jménem iterované funkce):

```
z <- map2(0:5, 1:6, rnorm, n = 1000)
```

Funkce opět umožňuje zadat místo funkce `.f()` pravostrannou formuli, kterou na funkci sama převede. Zpracovávaný prvek vektoru `.x` v zadáme jako `.x`, prvek vektoru `.y` jako `.y`. Řekněme tedy, že chce vytvořit

tisíc vektorů náhodných čísel s gaussovským rozdělením a různými středními hodnotami a směrodatnými odchylkami, a z těchto vektorů spočítat jejich střední hodnotu. To můžeme udělat takto:

```
map2_dbl(0:5, 1:6, ~ mean(rnorm(n = 1000, mean = .x, sd = .y)))
```

```
## [1] -0.04180974  1.03073488  1.90149734  2.95012582  4.28840271  4.98437730
```

Pro iterace nad větším počtem vektorů nabízí **purrr** funkci `pmap(.l, .f, ...)` a její zjednodušující varianty `pmap_lgl()` atd., kde `.l` je buď seznam nebo tabulka vektorů, nad kterými se má iterovat, a `.f` je buď funkce, která bere příslušný počet parametrů, nebo pravostranná formule, kterou `pmap()` převede na funkci. Pokud je `.f` funkce a jednotlivé vektory v `.l` nejsou pojmenované, pak se předávají do `.f` podle svého pořadí. Pokud jsou pojmenované, pak se předávají jménem, takže na jejich fyzickém pořadí v `.l` nezáleží.

Řekněme, že chceme opět vytvořit seznam náhodných výběrů z gaussovského rozdělení. Každý výběr bude mít různý počet prvků, různou střední hodnotu a různou směrodatnou odchylku. Pokud seznam parametrů nepojmenujeme, musíme mít jednotlivé parametry v seznamu v tom pořadí, v jakém je očekává funkce `rnorm()`, která vygeneruje náhodná čísla:

```
n <- (1:5) * 100  # počet pozorování je 100, 200, ..., 500
mu <- 0:4  # střední hodnota je 0, 1, ..., 4
sd <- 1:5  # směrodatná odchylka je 1, 2, ..., 5
pars <- list(n, mu, sd)  # nepojmenovaný seznam parametrů v pořadí
z <- pmap(pars, rnorm)
str(z)  # struktura výsledku
```

```
## List of 5
##  $ : num [1:100] -0.0937 0.0369 0.8324 0.9205 0.253 ...
##  $ : num [1:200] -1.526 3.452 -2.018 1.2 -0.787 ...
##  $ : num [1:300] 0.776 3.388 1.56 -4.816 2.411 ...
##  $ : num [1:400] -2.232 9.108 6.722 0.297 6.05 ...
##  $ : num [1:500] 13.14 1.95 7.36 3.69 -1.52 ...
```

Pokud jednotlivé parametry v seznamu pojmenujeme, na jejich pořadí nezáleží, protože se předají jménem:

```
pars <- list(sd = sd, mean = mu, n = n)  # pojmenovaný seznam parametrů
z <- pmap(pars, rnorm)
str(z)  # struktura výsledku
```

```
## List of 5
##  $ : num [1:100] 0.97 -0.46 -0.639 1.098 -0.203 ...
##  $ : num [1:200] -2.9111 -1.5736 0.4012 3.2568 0.0803 ...
##  $ : num [1:300] 2.511 1.197 1.587 -2.206 0.464 ...
##  $ : num [1:400] 2.15 3.29 2.54 -5.64 3.55 ...
##  $ : num [1:500] 4.47 6.74 -4.47 2.72 12.17 ...
```

Pohodlnější je však zadat parametry jako tabulku:

```
pars <- tibble::tibble(sd = sd, mean = mu, n = n)
z <- pmap(pars, rnorm)
str(z)  # struktura výsledku
```

```
## List of 5
##  $ : num [1:100] -0.0538 0.3623 -0.8932 -2.4716 -0.0818 ...
##  $ : num [1:200] 0.638 -2.157 -0.764 1.902 -2.49 ...
##  $ : num [1:300] 0.4203 3.0874 -0.0851 -2.011 1.6105 ...
##  $ : num [1:400] 5.024 1.214 -0.941 5.006 7.717 ...
##  $ : num [1:500] 6.51 -1.53 -5.23 -1.9 1.25 ...
```

Pokud místo funkce zadáme `.f` jako pravostrannou formuli, pak první vektor v seznamu nebo tabulce označíme jako `..1`, druhý jako `..2` atd.:

```
z <- pmap(pars, ~ rnorm(n = ..3, mean = ..2, sd = ..1))
str(z)  # struktura výsledku
```

```
## List of 5
##  $ : num [1:100] 0.1882 -0.5366 -0.0551 -0.2683 -0.518 ...
##  $ : num [1:200] -0.894 3.56 3.405 0.891 0.127 ...
##  $ : num [1:300] 2.12 4.16 5.92 1.88 1.7 ...
##  $ : num [1:400] 0.836 6.984 -3.232 2.645 5.301 ...
##  $ : num [1:500] 3.25 14.83 6.23 2.23 5.47 ...
```

Balík **purrr** implementuje i funkci `modify2()` a funkce `walk2()` a `pwalk()`, které umožňují iterovat vedlejší efekty nad více vektory.

10.3 Filtrace a detekce prvků vektorů

Balík **purrr** implementuje i několik funkcí určených k filtraci hodnot vektorů. Funkce `keep(.x, .p, ...)` vrací ty prvky vektoru `.x`, pro které predikátová funkce `.p()` vrací hodnotu `TRUE`. Naopak funkce `discard(.x, .p, ...)` vrací ty prvky vektoru `.x`, pro které predikátová funkce `.p()` vrací hodnotu `FALSE`, tj. zahazuje prvky, pro které podmínka platí. Funkce `head_while(.x, .p, ...)` a `tail_while(.x, .p, ...)` vrací všechny prvky od začátku nebo od konce, pro které funkce `.p()` souvisle vrací hodnotu `TRUE`. Ve všech těchto funkcích nemusí být `.p` funkce: může to být i logický vektor stejné délky jako `.x` nebo pravostranná formule, která vrací logickou hodnotu. Jejich použití ukazuje následující příklad:

```
v <- 1:10
is.odd <- function(x) x %% 2 != 0  # vrací TRUE, když je číslo liché
keep(v, is.odd)  # výběr lichých hodnot z vektoru v
```

```
## [1] 1 3 5 7 9
```

```
keep(v, ~ . %% 2 != 0)  # totéž pomocí pravostranné formule
```

```
## [1] 1 3 5 7 9
```

```
discard(v, is.odd)  # vrácení vektoru v bez lichých hodnot
```

```
## [1]  2  4  6  8 10
```

```
head_while(v, ~ . < 5)  # vrácení prvních hodnot menších než 5
```

```
## [1] 1 2 3 4
```

Funkce `compact(.x, .p = identity)` umožňuje ze seznamu vypustit ty prvky, které mají buď hodnotu `NULL` nebo nulovou délku.

```
compact(list(a = 1, b = 2, c = NULL, d = 4, e = numeric(0)))
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $d
## [1] 4
```

Parametr `.p` umožňuje zadat funkci nebo formuli. Pokud tato funkce vrátí `NULL` nebo prázdný vektor, pak funkce `compact()` vynechá odpovídající prvek. Zbývající hodnoty však nejsou funkcí `.p` nijak transformované. Použití ukazuje triviální příklad:

```
compact(1:5, .p = ~ .[. < 4])  # zachová pouze prvky menší než 4
```

```
## [1] 1 2 3
```

Funkce `detect(.x, .f, ..., .dir = c("forward", "backward"), .default = NULL)` vrací první položku vektoru `.x`, pro kterou vrací `.f` hodnotu `TRUE`. Funkce `detect_index(.x, .f, ..., .dir = c("forward", "backward"))` vrací index této položky. Stejně jako výše může `.f` být funkce nebo pravostranná formule, která vrací logickou hodnotu.

```
detect(v, is.odd)  # první lichá hodnota ve vektoru v
```

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

```
detect(v, ~ . > 1)  # první hodnota větší než 1
```

```
## [1] 2
```

```
detect_index(v, is.odd)  # index prvního lichého prvku vektoru v
```

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

```
detect_index(v, ~ . > 1)  # index prvního prvku většího než 1
```

```
## [1] 2
```

Dva zbývající parametry určují směr, odkud se budou hodnoty hledat (parametr `.dir`, implicitně zepředu), a jaká hodnota se vrátí, pokud žádný prvek vektoru nesplňuje zadaný predikát (parametr `.default`).

Funkce `every(.x, .p, ...)` a `some(.x, .p, ...)` zobecňují logické funkce `all()` a `any()`. `every()` vrací `TRUE`, pokud zadaná predikátová funkce `.p` vrací pro každý prvek vektoru `.x` hodnotu `TRUE`; funkce `some()` vrací `TRUE`, pokud `.f` vrací `TRUE` aspoň pro jeden prvek `.x`. Pomocí těchto funkcí můžeme např. otestovat, zda tabulka `df` obsahuje aspoň jeden numerický sloupec (`some()`) nebo jen numerické sloupce (`every()`):

```
df %>% some(is.numeric)  # obsahuje df aspoň jeden numerický sloupec?
```

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

```
df %>% every(is.numeric)  # obsahuje df pouze numerické sloupce?
```

```
## [1] FALSE
```

Funkce `has_element(.x, .y)` zobecňuje operátor `%in%`. Vrací `TRUE`, pokud vektor `.x` obsahuje objekt `.y`.

```
x <- list(1:5, "a")  # prvky x jsou vektory 1:5 a "a"
has_element(x, 1:5)
```

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

```
has_element(x, 3)
```

```
## [1] FALSE
```

Balík **purrr** nabízí i užitečnou funkci `negate()`, která transformuje zadanou funkci tak, že vrací její negaci. Pokud bychom chtěli pomocí `keep()` a naší funkce `is.odd()` vybrat sudé prvky, museli bychom použít formuli:

```
keep(1:10, ~ !is.odd(.))
```

```
## [1]  2  4  6  8 10
```

Pomocí funkce `negate()` však můžeme negovat celou predikátovou funkci `is.odd()`:

```
keep(1:10, negate(is.odd))
```

```
## [1]  2  4  6  8 10
```

10.4 Výběr a úpravy prvků vektorů

Často potřebujeme ze složitější struktury získat jeden její prvek. Obecně k tomu slouží funkce `[[` (dvojité hranaté závorky). Funkce `pluck(.x, ..., .default = NULL)` tuto myšlenku zobecňuje. Umožňuje vybrat libovolně zanořený prvek vektoru `.x`. Ukážeme si to na příkladu vektoru hráčů:

```
dungeon %>% str()
```

```
## List of 2
##  $ :List of 3
##   ..$ id   : num 11
##   ..$ name : chr "Karel"
##   ..$ items:List of 2
##   .. ..$ : chr "sword"
##   .. ..$ : chr "key"
##  $ :List of 3
```

```
##   ..$ id   : num 12
##   ..$ name : chr "Emma"
##   ..$ items:List of 3
##   .. ..$ : chr "mirror"
##   .. ..$ : chr "potion"
##   .. ..$ : chr "dagger"
```

První parametr `pluck()` je vektor, ze kterého vybíráme. Další parametry jsou pozice nebo jména prvků, které vybíráme. Pokud zadáme víc položek, pak výběr funguje rekurzivně: druhá položka vybírá z výsledku prvního výběru atd.:

```
pluck(dungeon, 1)
```

```
## $id
## [1] 11
##
## $name
## [1] "Karel"
##
## $items
## $items[[1]]
## [1] "sword"
##
## $items[[2]]
## [1] "key"
```

```
pluck(dungeon, 1, "name")
```

```
## [1] "Karel"
```

```
pluck(dungeon, 1, "items")
```

```
## [[1]]
## [1] "sword"
##
## [[2]]
## [1] "key"
```

```
pluck(dungeon, 1, "items", 1)
```

```
## [1] "sword"
```

K výběru můžeme použít i funkci, která vrací výběr z vektoru:

```
artefact <- function(x) x[["items"]]  # funkce vrací artefakty vybraného hráče
artefact(dungeon[[1]])  # seznam všech artefaktů prvního hráče
```

```
## [[1]]
## [1] "sword"
##
## [[2]]
## [1] "key"
```

```
pluck(dungeon, 1, artefact, 1)  # 1. artefakt prvního hráče
```

```
## [1] "sword"
```

Všechny tyto výběry můžeme samozřejmě provést i pomocí základních funkcí R, syntaxe `pluck()` je však přehlednější. Poslední výběr bychom např. museli zadat takto:

```
artefact(dungeon[[1]])[[1]]
```

```
## [1] "sword"
```

Pokud hledaný prvek ve vektoru neexistuje, funkce `pluck()` vrátí `NULL`. Tuto hodnotu můžeme změnit pomocí parametru `.default`:

```
pluck(dungeon, 3)
```

```
## NULL
```

```
pluck(dungeon, 3, .default = NA)
```

```
## [1] NA
```

Pokud bychom potřebovali, aby funkce raději zhavarovala, můžeme místo `pluck()` použít funkci `chuck(x, ...)`:

```
chuck(dungeon, 3)
```

```
## Error: Index 1 exceeds the length of plucked object (3 > 2)
```

Funkce `pluck()` umožňuje i měnit hodnotu vybraného prvku (zde bohužel není možné při výběru použít funkci jako je např. naše funkce `artefact()`):

```
pluck(dungeon, 1, "items", 1) <- "megaweapon"
str(dungeon)
```

```
## List of 2
##  $ :List of 3
##   ..$ id   : num 11
##   ..$ name : chr "Karel"
##   ..$ items:List of 2
##   .. ..$ : chr "megaweapon"
##   .. ..$ : chr "key"
##  $ :List of 3
##   ..$ id   : num 12
##   ..$ name : chr "Emma"
##   ..$ items:List of 3
##   .. ..$ : chr "mirror"
##   .. ..$ : chr "potion"
##   .. ..$ : chr "dagger"
```

První hráč má nyní místo meče nějakou “megazbraň”.

Pokud potřebujete změnit nějaký prvek vektoru, můžete použít funkce `assign_in()` a `modify_in()`. Na jejich použití se podívejte do dokumentace.

10.5 Zabezpečení iterací proti chybám

Pokud jedna z iterací ve funkci `map()` a spol. skončí chybou, skončí chybou celé volání této funkce. Pak může být obtížné zjistit, který prvek vektoru chybu způsobil. Jedním z přínosů balíku **purrr** je to, že zavádí několik speciálních funkcí, které umožňují tento problém vyřešit. Všechny tyto funkce fungují tak, že transformují funkci `.f` ještě před tím, než vstoupí do `map()` – na vstupu vezmou funkci a vrací její zabezpečenou verzi odolnou vůči selhání.

První z těchto funkcí je `safely(.f, otherwise = NULL, quiet = TRUE)`. Tato funkce bere na vstupu iterovanou funkci a vrací její modifikovanou verzi, která nikdy neskončí chybou a která vrací seznam dvou prvků: výsledku a chybového objektu. Pokud původní funkce proběhla, má chybová hlášení hodnotu `NULL`, pokud skončila chybou, má hodnotu `NULL` výsledek. Protože funkce vrací seznam, je možné ji použít pouze ve funkci `map()`, ne v jejích zjednodušujících variantách jako je `map_lgl()` apod.

Ukážeme si použití této funkce na příkladu. Máme seznam `v`, který obsahuj většinou čísla, mezi která je však přimíchaný jeden řetězec. Pro každý prvek `v` chceme spočítat vektor jeho logaritmů (pomocí funkce `map()` přesto, že funkce `log()` je vektorizovaná). Přímé volání funkce `log()` skončí chybou:

```
v <- list(1, 2, "a", 5)
map(v, log)
```

```
## Error in .Primitive("log")(x, base): non-numeric argument to mathematical function
```

Pokud funkci `log()` "obalíme" funkcí `safely()`, výpočet proběhne až do konce a výsledkem bude struktura popsaná výše. Všimněte si, že v 1., 2. a 4. prvku struktury má chybová složka hodnotu `NULL`. Ve 3. prvku, který zhavaroval, obsahuje chybová složka seznam, který obsahuje objekt třídy *error*. Ruční prohlídka našeho výsledku by nám umožnila zjistit, že je to 3. prvek vektoru `v`, který způsobuje chyby:

```
result <- map(v, safely(log))
str(result)  # struktura výsledku
```

```
## List of 4
##  $ :List of 2
##   ..$ result: num 0
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: num 0.693
##   ..$ error : NULL
##  $ :List of 2
##   ..$ result: NULL
##   ..$ error :List of 2
##   .. ..$ message: chr "non-numeric argument to mathematical function"
##   .. ..$ call   : language .Primitive("log")(x, base)
##   .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
##  $ :List of 2
##   ..$ result: num 1.61
##   ..$ error : NULL
```

Ruční hledání chyb je však možné jen v případě, že zpracovávaná data jsou velmi malá. Balík **purrr** naštěstí umožňuje proces hledání chyby zautomatizovat. Stačí na výsledek předchozího výpočtu použít funkci `transpose()`, která změní seznam párů na pár seznamů. Výraz `transpose(result)` tedy vrátí seznam dvou prvků: první obsahuje všechny výsledky a druhý všechna chybová hlášení (oboje uložené jako seznamy):

```
transpose(result)
```

```
## $result
## $result[[1]]
## [1] 0
##
## $result[[2]]
## [1] 0.6931472
##
## $result[[3]]
## NULL
##
## $result[[4]]
## [1] 1.609438
##
##
## $error
## $error[[1]]
## NULL
##
## $error[[2]]
## NULL
##
## $error[[3]]
## <simpleError in .Primitive("log")(x, base): non-numeric argument to mathematical function>
##
## $error[[4]]
## NULL
```

To nám umožní najít problematické prvky vektoru `v` např. tak, že vybereme pouze část chybových hlášení a z ní sestavíme logický vektor, který bude mít hodnotu `TRUE` tam, kde hodnota chyby není `NULL`, tj. volání funkce selhalo. Logický vektor pak můžeme použít k nalezení indexů prvků vektoru, kde k chybě došlo (pomocí funkce `which()`), nebo k vypsání hodnot, které chybu způsobily:

```
bugs <- transpose(result)$error  # transpozice a výběr chybové složky
bugs <- !map_lgl(bugs, is.null)  # TRUE, kde error není NULL, tj. kde je chyba
which(bugs)  # index prvků v, kde nastala chyba (jako vektor)
```

```
## [1] 3
```

```
x[bugs]  # hodnoty prvků v, kde nastala chyba (jako seznam)
```

```
## [[1]]
## NULL
```

Pokud nás nezajímají chyby, ale pouze ty výsledky, které se skutečně spočítaly, můžeme použít funkci `possibly(.f, otherwise, quiet = TRUE)`. Tato funkce zabezpečí funkci `.f` tak, že nikdy nezhavaruje. Pokud není schopná spočítat výsledek, vrátí místo něj hodnotu zadanou v parametru `otherwise` (pokud není zadána, funkce zhavaruje). Díky tomu funkce `possibly()` vrací jen vlastní výsledky, takže může být využitá i ve zjednodušujících variantách `map()`:

```
map_dbl(v, possibly(log, otherwise = NA_real_))  # chybná hodnota je nahrazena NA
```

```
## [1] 0.0000000 0.6931472        NA 1.6094379
```

Pokud byste místo zachytávání chyb potřebovali zachytit zprávy a varování, která vrací iterovaná funkce, můžete použít funkci `quietly(.f)`.

Poslední funkce, kterou balík **purrr** nabízí k ovlivnění výpočtu, je funkce `auto_browse(.f)`, která transformuje funkci `.f` tak, že v případě chyby, automaticky spustí ladící mechanismus:

```
map(v, auto_browse(log))
```

10.6 Rekurzivní kombinace prvků vektorů

Někdy máme seznam objektů, na které potřebujeme aplikovat funkci, která však bere jen dva vstupy. Patrně nejdůležitější příklad takového užití je spojení mnoha tabulek uložených v seznamu pomocí funkce `left_join()`, se kterým se seznámíte v kapitole 16. V takovém případě chceme aplikovat funkci postupně: nejprve spojit první dvě tabulky, pak k výsledku připojit třetí tabulku, k výsledku tohoto spojení čtvrtou atd. Balík **purrr** k tomuto účelu nabízí čtyři funkce: `reduce(.x, .f, ..., .init, .dir = c("forward", "backward"))` a `reduce2(.x, .y, .f, ..., .init)` a `accumulate(.x, .f, ..., .init, .dir = c("forward", "backward"))` a `accumulate2(.x, .y, .f, ..., .init)`. My se zde podíváme jen na funkce `reduce()` a `accumulate()`, které pracují nad jedním seznamem; druhé dvě funkce pracují paralelně nad dvěma.

Funkce `accumulate()` postupně aplikuje na vektor `.x` funkci `.f` a vrací seznam stejné délky jako `.x` (nebo o 1 delší, viz dále). Prvním prvkem výsledného vektoru je `.x[[1]]`, druhým `.f(.x[[1]], .x[[2]])`, třetím `.f(.f(.x[[1]], .x[[2]]), .x[[3]])` atd. Pokud tedy "akumulujeme" atomický vektor čísel $1, 2, \ldots, N$ pomocí funkce součtu `+`, pak dostaneme atomický vektor čísel $1, 1+2, 1+2+3, \ldots$, tedy totéž, co by nám vrátila funkce `cumsum()`:

```
accumulate(1:5, `+`)
```

```
## [1]  1  3  6 10 15
```

Funkce `reduce()` funguje podobně, ale vrací jen finální výsledek akumulace, zde tedy součet všech prvků vektoru:

```
reduce(1:5, `+`)
```

```
## [1] 15
```

Podívejme se nyní na realističtější příklad. Řekněme, že máme seznam atomických vektorů, které obsahují id jedinců, kteří se zúčastnili nějakých akcí. Zajímá nás, kteří jedinci, se zúčastnili *všech* těchto akcí, tj. hledáme průnik všech těchto množin. K tomu slouží funkce `intersect(x, y)`, která však umí vrátit pouze průnik dvou množin. Musíme ji tedy použít na všechny prvky seznamu rekurzivně. Protože nás zajímá jen finální průnik, použijeme funkci `reduce()`:

```
riots <- list(  # seznam id účastníků různých akcí
  c(1, 2, 3, 7, 9),
  c(1, 4, 8, 9),
  c(1, 3, 5, 9)
)
reduce(riots, intersect)  # celkový průsečík množin
```

```
## [1] 1 9
```

Funkce `reduce()` a `accumulate()` umožňují zadat i počáteční hodnotu pomocí parametru `.init`. To se hodí v případě, kdy by akumulovaný vektor mohl být prázdný a my nechceme, aby výpočet zhavaroval.

```
v <- numeric(0)
reduce(v, `+`)
```

```
## Error: `.x` is empty, and no `.init` supplied
```

```
reduce(v, `+`, .init = 0)
```

```
## [1] 0
```

Pokud zadáme parametr `.init`, bude výsledek funkce `accumulate()` o 1 delší než vstupní vektor.

Parametr `.dir` umožňuje nastavit směr akumulace (implicitně se akumuluje od prvního prvku vektoru po poslední). Detaily najdete v dokumentaci.

10.7 Paralelizace výpočtu

Funkce `map()` a spol. pracují s každým prvkem vektoru samostatně a izolovaně, takže nezáleží na pořadí, v jakém jsou tyto prvky zpracovány. To, mimo jiné, umožňuje paralelizaci výpočtu, tj. zpracování každého prvku na jiném jádře počítače nebo jiném prvku počítačového klastru. Pokud jsou data tak velká, že se to vyplatí, je možné použít balík **furrr**. Balík **furrr** implementuje paralelizované ekvivalenty funkcí `map()`, `map2()`, `pmap()`, `modify()` a všech jejich variant, které zjednodušují výsledek na atomický vektor nebo tabulku jako je `map_dbl()` nebo transformují jen vybrané prvky jako `map_at()`. Tyto alternativní funkce mají konzistentní pojmenování: před jméno dané funkce připojují `future_`, takže místo `map()` máme `future_map()` apod. Všechny tyto funkce také vracejí stejné výsledky a berou stejné parametry jako odpovídající funkce z balíku **purrr** (plus parametr `.progress`, který umožňuje sledovat průběh výpočtu pomocí progress baru, a parametr `.options`, který umožňuje předávat dodatečné informace paralelizačnímu stroji).

Před použitím těchto funkcí je třeba nejen načíst balík **furrr**, ale i nastavit paralelizaci. K tomu slouží funkce `plan()`. Její hlavní parametr určuje, jak se bude paralelizovat. Implicitní hodnota je `sequential`, tj. normální výpočet na jednom jádře bez paralelizace. Typické nastavení je

```
library(furrr)
plan(multiprocess)
```

které nastaví nejlepší dostupnou paralelizaci na daném stroji. Vlastní iterace jsou pak stejné jako s balíkem **purrr**, stačí jen přidat `future_` ke jménu funkce, tj. např. volat:

```
future_map_dbl(1:5, ~ . ^ 2)
```

```
## [1]  1  4  9 16 25
```

Na velmi stylizovaném příkladu si ukážeme, jak paralelizace zrychluje výpočet. Pomocí funkce `map()` a `future_map()` spustíme třikrát výraz `Sys.sleep(1)`, který na 1 sekundu pozastaví výpočet. Pomocí funkce `system.time()` změříme, jak dlouho tento "výpočet" trval:

```
system.time(map(1:3, ~ Sys.sleep(1)))
```

```
##    user  system elapsed
##   0.002   0.000   3.005
```

```
system.time(future_map(1:3, ~ Sys.sleep(1)))
```

```
##    user  system elapsed
##   0.049   0.012   1.103
```

Zatímco s pomocí `map()` trval, jak bychom očekávali, zhruba tři sekundy, s pomocí `future_map()` zabral jen o málo víc než 1 sekundu, protože každé jednotlivé volání `Sys.sleep(1)` proběhlo na mém osmijádrovém počítači v jiném sezení R. Výsledek by byl ještě rychlejší, kdybych na svém Linuxovém stroji nekompiloval tento text v RStudiu, viz dále.

Implicitně se používají všechna jádra počítače. To je možné změnit tak, že nejprve pomocí funkce `availableCores()` zjistíme počet dostupných jader, a pak nastavíme počet jader použitých výpočtu ve funkci `plan()` pomocí parametru `workers`. Pokud tedy chceme např. jedno jádro ušetřit pro ostatní procesy běžící na počítači, můžeme paralelizaci naplánovat takto:

```
n_cores <- availableCores()
plan(multiprocess, workers = n_cores - 1)
```

Popis podporovaných způsobů paralelizace najdete v dokumentaci k funkci `plan()` a v základní vinětě k balíku **future**, který se stará o backend. Základní způsoby paralelizace jsou čtyři: `sequential` provádí jednotlivé výpočty normálně bez paralelizace na jednom jádře, `multisession` spouští jednotlivé výpočty v nových kopiích R, `multicore` vytváří forky procesů R a `cluster` používá počítačový klastr. Ve většině případů budeme pracovat na jednom počítači, takže volíme mezi `multisession` a `multicore`. `mulitcore` je při tom efektivnější, protože nemusí kopírovat pracovní prostředí (globální proměnné, balíky apod.) do nového procesu, zatímco `multisession` musí překopírovat potřebné objekty do nového sezení R. `multicore` však není dostupný ve Windows, které fork vůbec nepodporují, ani při spuštění výpočtu v rámci RStudia. Nejrozumnější je proto použít `multiprocess`, který buď o zavolá `multicore` (na Linuxu nebo macOS při spuštění R mimo RStudio) nebo `multisession` (jinak).

Při spuštění `multisession` musí `future_map()` a spol. překopírovat do nového sezení potřebné proměnné. Většinou to funguje bezproblémově automaticky. Pokud však něco selže, přečtěte si viněty k balíku **future**, které vysvětlují, jak potřebné proměnné do nového sezení nakopírovat ručně a jak vyřešit i další případné problémy. Ani jeden z autorů tohoto textu však při použití **furrr** zatím na takový problém nenarazil.

Potřeba přesouvat data do a z nového procesu také znamená, že paralelní výpočet na čtyřech jádrech nebude typicky čtyřikrát rychlejší než na jednom. Pokud byste spouštěli jen malý počet velmi rychlých výpočtů na velkých datech, mohlo by se dokonce stát, že výpočet bude *pomalejší* než na jednom jádře. Paralelizovat se tak vyplatí jen výpočty, kde každé spuštění funkce nad prvkem seznamu trvá poměrně dlouho nebo se zpracovává poměrně hodně prvků.

10.8 Srovnání map() s cyklem for

Většinu začátečníků funkce typu `map()` děsí a snaží se daný problém řešit pomocí cyklů. To je samozřejmě možné. Vraťme se k příkladu, kdy máme nějakou tabulku a chceme zjistit, kolik který její sloupec obsahuje chybějících hodnot. Pomocí funkce `map_int()` to uděláme na jednom řádku:

```
map_int(df, ~ sum(is.na(.)))
```

```
##  names income   rent   loan
##      0      0      0      0
```

Pokud budeme místo toho chtít použít cyklus `for`, bude náš kód vypadat takto:

```
result <- integer(ncol(df))
for (k in seq_len(ncol(df)))
    result[k] <- sum(is.na(df[[k]]))
names(result) <- names(df)
result
```

```
##  names income   rent   loan
##      0      0      0      0
```

Výsledek je v obou případech stejný, ale kód napsaný pomocí cyklů má několik nevýhod: 1) Kód napsaný pomocí cyklu `for` bývá obvykle delší a podstatně méně přehledný. Funkce `map()` jasně ukazuje, na jakých datech se iteruje a co se na nich iteruje. V kódu cyklu to není zdaleka tak jasně vidět. 2) Při použití cyklu `for` musíte myslet na víc věcí: musíte si předalokovat vektor pro uložení výsledku, musíte vektor pojmenovat (pokud o to stojíte) a musíte přemýšlet, jestli použijete jednoduché nebo dvojité hranaté závorky (podle toho, zda je původní vektor a výsledek atomický vektor, seznam, nebo tabulka typu *data.frame* nebo *tibble*). 3) Zaplevelíte si pracovní prostředí nechtěnými proměnnými: možná proměnnou `result`, určitě však počítadlem cyklu `k`. A 4) Mapovací funkce je mnohem jednodušší paralelizovat. Obecně je proto lepší vždy používat funkce typu `map()` raději než explicitní cyklus `for`.

Přesto však existují situace, kdy je využití cyklů nezbytné. Hlavní případy jsou dva: 1) Když výpočet i-té hodnoty záleží na některé z hodnot spočítaných dříve a 2) když pro ušetření času nahrazujeme hodnoty přímo v původní datové struktuře, místo abychom celou strukturu vytvářeli znovu. V ostatních situacích je téměř vždy rozumnější použít některou z funkcí typu `map()`.

10.9 Aplikace

Někdy potřebujeme načíst velké množství tabulek z jednotlivých `.csv` souborů, zkontrolovat jejich konzistenci a spojit je dohromady. To dělá následující kód, který představuje zjednodušenou verze části jednoho mého projektu:

```
# načtení potřebných balíků
library(readr)
library(purrr)
# adresář s daty
DATADIR <- file.path("..", "testdata")
# seznam jmen souborů, které splňují určitou masku danou regulárním výrazem
PRODUCT_FILES <- dir(DATADIR, "products_.*\\.csv\\.gz", full.names = TRUE)
# načtení jednotlivých souborů, sloupcům vnutíme typ řetězec
product_data <- map(PRODUCT_FILES,
                    ~ read_csv2(file = ., col_types = cols(.default = "c")))
# kontrola, že všechny tabulky mají stejnou strukturu
product_col_names <- map(product_data, colnames)
product_col_names_test <- map_chr(product_col_names,
                                  ~ all.equal(product_col_names[[1]], .))
if (!(all(identical(product_col_names_test,
                    rep(TRUE, length = length(product_col_names))))))
    stop("Product data sets have different columns!")
# spojení jednotlivých tabulek do jedné
product_data <- reduce(product_data, rbind)
```

Vlastní spojení dat je možné provést efektivněji pomocí funkce `bind_rows()` z balíku **dplyr**, která je efektivnější a rychlejší:

```
product_data <- dplyr::bind_rows(product_data)
```

Part II

Načítání a ukládání dat

11 Načítání a ukládání dat ze souborů

Práce s daty začíná vždy jejich načtením a často končí jejich uložením. V této kapitole se naučíte:

- co jsou to textové delimitované tabulární soubory, jak je načíst a uložit
- jak pracovat s nedelimitovanými textovými tabulárními soubory
- jak pracovat s komprimovanými soubory textovými a soubory na webu
- jak uložit a načíst data do nativního R-kového binárního souboru
- jak pracovat s daty uloženými v balících
- jak načíst data uložená v souboru MS Excel
- jak pracovat s daty ve formátu statistických programů SPSS, SAS a Stata
- jak zkontrolovat integritu načtených dat

11.1 Textové tabulární delimitované soubory

Základní způsob výměny dat jsou textové tabulární delimitované formáty, kde jednotlivé řádky odpovídají pozorováním a kde jsou sloupce (proměnné) odděleny nějakým jasně definovaným znakem, např. mezerou, čárkou, středníkem nebo dvojtečkou. Velká výhoda těchto formátů je, že je možné je načíst téměř do každého softwaru, který pracuje s daty a stejně tak je z něj i vypsat. Zároveň jsou data "čitelná lidmi" a pokud se něco pokazí, je často možné data nějak zachránit. Nevýhodou je, že tyto formáty mohou obsahovat pouze tabulární data (tabulky), jednoduché datové typy (tj. ne složitější objekty, jako jsou např. odhady modelů), nemohou obsahovat metadata (např. atributy), jejich načítání může trvat dlouho (pokud jsou data velká) a že data v těchto formátech na disku zabírají hodně místa (pokud nejsou komprimovaná).

Příkladem dat v textovém tabulárním formátu je např. soubor "bmi_data.csv". Na disku vypadá takto:

```
id,height,weight,bmi
1,153,55,23.4952368747
2,207,97,22.6376344839
3,173,83,27.7322997761
4,181,92,28.0821708739
5,164,112,41.6418798334
```

Zde se podíváme na to, jak data v textovém tabulárním formátu načíst a uložit pomocí funkcí z balíku **readr**. Oproti podobným funkcím ze základního R, tyto funkce načítají data rychleji, umožňují přesněji určit, jak se mají data načíst a neprovádí některé nepříjemné úpravy.[1] Před použitím těchto funkcí musíme nejdříve načíst balík **readr**:

```
library(readr)
```

[1] Od verze 2.0 neumí funkce z balíku **readr** načítat soubory, které mají řádky oddělené pouze pomocí tzv. *carriage return* (znaku `\r`). Přestože žádný operační systém by už od roku 2001 neměl takové soubory vytvářet, můžete je v divočině občas dosud najít. Nejjednodušší způsob, jak takový soubor načíst, je použít funkce ze základního R.

11.1.1 Načítání dat

Všechny funkce z balíku **readr**, které sloužící k načítání dat, jsou pojmenované `read_XXX()`, kde `XXX` je jméno načítaného formátu. Data ve formátu CSV tedy načítá funkce `read_csv()`. (Pozor! Funkce ze základního R určené k načtení těchto dat jsou pojmenované téměř stejně, jen místo podtržítka je jméno formátu oddělené tečkou. Obdobou funkce `read_csv()` je funkce `read.csv()` ze základního balíku. Přestože se funkce jmenují podobně, mají výrazně odlišný interface i chování.) Všechny funkce `read_XXX()` z balíku **readr** vracejí tabulku třídy *tibble*.

Základní funkcí k načtení textových delimitovaných tabulárních dat je funkce `read_delim()`. Při jejím použití je nezbytné zadat pouze jediný parametr: jméno načítaného souboru včetně cesty. Při takovém použití funkce odhadne, který znak odděluje jednotlivé sloupce (proměnné) v datech. Bezpečnější je však tento oddělovací znak zadat explicitně pomocí parametru `delim`. V našem případě nastavíme `delim = ","`:

```
bmi_data <- read_delim("data/reading_and_writing/bmi_data.csv", delim = ",")
```

```
## Rows: 5 Columns: 4
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): id, height, weight, bmi
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
bmi_data
```

```
## # A tibble: 5 x 4
##      id height weight   bmi
##   <dbl>  <dbl>  <dbl> <dbl>
## 1     1    153     55  23.5
## 2     2    207     97  22.6
## 3     3    173     83  27.7
## 4     4    181     92  28.1
## 5     5    164    112  41.6
```

Textová tabulární data mohou mít jednotlivé proměnné oddělené různě. Existují však tři nejobvyklejší varianty, pro které má **readr** speciální varianty funkce `read_delim()`:

- `read_csv()` načítá klasický americký standard CSV, kde jsou jednotlivé proměnné oddělené čárkami a celou a desetinnou část čísla odděluje desetinná tečka,
- `read_csv2()` načítá "evropskou" variantu CSV, kde jsou jednotlivé proměnné oddělené středníky a desetinná místa v číslech odděluje čárka a
- `read_tsv()` načítá variantu formátu, kde jsou proměnné oddělené tabelátorem.

V těchto funkcích není potřeba nastavovat parametr `delim`; většinu ostatních parametrů mají tyto funkce stejné jako funkce `read_delim()`.

Protože dataset "bmi_data.csv" splňuje klasický formát CSV, můžeme jej pohodlněji načíst i pomocí funkce `read_csv()`:

```
bmi_data <- read_csv("data/reading_and_writing/bmi_data.csv")
```

```
## Rows: 5 Columns: 4

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): id, height, weight, bmi

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Všechny funkce `read_XXX()` berou první řádek datového souboru implicitně jako jména proměnných. Pokud soubor jména proměnných neobsahuje, je to třeba funkcím říct pomocí parametru `col_names`. Ten může nabývat jedné ze tří hodnot. Pokud má hodnotu `TRUE`, pak první řádek souboru chápe jako jména proměnných. Pokud má hodnotu `FALSE`, pak se předpokládá, že první řádek souboru jména proměnných neobsahuje; z řádku se načtou data a funkce sama jednotlivé proměnné pojmenuje `X1`, `X2` atd. Třetí možností je zadat jména proměnných ručně. V tomto případě musí být parametr `col_names` zadán jako vektor řetězců, který obsahuje jména jednotlivých proměnných. Na rozdíl od funkcí ze základního R, funkce z balíku **readr** jména proměnných nijak nemodifikují. V důsledku toho nemusejí být jména proměnných platnými názvy proměnných v R. To je však vždy možné opravit dvěma různými způsoby: Jednou možností je dodatečně změnit jména ručně pomocí funkcí `names()` nebo `setNames()`. Druhou možností je pomocí parametru `name_repair` nastavit funkci, která jména sloupců opraví automaticky. Jednou z možností je zde použít funkci `make_clean_names()` z balíku **janitor**. Tato funkce odstraní divné znaky, písmena s háčky, čárkami a jinými akcenty zbaví těchto akcentů, mezery nahradí podtržítky apod. Volání funkce `read_csv()` pak bude vypadat takto:

```
bmi_data <- read_csv("data/reading_and_writing/bmi_data.csv",
                     name_repair = janitor::make_clean_names)
```

Ukážeme si to na příkladu. Soubor "bmi_data_headless.csv" obsahuje stejná data jako soubor "bmi_data.csv", na prvním řádku však nejsou uvedena jména sloupců, ale dataset začíná přímo daty. Pokud bychom dataset načetli stejně jako výše, funkce `read_csv()` by první řádek použila k pojmenování proměnných (jména sloupců by navíc v našem případě byla syntakticky nepřípustná, takže bychom je museli používat spolu se zpětnými apostrofy):

```
read_csv("data/reading_and_writing/bmi_data_headless.csv")
```

```
## Rows: 4 Columns: 4

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): 1, 153, 55, 23.4952368747

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

## # A tibble: 4 x 4
##     `1` `153`  `55` `23.4952368747`
##   <dbl> <dbl> <dbl>           <dbl>
## 1     2   207    97            22.6
## 2     3   173    83            27.7
## 3     4   181    92            28.1
## 4     5   164   112            41.6
```

Tomu můžeme zabránit tak, že nastavíme parametr `col_names = FALSE`, takže se první řádek souboru považuje za data:

```
read_csv("data/reading_and_writing/bmi_data_headless.csv",
         col_names = FALSE)
```

```
## Rows: 5 Columns: 4
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): X1, X2, X3, X4
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 5 x 4
##      X1    X2    X3    X4
##   <dbl> <dbl> <dbl> <dbl>
## 1     1   153    55  23.5
## 2     2   207    97  22.6
## 3     3   173    83  27.7
## 4     4   181    92  28.1
## 5     5   164   112  41.6
```

V takovém případě se sloupce tabulky pojmenují `X1`, `X2` atd. Alternativně můžeme pomocí parametru `col_names` přímo zadat jména jednotlivých sloupců, např. jako `col_names = c("id", "výška", "váha", "bmi")` (první řádek souboru se bude opět považovat za data):

```
read_csv("data/reading_and_writing/bmi_data_headless.csv",
         col_names = c("id", "výška", "váha", "bmi"))
```

```
## Rows: 5 Columns: 4
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): id, výška, váha, bmi
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 5 x 4
##      id výška  váha   bmi
##   <dbl> <dbl> <dbl> <dbl>
## 1     1   153    55  23.5
## 2     2   207    97  22.6
## 3     3   173    83  27.7
## 4     4   181    92  28.1
## 5     5   164   112  41.6
```

Některé datové soubory obsahují v prvních řádcích nějaké balastní informace, např. údaje o pořízení dat, copyrightu, parametrech simulace apod. Tyto řádky je možné přeskočit pomocí parametru `skip`: např. `skip = 5` znamená, že se má přeskočit prvních pět řádků. Podobně některé soubory obsahují mezi skutečnými daty komentáře. Pokud jsou tyto komentáře uvozeny nějakým jasným znakem, můžeme je při načítání dat přeskočit pomocí parametru `comment`: např. `comment = "#"` znamená, že se přeskočí všechny řádky, které začínají "křížkem".

Opět si to ukážeme na příkladu. Soubor "bmi_data_comments.csv" opět obsahuje stejná data jako "bmi_data.csv", na prvních dvou řádcích však obsahuje nějaké informace o experimentu. Na disku vypadá soubor takto:

```
Experiment no. 747.
Top secret
id,height,weight,bmi
1,153,55,23.4952368747
2,207,97,22.6376344839
3,173,83,27.7322997761
4,181,92,28.0821708739
5,164,112,41.6418798334
```

Tento soubor musíme načíst s pomocí parametru `skip`; v opačném případě se věci opravdu pokazí (vyzkoušejte si to):

```
read_csv("data/reading_and_writing/bmi_data_comments.csv", skip = 2)
```

```
## Rows: 5 Columns: 4
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): id, height, weight, bmi
```

```
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 5 x 4
##      id height weight   bmi
##   <dbl>  <dbl>  <dbl> <dbl>
## 1     1    153     55  23.5
## 2     2    207     97  22.6
## 3     3    173     83  27.7
## 4     4    181     92  28.1
## 5     5    164    112  41.6
```

Soubor "bmi_data_comments2.csv" obsahuje opět stejná data; nyní jsou však dvě množiny pozorování nadepsány komentáři a jeden řádek má také vlastní komentář:

```
id,height,weight,bmi
# the first set of observations
1,153,55,23.4952368747
2,207,97,22.6376344839
3,173,83,27.7322997761  # suspicious
# the second set of observations
4,181,92,28.0821708739
5,164,112,41.6418798334
```

V tomto případě musíme zadat znak, kterým začínají komentáře, pomocí parametru `comment`. Funkce `read_XXX()` pak ignorují vše od tohoto znaku do konce řádku:

```
read_csv("data/reading_and_writing/bmi_data_comments2.csv", comment = "#")
```

```
## Rows: 5 Columns: 4

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): id, height, weight, bmi

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

## # A tibble: 5 x 4
##      id height weight   bmi
##   <dbl>  <dbl>  <dbl> <dbl>
## 1     1    153     55  23.5
## 2     2    207     97  22.6
## 3     3    173     83  27.7
## 4     4    181     92  28.1
## 5     5    164    112  41.6
```

Různé datové soubory kódují chybějící hodnoty různě. Standardně funkce `read_XXX()` očekávají, že chybějící hodnota je buď prázdná, nebo obsahuje řetězec "NA". To je však možné změnit parametrem `na`. Do něj uložíte vektor všech hodnot, které má funkce `read_XXX()` považovat za `NA`. Pokud např. váš datový soubor kóduje chybějící hodnoty pomocí tečky, nastavíte `na = "."`. Pokud navíc může být zadaná chybějící hodnota i jako prázdný řetězec nebo řetězec "NA", zadáte `na = c("", ".", "NA")`. (Funkce `read_XXX()` při načítání jednotlivých hodnot implicitně odstraní všechny úvodní a koncové mezery, takže našemu "tečkovému" pravidlu vyhoví jak řetězce ".", tak i " . ". Toto chování můžete změnit parametrem `trim_ws`.) Následující kód ukazuje příklad:

```
read_csv("data/reading_and_writing/bmi_data_na.csv", na = c("", ".", "NA"))
```

```
## Rows: 5 Columns: 4

## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (4): id, height, weight, bmi

##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.

## # A tibble: 5 x 4
##      id height weight   bmi
##   <dbl>  <dbl>  <dbl> <dbl>
## 1     1    153     55  23.5
## 2     2    207     97  22.6
## 3     3     NA     83  27.7
## 4     4    181     NA  28.1
## 5     5    164    112  41.6
```

Jak jste si asi všimli, funkce `read_XXX()` při načítání dat samy odhadnou, jaká data obsahují jednotlivé proměnné, a převedou je na daný datový typ. (Výsledná tabulka třídy *tibble* tyto typy vypisuje pod jmény proměnných.) Informace o tom, jaký typ funkce zvolila pro který sloupec, se vypíše při načítání: informace má tvar "dbl (4): id, height, weight, bmi" apod. V našem případě se všechny proměnné převedly na reálná čísla. Funkce `read_XXX()` odhadují typ proměnné na základě prvních 1 000 načtených řádků (toto číslo je možné změnit pomocí parametru `guess_max`). Nechat si datový typ odhadnout, je pohodlné, ale nepříliš bezpečné. Mnohem rozumnější je zadat datový typ jednotlivých sloupců ručně pomocí parametru `col_types`. Ten je možné zadat třemi způsoby. Zaprvé, hodnota `NULL` indikuje, že se typy proměnných mají odhadnout. Zadruhé, datové typy jednotlivých sloupců je možné zadat plnými jmény ve funkci `cols()`. A zatřetí, tato jména můžeme zkrátit do jednopísmenných zkratek zadaných v jednom řetězci. Seznam typů sloupců a jejich zkratek uvádí tabulka 11.1. Rozdíl mezi funkcemi `col_double()` a `col_number()` spočívá v tom, že `col_number()` umožňuje mít před a za vlastním číslem nějaké nečíselné znaky, které zahodí. To se hodí např. při načítání peněžních údajů, kde `col_number()` zvládne převést na číslo i řetězce jako `$15.30` nebo `753 Kč`.

Table 11.1: Seznam jmen funkcí pro určení datového typu načítané proměnné.

funkce	význam	zkratka	parametry
`col_logical()`	logická proměnná	"l"	
`col_integer()`	striktní celé číslo	"i"	
`col_double()`	striktní reálné číslo	"d"	
`col_number()`	flexibilní číslo	"n"	
`col_character()`	řetězec	"c"	
`col_date()`	datum (jen den bez hodin)	"D"	`format`
`col_datetime()`	datum (den i hodina)	"T"	`format`
`col_time()`	čas (jen hodiny, minuty a sekundy)	"t"	`format`
`col_factor()`	faktor	"f"	`levels`, `ordered`, `include_na`
`col_guess()`	typ odhadne R	"?"	
`col_skip()`	proměnná se přeskočí a nenačte	"_" nebo "-"	

Řekněme, že proměnnou `id` chceme mít uloženou jako typ *integer*, zatímco ostatní hodnoty jako reálná čísla. Toho můžeme dosáhnout ručně takto (v tomto případě R nevypíše informaci o sloupcích):

```
bmi_data <- read_csv("data/reading_and_writing/bmi_data.csv",
                     col_types = cols(id = col_integer(),
                                      height = col_double(),
                                      weight = col_double(),
                                      bmi = col_double()))
```

nebo stručněji takto:

```
bmi_data <- read_csv("data/reading_and_writing/bmi_data.csv",
                     col_types = "iddd")
```

Specifikací sloupců můžeme samozřejmě změnit i typ sloupců. Někdy např. můžeme chtít načíst všechny proměnné jako řetězce, a pak si je zkonvertovat sami. To můžeme udělat třemi způsoby: 1) sloupce můžeme nastavit pomocí plného volání funkcí `col_types = cols(id = col_character(), height = col_character(), weight = col_character(), bmi = col_character())`, 2) je můžeme nastavit pomocí zkratkových písmen jako `col_types = "cccc"` nebo 3) nastavíme implicitní typ sloupce pomocí speciálního parametru `.default` buď plným voláním funkce `col_types = cols(.default = col_character())` nebo pomocí zkratkového písmene:

```
read_csv("data/reading_and_writing/bmi_data.csv",
         col_types = cols(.default = "c"))
```

```
## # A tibble: 5 x 4
##   id    height weight bmi
##   <chr> <chr>  <chr>  <chr>
## 1 1     153    55     23.4952368747
## 2 2     207    97     22.6376344839
## 3 3     173    83     27.7322997761
## 4 4     181    92     28.0821708739
## 5 5     164    112    41.6418798334
```

Nejjednodušší způsob, jak explicitně zadat datové typy sloupců tabulky, je nechat funkci `read_XXX()` odhadnout datový typ sloupců, a pak specifikaci `cols()` zkopírovat a případně upravit. Po vlastním načtení je možné specifikaci získat i pomocí funkce `spec()`, jejímž jediným argumentem je načtená tabulka. Specifikaci je možné zjednodušit pomocí funkce `cols_condense()`, která nejčastěji používaný datový typ shrne do parametru `.default`:

```
cols_condense(spec(bmi_data))
```

```
## cols(
##   .default = col_double(),
##   id = col_integer()
## )
```

Pokud bychom chtěli vypsat rychlý přehled o tom, v jakém formátu se data načetla, můžeme na výsledek funkce `spec()` spustit funkci `summary()`. Výsledek bude ve stejném formátu, v jakém tuto specifikaci vypíší funkce `read_XXX()`:

```
summary(spec(bmi_data))
```

```
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## dbl (3): height, weight, bmi
## int (1): id
```

Některé funkce `col_XXX` umožňují zadat parametry. Funkce pro načítání data a času umožňují zadat formát, ve kterém je datum nebo čas uložen. Funkce pro načítání faktorů umožňují zadat platné úrovně faktoru, to, zda je faktor ordinální, a to, zda má být hodnota `NA` součástí úrovní faktoru. Úplný seznam parametrů uvádí tabulka 11.1. Pokud bychom např. chtěli načíst proměnnou `id` jako faktor se třemi úrovněmi (“1”, “2” a “3”), můžeme to udělat takto (funkce `read_XXX()` nikdy nic nepřevádí na faktory samy od sebe – pokud chcete proměnnou převést na faktor, musíte o to požádat):

```
bmi_data <- read_csv("data/reading_and_writing/bmi_data.csv",
                     col_types = cols(id = col_factor(levels = as.character(1:3)),
                                      height = col_integer(),
                                      weight = col_integer(),
                                      bmi = col_double()))
```

Funkce vypíše varování, protože proměnná `id` obsahuje i jiné než povolené úrovně, a nepovolené úrovně nahradí hodnotami `NA`. (Pokud bychom chtěli nechat funkci `read_XXX()` odhadnout úrovně faktoru z dat, museli bychom zadat parametr `levels = NULL`.)

Funkcím `read_XXX()` je možné sdělit i to, které proměnné z tabulky vyloučit. K tomu slouží funkce `col_skip()`. Řekněme, že bychom chtěli náši tabulku anonymizovat tím, že z dat odstraníme identifikátory `id`. Toho můžeme dosáhnout např. takto:

```
bmi_data <- read_csv("data/reading_and_writing/bmi_data.csv",
                     col_types = cols(id = col_skip(),
                                      height = col_integer(),
                                      weight = col_integer(),
                                      bmi = col_double()))
bmi_data
```

```
## # A tibble: 5 x 3
##   height weight   bmi
##    <int>  <int> <dbl>
## 1    153     55  23.5
## 2    207     97  22.6
## 3    173     83  27.7
## 4    181     92  28.1
## 5    164    112  41.6
```

Pokud bychom chtěli načíst jen některé proměnné, stačí nahradit funkci `cols()` funkcí `cols_only()` a v ní uvést pouze proměnné, které se mají načíst. Následující kód načte pouze proměnné `height` a `bmi`:

```
bmi_data <- read_csv("data/reading_and_writing/bmi_data.csv",
                     col_types = cols_only(height = col_integer(),
                                           bmi = col_double()))
bmi_data
```

```
## # A tibble: 5 x 2
##   height   bmi
##    <int> <dbl>
## 1    153  23.5
## 2    207  22.6
## 3    173  27.7
## 4    181  28.1
## 5    164  41.6
```

K výběru sloupců, které se mají načíst, je možné použít i sofistikovanější parametr `col_select`, která používá stejnou syntaxi jako funkce `select()` z balíku **dplyr**, viz oddíl 16.1.2. Vybrané sloupce je možné vyjmenovat (bez uvozovek) nebo zavolat pořadím sloupce:

```
read_csv("data/reading_and_writing/bmi_data.csv",
         col_types = "iddd",
         col_select = c(height, bmi))
```

```
## # A tibble: 5 x 2
##   height   bmi
##    <dbl> <dbl>
## 1    153  23.5
## 2    207  22.6
## 3    173  27.7
## 4    181  28.1
## 5    164  41.6
```

```
read_csv("data/reading_and_writing/bmi_data.csv",
         col_types = "iddd",
         col_select = c(2, 4))
```

```
## # A tibble: 5 x 2
##   height   bmi
##    <dbl> <dbl>
## 1    153  23.5
## 2    207  22.6
## 3    173  27.7
## 4    181  28.1
## 5    164  41.6
```

Pomocí symbolu `:` je možné vybrat souvisle sloupce od vybraného počátečního po vybraný koncový:

```
read_csv("data/reading_and_writing/bmi_data.csv",
         col_types = "iddd",
         col_select = height:bmi)
```

```
## # A tibble: 5 x 3
##   height weight   bmi
##    <dbl>  <dbl> <dbl>
## 1    153     55  23.5
## 2    207     97  22.6
## 3    173     83  27.7
## 4    181     92  28.1
## 5    164    112  41.6
```

Pomocí symbolu `-` je možné sloupce vypouštět:

```
read_csv("data/reading_and_writing/bmi_data.csv",
         col_types = "iddd",
         col_select = -c(height, bmi))
```

```
## # A tibble: 5 x 2
##      id weight
##   <int>  <dbl>
## 1     1     55
## 2     2     97
## 3     3     83
## 4     4     92
## 5     5    112
```

Speciální funkce `starts_with()`, `ends_with()`, `contains()`, `matches()` a `num_range()` umožňují vybírat sloupce podle části jejich názvu. Sloupce, jejichž název končí na "ht" je např. možné načíst takto:

```
read_csv("data/reading_and_writing/bmi_data.csv",
         col_types = "iddd",
         col_select = ends_with("ht"))
```

```
## # A tibble: 5 x 2
##   height weight
##    <dbl>  <dbl>
## 1    153     55
## 2    207     97
## 3    173     83
## 4    181     92
## 5    164    112
```

Tyto možnosti je samozřejmě možné i kombinovat:

```
read_csv("data/reading_and_writing/bmi_data.csv",
         col_types = "iddd",
         col_select = c(id, ends_with("ht")))
```

```
## # A tibble: 5 x 3
##      id height weight
##   <int>  <dbl>  <dbl>
## 1     1    153     55
## 2     2    207     97
## 3     3    173     83
## 4     4    181     92
## 5     5    164    112
```

Jak jsme viděli výše, pokud se při načítání jednotlivých sloupců tabulky něco pokazí, funkce `read_XXX()` o tom vypíší varování a příslušnou "buňku" tabulky doplní hodnotou `NA`. K tomu může dojít z různých důvodů: Datový typ, který jste zadali (nebo který R odhadlo na základě prvních 1 000 řádků) není dost obecný pro všechny řádky tabulky. První řádky např. obsahují jen celá čísla, zatímco pozdější řádky i reálná čísla. Faktor obsahuje i jiné hodnoty, než zadané platné úrovně atd. Pokud je varování hodně, zobrazí se jen několik prvních varování. Všechna varování je možné vypsat funkcí `problems()`, jejímž jediným argumentem je načtená tabulka.

Většinou je moudré ladit zadání typů sloupců ve specifikaci `cols()` tak dlouho, až R nevypisuje žádná varování. To je užitečné zejména v případě, že načítáte různé datové soubory se stejnou strukturou. Když se potom u některého souboru objeví varování, znamená to, že se v datech něco změnilo. Pokud chcete být opravdu důkladně varováni, že k tomu došlo, můžete použít funkci `stop_for_problems()`. Jejím jediným argumentem je načtená tabulka. Pokud při jeho načtení došlo k nějakým varováním, funkce `stop_for_problems()` zastaví běh skriptu.

Tabulární datové soubory jsou obyčejné textové soubory bez speciálního formátování. Proto celá řada prvků těchto souborů není standardizovaná. Jednotlivé datové soubory se mohou lišit kódováním, znakem použitým pro oddělení celých a desetinných čísel, znakem použitým pro oddělení tisíců, jmény měsíců a dnů v týdnu apod. Balík **readr** implicitně používá americkou konvenci: kódování je UTF-8, desetinná místa odděluje tečka, tisíce čárka a veškerá jména jsou anglická. Toto chování můžete změnit pomocí parametru `locale`, do kterého vložíte objekt vytvořený funkcí `locale()`. Její základní parametry uvádí tabulka 11.2.

Table 11.2: Seznam jmen funkcí pro určení datového typu načítané proměnné.

parametr	význam	impl. hodnota
`date_names`	řetězec pro jména datumů (česká "cs", slovenská "sk")	"en"
`date_format`	formát data	"%AD"
`time_format`	formát času	"%AT"
`decimal_mark`	znak pro oddělení desetinných míst	"."
`grouping_mark`	znak pro oddělení tisíců	","
`tz`	časová zóna	"UTC"

parametr	význam	impl. hodnota
`encoding`	kódování souboru	"UTF-8"
`asciify`	jestli soubor odstranil diakritiku ze jmen datumů	`FALSE`

Nastavení ve funkci `locale` si můžete vyzkoušet tak, že je necháte vypsat do konzoly. Typické české nastavení data vytvořená z LibreOffice na Linuxu s formátem data typu 31. 12. 2017 by vypadalo takto:

```
locale("cs",
       date_format = "%d.%*%m.%*%Y",
       decimal_mark = ",", grouping_mark = ".",
       tz = "Europe/Prague")
```

```
## <locale>
## Numbers:  123.456,78
## Formats:  %d.%*%m.%*%Y / %AT
## Timezone: Europe/Prague
## Encoding: UTF-8
## <date_names>
## Days:   neděle (ne), pondělí (po), úterý (út), středa (st), čtvrtek (čt), pátek
##         (pá), sobota (so)
## Months: ledna (led), února (úno), března (bře), dubna (dub), května (kvě),
##         června (čvn), července (čvc), srpna (srp), září (zář), října
##         (říj), listopadu (lis), prosince (pro)
## AM/PM:  dopoledne/odpoledne
```

Složitější nastavení `locale`, např. jak nastavit jména dnů a měsíců v nepodporovaném jazyce nebo jak zjistit časovou zónu, najdete v příslušné vinětě balíku **readr**.

Největší oříšek je v našich končinách zjistit, jaké kódování má daný soubor. K tomu může pomoci funkce `guess_encoding()`, jejímž jediným parametrem je jméno souboru, jehož kódování chceme odhadnout. Funkce vypíše seznam několika kódování a u nich pravděpodobnost, kterou tomuto kódování přikládá. Pokud funkci vyzkoušíme na našem testovacím souboru, dostaneme následující výsledek:

```
guess_encoding("data/reading_and_writing/bmi_data.csv")
```

```
## # A tibble: 1 x 2
##   encoding confidence
##   <chr>         <dbl>
## 1 ASCII             1
```

Protože náš soubor neobsahuje žádnou diakritiku, je si funkce `guess_encoding()` zcela jistá, že se jedná o ASCII soubor. (Protože ASCII je podmnožinou UTF-8, nemusíme kódování explicitně deklarovat.)

11.1.2 Načítání velkých souborů

Načítání velkých souborů funguje stejně jako načítání malých souborů – se dvěma specifiky: trvá déle a zabírá více operační paměti počítače. Balík **readr** zde pomáhá dvěma způsoby. Předně implicitně načítá data do paměti "lenivě". To znamená, že když načtete datový soubor, funkce `read_XXX()` ve skutečnosti jen projde strukturu datového souboru (zjistí počet řádků a sloupců, typy sloupců apod.), ale žádná data ve skutečnosti do paměti počítače nenačte. To udělá teprve ve chvíli, kdy o tato data skutečně požádáte, tj. pokusíte si je vypsat, něco s nimi spočítat apod. Přitom se načtou pouze ta data, o která požádáte. To např. znamená, že když po načtení souboru vypíšete jen několik prvních nebo posledních řádků pomocí funkce `head()` nebo `tail()`, načtou se pouze tyto řádky.

Líné načítání dat může někdy způsobit určité problémy. Načítaný soubor je např. otevřený do té doby, dokud není do paměti načtený celý. Ve Windows není možné otevřený soubor smazat. Některé operace mohou (aspoň v současné implementaci) vést k tomu, že se data načtou několikrát. Línému načítání dat je možné zabránit pomocí parametru `lazy = FALSE`.

Zadruhé, R dokáže pracovat pouze s proměnnými, které má v operační paměti. Pokud se pokusíte načíst data, která se do paměti počítače nevejdou, R spadne. Pokud tedy máte opravdu hodně velký datový soubor, je užitečné dopředu zjistit, kolik paměti zabere. Objem dat v paměti můžete přibližně odhadnout následujícím způsobem:

1. zjistíte, kolik řádků má vaše tabulka (v Linuxu to můžete udělat pomocí prográmku `wc`),
2. načtete prvních 1 000 řádků tabulky (počet řádků omezíte pomocí parametru `n_max`); přitom odladíte datové typy jednotlivých sloupců,
3. zjistíte, kolik paměti tabulka zabírá pomocí funkce `object.size()` a
4. vypočítáte, kolik paměti zabere celý datový soubor: vydělíte velikost vzorku dat tisícem a vynásobíte ji počtem řádků souboru.

Balík **readr** umožňuje do jisté míry omezení proměnných na operační paměť počítače obejít. K tomu slouží speciální funkce `read_XXX_chunked()` (ekvivalentem funkce `read_delim()` je funkce `read_delim_chunked()`). Tyto funkce čtou datový soubor po kusech. Nad každým kusem provedou nějakou agregační operaci (např. vyberou jen řádky, které splňují určitou podmínku, spočítají nějaké agregátní statistiky apod.) a do paměti ukládají jen výsledky těchto operací. Na detaily použití těchto funkcí se podívejte do jejich dokumentace.

Naštěstí pro vás, valná většina dat, se kterou budete v blízké budoucnosti pracovat, bude nejspíše tak malá, že nic z těchto triků nebudete muset používat. Například tabulka s milionem řádků a 20 sloupci, které všechny obsahují reálná čísla bude v paměti počítač zabírat 1 000 000 $\times$ 20 $\times$ 8 = 160 000 000 bytů, tj. asi jen 153 MB.

11.1.3 Ukládání dat do textových tabulárních delimitovaných souborů

K zapsání dat do tabulárních formátů slouží funkce `write_delim()`, `write_csv()`, `write_excel_csv()` a `write_tsv()`. Prvním parametrem všech těchto funkcí je vždy tabulka, který se má zapsat na disk, druhým je cesta, kam se mají data zapsat. Dále je možné nastavit pomocí parametru `na` znak pro uložení hodnoty `NA` a určit, zda se mají data připsat na konec existujícího souboru. Funkce `read_delim()` umožňuje navíc nastavit pomocí parametru `delim` znak, který bude oddělovat jednotlivé sloupce tabulky. Funkce `write_excel_csv()` přidá do souboru kromě běžného CSV i znak, podle kterého MS Excel pozná, že soubor je v kódování UTF-8.

Tyto funkce bohužel neumožňují nastavit `locale`, takže výsledkem je vždy soubor v kódování UTF-8 s desetinnými místy oddělenými pomocí tečky, tisící oddělenými pomocí čárky atd. Pokud potřebujete něco jiného. podívejte se do dokumentace na odpovídající funkce ze základního R (tj. např. funkci `write.csv()`).

11.1.4 Spojení (connections)

Když R čte nebo zapisuje data, nepracuje přímo se soubory, ale se "spojeními" (connections). Spojení zobecňuje myšlenku souboru – spojení může být lokální soubor, soubor komprimovaný algoritmem `gzip`, `bzip2` apod., URL a další.

Funkce `read_XXX()` vytvářejí spojení pro čtení dat automaticky. To znamená, že jméno souboru může obsahovat jak lokální soubor, tak URL. Internetový odkaz funkce pozná podle úvodního "http://", "https://", "ftp://" nebo "ftps://". V takovém případě vzdálený soubor automaticky stáhne. Funkce umí číst i některý typy komprimovaných souborů. Ty pozná podle koncovky ".gz", ".bz2", ."xz" nebo ".zip". Takový soubor funkce automaticky dekomprimuje.

Předpokládejme, že náš datový soubor je pro úsporu místa na disku komprimovaný pomocí programu `gzip`. Pak data načteme jednoduše takto:

```
bmi_data <- read_csv("data/reading_and_writing/bmi_data.csv.gz",
                     col_types = "iiid")
bmi_data
```

```
## # A tibble: 5 x 4
##      id height weight   bmi
##   <int>  <int>  <int> <dbl>
## 1     1    153     55  23.5
## 2     2    207     97  22.6
## 3     3    173     83  27.7
## 4     4    181     92  28.1
## 5     5    164    112  41.6
```

Podobně je možné načíst i data z internetu, stačí nahradit jméno souboru jeho URL. Řekněme, že v proměnné `estat_gdp_adr` máte uloženou cestu k jedné z datových tabulek Eurostatu (zde jedna z tabulek s hodnotami GDP: "https://ec.europa.eu/eurostat/estat-navtree-portlet-prod/BulkDownloadListing?file=data/namq_10_gdp.tsv.gz"). Pak můžete tato data načít do R takto:

```
gdp <- read_tsv(estat_gdp_adr, show_col_types = FALSE)
dim(gdp)
```

```
## [1] 51429   187
```

Na rozdíl od funkcí ze základního R umí funkce `write_XXX()` i přímo zapisovat do komprimovaných souborů bez toho, aby bylo potřeba *connection* explicitně deklarovat (např. pomocí funkce `gzfile()`). Stačí uvést jméno souboru ukončené odpovídající koncovkou. Pokud tedy chceme uložit naši tabulku do komprimovaného souboru, můžeme to provést např. takto:

```
write_csv(bmi_data, "moje_data.csv.gz")
```

Víc detailů ke spojením najdete např. v (Spector, 2008, s. 23–25) nebo (Peng, 2016, s. 33–35).

11.1.5 Načtení více souborů najednou

Někdy máme data se stejnou strukturou rozdělená do více dílčích souborů. To vzniká typicky v situaci, kdy data zapisuje nějaký senzor nebo robot. Funkce `read_XXX()` umožňují načíst takové soubory naráz a automaticky je spojit do jedné tabulky. Stačí místo cesty k souboru zadat vektor řetězců, který obsahuje cesty ke všem načítaným souborům:

```
files <- fs::dir_ls(path = "rawdata", glob = "data*.tsv")
rawdata <- readr::read_tsv(files, id = "path")
```

Funkce `dir_ls()` z balíku **fs** najde všechny soubory v adresáři `rawdata`, která odpovídají zadané masce, tj. např. soubory `data0001.tsv`, `data0002.tsv` atd. a absolutní cesty k těmto souborům vrátí jako vektor řetězců. Následně funkce `read_tsv()` všechny tyto soubory načte a spojí do jedné tabulky `rawdata`. Pokud je zadán parametr `id`, pak funkce k vlastním datům přidá i sloupec, do kterého uloží zadanou cestu k načítaným souborům. Jméno tohoto sloupce určí parametr `id`; zde se tedy bude jmenovat "path". Přidání tohoto sloupce je užitečné např. v případě, že vlastní data neobsahují všechny proměnné a chybějící proměnné jsou součástí jména datového souboru nebo cesty k němu.

11.1.6 Načtení tabulárních dat v RStudiu

Při interaktivní práci je možné využít toho, že RStudio umí načíst tabulární textová data. V záložce `Environment` klikněte na `Import Dataset` a zvolte `From Text (readr)...`, viz levá část obrázku 11.1. Nástroj odhadne většinu potřebných parametrů a zbytek (včetně `locale`) je možné konfigurovat, viz pravá část obrázku 11.1. Příjemné je, že nástroj vygeneruje i potřebný kód, který můžete překopírovat do svého skriptu. Nástroj vám tedy umožní vizuálně odladit základní nastavení načítání dat, detaily můžete doladit později ve skriptu.

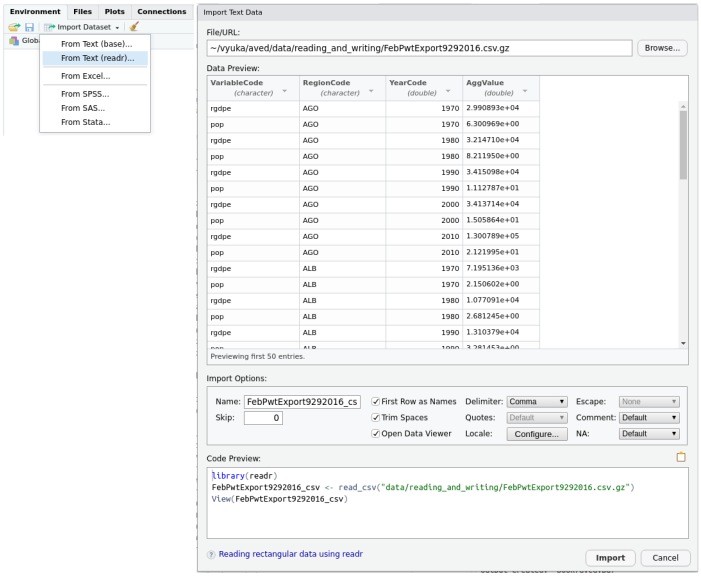


Figure 11.1: Načtení dat z tabulárního formátu v RStudiu.

11.2 Další textové formáty

Balík **readr** umožňuje číst data i z dalších typů textových formátů. Nejjednodušším případem jsou tabulární formáty, které nejsou delimitované, tj. jednotlivé sloupce nejsou oddělené jasně definovaným znakem. Sem patří zejména dva typy formátů. V prvním jsou jednotlivé sloupce oddělené mezerami (nebo jinými "bílými znaky", např. tabelátory). V ideálním případě jsou mají všechny sloupce díky bílým znakům právě stejnou šířku. K načtení těchto formátů slouží funkce `read_table()`. Druhou možností jsou formáty s pevnou šířkou sloupců. K jejich načtení můžete použít funkci `read_fwf()`. Všechny tyto funkce mají syntaxi velmi podobnou funkcím na čtení delimitovaných formátů.

Soubor "bmi_data.txt" obsahuje stejná data jako výše, ovšem tentokrát jsou oddělená jen bílými místy, v našem případě mezerami:

```
id height weight           bmi
 1    153     55 23.4952368747
 2    207     97 22.6376344839
 3    173     83 27.7322997761
 4    181     92 28.0821708739
 5    164    112 41.6418798334
```

Použít funkce `read_delim()` a její speciální varianty není možné, protože nemáme jasný oddělující znak (dokonce ani mezera by nefungovala, vyzkoušejte si to a vysvětlete, proč). Naštěstí pro nás je možné tato data snadno načíst pomocí funkce `read_table()`:

```
read_table("data/reading_and_writing/bmi_data.txt",
           col_types = "iiid")
```

```
## # A tibble: 5 x 4
##      id height weight   bmi
##   <int>  <int>  <int> <dbl>
## 1     1    153     55  23.5
## 2     2    207     97  22.6
## 3     3    173     83  27.7
## 4     4    181     92  28.1
## 5     5    164    112  41.6
```

Naše tabulka má všechny sloupce stejně široké, můžeme tedy použít i funkci `read_fwf()`. V tomto případě je však třeba specifikovat hranice jednotlivých sloupců. K tomu slouží pomocné funkce `fwf_empty()`, `fwf_widths`, `fwf_positions()` a `fwf_cols()`. Nejjednodušší z nich je funkce `fwf_widths()`, do které se zadává šířka jednotlivých sloupců. Formát FWF také nepodporuje jména sloupců, takže musíme přeskočit první řádek, který jména obsahuje, a jména sloupců zadat ručně ve funkci `fwf_widths()`:

```
read_fwf("data/reading_and_writing/bmi_data.txt",
         fwf_widths(c(2, 7, 7, 13), c("id", "výška", "váha", "bmi")),
         col_types = "iiid",
         skip = 1)
```

```
## # A tibble: 5 x 4
##      id výška  váha   bmi
##   <int> <int> <int> <dbl>
## 1     1   153    55  23.5
## 2     2   207    97  22.6
## 3     3   173    83  27.7
## 4     4   181    92  28.1
## 5     5   164   112  41.6
```

Na další detaily se podívejte do dokumentace. Balík **readr** neobsahuje žádné funkce pro výpis do těchto formátů.

V případě, že textová data nejsou formátována ani jedním z výše popsaných formátů, je možné načíst textový soubor jako řetězec a data si zpracovat sami pomocí funkcí, které se naučíte v kapitole 13. K načtení textového souboru po jednotlivých řádcích slouží funkce `read_lines()` a `read_lines_raw()`, které načtou jednotlivé řádky ze souboru a uloží je do vektoru řetězců, kde každý prvek odpovídá jednomu řádku. Obě funkce umožní zadat, kolik řádků načíst (parametr `n_max`) a kolik počátečních řádků přeskočit (parametr `skip`). Funkce `read_lines()` umožňuje zadat i kódování textu pomocí `locale` způsobem popsaným výše. Náš testovací datový soubor bychom načetli takto:

```
read_lines("data/reading_and_writing/bmi_data.csv")
```

```
## [1] "id,height,weight,bmi"    "1,153,55,23.4952368747"
## [3] "2,207,97,22.6376344839"  "3,173,83,27.7322997761"
## [5] "4,181,92,28.0821708739"  "5,164,112,41.6418798334"
```

Funkce `write_lines()` umožňuje uložit vektor řetězců do souboru. Na detaily se podívejte do dokumentace.

Pokud není způsob, jakým jsou v textovém souboru data rozdělena do řádků, užitečný, je možné načíst celý soubor do jednoho řetězce. K tomu slouží funkce `read_file()` a `read_file_raw()`. Funkce `write_file()` zapíše řetězec do souboru. Detaily použití opět najdete v dokumentaci.

11.3 Nativní R-kové binární soubory

Textové tabulární datové formáty jsou výborné při předávání dat mezi lidmi. Při vlastním zpracování dat je však výhodnější použít vlastní binární datový formát R. Jeho výhodou je, že data zabírají na disku méně místa, načítají se rychleji, mohou jakékoli datové typy včetně složitých objektů, obsahovat metadata včetně atributů a jde v nich ukládat i jiná data, než jsou tabulky (např. seznamy, pole, různé objekty apod.).

K uložení jednotlivých proměnných slouží funkce `save()`. Nejdříve se uvedou jména všech proměnných, které se mají uložit, a pak cesta k souboru, kam se mají uložit (další parametry jsou popsány v dokumentaci). Cestu je nutné zadat pojmenovaných parametrem `file`. Data se do těchto souborů obvykle ukládají s koncovkami `.RData` nebo `.rda`.

```
save(bmi_data, file = "bmi_data.RData")
```

Pokud je proměnných více, oddělí se čárkou:

```
save(var1, var2, var3, var4, file = "somefile.RData")
```

Někdy chcete uložit všechny proměnné, které máte v paměti R. K tomu slouží funkce `save.image()` (detaily viz dokumentace).

Data uložená pomocí funkcí `save()` nebo `save.image()` načteme do pracovního prostředí R pomocí funkce `load()`. Funkce načte všechny proměnné obsažené v daném datovém soubory včetně metadat, tj. není možné si vybrat jen jednotlivé proměnné uložené v souboru. Proměnným zůstanou jejich původní jména, tj. není možné načíst vybranou proměnnou do nové proměnné. Funkce data načte do pracovního prostředí, tj. nevrací je jako hodnotu funkce. Jediným důležitým parametrem funkce `load()` je cesta k datovému souboru:

```
load("bmi_data.RData")
```

Pokud chcete do nativního binárního souboru uložit jen obsah jedné proměnné a ten pak opět načíst do nové proměnné, můžete použít dvojici funkcí `saveRDS()` a `readRDS()`:

```
saveRDS(bmi_data, file = "bmi_data.RData")
bmi <- readRDS("bmi_data.RData")
```

11.4 Načítání dat z balíků

Mnoho balíků v R obsahuje nějaká data. K načtení těchto dat slouží funkce `data()`. Stejně jako funkce `load()` ani tato funkce data nevrací, nýbrž je jako svůj vedlejší efekt načte přímo do pracovního prostředí R:

```
library(ggplot2)
data("diamonds")
head(diamonds)
```

```
## # A tibble: 6 x 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
## 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
```

Funkce `data()` může vypsat i seznam dat obsažených v daném balíku:

```
data(package = "ggplot2")
```

Funkce `data()` umožňuje i načíst data bez načtení balíku. K tomu stačí zadat jméno balíku:

```
data("economics", package = "ggplot2")
head(economics)
```

```
## # A tibble: 6 x 6
##   date         pce    pop psavert uempmed unemploy
##   <date>     <dbl>  <dbl>   <dbl>   <dbl>    <dbl>
## 1 1967-07-01  507. 198712    12.6     4.5     2944
## 2 1967-08-01  510. 198911    12.6     4.7     2945
## 3 1967-09-01  516. 199113    11.9     4.6     2958
## 4 1967-10-01  512. 199311    12.9     4.9     3143
## 5 1967-11-01  517. 199498    12.8     4.7     3066
## 6 1967-12-01  525. 199657    11.8     4.8     3018
```

Některé balíky zpřístupní svá data hned při svém načtení, takže není třeba je zpřístupňovat pomocí funkce `data()`. Tato data jsou obvykle zpřístupněna "líně": implicitně se data nenačtou do paměti, ale pouze do vyhledávací cesty. Do paměti počítače se načtou až ve chvíli, kdy je poprvé jakkoli použijete.

Data v balících bývají obvykle dokumentovaná stejným způsobem, jako funkce. Nápověda je tedy dostupná pomocí funkce `help()` i pomocí otazníku. Po načtení dat `economics` můžete dokumentaci zobrazit např. takto: `?economics`.

11.5 Načítání dat z MS Excelu

Někteří lidé (a některé statistické úřady) dávají data k dispozici jako soubory uložené z Microsoft Excelu. Používat tato data může být poněkud ošidné, protože datové "buňky" jsou často obalené různým balastem, který je potřeba odstranit. Nejbezpečnějším způsobem načtení dat z Excelu je tak data v Excelu ručně upravit, následně vyexportovat do CSV souboru a ten pak načíst způsobem popsaným v oddíle 11.1. Tak máte největší kontrolu nad tím, co se děje.

Data z Excelu je však v R možné načíst i přímo pomocí balíku **readxl** (umí načíst jak soubory `.xls`, tak `.xlsx`). Balík poskytuje dvě hlavní funkce: `excel_sheets()` a `read_excel()`. Funkce `excel_sheets()` má jediný argument (jméno souboru) a vypíše seznam listů, které Excelový soubor obsahuje.

Funkce `read_excel()` načte jeden list z Excelového souboru a vrátí jej jako tabulku třídy *tibble*. Jejím prvním parametrem je opět název excelového souboru. Druhým parametrem (`sheet`) určíte, který list se má načíst (implicitně první). List je možné zadat jménem (jako řetězec), nebo pozicí (jako celé číslo). Parametr `range` určuje oblast, která se má z excelového listu načíst (implicitně celý list). Nejjednodušší způsob určení oblasti je pomocí excelových rozsahů. Pokud např. chceme načíst buňky od B5 po E17, nastavíme `range = "B5:E17"`. Sofistikovanější výběry jsou popsané v dokumentaci k balíku **cellranger**. Pokud rozsah zabere i oblast "prázdných buněk", pak jsou chybějící hodnoty označeny jako `NA`.

Funkce `read_excel()` se v ostatních ohledech chová podobně jako funkce `read_csv()` z balíku **readr**. První řádek výběru interpretuje funkce jako názvy sloupců. Pokud to nevyhovuje, je možné toto chování modifikovat pomocí parametru `col_names`, jehož specifikace je stejná jako u funkce `read_csv()`.

Typy jednotlivých sloupců funkce `read_excel()` odhaduje z prvních 1 000 řádků dat. Pomocí parametru `col_types` je však možné jednotlivé sloupce zadat, a to jako vektor následujících řetězců: "skip", "guess", "logical", "numeric", "date", "text" a "list". První možnost sloupec přeskočí, druhá odhadne typ. Typ "list" vytvoří sloupec typu seznam (normálně je každý sloupec tabulky atomický vektor), což umožní, aby každá buňka sloupce mohla mít vlastní datový typ zjištěný z dat. Ostatní typy mají očividný význam. Pokud je zadán právě jeden typ, R jej recykluje a použije pro všechny sloupce.

Funkce umožňuje také přeskočit několik prvních řádků výběru pomocí parametru `skip`, načíst maximálně určitý počet řádků pomocí parametru `n_max` a zadat, jaká hodnota z excelového souboru se převede na hodnotu `NA` v R pomocí parametru `na`. Na další parametry funkce se podívejte do dokumentace.

Následující kód ukazuje příklad, jak funkce použít:

```
library(readxl)
excel_sheets("data/reading_and_writing/FebPwtExport9292016.xlsx")
```

```
## [1] "Preface"   "Data"      "Variables" "Regions"
```

```
dt <- read_excel("data/reading_and_writing/FebPwtExport9292016.xlsx",
                 sheet = "Data",
                 col_types = c("text", "text", "numeric", "numeric"))
head(dt)
```

```
## # A tibble: 6 x 4
##   VariableCode RegionCode YearCode AggValue
##   <chr>        <chr>         <dbl>    <dbl>
## 1 rgdpe        AGO            1970 29909.
## 2 pop          AGO            1970     6.30
## 3 rgdpe        AGO            1980 32147.
## 4 pop          AGO            1980     8.21
## 5 rgdpe        AGO            1990 34151.
## 6 pop          AGO            1990    11.1
```

Pokud je balík **readxl** nainstalován, pak RStudio umožňuje načíst excelové soubory pomocí interaktivního nástroje `Import Dataset` dostupného v panelu `Environment`, viz levá část obrázku 11.2. Po jeho spuštění zvolte `From Excel...`. Nástroj umožňuje dokonce vybrat list a rozsah dat, nastavit jména sloupců a několik dalších věcí, viz pravá část obrázku 11.2. Nástroj opět vygeneruje kód, který je možné zkopírovat a vložit do skriptu.

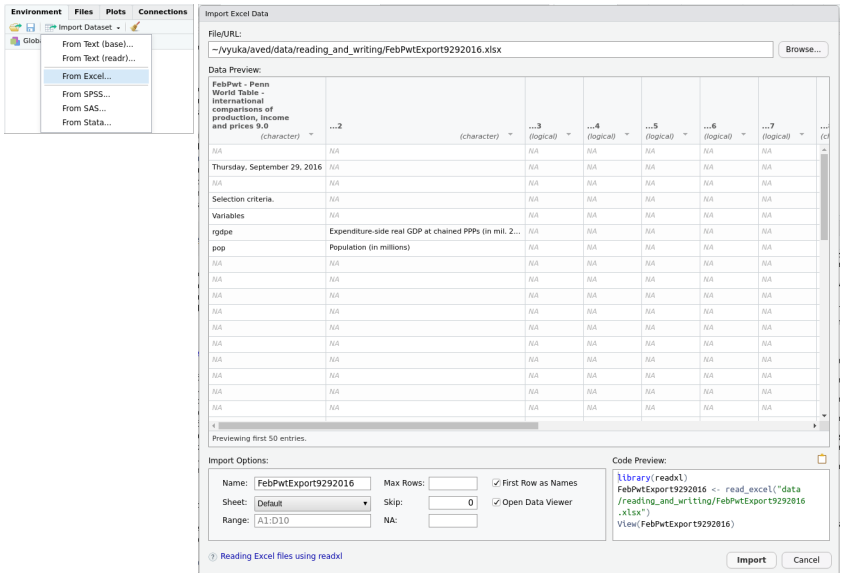


Figure 11.2: Načtení dat z Excelu v RStudiu.

Balík **readxl** zatím neumožňuje data do excelových souborů zapisovat. Pokud to potřebujete, podívejte se na balík **openxlsx**.

11.6 Data z jichých statistických programů

Někdy jsou data distribuovaná v nativním formátu některého statistického softwaru. K jejich načtení je možné použít balíky **haven** a **foregin**. Balík **haven** slouží k načítání a ukládání souborů v nativních formátech programů SPSS, SAS a Stata. Balík **foreign** umí načítat některé další datové formáty (a do vybraných formátů i zapisovat). Pokud nějaký formát umí číst oba balíky, doporučuji použít balík **haven**, na který se zde zaměříme. Jak se používá balík **foreign**, najdete v jeho dokumentaci.

K načítání dat poskytuje balík **haven** následující funkce: funkce `read_dta()` a `read_stata()` čtou soubory Stata DTA, funkce `read_por()`, `read_sav()` a `read_spss()` čtou soubory SPSS POR a SAV a funkce

`read_xpt()` a `read_sas()` čtou soubory SAS. Všechny tyto funkce vracejí načtená data jako tabulku třídy *tibble*. Funkce `write_dta`, `write_sas`, `write_sav` a `write_xpt()` zapisují odpovídající formáty. Před jejich použitím je potřeba balík **haven** načíst; užitečné je zároveň načíst i balík **labelled**:

```
library(haven)
library(labelled)
```

Data pak načteme očekávávaným způsobem, kdy jméno souboru je prvním parametrem funkce `read_xxx()`:

```
dt <- read_spss("nama_10_gdp.sav")
```

Přitom mohou nastat dva problémy. První spočívá v tom, že velké množství dat uložených v těchto formátech zřejmě nebylo vytvořených přímo v programech Stata, SPSS nebo SAS, nýbrž je vytvořily nějaké konvertory, které braly data z databáze. Tyto konvertory často daný formát zcela nedodržují, takže vlastní software takto vytvořená data načte, ale balík **haven** si s nimi neporadí. V tom případě je potřeba načíst data do originálního softwaru (někdy stačí jeho svobodná obdoba, např. pro software SPSS program PSPP) a data uložit do formátu CSV. Někdy také pomůže data znovu uložit do nativního formátu daného softwaru – balík **haven** si s nimi potom poradí. Alternativně je možné vyzkoušet balík **foreign**. V některých případech uspěje tam, kde **haven** selhal.

Druhý problém spočívá v tom, že všechny tři výše zmíněné programy používají poněkud jiné datové struktury než R. Zejména se to týká "popisků dat" a kódování chybějících hodnot. Detaily rozdílů najdete ve vinětě balíku **haven** s názvem "Conversion semantics". Zde se zaměříme pouze na popisky hodnot ve vektorech. Stata, SPSS a další statistické programy umožňují "přibalit" k proměnné (sloupci v tabulce) nějaké dodatečné informace. Ukázkovým příkladem může být uchovávání dotazníkových dat. Předpokládejme, že respondenti odpovídali na následující otázku:

> Máte rádi zpěv ve sprše?
>
> 1. Určitě ano
> 2. Spíše ano
> 3. Spíše ne
> 4. Určitě ne

Samotná odpověď je typicky uložena jako číslo kódující odpověď nebo druh chybějící odpovědi (např. "irelevantní", nebo "odmítl odpovědět" a podobně). Význam těchto číselných kódů může být popsán právě v těchto "přibalených" informacích. Ty často obsahují i samotné znění otázky. Balík **haven** převede při načítání dat sloupce s přibalenými informacemi do třídy **labelled**. Balík zároveň implementuje i základní funkce a metody pro práci s daty této třídy; další funkce jsou obsaženy v balíku **labelled**.

Vektor třídy *labelled* je možné vytvořit i uměle pomocí `labelled()` z balíku **haven**:

```
lvector <- labelled(c(1,1,2,3,1,4,-99),
                    c("Určitě ano" = 1,
                      "Spíše ano" = 2,
                      "Spíše ne" = 3,
                      "Určitě ne" = 4,
                      "Odmítl odpovědět" = -99)
                    )
class(lvector)
```

```
## [1] "haven_labelled" "vctrs_vctr"     "double"
```

Po načtení jsou vektory této třídy součástí načtené tabulky. Zde si pro jednoduchost takovou tabulku vytvoříme ručně:

```
ltable <- tibble::tibble(lvector)
print(ltable)
```

```
## # A tibble: 7 x 1
##                  lvector
##                 <dbl+lbl>
## 1   1 [Určitě ano]
## 2   1 [Určitě ano]
## 3   2 [Spíše ano]
## 4   3 [Spíše ne]
## 5   1 [Určitě ano]
## 6   4 [Určitě ne]
## 7 -99 [Odmítl odpovědět]
```

Číst informace z "labels" umožňují funkce `val_labels()` a `var_label()` z balíku **labelled**. Funkce `var_labels()` vrací "labels" přiřazené proměnné – tj. celému sloupci (typicky znění otázky). `val_labels()` vrací "labels" pro kódovací hodnoty v tabulce:

```
val_labels(ltable)
```

```
## $lvector
##       Určitě ano        Spíše ano         Spíše ne        Určitě ne
##                1                2                3                4
## Odmítl odpovědět
##              -99
```

Pro třídu `labelled` je v R dostupné jen velmi málo metod. Proto bývá obvyklým krokem jejich konverze do jiného datové typu. Při této operaci je užitečné být opatrný. Pouhé odstranění "lables" totiž za sebou zanechá vektory celých čísel nebo tajemně kódovaných řetězců. K touto účelu slouží funkce `zap_labels()`:

```
zap_labels(ltable)
```

```
## # A tibble: 7 x 1
##   lvector
##     <dbl>
## 1       1
## 2       1
## 3       2
## 4       3
## 5       1
## 6       4
## 7     -99
```

Lepším způsobem je konverze kódovacích hodnot na faktor. K tomu slouží funkce `as_factor()`, která umí převést jeden vektor i celou tabulku. Alternativně je možné popisky převést na faktor i pomocí funkce `to_factor()` nebo na vektor řetězců pomocí funkce `to_character()` z balíku **labelled**.

```
as_factor(ltable)
```

```
## # A tibble: 7 x 1
##   lvector
##   <fct>
```

```
## 1 Určitě ano
## 2 Určitě ano
## 3 Spíše ano
## 4 Spíše ne
## 5 Určitě ano
## 6 Určitě ne
## 7 Odmítl odpovědět
```

Podobným způsobem je možné se vyrovnat i s různě kódovanými chybějícími hodnotami, viz vinněta “Conversion semantics”.

Pokud máte balík **haven** nainstalovaný, můžete v RStudiu načítat data z SPSS, SASu a Staty i interaktivně, když v záložce `Import Dataset` v tabu `Environment` zvolíte `From SPSS...`, `From SAS...` nebo `From Stata...`, viz levá strana obrázku 11.3. Nástroj nemá žádná zvláštní nastavení, opět však vygeneruje kód, který můžete vložit do svého skriptu, viz , viz pravá strana obrázku 11.3.

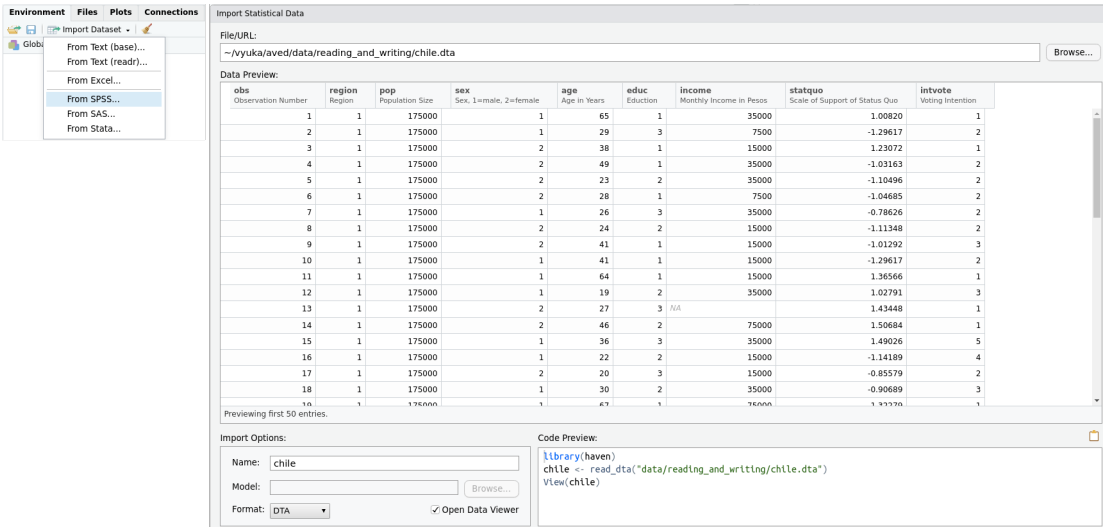


Figure 11.3: Načtení dat z SPSS, SAS a Stata v RStudiu.

11.7 Rychlá cesta k datům: balík rio

Pokud chcete rychle načíst data z datového souboru (lokálního nebo na webu, nekomprimovaného nebo komprimovaného) a nepotřebujete mít důkladnou kontrolu nad zpracováním dat, můžete použít balík **rio**. Tento balík představuje wrapper nad balíky, které načítají data včetně balíků **readxl**, **haven**, **foreign** a mnoha dalších, takže umí načíst všechna data, která umí načíst funkce implementované v těchto balících. Balík implementuje především dvě funkce: `import()` a `export()`. Funkce `import()` načte data z datového souboru a uloží je do proměnné třídy *data.frame*. Funkce `export()` uloží data do datového souboru. Typ datového souboru odhadují obě funkce z koncovky dat. Načítání a ukládání některých méně obvyklých formátů může vyžadovat doinstalování potřebných balíků. Seznam všech podporovaných formátů obsahuje vinněta k balíku.

Funkce `import()` vyžaduje nutně jen jeden parametr: cestu k načítanému souboru. Pokud by se formát dat odhadl z koncovky souboru špatně, umožňuje parametr `format` zadat typ dat ručně. Parametr `setclass` umožňuje změnit datovou strukturu, kterou funkce vrátí. Implicitně je to *data.frame*, ale povolený je mimo jiné i *tibble*. Další parametry najdete v dokumentaci k funkci. Funkce `export()` vyžaduje nutně jméno ukládané tabulky a jméno souboru včetně cesty. Další parametry opět najdete v dokumentaci.

Několik příkladů použití:

```
library(rio)
bmi <- import("data/reading_and_writing/bmi_data.csv.gz")
```

```
chile <- import("data/reading_and_writing/chile.dta")
xls <- import("data/reading_and_writing/FebPwtExport9292016.xlsx")
export(bmi, "test.csv")
export(bmi, "test.sav")
```

11.8 Kontrola načtených dat

Když načtete data ze souboru do paměti počítače, je obvykle moudré zkontrolovat, že se načetla všechna data, že se načetla správně a že znamenají to, co si myslíte. Vyplatí se projít minimálně těchto několik kroků:

1. Zkontrolujte, že má tabulka správný počet řádků (`nrow()`) a sloupců (`ncol()`), tj. že se načetlo vše a nenačetl se nějaký "odpad" uložený na začátku nebo konci datového souboru.
2. Podívejte se na začátek (`head()`) a konec (`tail()`) tabulky; to opět pomůže zkontrolovat, zda se nenačetl nějaký zmatek na začátku nebo konci souboru a že vše vypadá tak, jak má.
3. Zkontrolujte strukturu tabulky (v tabulce třídy *tibble* vypsané nahoře, jinak pomocí funkce `str()`) – jména sloupců, jejich typy a hodnoty.
4. Podívejte se na souhrnné statistiky dat (`summary()`): Jaké jsou hodnoty proměnných, zda dávají smysl a zda jsou správně velké. Porovnejte hodnoty s tím, co víte odjinud. Zkontrolujte také, kde hodnoty chybí apod.

12 Práce se soubory a adresáři

TODO

Part III

Pokročilé transformace dat

13 Práce s řetězci

Analýza dat nezahrnuje jen práci s čísly, ale i s řetězci (texty). Když pomineme textovou analýzu jako takovou, velmi často jsou data zabalena v textovém balastu a je třeba je z něj extrahovat. R má v základním balíku **base** mnoho užitečných funkcí pro práci s řetězci. Tyto funkce však mají často složité a vzájemně nekonzistenční rozhraní. Proto se zde místo nich podíváme na funkce implementované v balíku **stringr** (a také jednu funkci z balíku **glue**), který výrazně zjednodušuje práci s řetězci a stále pokrývá velkou většinu toho, co člověk potřebuje. (Balík **stringr** je uživatelsky přívětivý wrapper nad funkcemi balíku **stringi**; proto často vypisuje chyby ze **stringi** a stejně tak část dokumentace je třeba hledat ve **stringi**.) Pro práci s touto kapitolou je tedy nezbytné načíst balík **stringr** do paměti počítače:

```
library(stringr)
```

Jména všech funkcí z balíku **stringr** začínají `str_`.

V této kapitole se naučíte

- základy práce se řetězci
- jak zjistit délku řetězce,
- jak řetězce spojovat, duplikovat a zalamovat,
- jak řetězce setřídit,
- jak nahrazovat části řetězců,
- jak pracovat s regulárními výrazy a
- jak měnit chování regulárních výrazů

a mnoho dalšího.

13.1 Základy: řetězce v R

R ukládá řetězce ve vektorech datového typu *character*. Řetězec se zadává mezi dvěma uvozovkami nebo dvěma apostrofy. Uvozovky a apostrofy nejde míchat, ale je možné uzavřít apostrof mezi uvozovky nebo naopak:

```
"Petr řekl: 'Už tě nemiluji.'"
'Agáta odpověděla: "Koho to zajímá?"'
```

Kromě uvozovek a apostrofů má v řetězcích zvláštní význam i zpětné lomítko \. To slouží ke dvěma účelům: 1) uvozuje speciální znaky jako např. `"\n"` (konec řádku), `"\t"` (tabelátor) apod., a 2) zbavuje speciální znaky jejich zvláštního významu. To provedeme tak, že zvláštnímu znaku předřadíme zpětné lomítko. Tímto způsobem je možné zbavit zvláštního významu i uvozovky, apostrofy a zpětná lomítka a uvést je uvnitř řetězce:

```
s <- "Petr na to odvětil: \"Je to proto, že vypadáš jako \\.\""
```

Podobně můžete v řetězcích zadat i znaky Unicode pomocí `\u` a příslušného čísla. Znak velká omega (Ω) je např. možné zadat jako `\u03a9`. (Tabulku Unicode najdete na adrese https://unicode-table.com/.)

Funkce `print()`, která se používá při implicitním vypsání obsahu proměnné, vypisuje na začátku i na konci řetězců uvozovky a všechny speciální znaky vypisuje se zpětnými lomítky. Pokud chcete vypsat řetězec "normálně", můžete použít funkci `cat()`:

```
print(s)
```

```
## [1] "Petr na to odvětil: \"Je to proto, že vypadáš jako \\.\""
```

```
cat(s)
```

```
## Petr na to odvětil: "Je to proto, že vypadáš jako \."
```

Kromě funkcí `print()` a `cat()` existuje mnoho dalších funkcí pro výpis řetězců. Nejvýznamnější z nich uvádí tabulka 13.1. Na detaily použití těchto funkcí se podívejte do dokumentace.

Table 13.1: Vybrané funkce pro výpis řetězců.

funkce	účel
`print()`	generická funkce pro tisk objektů
`noquote()`	tisk bez uvozovek
`cat()`	tisk obsahu řetězců; spojuje více řetězců
`format()`	formátování proměnných před tiskem
`toString()`	konverze na řetězec
`sprintf()`	formátování řetězců ve stylu jazyka C

V R je potřeba rozlišovat mezi prázdným řetězcem (tj. řetězcem `""`) a prázdným vektorem řetězců:

```
s1 <- ""            # vektor délky 1 obsahující prázdný řetězec
length(s1)
```

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

```
s2 <- character(0)  # vektor typu character, který má nulovou délku,
length(s2)          # tj. neobsahuje žádné řetězce
```

```
## [1] 0
```

```
s3 <- character(5)  # vektor pěti prázdných řetězců
length(s3)
```

```
## [1] 5
```

```
s3
```

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

Všimněte si, že funkce `length()` vrací délku vektoru, ne délku řetězce, který vektor snad obsahuje.

R má dva speciální vektory, které obsahují písmena anglické abecedy:

```
letters  # malá písmena anglické abecedy
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```
LETTERS  # velká písmena anglické abecedy
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S"
## [20] "T" "U" "V" "W" "X" "Y" "Z"
```

Pokud potřebujete porovnat, zda jsou dva řetězce stejné, můžete použít operátor == i funkci `all.equal()`. (Samozřejmě můžete použít i >, < apod. pro zjištění, který řetězec by by byl umístěn ve slovníku dříve; později v této kapitole však uvidíte funkce, které vám dají nad řazením řetězců větší kontrolu.)

```
"abc" == "abc"
```

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

```
"abc" == "ABC"
```

```
## [1] FALSE
```

```
all.equal("abc", "abc", "abc")
```

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

Poznámka: Počítač dokáže zpracovávat jen sekvence jedniček a nul. Aby bylo jasné, jak binární čísla interpretovat jako text, musí být jasné kódování tohoto textu. Každý řetězec může mít v R jiné kódování. Nativně umí R dvě kódování: ISO Latin 1 a UTF-8. Kromě toho zvládá i nativní kódování operačního systému počítače, na kterém běží. Většinou se o kódování nemusíte nijak starat. Problém může nastat jen v případě přenosu dat mezi různými operačními systémy (v rámci Windows i mezi různými lokalizacemi). Naštěstí umí R změnit kódování i převést text z jednoho kódování do jiného. Pokud potřebujete data přenášet mezi počítači, zvažte použití standardního kódování UTF-8. Základní dokumentaci ke kódování poskytuje `help("Encoding")`. Ke konverzi kódování řetězců je možné použít funkci `iconv()` z balíku **base** nebo funkci `str_conv()` z balíku **stringr**. Detaily použití obou funkcí najdete v dokumentaci.

13.2 Základní operace

Nejdříve se podíváme na základní operace s řetězci: na to, jak řetězce spojovat, třídit, replikovat, nahrazovat, srovnávat a zjišťovat jejich délku. Ve zbytku kapitoly se pak budeme věnovat pokročilejší práci s řetězci pomocí regulárních výrazů.

13.2.1 Zjištění délky řetězce

Často potřebujeme zjistit délku jednotlivých řetězců. To nemůžeme udělat pomocí funkce `length()` protože ta nevrací délku řetězce, ale délku vektoru řetězců. Nebude to fungovat ani v případě jediného řetězce, protože i to je vektor o délce 1. (Pamatujte, že R nemá skalár: to, co vypadá jako jeden řetězec, je ve skutečnosti vektor řetězců o délce 1.)

Ke zjištění délky řetězce slouží funkce `str_length()`, která vrací vektor délek jednotlivých řetězců ve vektoru:

```
s1 <- "Ahoj!"
s2 <- c("A", "Bb", "Ccc", NA)
length(s1)  # délka (= počet pruků) vektoru s1
```

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

```
length(s2)  # délka (= počet pruků) vektoru s2
```

```
## [1] 4
```

```
str_length(s1)  # vektor délek (= počtu znaků) řetězce s1
```

```
## [1] 5
```

```
str_length(s2)  # vektor délek (= počtu znaků) řetězců ve vektoru s2
```

```
## [1]  1  2  3 NA
```

Pozor: Technicky vzato vrací funkce `str_length()` počet tzv. "code points". Ty většinou odpovídají jednotlivým znakům, ne však vždy. Např. znak á může být v paměti počítače reprezentován jako jeden znak nebo dva znaky (a a akcent). Ve druhém případě vrátí funkce `str_length()` délku 2. V takovém případě je bezpečnější počítat počet znaků pomocí funkce `str_count()`, viz dále. Příklad najdete v dokumentaci funkce. Pokud jsou však anglické i české znaky zadané obvyklým způsobem, pak bude funkce `str_length()` vracet počet znaků v jednotlivých řetězcích.

13.2.2 Spojení řetězců

Ke spojení více řetězců do jednoho slouží funkce `str_c(..., sep = "", collapse = NULL)`. Funkce bere libovolný počet řetězců, vypustí z nich prázdné vektory řetězců a `NULL` (ale ne prázdné řetězce `""`) a zbylé řetězce spojí do jednoho řetězce. Funkce `str_c()` implicitně odděluje jednotlivé řetězce prázdným řetězcem, tj. spojí jednotlivé řetězce těsně za sebe. Oddělovací řetězec je možné nastavit pomocí pojmenovaného parametru `sep`. (Všimněte si, že prázdný řetězec `""` funkce nevypustila.)

```
str_c("Jednou", "", "budem", character(0), "možná", NULL, "dál!")
```

```
## [1] "Jednoubudemmožnádál!"
```

```
str_c("Jednou", "", "budem", character(0), "možná", NULL, "dál!", sep = " ")
```

```
## [1] "Jednou  budem možná dál!"
```

```
str_c("Jednou", "", "budem", character(0), "možná", NULL, "dál!", sep = "-")
```

```
## [1] "Jednou--budem-možná-dál!"
```

Někdy potřebujeme spojit vektory řetězců. Funkce `str_c()` spojí odpovídající prvky jednotlivých vektorů (s obvyklou recyklací kratších vektorů, při které funkce vypíše varování) a vrátí vektor (rozumnější použití uvidíte dále):

```
s1 <- c("a", "b", "c", "d")
s2 <- 1:3  # automatická koerze převede čísla na řetězce
str_c(s1, s2)
```

```
## Warning in stri_c(..., sep = sep, collapse = collapse, ignore_null = TRUE):
## longer object length is not a multiple of shorter object length
```

```
## [1] "a1" "b2" "c3" "d1"
```

Pokud navíc chceme výsledný vektor spojit do jednoho řetězce, můžeme použít parametr `collapse`, kterým se nastavuje řetězec oddělující jednotlivé dílčí vektory (funguje i prázdný řetězec `""`):

```
str_c(s1, s2, collapse = "-")
```

```
## Warning in stri_c(..., sep = sep, collapse = collapse, ignore_null = TRUE):
## longer object length is not a multiple of shorter object length
```

```
## [1] "a1-b2-c3-d1"
```

```
str_c(s1, s2, collapse = "")
```

```
## Warning in stri_c(..., sep = sep, collapse = collapse, ignore_null = TRUE):
## longer object length is not a multiple of shorter object length
```

```
## [1] "a1b2c3d1"
```

Celkově funguje funkce `str_c()` takto: své parametry chápe jako vektory řetězců. Tyto vektory sestaví do matice, kde každý vstupní vektor tvoří jeden sloupec této matice (přitom prodlouží kratší vektory recyklací jeho prvků). Pak vloží mezi jednotlivé sloupce řetězec `sep` a spojí každý řádek do jednoho řetězce, takže výsledkem je jeden vektor řetězců. Pokud je navíc zadán řetězec `collapse` (tj. není `NULL`), funkce vloží tento řetězec mezi jednotlivé prvky tohoto vektoru a spojí je do jednoho řetězce. Tabulka 13.2 ukazuje, co udělá výraz `str_c(s1, s2, sep = "-")`.

Table 13.2: Ukázka toho, jak funguje výraz `str_c(s1, s2, sep = "-")`.

s1	(sep)	s2	výsledek
a	-	1	a-1
b	-	2	b-2
c	-	3	c-3
d	-	1	d-1

Pokud má kterýkoli řetězec ve spojovaných vektorech hodnotu `NA`, pak je výsledné spojení také `NA`:

```
s1 <- c("a", "b", NA)
s2 <- 1:3
str_c(s1, s2)
```

```
## [1] "a1" "b2" NA
```

```
str_c(s1, s2, collapse = "-")
```

```
## [1] NA
```

Některé zajímavé příklady použití funkce `str_c()` jsme vybrali z dokumentace funkce:

```
str_c("Letter", letters, sep = ": ")
```

```
##  [1] "Letter: a" "Letter: b" "Letter: c" "Letter: d" "Letter: e" "Letter: f"
##  [7] "Letter: g" "Letter: h" "Letter: i" "Letter: j" "Letter: k" "Letter: l"
## [13] "Letter: m" "Letter: n" "Letter: o" "Letter: p" "Letter: q" "Letter: r"
## [19] "Letter: s" "Letter: t" "Letter: u" "Letter: v" "Letter: w" "Letter: x"
## [25] "Letter: y" "Letter: z"
```

```
str_c(letters[-26], " comes before ", letters[-1])
```

```
##  [1] "a comes before b" "b comes before c" "c comes before d" "d comes before e"
##  [5] "e comes before f" "f comes before g" "g comes before h" "h comes before i"
##  [9] "i comes before j" "j comes before k" "k comes before l" "l comes before m"
## [13] "m comes before n" "n comes before o" "o comes before p" "p comes before q"
## [17] "q comes before r" "r comes before s" "s comes before t" "t comes before u"
## [21] "u comes before v" "v comes before w" "w comes before x" "x comes before y"
## [25] "y comes before z"
```

```
str_c(letters, collapse = ", ")
```

```
## [1] "a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z"
```

13.2.3 Doplnění hodnot do řetězců

Někdy potřebujeme spojit několik řetězců tak, že na vybraná místa do textu "vlepíme" hodnoty z vybraných proměnných – např. když chceme sestavit výpis z účtu:

```
cislo_uctu <- 854397
zustatek <- 12365.30
str_c("Na účtu číslo ", cislo_uctu, " je aktuální zůstatek ",
      format(zustatek, big.mark = ".", decimal.mark = ",", nsmall = 2),
      " Kč.")
```

```
## [1] "Na účtu číslo 854397 je aktuální zůstatek 12.365,30 Kč."
```

Využít ke "vlepení" hodnot do řetězce funkci `str_c()`, jak to ukazuje výše uvedený příklad, je možné, ale výsledek není právě přehledný. Místo toho můžeme použít speciální funkci `glue()` z balíku **glue**:

```
library(glue)
hezky_zustatek <- format(zustatek, big.mark = '.',
                                   decimal.mark = ',', nsmall = 2)
glue("Na účtu číslo {cislo_uctu} je aktualni zustatek {hezky_zustatek} Kč.")
```

```
## Na účtu číslo 854397 je aktualni zustatek 12.365,30 Kč.
```

Funkce `glue()` vezme jeden nebo více řetězců, slepí je dohromady a na místa ve složených závorkách vloží hodnoty proměnných z aktuálního prostředí R, přesněji výsledek výpočtu, takže předchozí kód bychom mohli napsat i takto:

```
glue("Na účtu číslo {cislo_uctu} je aktualni zustatek ",
     "{format(zustatek, big.mark = '.', decimal.mark = ',', nsmall = 2)} Kč.")
```

```
## Na účtu číslo 854397 je aktualni zustatek 12.365,30 Kč.
```

Hodnoty vložených proměnných můžeme zadat přímo jako parametry funkce `glue()`:

```
glue("Na účtu číslo {cislo_uctu} je aktualni zustatek {zustatek} Kč.",
     cislo_uctu = 24683, zustatek = 123)
```

```
## Na účtu číslo 24683 je aktualni zustatek 123 Kč.
```

Funkce `glue()` má několik zajímavých vlastností: zahazuje počáteční a koncová bílá místa (mezery apod.) včetně prvního a posledního zalomení řádku, ostatní zalomení a mezery však respektuje:

```
glue("
   Toto je jakýsi text.
        A zde pokračuje...
     ")
```

```
## Toto je jakýsi text.
##      A zde pokračuje...
```

```
glue("

   Toto je jakýsi text.
        A zde pokračuje...

        ")
```

```
##
## Toto je jakýsi text.
##      A zde pokračuje...
```

Zalomení řádků je možné zabránit pomocí (zdvojeného) lomítka na jeho konci:

```
glue("Zde nějaký text začíná, \\
     a zde pokračuje.")
```

```
## Zde nějaký text začíná, a zde pokračuje.
```

Pokud z nějakého důvodu potřebujeme v textu použít složené závorky, máme dvě možnosti: buď je zdvojit, nebo pomocí parametrů `.open` a `.close` nastavit jiný počátek a konec nahrazované oblasti na jiný znak:

```
glue("Zde je {{jméno}}.")
```

```
## Zde je {jméno}.
```

```
glue("Jméno výherce soutěže je {\\bf <<name>>}.", .open = "<<", .close =">>",
     name = "Joe") # vhodné např. při exportu do LaTeXu
```

```
## Jméno výherce soutěže je {\bf Joe}.
```

Balík **stringr** nabízí podobnou funkcionalitu ve funkcích `str_glue()` a `str_interp()`. Funkce `str_glue()` je vpodstatě jen wrapper nad funkcí `glue()` a používá se stejně:

```
str_glue("Jmenuji se {name} a příští rok mi bude {age + 1} let.",
         name = "Jana", age = 40)
```

```
## Jmenuji se Jana a příští rok mi bude 41 let.
```

Naproti tomu funkce `str_interp()` má poněkud jiné použití. V jejím případě musí mít nahrazovaný text buď podobu `${}`, kde ve složených závorkách je nějaký výraz, nebo `$[]{}`, kde v hranatých závorkách je formát a v složených výraz. Formát určuje typicky formátování čísel – např. `.2f` znamená, že se vytisknou dvě desetinná místa (detaily formátu viz nápověda k funkci `sprintf()`). Pokud chceme zadat vkládané hodnoty přímo v této funkci, je třeba je zabalit do seznamu.

```
str_interp("Na účtu číslo ${cislo_uctu} je aktualni zustatek ${zustatek} Kč.")
```

```
## [1] "Na účtu číslo 854397 je aktualni zustatek 12365.3 Kč."
```

```
str_interp("Na účtu číslo $[d]{cislo_uctu} je aktualni zustatek $[.2f]{zustatek} Kč.")
```

```
## [1] "Na účtu číslo 854397 je aktualni zustatek 12365.30 Kč."
```

```
str_interp("Na účtu číslo $[d]{cislo_uctu} je aktualni zustatek $[.2f]{zustatek} Kč.",
           list(cislo_uctu = 51349, zustatek = 17))
```

```
## [1] "Na účtu číslo 51349 je aktualni zustatek 17.00 Kč."
```

13.2.4 Řazení řetězců

Pro setřídění řetězců většinou stačí základní funkce `sort()` a `order()`. Pro složitější případy, kdy je např. třeba třídit v cizím *locale*, nabízí balík **stringr** dvě funkce:

- `str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)` vrací celé číslo, které odpovídá pořadí daného řetězce ve vektoru `x` (podobně jako funkce `order()`)
- `str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)` setřídí vektor řetězců `x`

Přitom `x` je vektor řetězců, který má být setříděn. `decreasing` je logická hodnota (implicitní hodnota je `FALSE`; pak třídí od nejnižšího k nejvyššímu; `TRUE` třídí od nejvyššího k nejnižšímu). `na_last` je logická hodnota (implicitní hodnota je `TRUE`, při které funkce umístí hodnoty `NA` na konec vektoru; `FALSE` je umístí na začátek na začátek vektoru; `NA` je vyhodí). `locale` označuje v jakém *locale* se má třídit (implicitně v systémovém). Pokud je logická hodnota `numeric` nastavena na `TRUE`, pak číslice řadí jako čísla, ne jako řetězce; implicitní hodnota je `FALSE`. `...` označují další parametry přidané do `stri_opts_collator`.

```
str_order(letters, locale = "en")
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
## [26] 26
```

```
str_sort(letters, locale = "en")
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

13.2.5 Výběr a náhrada pomocí indexů

Někdy je třeba z řetězce vybrat jeho část. Pokud známe pozici prvního a posledního znaku, který chceme vybrat, můžeme použít funkce `str_sub(string, start = 1L, end = -1L)`, kde `string` je řetězec, ze kterého vybíráme, `start` je pozice znaku začátku výběru a `end` je pozice konce výběru (včetně). Implicitní hodnota `start` je 1 (tj. od začátku vektoru), implicitní hodnota `end` je −1 (tj. do konce vektoru). Pozice znaků mohou být zadány i jako záporná čísla – ta se počítají od konce vektoru, takže např. −1 je poslední znak vektoru.

```
s1 <- "Sol 6: Katastrofa na Marsu."
str_sub(s1, start = 8, end = str_length(s1))
```

```
## [1] "Katastrofa na Marsu."
```

```
str_sub(s1, start = 8, end = -1)  # totéž
```

```
## [1] "Katastrofa na Marsu."
```

```
str_sub(s1, 8)  # totéž
```

```
## [1] "Katastrofa na Marsu."
```

```
str_sub(s1, end = 5)
```

```
## [1] "Sol 6"
```

Funkce `str_sub()` recykluje všechny své parametry. Pokud chceme např. vybrat stejné pozice z každého vektoru v řetězci:

```
s2 <- c(s1, "Sol 7: Nová naděje.")
str_sub(s2, start = c(8, 8))
```

```
## [1] "Katastrofa na Marsu." "Nová naděje."
```

```
str_sub(s2, 8)  # totéž
```

```
## [1] "Katastrofa na Marsu." "Nová naděje."
```

Stejně tak je však možné recyklovat i řetězec a vybrat z něj naráz dva pod-řetězce:

```
str_sub(s1, start = c(8, 22), end = c(17, 26))
```

```
## [1] "Katastrofa" "Marsu"
```

Pokud je řetěz kratší než výběr znaků, vrátí funkce `str_sub()` tolik znaků, kolik může (někdy i prázdný řetězec):

```
str_sub(s1, start = 25, end = 50)
```

```
## [1] "su."
```

```
str_sub(s1, start = 40, end = 50)
```

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

Funkci `str_sub()` je možné použít i k náhradě části řetězce:

```
str_sub(s1, 22, 26) <- "rudé planetě"
s1
```

```
## [1] "Sol 6: Katastrofa na rudé planetě."
```

13.2.6 Replikace řetězců

Někdy je potřeba nějaký řetězec "zmnožit". K tomu slouží funkce `str_dup(string, times)`, který vezme vektor `string`, zopakuje jej `times`-krát a výsledek spojí. To se hodí např. při načítání mnoha textových sloupců pomocí balíku **readr**. Při tom je někdy užitečné říct funkci `read_csv()`, že všechny sloupce tabulky mají typ `character`. K tomu slouží řetězec mnoha "c":

```
str_dup("c", 28)
```

```
## [1] "ccccccccccccccccccccccccccc"
```

Funkce `str_dup()` také recykluje všechny své parametry:

```
str_dup(c("a", "b", "c"), 1:3)
```

```
## [1] "a"   "bb"  "ccc"
```

13.2.7 Odstranění okrajových mezer

Někdy dostaneme řetězec, který začíná nebo končí "bílými znaky" (mezerami, tabelátory, novými řádky "\n" apod.). Tato situace vznikne např. tehdy, když řetězec vznikl rozdělením delšího řetězce na části. Tyto bílé znaky je často vhodné odstranit. K tomu slouží funkce `str_trim(string, side)`, kde `string` je řetězec a `side` označuje stranu, ze které se mají bílé znaky odstranit:

```
s1 <- c("Ahoj,", " lidi, ", "jak ", "se", " máte?")
str_trim(s1, "left")   # odstraní mezery zleva
```

```
## [1] "Ahoj,"  "lidi, " "jak "   "se"     "máte?"
```

```
str_trim(s1, "right")  # odstraní mezery zprava
```

```
## [1] "Ahoj,"  " lidi," "jak"    "se"     " máte?"
```

```
str_trim(s1, "both")   # odstraní mezery z obou stran
```

```
## [1] "Ahoj," "lidi," "jak"   "se"    "máte?"
```

```
str_trim(s1)   # totéž -- "both" je implicitní hodnota side
```

```
## [1] "Ahoj," "lidi," "jak"   "se"    "máte?"
```

Funkce `str_squish(string)` funguje podobně, ale odstraní navíc i všechny přebytečné prázdné znaky – na začátku řetězce `string`, na jeho konci i opakované znaky uvnitř:

```
str_squish("  Toto  je   koktavý  text.   ")
```

```
## [1] "Toto je koktavý text."
```

13.2.8 Zarovnání a ořež řetězců na stejnou délku a do odstavce

Někdy je užitečné zarovnat řetězce na stejnou délku přidáním bílých (nebo jiných) znaků. K tomu slouží funkce `str_pad(string, width, side, pad = " ")`, kde `string` je vektor zarovnávaných řetězců, `width` je minimální délka výsledného řetězce, `side` je strana, na kterou se mají výplňové znaky přidat (implicitně je to `left`) a `pad` je výplňový řetězec (implicitně mezera).

```
str_pad(c("Ahoj", "lidi"), 7)
```

```
## [1] "   Ahoj" "   lidi"
```

```
str_pad(c("Ahoj", "lidi"), 9, side = "both", pad = "-")
```

```
## [1] "--Ahoj---" "--lidi---"
```

Delší řetězce funkce nemění:

```
str_pad(c("Ahoj", "malé zelené bytosti z Viltvodlu VI"), width = 7)
```

```
## [1] "   Ahoj"                            "malé zelené bytosti z Viltvodlu VI"
```

V některých případech potřebujeme delší řetězec zkrátit na určitou délku tak, že se jeho část vynechá. K tomu slouží funkce `str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...")`, která zkrátí řetězec `string` na délku `width` znaků. Přitom přebytečnou část textu na straně `side` vynechá a nahradí znaky `ellipsis`:

```
s <- "Toto je přehnaně dlouhý text, který toho mnoho neříká."
str_trunc(s, 20, "right")
```

```
## [1] "Toto je přehnaně ..."
```

```
str_trunc(s, 20, "left")
```

```
## [1] "...oho mnoho neříká."
```

```
str_trunc(s, 20, "center")
```

```
## [1] "Toto je p... neříká."
```

Zarovnat řetězec do odstavce umožňuje funkce `str_wrap(string, width = 80, indent = 0, exdent = 0)`, kde `strings` je zarovnávaný řetězec, `width` je cílová šířka sloupce (implicitně 80 znaků), `indent` je odsazení prvního řádku a `exdent` je odsazení následujících řádků (oboje implicitně 0 znaků):

```
s1 <- "Na počátku bylo Slovo, to Slovo bylo u Boha, to Slovo byl Bůh. To bylo na počátku u Boha. Vše
cat(str_wrap(s1, 60))
```

```
## Na počátku bylo Slovo, to Slovo bylo u Boha, to Slovo byl
## Bůh. To bylo na počátku u Boha. Všechno povstalo skrze ně
## a bez něho nepovstalo nic, co jest. V něm byl život a život
## byl světlo lidí. To světlo ve tmě svítí a tma je nepohltila.
```

13.2.9 Konverze malých a velkých písmen

Ke konverzi malých písmen na velká slouží funkce `str_to_upper(string, locale = "en")`, ke konverzi na malá písmena funkce `str_to_lower(string, locale = "en")`, kde `string` je převáděný řetězec. Parametr `locale` umožňuje zadat popis jazykových pravidel pro převod znaků. Parametr je nepovinný a pokud není zadán, použije se aktuální *locale* počítače.

```
s <- "Nějaký malý textík..."
str_to_lower(s, locale = "cz")
```

```
## [1] "nějaký malý textík..."
```

```
str_to_upper(s, locale = "cz")
```

```
## [1] "NĚJAKÝ MALÝ TEXTÍK..."
```

13.2.10 Nahrazení chybějících hodnot řetězcem

Někdy máme vektor řetězců, ve kterém některé hodnoty chybí, tj. mají hodnotu `NA`, a my potřebujeme tyto hodnoty nahradit nějakým textem. K tomu slouží funkce `str_replace_na(string, replacement = "NA")`, kde `string` je vektor řetězců a `replacement` je řetězec, který má v řetězci `string` nahradit chybějící hodnoty. Pokud chceme např. nahradit chybějící hodnoty slovem "neznámé", můžeme to udělat takto:

```
poloha <- c("nahoře", "dole", NA, "vlevo")
str_replace_na(poloha, replacement = "neznámá")
```

```
## [1] "nahoře"  "dole"    "neznámá" "vlevo"
```

13.2.11 Formátování čísel

Někdy potřebujeme vytisknout čísla v "pěkném formátu". To znamená, že musíme převést dané číslo na zformátovaný řetězec, který dané číslo prezentuje v požadovaném tvaru. K tomu slouží funkce `format()` ze základního balíku **base**. Funkce `format()` je *generická* a umí formátovat mnoho různých objektů. Pro čísla má následující parametry:

```
format(x, trim = FALSE, digits = NULL, nsmall = 0L,
       justify = c("left", "right", "centre", "none"),
       width = NULL, na.encode = TRUE, scientific = NA,
       big.mark   = "",   big.interval = 3L,
       small.mark = "", small.interval = 5L,
       decimal.mark = getOption("OutDec"),
       zero.print = NULL, drop0trailing = FALSE, ...)
```

kde `x` je numerický vektor. Další užitečné parametry jsou zejména `nsmall` (minimální počet desetinných míst), `big.mark` a `decimal.mark` (znak oddělující tisíce a desetinná místa), `scientific` (pokud je `TRUE`, pak vypisuje čísla ve vědecké notaci), `width` (minimimální počet znaků) a `justify` (zarovnání). Další parametry jsou popsané v dokumentaci.

```
x <- 123456.3
print(x)
```

```
## [1] 123456.3
```

```
print(format(x, big.mark = ".", decimal.mark = ",", nsmall = 2))
```

```
## [1] "123.456,30"
```

```
print(format(x, scientific = TRUE))
```

```
## [1] "1.234563e+05"
```

```
print(format(x, width = 10, justify = "left"))
```

```
## [1] "  123456.3"
```

13.3 Regulární výrazy

Jednoduché úpravy řetězců popsané výše a hledání, jaké je obvyklé v programech typu Word nebo Excel nestačí, když řešíme nějakou složitější situaci. Řekněme, že potřebujeme v textu najít každé telefonní číslo a převést je na nějaký standardní tvar. Každé telefonní číslo se však skládá z jiných číslic a navíc mohou být tyto číslice i různě oddělené: telefonní číslo `123456789` je totéž, co `123 456 789`, `+420 123 456 789` a `+420-123-456-789`. Podobné problémy nastávají i při zjišťování data: stejné datum může být např. zadané jako `1.6.2006`, `1. 6. 2006` i `1. června 2006`. K hledání a úpravám takto volně definovaných skupin znaků slouží regulární výrazy.

Regulární výraz je řetězec, který obsahuje vzor (*pattern*) toho, jak vypadá hledaný kus textu. Je to tedy jakási abstraktní maska, která ukazuje, jaké vlastnosti musí kus textu splňovat, aby byl vybrán. Zkoumaný řetězec se pak prochází a hledají se ty jeho části, které splňují daný vzor. Bohužel se regulární výrazy implementované v různých programech od sebe mírně liší. Z historických důvodů se dnes v R používají tři různé typy regulárních výrazů: základní funkce v R používají regulární výrazy standardu POSIX 1003.2 nebo (na požádání) regulární výrazy podle standardu jazyka Perl. Balík **stringr** používá regulární výrazy založené na ICU. Naštěstí se od sebe různé standardy liší jen v různých nadstavbách. Zde se podíváme na základ, který je společný všem třem výše zmíněným formám regulárních výrazů. (Plné definice těchto standardů najdete v dokumentaci funkcí: `help("regex", package = "base")` pro POSIX a `help("stringi-search-regex", package = "stringi")` pro ICU.)

Jak už bylo řečeno, regulární výraz je řetězec. Některé znaky v něm se berou doslovně, jiné mají speciální význam, který může záviset na kontextu, ve kterém jsou použité. Prakticky používají regulární výrazy čtyři operace:

- spojování – znaky zapsané za sebou se spojí do jednoho výrazu; např. `abcd` je spojení čtyř znaků, které fungují jako celek, tj. jako řetězec "abcd"
- logické "nebo" označené symbolem `|` vybírá jednu z možností; např. `ab|cd` znamená řetězec "ab" nebo řetězec "cd", protože spojování má vyšší prioritu než "nebo"
- opakování – umožňuje říct, kolikrát se má nějaký řetězec v textu vyskytovat; k tomu poskytují regulární výrazy *kvantifikátory*, viz dále; např. `abc+` znamená řetězce `abc`, `abcc`, `abccc` atd.; opakuje se jen poslední znak, protože operace spojování i "nebo" mají nižší prioritu než kvantifikátory
- seskupování – znaky zapsané do obyčejných závorek `( )` tvoří skupinu, tj. jeden celek, který se může např. opakovat, měnit priority vyhodnocování výrazů apod., protože seskupení má nejvyšší prioritu ze všech operací; skupiny mají i speciální význam při vybírání částí regulárních výrazů, viz dále

Všechny znaky kromě `[\^$.|?*+(){}` se berou doslovně, tj. např. `a` znamená písmeno "a", `1` znamená číslici "1" atd. Regulární výraz `"jak"` tedy najde v textu všechny výskyty slova "jak" všechně výskytů v jiných slovech, např. "jakkoli"; nenajde však "Jak", protože regulární výraz chápe malá a velká písmena jako různé znaky.

Znaky `[\^$.|?*+(){}` mají speciální význam. To znamená, že výraz `cože?` nenajde doslovně řetězec "cože?" (ve skutečnosti najde řetězec "což" nebo řetězec "cože"). Stejně tak `Ano.` neznamená doslovně "Ano.", nýbrž jakýkoli řetězec, ve kterém za "Ano" následuje jeden znak (např. "Ano!"). Konkrétně tečka `.` označuje libovolný znak včetně mezer, tabelátorů apod. Výraz `ma.ka` najde slova jako "matka", "maska" nebo "marka", ale ne "maka".

Pokud chcete zadat přímo speciální znak, je třeba jej zabezpečit pomocí zpětného lomítka. Skutečná tečka je tedy `\.`, hranatá závorka je `\[` atd. Bohužel je v R zpětné lomítko samo aktivním znakem. To znamená,

že je třeba jej zabezpečit dalším zpětným lomítkem. Místo `\.` je tak třeba zadat `"\\."` apod. Nepříjemné je to v případě, kdy potřebujete hledat doslovně zpětné lomítko. V regulárním výrazu je musíte zabezpečit jiným zpětným lomítkem, takže dostanete `\\`. V R však dále musíte každé zpětné lomítko zabezpečit dalším zpětným lomítkem, takže regulární výraz, který hledá zpětné lomítko musí být v R zadaný jako čtyři zpětná lomítka: `\\\\`.

13.3.1 Rozsahy

Někdy chceme najít řetězec, který obsahuje jakýkoli znak z určité množiny. K tomu slouží rozsahy (*character classes*). Nejjednodušší způsob, jak v regulárním výrazu zadat rozsah, je použít operátor hranatá závorka. Hranaté závorky určují výčet platných znaků. Tak např. `[Aa]` najde všechny výskyty malého "a" a velkého "A". Takže výrazu `[Aa]dam` vyhoví jak "Adam", tak "adam". V případě číslic a ASCII znaků mohou hranaté závorky obsahovat i rozsah zadaný pomocí pomlčky. Pro nalezení číslic 0 až 4 tedy není třeba psát `[01234]`, nýbrž stačí `[0-4]`. Pokud se má pomlčka chápat v rozsahu doslovně, je třeba ji napsat jako první znak. Hranaté závorky mohou obsahovat i negaci, ke které slouží znak `^` uvedený hned za otevírající hranatou závorkou. Výraz `[^Aa]` splňují všechny znaky mimo malého "a" a velkého "A".

Pro některé často užívané výčty existují speciální symboly. Tabulka 13.3 uvádí nejdůležitější z nich. Kromě těchto rozsahů mohou regulární výrazy v R obsahovat i POSIXové rozsahy, které mají zvláštní tvar `[:jméno:]` a mohou se vyskytovat jen ve vnějším rozsahu, tj. jako `[[:jméno:]]` nebo např. `[a[:jméno:]b]` apod. Jejich seznam uvádí tabulka 13.4. POSIXové rozsahy se od základních rozsahů liší v tom, že respektují *locale* počítače, takže mezi písmena počítají i znaky národní abecedy. Rozdíl mezi `\w` a `[:alnum:]` tedy spočívá v tom, že `[:alnum:]` obsahuje v českém *locale* i písmena s háčky a čárkami, zatímco `\w` obsahuje jen písmena anglické abecedy (zahrnutá v ASCII). Pokud tedy uvažujete písmena, doporučuji používat vždy POSIXové rozsahy.

Table 13.3: Přehled základních rozsahů používaných v regulárních výrazech.

symbol	význam	výčtem
`\d`	číslice	`[0-9]`
`\D`	není číslice	`[^0-9]`
`\w`	písmeno, číslice nebo podtržítko	`[a-zA-z0-9_]`
`\W`	cokoli, co není `\w`	`[^a-zA-z0-9_]`
`\s`	prázdný znak	`[ \t\n\r\f]`
`\S`	není prázdný znak	`[^ \t\n\r\f]`
`\b`	hranice slova	
`\B`	není hranice slova	
`\h`	horizontální mezera	
`\H`	není horizontální mezera	
`\v`	vertikální mezera	
`\V`	není vertikální mezera	

Table 13.4: Přehled POSIXových rozsahů používaných v regulárních výrazech.

rozsah	význam
`[:lower:]`	malá písmena v *locale* počítače
`[:upper:]`	velká písmena v *locale* počítače
`[:alpha:]`	malá i velká písmena v *locale* počítače
`[:digit:]`	číslice, tj. 0, 1, ..., 9
`[:alnum:]`	alfanumerické znaky, tj. `[:alpha:]` a `[:digit:]`
`[:blank:]`	mezera a tabelátor
`[:cntrl:]`	řídící znaky
`[:punct:]`	`!`, `"`, `#`, `%`, `&`, `'`, `(`, `)`, `*`, `+`, `,`, `-`, `.`, `/`, `:`, `;`, `<`, `>`, `?`, `@`, `[`, `\`, `]`, `^`, `_`, `‘`, `{`,

13.3.2 Kvantifikátory

Znaky `*+?{` jsou kvantifikátory, které řídí počet opakování předchozího znaku, rozsahu nebo skupiny. Jejich význam shrnuje tabulka 13.5.

Table 13.5: Přehled POSIXových rozsahů používaných v regulárních výrazech.

znak	význam
`*`	libovolný počet opakování (tj. vůbec, jednou nebo víckrát)
`+`	aspoň jeden výskyt (tj. jednou nebo víckrát)
`?`	maximálně jeden výskyt (tj. vůbec nebo jednou)
`{5}`	právě pětkrát
`{3,5}`	nejméně třikrát a nejvýše pětkrát
`{3,}`	nejméně třikrát
`{,5}`	maximálně pětkrát

Podívejme se na příklady:

- `.*` odpovídá jakémukoli znaku, který je opakovaný libovolně krát, tj. jakýkoli řetězec včetně `""`
- `\w+` odpovídá aspoň jednomu písmenu anglické abecedy
- `[+-]?\d+` odpovídá celému číslu (najde např. “+12”, “1”, “-3” apod.; z “3.14” najde “3” a “14”)

Kvantifikátory jsou “hladové”. To znamená, že najdou řetězec s maximální délkou, která vyhovuje vzoru. To se nemusí vždy hodit. Např. pokud chceme najít všechny přímé řeči, regulární výraz `".*"` nebude fungovat, protože najde celý výraz mezi první a poslední uvozovkou:

```
s1 <- '"Už je to dobré," řekl Josef. "Pojdme si zaplavat."'
str_extract_all(s1, '".*"')  # jeden řetězec od první po poslední uvozovku
```

```
## [[1]]
## [1] "\"Už je to dobré,\" řekl Josef. \"Pojdme si zaplavat.\""
```

Problém vyřeší “líný” kvantifikátor, který najde řetězec s minimální délkou, který splňuje zadaný vzor. Z kvantifikátoru uděláte líný tak, že za něj připojíte otazník. Např. kvantifikátor pro nula až nekonečný počet výskytů `*` je hladový; jeho líná verze je `*?`:

```
str_extract_all(s1, '".*?"')  # vektor dvou řetězců, v každém je jedna přímá řeč
```

```
## [[1]]
## [1] "\"Už je to dobré,\""     "\"Pojdme si zaplavat.\""
```

13.3.3 Začátek a konec řetězce

Normálně regulární výrazy najdou jakoukoli část řetězce. Někdy je však užitečné hledat jen za začátku nebo na konci řetězce. K tomu slouží “kotvy” (*anchors*). Znak `^` uvedený na začátku regulárního výrazu znamená začátek řetězce (nebo ve speciálním případě řádku, viz dále). Znak `$` uvedený na konci regulárního výrazu znamená konec řetězce (nebo ve speciálním případě řádku, viz dále).

Pokud tedy chceme najít jen řádky, které začínají písmenem “A”, můžeme použít regulární výraz `^A`. Pokud chceme najít řádky, které končí tečkou, můžeme použít regulární výraz `\.$` (v R bude potřeba zpětné lomítko zdvojit a zadat `"\\.$"`). Výraz `^-+$` splní pouze řádky, které obsahují pouze pomlčky.

13.3.4 Skupiny

Někdy je potřeba některé znaky seskupit. K tomu slouží kulaté závorky. Skupiny ovlivňují priority vyhodnocování regulárního výrazu. To je užitečné např. při alternaci. Např. regulárnímu výrazu `abc|def` vyhoví řetězce "abc" a "def". Naproti tomu výrazu `ab(c|d)ef` vyhoví řetězce "abcef" a "abdef". (Normálně má spojení prioritu před alternací.) Skupiny ovlivňují priority i při použití kvantifikátorů. Např. výrazu `abc*` vyhoví řetězce "ab", "abc", "abcc" atd. Naproti tomu výrazu `(abc)*` vyhoví prázdný řetězec `""`, "abc", "abcabc" atd. (Normálně má kvantifikátor přednost před spojením.)

Skupiny navíc dávají regulárním výrazům "paměť". První skupině odpovídá `\1`, druhé skupině `\2` atd. (Pokud jsou do sebe skupiny vložené, pak se jejich čísla počítají podle pořadí levé závorky.) Tuto paměť je možné využít jak ve vyhledávacím řetězci, tak při náhradě regulárního výrazu, viz dále. Např. k nalezení pětiznakového palindromu (tj. slova, které je stejné, ať ho čteme zleva nebo zprava, např. kajak nebo madam) můžeme použít regulární výraz `(.)(.).\2\1`. V něm musí být znak `\1` stejný jako první znak palindromu (odpovídající první skupině s tečkou) a `\2` musí být znak stejný jako druhý znak palinromu (odpovídající druhé skupině s tečkou). (Náš regulární výraz najde ovšem i palindromy složené z čísel apod. Pro hledání skutečných palindromů by bylo bezpečnější nahradit v regulárním výrazu tečky např. rozsahem `[[:alpha:]]`.)

13.3.5 Příklady

Podívejme se nyní na regulární výrazy pro dva případy zmíněné výše: pro nalezení telefonního čísla a datumu. Regulární výraz, který pozná všechny výše uvedené formáty telefonního čísla, vypadá takto: `(\+420)?[\s-]*\d{3}[\s-]*\d{3}[\s-]*\d{3}`. V R však musejí být všechna zpětná lomítka zdvojena, takže regulární výraz musí být zapsán takto:

```
r <- "(\\+420)?[-\\s]*\\d{3}[-\\s]*\\d{3}[-\\s]*\\d{3}"
cisla <- c("Leoš: 777 666 555 444",
           "Lída: domů +420 734 123 456, práce 777-666-555",
           "Leona: nevím")
str_extract_all(cisla, r)
```

```
## [[1]]
## [1] " 777 666 555"
##
## [[2]]
## [1] "+420 734 123 456" " 777-666-555"
##
## [[3]]
## character(0)
```

Regulární výraz, který pozná všechny výše uvedené formáty data:

```
r <- str_c("\\d{1,2}\\.\\s*(\\d{1,2}\\.|leden|únor|březen|duben|červen|červenec|srpen|",
           "září|říjen|listopad|prosinec)\\s*\\d{4}")
dat <- c("1. 6. 2001", "1.1.2012", "1. červen 2016", "ahoj")
str_detect(dat, r)
```

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

13.3.6 Praktická práce s regulárními výrazy

Regulární výrazy jsou velmi mocné, ale poněkud nepřehledné. Vždy byste si proto měli vyzkoušet, zda váš regulární výraz 1) nachází to, co chcete, a 2) nenachází to, co nechcete. Je proto velmi užitečné, abyste si připravili několik příkladů řetězců, které by výraz měl a které by neměl najít.

K otestování regulárního výrazu pak můžete použít buď funkce popsané v dalším oddílu, nebo uživatelsky přívětivé funkce `str_view(string, pattern, match = NA)` a `str_view_all(string, pattern, match =`

NA), kde string je prohledávaný vektor řetězců, pattern je zkoušený regulární výraz a match je logická hodnota. Pokud je TRUE, zobrazí jen řetězce, které splňují daný regulární výraz, pokud FALSE, pak zobrazí jen řetězce, které daný regulární výraz nesplňují. Pokud necháte implicitiní hodnotu NA, pak zobrazí všechny zadané řetězce. Funkce zobrazí v RStudiu v záložce Viewer zadané řetězce a barevně vyznačí části, které odpovídají zadanému regulárnímu výrazu. První funkce zobrazí pouze první výskyt, zatímco druhá funkce všechny výskyty. Pohodlné testování regulárních výrazů umožňuje i doplněk RStudia RegExplain. Protože není zatím součástí CRANu, nainstalujete jej takto:

```
devtools::install_github("gadenbuie/regexplain")
```

I když váš regulární výraz funguje, měli byste být opatrní a pamatovat na to, že funguje jen za předem určených předpokladů: náš výraz pro datum např. najde jen datum zadané tak, jak je zvykem v České republice, a to ještě jen v případě, že je zadán i rok. Pokud byste chtěli obecnější řešení, začal by váš regulární výraz být složitější a složitější. Pak se často vyplatí výraz zjednodušit a hledaný řetězec hledat na dvakrát nebo na třikrát a nebo využít i jiné nástroje než jen regulární výrazy. Některé texty byste neměli pomocí regulárních výrazů procházet téměř nikdy. Sem patří např. webové stránky, které je možné procházet mnohem elegantněji s použitím syntaxe *XPath* nebo *CSS*.

Pamatujte také, že regulární výrazy obsahují mnohem více než to, co jste viděli v tomto oddílu. Plný výčet detailů je na těchto (podle mého mínění velmi nepřehledných) stránkách http://www.regular-expressions.info/reference.html. Ve většině případů vám však bude obsah tohoto oddílu bohatě stačit.

13.4 Funkce pro práci s regulárními výrazy

Nyní se podíváme na to, jak využít regulární výrazy pro práci s řetězci v R. Většina funkcí pro práci s regulárními výrazy definovaná v balíku **stringr** má podobnou syntaxi: `str_XXX(string, pattern, ...)`, kde XXX je část jména funkce, string je vektor zpracovávaných řetězců, pattern je použitý regulární výraz a ... případné další parametry. Pokud některá funkce vrací jen první výskyt hledaného vzoru v řetězci, pak většinou existuje i podobná funkce s koncovkou _all, která vrací všechny výskytu zadaného vzoru.

13.4.1 Detekce vzoru

Nejjednodušším případem použití regulárních výrazů je nalezení řetězců, které odpovídají danému regulárnímu výrazu. K detekci, zda řetězec odpovídá zvolenému regulárnímu výrazu, slouží funkce `str_detect(string, pattern, negate = FALSE)`, kde string je prohledávaný vektor řetězců a pattern je hledaný vzor. Funkce vrací logický vektor stejné délky jako string s hodnotou TRUE, pokud byl vzor nalezen, a hodnotou FALSE v opačném případě. Nastavení parametru negate na TRUE výsledek obrací: nyní funkce vrací TRUE, pokud řetězec daný vzor neobsahuje. Řekněme, že chceme zjistit, které řetězce ve vektoru s1 obsahují slova o pěti písmenech, kde první dvě jsou "ma" a poslední dvě "ka" (další příklady najdete výše):

```
s1 <- c("matka", "mačka", "mačká", "maska", "Matka", "marka!", "ma ka")
str_detect(s1, "ma.ka")  # TRUE, pokud obsahuje vzor
```

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

```
str_detect(s1, "ma.ka", negate = TRUE)  # TRUE, pokud neobsahuje vzor
```

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

Funkce `str_which(string, pattern, negate = FALSE)` je podobná a má i stejné parametry. Místo logických hodnot však vrací celočíselný vektor pozic prvků vektoru string, které splňují zadaný regulární výraz:

```
str_which(s1, "ma.ka")
```

```
## [1] 1 2 4 6 7
```

Často potřebujeme řetězce, které odpovídají regulárnímu výrazu vybrat. To by bylo možné provést pomocí subsetování:

```
s1[str_detect(s1, "ma.ka")]
```

```
## [1] "matka"  "mačka"  "maska"  "marka!" "ma ka"
```

Balík **stringr** však k tomuto účelu nabízí komfortnější funkci `str_subset(string, pattern, negate = FALSE)`, která vrací ty prvky vektoru `s`, které obsahují vzor `p` (parametry funkce mají stejný význam jako u `str_detect()`):

```
str_subset(s1, "ma.ka")
```

```
## [1] "matka"  "mačka"  "maska"  "marka!" "ma ka"
```

Podívejme se na praktičtější příklad. Řekněme, že chceme vybrat ty řetězce z vektoru `cisla`, které obsahují telefonní čísla:

```
r <- "(\\+420)?[-\\s]*\\d{3}[-\\s]*\\d{3}[-\\s]*\\d{3}"
cisla <- c("Leoš: 777 666 555 444",
           "Lída: domů +420 734 123 456, práce 777-666-555",
           "Leona: nevím")
str_subset(cisla, r)
```

```
## [1] "Leoš: 777 666 555 444"
## [2] "Lída: domů +420 734 123 456, práce 777-666-555"
```

Mohl by nás zajímat i počet výskytů regulárního výrazu v řetězci. K tomu slouží funkce `str_count(string, pattern)`, která spočítá počet výskytů regulárního výrazu `pattern` v řetězci `string`. Pracuje vektorově přes zadaný vektor řetězců `string` i přes regulární výraz `pattern`. Implicitní hodnota `pattern` je prázdný řetězec `""`. V tomto případě vrací funkce `str_count()` počet znaků v řetězci (počítaných podle aktuálního *locale*). Podívejme se na několik příkladů:

```
ovoce <- c("jablko", "ananas", "hruška", "rybíz")
str_count(ovoce, "a")  # počet výskytů "a" v každém slově
```

```
## [1] 1 3 1 0
```

```
# počet "a" v jablku, "a" v ananasu, "b" v hrušce a "r" v rybízu
str_count(ovoce, c("a", "a", "b", "r"))
```

```
## [1] 1 3 0 1
```

```
str_count(c("Ahoj", "lidičky!"))  # počet znaků v každém řetězci
```

```
## [1] 4 8
```

13.4.2 Získání částí řetězců, které splnují vzor

Většinou nám nestačí najít výskyt řetězce, který odpovídá regulárnímu výrazu, ale chceme tento řetězec získat. K tomu slouží funkce `str_extract(string, pattern)`, která získá z každého řetězce ve vektoru `string` tu jeho část, která odpovídá prvnímu výskytu vzoru `pattern`. Funkce vrací vektor stejné délky jako `string`; pokud není vzor nalezen, vrací `NA`.

Řekněme, že chceme např. vybrat z tweetů jednotlivé hashtagy (zjednodušíme si život a budeme předpokládat, že hashtag začíná křížkem a tvoří ho písmena a čísla a končí s prvním výskytem ne-alfanumerického znaku jako je mezera, tečka, čárka apod.):

```
r <- "#[[:alpha:]]+"  # hashtag následovaný jedním nebo více písmeny
tweet <- c("#Brno je prostě nejlepší a #MU je nejlepší v Brně.",
           "Někdy je možné najít zajímavé podcasty na #LSE.",
           "Stupnování 'divnosti': divný, divnější, #ParisHilton.",
           "Docela prázdný tweet.")
str_extract(tweet, r)
```

```
## [1] "#Brno"        "#LSE"         "#ParisHilton" NA
```

Funkce `str_extract()` vrací pouze první výskyt výrazu v každém řetězci. Pokud chceme získat všechny výskyty, musíme použít funkci `str_extract_all(string, pattern, simplify = FALSE)`. Implicitně funkce vrací seznam, jehož prvky odpovídají prvkům vektoru `string`; nenalezené výskyty pak indikuje prázdný vektor řetězců (`character(0)`). Funkce však umí zjednodušit výsledek na matici. K tomu slouží parametr `simplify` nastavený na hodnotu `TRUE`. V tomto případě odpovídají řádky výsledné matice prvkům vektoru `string`; nenalezené výskyty jsou pak označené jako prázdné řetězce `""`.

```
str_extract_all(tweet, r)
```

```
## [[1]]
## [1] "#Brno" "#MU"
##
## [[2]]
## [1] "#LSE"
##
## [[3]]
## [1] "#ParisHilton"
##
## [[4]]
## character(0)
```

```
str_extract_all(tweet, r, simplify = TRUE)
```

```
##      [,1]           [,2]
## [1,] "#Brno"        "#MU"
## [2,] "#LSE"         ""
## [3,] "#ParisHilton" ""
## [4,] ""             ""
```

Někdy nechceme získat celý vzor, nýbrž pouze jeho části. K tomu slouží funkce `str_match(string, pattern)` a `str_match_all(string, pattern)`, kde `string` je vektor prohledávaných řetězců a `pattern` regulární výraz. K rozdělení regulárního výrazu do částí se používají skupiny. Funkce `str_match()` hledá první výskyt regulárního výrazu `pattern` v řetězci a vrací matici, jejíž řádky odpovídají prvkům vektoru `string`. První sloupec je celý regulární výraz, druhý sloupec první skupina v regulárním výrazu, třetí sloupec druhá

skupina atd. (Pokud jsou skupiny zanořené jedna do druhé, pak se jejich pořadí počítá podle pořadí levé závorky.) Nenalezené prvky mají hodnotu `NA`.

Pokud např. chceme získat jméno hashtagu bez křížku, můžeme vzít druhý sloupec matice, kterou v našem případě vrátí funkce `str_match()`:

```
r <- "#([[:alpha:]]+)"  # totéž, co výše, ale všechna písmena tvoří skupinu
str_match(tweet, r)
```

```
##      [,1]           [,2]
## [1,] "#Brno"        "Brno"
## [2,] "#LSE"         "LSE"
## [3,] "#ParisHilton" "ParisHilton"
## [4,] NA             NA
```

Poznámka: Vždy stojí za zvážení, zda provést nějakou operaci naráz pomocí složitého regulárního výrazu, nebo ji rozdělit do několika kroků. Extrakci hashtagů z minulého příkladu můžeme snadno provést ve dvou jednodušších krocích: nejdříve extrahovat celý hashtag, a pak z něj odstranit křížek:

```
str_extract(tweet, r) %>% str_remove("#")
```

```
## [1] "Brno"        "LSE"         "ParisHilton" NA
```

Funkce `str_match_all()` vrací všechny výskyty regulárního výrazu. Jejím výsledkem je seznam matic. Řádky těchto matic odpovídají jednotlivým nálezům. Sloupce mají význam jako výše. Pokud není regulární výraz v daném řádku nalezen, je výsledkem prázdná matice.

```
str_match_all(tweet, r)
```

```
## [[1]]
##      [,1]    [,2]
## [1,] "#Brno" "Brno"
## [2,] "#MU"   "MU"
##
## [[2]]
##      [,1]   [,2]
## [1,] "#LSE" "LSE"
##
## [[3]]
##      [,1]           [,2]
## [1,] "#ParisHilton" "ParisHilton"
##
## [[4]]
##      [,1] [,2]
```

Podívejme se opět na poněkud komplexnější příklad. Vektor `cisla` obsahuje řetězce, které obsahují telefonní čísla. Pro každého člověka chceme získat jeho první telefonní číslo a převést je do standardního tvaru (řekněme, že standardní tvar vynechává předčíslí země a trojice čísel odděluje pomlčkou). Postup může být následující: nejdříve získáme jednotlivé výskyty regulárního výrazu pomocí funkce `str_match()`. Z výsledku si vezmeme pouze skupiny, které odpovídají trojicím čísel (v našem případě třetí, čtvrtý a pátý sloupec výsledku). Nakonec spojíme jednotlivá trojčíslí do jednoho řetězce pomocí funkce `str_c()`; abychom jí zabránili spojit všechna trojčíslí pro všechny lidi dohromady, aplikujeme funkci `str_c()` na jednotlivé řádky matice pomocí funkce `map_chr()` z balíku **purrr**:

```
r <- "(\\+420)?[-\\s]*(\\d{3})[-\\s]*(\\d{3})[-\\s]*(\\d{3})"
cisla <- c("Leoš: 777 666 555 444",
           "Lída: domů +420 734 123 456, práce 777-666-555",
           "Leona: nevím")
cisla2 <- str_match(cisla, r)[, 3:5]
purrr::map_chr(1:nrow(cisla2), ~ str_c(cisla2[., ], collapse = "-"))
```

```
## [1] "777-666-555" "734-123-456" NA
```

13.4.3 Indexy řetězců splnujících vzor

Někdy se hodí najít indexy, kde začíná a končí vzor v daném řetězci. K tomu slouží funkce `str_locate()` a `str_locate_all`. Funkce `str_locate(string, pattern)` najde indexy prvního výskytu regulárního výrazu `pattern` v řetězci `string`. Výsledkem je matice, jejíž řádky odpovídají prvkům vektoru `string`. První sloupec je index začátku prvního výskytu vzoru, druhý sloupec je index konce prvního výskytu vzoru. Pokud není vzor v řetězci nalezen, vrátí `NA`. Nalezené indexy výskytu řetězce je možné následně použít k extrakci daného řetězce pomocí funkce `str_sub()`. To je však ekvivalentní k použití funkce `str_extract()`.

```
r <- "#[[:alpha:]]+"  # hashtag následovaný jedním nebo více písmeny
tweet <- c("#Brno je prostě nejlepší a #MU je nejlepší v Brně.",
           "Někdy je možné najít zajímavé podcasty na #LSE.",
           "Stupnování 'divnosti': divný, divnější, #ParisHilton.",
           "Docela prázdný tweet.")
str_locate(tweet, r)  # vrací matici indexů prvních výskytů klíčových slov v tweetu
```

```
##      start end
## [1,]     1   5
## [2,]    43  46
## [3,]    41  52
## [4,]    NA  NA
```

```
str_sub(tweet, str_locate(tweet, r))  # vyřízne tato klíčová slova
```

```
## [1] "#Brno"        "#LSE"         "#ParisHilton" NA
```

```
str_extract(tweet, r)  # totéž
```

```
## [1] "#Brno"        "#LSE"         "#ParisHilton" NA
```

K nalezení všech výskytů vzoru ve vektoru řetězců `string` slouží funkce `str_locate_all(string, pattern)`, která vrací seznam matic indexů. Prvky seznamu odpovídají prvkům vektoru `string`. Řádky každé matice odpovídají jednotlivým výskytům vzoru v jednom prvku vektoru `string`. První sloupec matice je index začátku výskytu, druhý sloupec je index konce výskytu. Pokud není vzor nalezen, vrací prázdnou matici:

```
str_locate_all(tweet, r)
```

```
## [[1]]
##      start end
## [1,]     1   5
## [2,]    28  30
##
```

```
## [[2]]
##      start end
## [1,]    43  46
##
## [[3]]
##      start end
## [1,]    41  52
##
## [[4]]
##      start end
```

Funkce `invert_match(loc)` invertuje matici indexů vrácenou funkcí `str_locate_all()`, takže obsahuje indexy částí řetězce, které *neodpovídají* vzoru. Detaily najdete v dokumentaci.

13.4.4 Nahrazení vzoru

Regulární výrazy umožňují i nahrazení částí řetězců, které odpovídají danému regulárnímu výrazu. K tomu slouží funkce `str_replace()` a `str_replace_all()`. Funkce `str_replace(string, pattern, replacement)` nahradí ve vektoru řetězců `string` první výskyt vzoru `pattern` řetězcem `replacement`. Funkce `str_replace_all()` má stejnou syntaxi, ale nahradí všechny výskyty vzoru ve vektoru `string`.

Začněme nejjednodušším případem. Řekněme, že chceme v každém tweetu skrýt hashtagy: nahradit je řetězcem "XXX" nebo je úplně vymazat. To můžeme udělat např. takto:

```
r <- "#[[:alpha:]]+"
str_replace_all(tweet, r, "#XXX")  # náhrada hashtagu pomocí #XXX
```

```
## [1] "#XXX je prostě nejlepší a #XXX je nejlepší v Brně."
## [2] "Někdy je možné najít zajímavé podcasty na #XXX."
## [3] "Stupnování 'divnosti': divný, divnější, #XXX."
## [4] "Docela prázdný tweet."
```

```
str_replace_all(tweet, r, "")  # vymazání hashtagu
```

```
## [1] " je prostě nejlepší a  je nejlepší v Brně."
## [2] "Někdy je možné najít zajímavé podcasty na ."
## [3] "Stupnování 'divnosti': divný, divnější, ."
## [4] "Docela prázdný tweet."
```

Pokud chceme nějaký text odstranit, můžeme jej nahradit prázdným řetězcem (jak ukazuje příklad výše), nebo použít užitečnou zkratku, kterou nabízí funkce `str_remove(string, pattern)` a `str_remove_all(string, pattern)`. Ta první odstraní první výskyt regulárního výrazu `pattern` z řetězce `string`, druhá všechny výskyty. Vymazat hashtag můžeme tedy i takto:

```
str_remove_all(tweet, r)
```

```
## [1] " je prostě nejlepší a  je nejlepší v Brně."
## [2] "Někdy je možné najít zajímavé podcasty na ."
## [3] "Stupnování 'divnosti': divný, divnější, ."
## [4] "Docela prázdný tweet."
```

Funkce `str_replace_all()` dokáže nahradit více vzorů naráz. K tomu stačí nahradit vzor `pattern` a nahrazující řetězec `replacement` pojmenovaným vektorem řetězců: jména prvků vektorů určují, co se nahrazuje, hodnoty určují, čím se nahrazuje. Řekněme, že chceme nahradit slova "jeden", "dva" a "tři" odpovídajícími číslicemi:

```
ovoce <- c("jeden banán", "dva pomeranče", "tři mandarinky")
str_replace_all(ovoce,
                c("jeden" = "1", "dva" = "2", "tři" = "3"))
```

```
## [1] "1 banán"      "2 pomeranče"  "3 mandarinky"
```

Regulární výrazy však dokáží víc než jen nahradit kus řetězce jiným fixním řetězcem (např. "XXX"): dokáží náhradu zkonstruovat přímo z nahrazovaného textu. K tomu opět slouží skupiny. Funkce `str_replace()` i `str_replace_all()` si zapamatují obsah skupin obsažených v regulárním výrazu a mohou jej použít v nahrazujícím řetězci `r`. Obsah první skupiny je `\1`, druhé skupiny `\2` atd. (v R je ovšem třeba zpětné lomítko zdvojit). Řekněme, že chceme hashtagy v tweetech upravit tak, že hashtag bude nejdříve uveden bez křížku, a pak v závorce s křížkem:

```
r <- "#([[:alpha:]]+)"
str_replace_all(tweet, r, "\\1 (#\\1)")
```

```
## [1] "Brno (#Brno) je prostě nejlepší a MU (#MU) je nejlepší v Brně."
## [2] "Někdy je možné najít zajímavé podcasty na LSE (#LSE)."
## [3] "Stupnování 'divnosti': divný, divnější, ParisHilton (#ParisHilton)."
## [4] "Docela prázdný tweet."
```

Paměť v regulárních výrazech se dá použít na celou řadu praktických úprav řetězců. Řekněme například, že máme vektor datumů v anglosaském formátu "MM-DD-YYYY", a chceme je převést do českého formátu "DD. MM. YYYY". Bez regulárních výrazů to může být pracná záležitost. S použitím regulárních výrazů je to hračka:

```
datumy <- c("born: 06-01-1921", "died: 02-01-2017", "no information at all")
str_replace_all(datumy, "(\\d{2})-(\\d{2})-(\\d{4})", "\\2. \\1. \\3")
```

```
## [1] "born: 01. 06. 1921"    "died: 01. 02. 2017"    "no information at all"
```

13.4.5 Rozdělení řetězců podle vzoru

Často je potřeba rozdělit řetězec do několika částí oddělených nějakým vzorem. K tomu slouží funkce `str_split(string, pattern, n = Inf, simplify = FALSE)` a `str_split_fixed(string, pattern, n)`, které rozdělí řetězec `string` v bodech, kde je nalezen vzor `pattern`. Celé číslo `n` určuje, na kolik částí je řetězec rozdělen. Funkce `str_split()` nepotřebuje mít `n` zadané (implicitně rozdělí řetězec do tolika částí, do kolika je potřeba). Funkce vrací seznam, jehož každý prvek odpovídá jednomu prvku vektoru `string`. Funkce `str_split_fixed()` musí mít zadaný počet `n` a vrací matici, jejíž řádky odpovídají prvkům vektoru `string` a sloupce jednotlivým nálezům (přebytečné sloupce jsou naplněné prázdným řetězcem `""`). Pokud je počet `n` nedostatečný, je nerozdělený zbytek řetězce vložen do posledního zadaného sloupce. Pokud však funkci `str_split()` nastavíme parameter `simplify` na `TRUE`, pak také zjednoduší svůj výsledek do matice.

```
ovoce <- c("jablka a hrušky a hrozny", "pomeranče a fíky a datle a pomela")
str_split(ovoce, " a ")
```

```
## [[1]]
## [1] "jablka" "hrušky" "hrozny"
##
## [[2]]
## [1] "pomeranče" "fíky"      "datle"     "pomela"
```

```
str_split(ovoce, " a ", 3)
```

```
## [[1]]
## [1] "jablka" "hrušky" "hrozny"
##
## [[2]]
## [1] "pomeranče"      "fíky"           "datle a pomela"
```

```
str_split_fixed(ovoce, " a ", 4)
```

```
##      [,1]        [,2]     [,3]     [,4]
## [1,] "jablka"    "hrušky" "hrozny" ""
## [2,] "pomeranče" "fíky"   "datle"  "pomela"
```

```
str_split_fixed(ovoce, " a ", 3)
```

```
##      [,1]        [,2]     [,3]
## [1,] "jablka"    "hrušky" "hrozny"
## [2,] "pomeranče" "fíky"   "datle a pomela"
```

```
str_split(ovoce, " a ", simplify = TRUE)
```

```
##      [,1]        [,2]     [,3]     [,4]
## [1,] "jablka"    "hrušky" "hrozny" ""
## [2,] "pomeranče" "fíky"   "datle"  "pomela"
```

Řekněme, že potřebujeme rozdělit řetězce, které obsahují pouze dobře formátované datum v české konvenci na den, měsíc a rok. To můžeme udělat např. takto:

```
datum <- c("1.6.2001", "1. 2. 2003", "blábol")
str_split_fixed(datum, "\\.\\s*", 3)
```

```
##      [,1]     [,2] [,3]
## [1,] "1"      "6"  "2001"
## [2,] "1"      "2"  "2003"
## [3,] "blábol" ""   ""
```

Pokud řetězce s daty nejsou formátované dobře, ale mohou být v jednom z formátů "DD.MM.YYYY", "DD. MM. YYYY" nebo "DD-MM-YYYY", pak stačí jen zobecnit regulární výraz, který jednotlivé složky data odděluje:

```
datum <- c("10-5-1999", "1.6.2001", "1. 12. 2003", "blábol")
str_split_fixed(datum, "(\\.\\s*|-)", 3)
```

```
##      [,1]     [,2] [,3]
## [1,] "10"     "5"  "1999"
## [2,] "1"      "6"  "2001"
## [3,] "1"      "12" "2003"
## [4,] "blábol" ""   ""
```

(Pro práci s volně formátovanými daty má mnoho užitečných funkcí balík **lubridate**, viz oddíl 6.2.)

Někdy chceme s řetězci pracovat po slovech. K tomu slouží funkce `word()`, která rozdělí řetězec na slova a vrátí slova se zadanými indexy. Použití je

```
word(string, start = 1L, end = start, sep = fixed(" "))
```

kde `string` je vektor rozdělovaných řetězců, `start` je index prvního vraceného slova, `end` je index posledního vraceného slova a `sep` je regulární výraz, který odděluje slova (implicitně je to jedna mezera). Indexy mohou být i záporné – pak se počítají odzadu, tj. `-1` je poslední slovo. Všechny parametry se recyklují. Na příklady se podívejte do dokumentace.

13.5 Modifikace chování regulárních výrazů

Chování regulárních výrazů je možné ve funkcích z balíku **stringr** modifikovat pomocí čtyř funkcí: `regex()`, `fixed()`, `coll()` a `boundary()`. Nejdůležitější z nich je funkce `regex()`. Pokaždé, když zadáte regulární výraz, funkce z balíku **stringr** na něj tiše uplatní funkci `regex()`, takže dva následující výrazy jsou zcela ekvivalentní:

```
s1 <- c("máma", "Máma", "MÁMA")
str_detect(s1, "(.)á\\1a")
```

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

```
str_detect(s1, regex("(.)á\\1a"))
```

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

Funkce `regex()` umožňuje poněkud modifikovat chování regulárního výrazu. K tomu jí slouží čtyři parametry. Parametr `ignore_case` umožňuje vypnout rozdíl mezi malými a velkými písmeny (*case sensitivity*):

```
str_detect(s1, regex("(.)á\\1a", ignore_case = TRUE))
```

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

Normálně v regulárních výrazech v balíku **stringr** znamenají znaky `^` a `$` začátek a konec řetězce. Pokud mají znamenat začátek a konec řádku (což je obvyklejší), je třeba použít parametr `multiline`:

```
s2 <- "1. nakoupíš 7 vajec.\n2. umyješ podlahu\n3. nakrmíš králík"
cat(s2)
```

```
## 1. nakoupíš 7 vajec.
## 2. umyješ podlahu
## 3. nakrmíš králík
```

```
str_extract_all(s2, "^\\d+\\.")
```

```
## [[1]]
## [1] "1."
```

```
str_extract_all(s2, regex("^\\d+\\.", multiline = TRUE))
```

```
## [[1]]
## [1] "1." "2." "3."
```

Zbývající dva parametry jsou méně důležité: parametr `dotall` způsobí, že `.` zahrne všechny znaky, včetně konce řádku `\n`. Parametr `comments` umožňuje přidávat do regulárního výrazu komentáře.

Alternativou funkce `regex()` je funkce `fixed()`. Ta zajistí, že se výraz bere doslovně. Řekněme, že chceme spočítat počet souvětí v řetězci a že počet souvětí aproximujeme počtem teček (naše souvětí vždy končí tečkou, ne vykřičníkem nebo otazníkem, a text neobsahuje žádné zkratky, pořadová čísla ani tečky nevyužívá nijak jinak než na konci vět). Abychom našli všechny výskyty tečky v řetězci, můžeme ji buď zbavit speciálního významu pomocí zpětného lomítka, nebo použít funkci `fixed()`:

```
s3 <- "Máma má máso. Ema má mísu. Ó my se máme."
str_count(s3, "\\.")
```

```
## [1] 3
```

```
str_count(s3, fixed("."))
```

```
## [1] 3
```

Podobně jako `regex()` je i ve funkci `fixed()` možné vypnout rozdíl mezi malými a velkými písmeny pomocí parametru `ignore_case`.

Funkce `fixed()` porovnává řetězec doslovně, takže znaky bere doslovně jako byty, což nemusí vždy správně fungovat pro znaky, které nejsou součástí kódu ASCII (zhruba pro znaky, které nejsou součástí anglické abecedy). Pokud chceme takové znaky brát doslovně, je potřeba použít funkci `cols()`, které je navíc možné nastavit *locale* jazyka (implicitní jazyk je angličtina).

Poslední funkce pro modifikaci chování regulárních výrazů je funkce `boundary()`, která umožňuje nastavit okraje regulárního výrazu. Platné volby jsou "character", "line_break", "sentence" a "word"). Nejužitečnější je `boundary("word")`, které hledá hranice mezi slovy:

```
s4 <- "Svítilo zářilo zlaté slunce na pobřeží na lagunce."
str_split(s4, boundary("word"))
```

```
## [[1]]
## [1] "Svítilo" "zářilo"  "zlaté"   "slunce"  "na"      "pobřeží" "na"
## [8] "lagunce"
```

```
str_extract_all(s4, boundary("word"))
```

```
## [[1]]
## [1] "Svítilo" "zářilo"  "zlaté"   "slunce"  "na"      "pobřeží" "na"
## [8] "lagunce"
```

14 Co je to tidyverse?

tidyverse je soubor balíků obashujících všechny nástroje potřebné pro základní úkony v datové analýze. Balíky z *tidyverse* Vám umožní:

- načíst data (balík *reader*, *haven*, *xml2*, *rvest* a další),
- přeskládat je do požadované formy (balík *tidyr*),
- modifikovat je (balík *dplyr*)
- a vizualizovat (balík *ggplot2*).

Součástí *tidyverse* jsou i další balíky. Některé z nich Vám jsou již známé (např. *purrr*, *stringr*, *forcats* nebo *tibble*). Všechny balíky v *tidyverse* sdílejí logiku ovládání a datové struktury. Dohromady tak poskytují uživateli dobře provázané, kooperující a moderní prostředí pro datovou analýzu. Nevýhodou *tidyverse* je, že jejich vývoj stále není zcela ukončen a dohcází v něm k zásadním změnám.

Balíky z tidyverse je možné načítat jednotlivě podle potřeby, nebo načtením metabalíku *tidyverse*:

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------------------------------------
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.4     v dplyr   1.0.7
## v tidyr   1.1.3     v stringr 1.4.0
## v readr   2.0.1     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------------------------------------------ tid
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

14.1 Práce s tidyverse

14.1.1 Trubky/pipes

S pomocí *tidyverse* je možné psát velmi dobře srozumitelný kód. Hadley Wickham, který stojí za vytvořením podstatné části *tidyverse*, při popisu balíků mluví o konceptu "gramatiky práce s daty" a "gramatice vizualizace dat".

Průběh datové analýzy můžeme popsat jako sekvenci úkonů. Například:

1. Načti data.
2. Vezmi načtenou tabulku a vyber z ní několik sloupců.
3. Vezmi upravenou tabulku a vyber pouze řádky, které splňují určité podmínky.
4. Vezmi upravenou tabulku a na jejím základě vykresli obrázek.
5. Výsledek ulož do proměnné `x`.

Každý krok přitom využívá výsledek kroku předchozího. Pro podobné řetezení kroků využívá *tidyverse* speciální funkci %>% (tzv. trubku/pipe), které spojuje dva kroky. Výstup prvního kroku vkládá jako vstup do následujícího kroku. Předchozí sekvenci kroků bychom tedy mohli schématicky zapsat následujícím způsobem:

```
x <- read_tsv() %>%  # Načti data
  select() %>%       # Vyber sloupce
  filter() %>%       # Vyber řádky
  ggplot()           # Vykresli data
```

Ukázka kódu není funkční – funkce nemají zadané žádné parametry, ale ukazuje logiku celého přístupu. Výsledný kód svou strukturou připomíná strukturu psaného jazyka. Intrukce přehledně plyne zleva do prava, podobně jako text v knize.

Bez použití funkce %>% by postup vypadal následovně:

```
x <- read_tsv()
x <- select(x)
x <- filter(x)
x <- ggplot(x)
```

Výsledek by byl zcela identický (a provedení kódu u velkých tabulek o maličko rychlejší). Výhodou trubek je však čitelnost a přehlednost. Praxe ukazuje, že mít jasný a čistý kód je v praxi datové analýzy často důležitější než marginální ztráta rychlosti.

V tomto kurzu budeme od této chvíle často používat funkci %>%. Pokud si oblíbíte používání trubek, můžete se podívat na balík `magrittr`, který kromě %>% implementuje i další trubkoidní funkce.

14.1.1.1 Placeholder "."

Funkce %>% ja nastavená tak, že obsah, který ji protéká vždy umisťuje do prvního parametru přijímající funkce. To je skoro vždy žádoucí chování, ale ne vždy. Modifikovat je ho možné s použítím symbolu `.`, který určuje místo, na které má trubka přenášený obsah umístit.

Můžeme si například představit situaci, kdy máme vektory `x` a `w`. Chceme spočítát vážený průměr `x`, ale trubkou nám putuje vektor vah `w`. Požadovaného výsledku dosáhneme takto:

```
w %>% weighted.mean(x,.)
```

14.1.2 Subsetování

tidyverse, konkrétně balík *dplyr* obsahuje funkce pro subsetování. Zběžně se nyní seznámí s funkcí `select()`, která slouží k výběru sloupců. Na jejím příkladě si ukážeme, jak se v *tidyverse* zachází se jmény sloupců.

Základní syntaxe `select()` je triviální:

```
select(.data, ...)
```

Prvním parametrem je vstupní tabulka. V parametru `...` jsou specifikovány sloupce.

Jako ukázkovou tabulku budeme používat `us_rent_income` z balíku *tidyr*:

```
us_rent_income
```

```
## # A tibble: 104 x 5
##    GEOID NAME       variable estimate   moe
##    <chr> <chr>      <chr>       <dbl> <dbl>
```

```
##  1 01    Alabama    income      24476   136
##  2 01    Alabama    rent          747     3
##  3 02    Alaska     income      32940   508
##  4 02    Alaska     rent         1200    13
##  5 04    Arizona    income      27517   148
##  6 04    Arizona    rent          972     4
##  7 05    Arkansas   income      23789   165
##  8 05    Arkansas   rent          709     5
##  9 06    California income      29454   109
## 10 06    California rent         1358     3
## # ... with 94 more rows
```

Poznámka: Speciální metody výběru sloupců jsou implementovány v balíku *tidyselect*, Ten si však nemusíte vždy loadovat. Balíky z *tidyverse* si vše vyřeší samy.

14.1.2.1 Non-standard evaluation a identifikace jménem sloupce

tidyverse používá tzv. nestandardní evaluaci. Ta je výhodná zvláště při interaktivní práci s daty. Praktickým dopadem je, že se jména sloupců píšou bez úvozovek. První možností jak identifikovat sloupec je tedy jménem bez úvozovek. Můžeme vybrat jeden:

```
us_rent_income %>%
  select(variable)
```

```
## # A tibble: 104 x 1
##    variable
##    <chr>
##  1 income
##  2 rent
##  3 income
##  4 rent
##  5 income
##  6 rent
##  7 income
##  8 rent
##  9 income
## 10 rent
## # ... with 94 more rows
```

Nebo více sloupců:

```
us_rent_income %>%
  select(variable,estimate)
```

```
## # A tibble: 104 x 2
##    variable estimate
##    <chr>       <dbl>
##  1 income      24476
##  2 rent          747
##  3 income      32940
##  4 rent         1200
##  5 income      27517
##  6 rent          972
##  7 income      23789
##  8 rent          709
```

```
##  9 income      29454
## 10 rent         1358
## # ... with 94 more rows
```

14.1.2.2 Identifikace pozicí

Vedle jména je možné pro identifikaci sloupce použít číslo jeho pozice:

```
us_rent_income %>%
  select(1,2,4)
```

```
## # A tibble: 104 x 3
##    GEOID NAME       estimate
##    <chr> <chr>         <dbl>
##  1 01    Alabama       24476
##  2 01    Alabama         747
##  3 02    Alaska        32940
##  4 02    Alaska         1200
##  5 04    Arizona       27517
##  6 04    Arizona         972
##  7 05    Arkansas      23789
##  8 05    Arkansas        709
##  9 06    California    29454
## 10 06    California     1358
## # ... with 94 more rows
```

14.1.2.3 Speciální funkce

Funkce, které potřebují specifikaci sloupců umí pracovat se speciálními funkcemi, které budou fungovat pouze v jejich rámci. Základní speciální funkce jsou dvě: `-` a `:`.

Funkce `-` umožňuje negativní výběr – "vyber všechno až na". Následující volání funkce `select()` tak vrátí všechny sloupce až na `GEOID` a `NAME`:

```
us_rent_income %>%
  select(-GEOID,-NAME)
```

```
## # A tibble: 104 x 3
##    variable estimate   moe
##    <chr>       <dbl> <dbl>
##  1 income      24476   136
##  2 rent          747     3
##  3 income      32940   508
##  4 rent         1200    13
##  5 income      27517   148
##  6 rent          972     4
##  7 income      23789   165
##  8 rent          709     5
##  9 income      29454   109
## 10 rent         1358     3
## # ... with 94 more rows
```

Funkce `:` umožňuje specifikovat rozsah sloupců. To lze využít pokud chceme například vybrat sloupce `GEOID` a všechny sloupce mezi `variable` a `moe`:

```
us_rent_income %>%
  select(GEOID, variable:moe)
```

```
## # A tibble: 104 x 4
##    GEOID variable estimate   moe
##    <chr> <chr>       <dbl> <dbl>
##  1 01    income      24476   136
##  2 01    rent          747     3
##  3 02    income      32940   508
##  4 02    rent         1200    13
##  5 04    income      27517   148
##  6 04    rent          972     4
##  7 05    income      23789   165
##  8 05    rent          709     5
##  9 06    income      29454   109
## 10 06    rent         1358     3
## # ... with 94 more rows
```

Speciální funkce fungují i při identifikace sloupců jejich pozicí…

```
us_rent_income %>%
  select(-1,-2)
```

```
## # A tibble: 104 x 3
##    variable estimate   moe
##    <chr>       <dbl> <dbl>
##  1 income      24476   136
##  2 rent          747     3
##  3 income      32940   508
##  4 rent         1200    13
##  5 income      27517   148
##  6 rent          972     4
##  7 income      23789   165
##  8 rent          709     5
##  9 income      29454   109
## 10 rent         1358     3
## # ... with 94 more rows
```

```
us_rent_income %>%
  select(1,3:5)
```

```
## # A tibble: 104 x 4
##    GEOID variable estimate   moe
##    <chr> <chr>       <dbl> <dbl>
##  1 01    income      24476   136
##  2 01    rent          747     3
##  3 02    income      32940   508
##  4 02    rent         1200    13
##  5 04    income      27517   148
##  6 04    rent          972     4
##  7 05    income      23789   165
##  8 05    rent          709     5
##  9 06    income      29454   109
```

```
## 10 06    rent         1358     3
## # ... with 94 more rows
```

...a je možné je kombinovat

```
us_rent_income %>%
  select(-variable:-moe)
```

```
## # A tibble: 104 x 2
##    GEOID NAME
##    <chr> <chr>
##  1 01    Alabama
##  2 01    Alabama
##  3 02    Alaska
##  4 02    Alaska
##  5 04    Arizona
##  6 04    Arizona
##  7 05    Arkansas
##  8 05    Arkansas
##  9 06    California
## 10 06    California
## # ... with 94 more rows
```

14.1.2.4 Funkce pomocníčci: select helpers

Balík *tidyselect* obshauje funkce, které umožňují identifikovat sloupce jinak než přímým zadáním jména nebo pozice:

- `starts_with()` vybírá sloupce, jejichž jméno začíná na řetězec, který je argumentem funkce `starts_with()`
- `ends_with()` vybírá sloupce, jejichž jméno končí na řetězec, který je argumentem funkce `ends_with()`
- `contains()` vybírá sloupce, jejichž jméno obsahuje řetězec, který je argumentem funkce `contains()`
- `matches()` vybírá sloupce, jejichž jméno odpovídá zadanému regulárnímu výrazu
- `num_range()` slouží pro výběr sloupců, jejichž jméno je tvořeno kombinací řetězce a čísla – například `trial_1`, `trial_2`,...
- `one_of()` vrátí sloupce, jejichž jména jsou obsažena ve vektoru, který je vstupem funkce
- `everything()` vrací všechny sloupce
- `last_col()` vrací poslední sloupec

Tyto funkce opět fungují jenom "uvnitř" kompatibilních funkcí. K subsetování se v mírně větším detailu vrátíme znovu v kapitole věnované balíku *dplyr*.

15 Správně formátovaná data a balík `tidyr`

"It is often said that 80 % of data analysis is spent on the cleaning and preparing data."

"Tidy datasets are all alike but every messy dataset is messy in its own way."

– Hadley Wickham

Představte si výzkum vývoje dětí. Z populace si vybereme vzorek dětí, které budeme sledovat a následně u každého z nich každý měsíc naměříme řadu ukazatelů: výšku, váhu, počet červených krvinek, motorické a kognitivní schopnosti, počet prstů, atp.

Získáme tak soubor dat s mnoha pozorováními a mnoha dimenzemi. Jedno pozorování můžeme chápat jako moment měření – definuje ho tedy identita pozorovaného subjektu (průřezová jednotka) a čas pozorování (věk). Každá sledovaná charakteristika potom představuje samostatnou dimenzi.

Množství pozorování a dimenzí umožňuje nejrůznější organizaci naměřených dat. Data jsou typicky organizována do formátu pravoúhlé tabulky, kde jsou data zapsána v buňkách organizovaných v řádcích a sloupcích. Tabulky však mohou být různě vnitřně organizované. Mohou se lišit v tom, které údaje se zapisují do sloupců, které do řádků a podobně

V této lekci se naučíte:

- Jak vhodně organizovat data do tabulek – tzv. *tidy* formát
- S pomocí nástrojů z balíku *tidyr* upravovat data do *tidy* formátu

Balík *tidyr* je součástí *tidyverse* a je třeba si ho nejprve načíst do paměti.

15.1 Tidy data

Formát *"tidy data"* popisuje organizační strukturu dat v tabulkách. Data v tidy formátu splňují následující charakteristiky:

1. Každé pozorování je popsáno jedním řádkem
2. Každá proměnná je obsažena v jednom sloupci
3. Každý typ pozorování má vlastní tabulku

Wickham (2016) ilustruje tidy fromát pomocí následujícího schématu:

Uvažujme příklad statistik o trhu práce. Statistický úřad sleduje na roční bázi počet nezaměstnaných a velikost dopělé populace pro obce, okresy a kraje. Pokud by ukládal data v tidy struktuře potom by:

1. Data byla skladována ve třech tabulkách – v jedné tabulce by byly údaje pro kraje, v druhé pro okresy a ve třetí pro obce.
2. Struktura každé tabulky by byla následující:

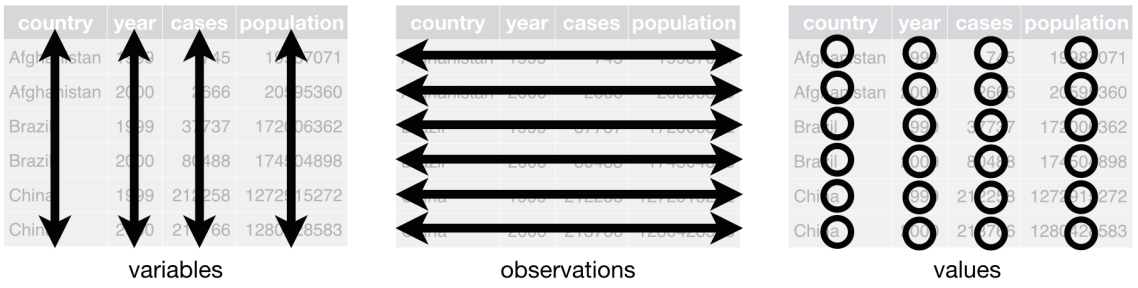


Figure 15.1: Struktura tidy dat (Wickham, 2016)

```
## # A tibble: 6 x 4
##   area        year adult_population unemployment_rate
##   <chr>      <int>            <dbl>             <dbl>
## 1 Kostelec    2001              301               7.6
## 2 Kostelec    2002              305               6
## 3 Kostelec    2003              295               8.5
## 4 Valtrovice  2001              656               6.3
## 5 Valtrovice  2002              650               9.3
## 6 Valtrovice  2003              660               7.8
```

Každé pozorování je identifikováno správní jednotkou (`area`) a rokem (`year`). Každá sledovaná proměnná je potom uložena ve vlastním sloupci.

15.2 Transformace tabulek do tidy formátu

Ne všechny dostupné datasety jsou organizované v tidy formátu. Součástí *tidyverse* je balíček *tidyr*, který obsahuje nástroje pro transformaci tabulek do tidy formátu.

Základními funkcemi v balíku *tidyr*, které zvládnou většinu obvyklých problémů, jsou `pivot_wider()` a `pivot_longer()`.

15.2.0.1 Poznámka 1

Funkce `pivot_wider()` a `pivot_longer()` přišly do *tidyr* od verze 1.0.0 (podzim 2019). Ve střeších verzích balíku jejich funkci plnily funkce `gather()` a `spread()`, které byly vlastně speciálním případem obecnějších `pivot_*` funkcí. Funkce `gather()` a `spread()` byly v balíku podrženy pouze pro zachování zpětné kompatibility.

15.2.0.2 Poznámka 2

Existují balíky, které mají podobné funkcionality jako *tidyr*. Zejména jde o *reshape* a *reshape2*. Oba balíky jsou v podstatě starými evolučními verzemi balíku *tidyr*. Zejména *reshape2* je stále v závislostech mnoha dalších balíků. *tidyr* je však obecně pokročilejší (co do rychlosti, elegance i rozsahu funkcí).

15.2.1 Transformace mnoha sloupců do jednoho s funkcí `pivot_longer()`

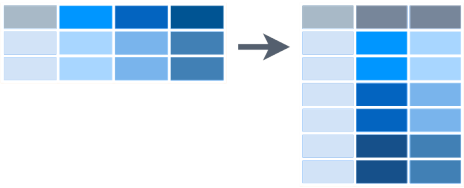
Mnoho datasetů obsahuje sloupce, jejichž jména nejsou samostatné proměnné, ale ve skutečnosti jde o hodnoty jedné proměnné. Jako příklad můžeme použít tabulku `table4a` z balíku *tidyr*, která zachycuje počet pozorovaných případů v několika letech a zemích:

```
print(table4a)
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

V tomto formátu obsahuje jeden řádek hned dvě pozorování (dvě pozorování z jedné země) a dva sloupce obsahují stejnou proměnnou (počet případů). Pro transformaci takové tabulky do tidy formátu slouží funkce `pivot_longer()`.

`pivot_longer()` skládá hodnoty z vybraných sloupců do nově vytvořeného sloupce. Jména vybraných sloupců vytvoří další dodatečný sloupec. `pivot_longer()` tak nahradí vybrané sloupce dvěma novými:



Výsledná tabulka je proto "delší". Pro snadnější zapamatování se se proto funkce jmenuje `pivot_longer()`.

Ukázkou praktické aplikace `pivot_longer()` může být transformace tabulky `table4a`:

```
table4a %>%
    pivot_longer(-country)
```

```
## # A tibble: 6 x 3
##   country     name   value
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

Funkce `pivot_longer()` provedla transformaci ilustrovanou následujícím obrázkem:

Původní tabulka měla 3 řádky. Nyní jich má 6. Každý původní řádek se rozpadl na dva nové.

Funkce `pivot_longer()` má následující syntax a parametry (více viz `?pivot_longer`):

```
pivot_longer(data, cols,
             names_to = "name",
             names_prefix = NULL,
             names_sep = NULL,
             names_pattern = NULL,
             names_ptypes = list(),
             names_repair = "check_unique",
             values_to = "value",
             values_drop_na = FALSE,
             values_ptypes = list()
             )
```

Figure 15.2: Fungování `gather()` (Wickham, 2016)

Základní parametry jsou následující: - `data`... vstupní tabulka (data frame), která má projít transformací, - `cols`... identifikace sloupců, které mají být transformovány, - `names_to`... jméno sloupce, který ve výsledné tabulce bude obsahovat jména transformovaných sloupců, - `values_to`... jméno sloupce, který v transformované tabulce bude obsahovat hodnoty z původních sloupců.

Nyní se vraťme k úvodnímu příkladu:

```
table4a %>%
    pivot_longer(-country)
```

Vstupem do funkce `pivot_longer()` byla tabulka `table4a`. Parametr `cols` byl s využitím pomocné funkce – nastaven na hodnotu `-country`. To znamená, že transformovány byly všechny sloupce až na `country`. Nově vytvořená tabulka má tři sloupce: netransformovaný sloupec `country` a nově vytvořené sloupce `name` a `value`. Jména těchto sloupců jsou dána defaultním nastavením funkce `pivot_longer()`.

Nyní se podíváme na složitější případ. Pro ilustraci upravenou tabulku `table4a`, které přidáme sloupec obsahující ID:

```
## # A tibble: 3 x 4
##   country     `1999` `2000`    id
## * <chr>        <int>  <int> <int>
## 1 Afghanistan    745   2666     1
## 2 Brazil       37737  80488     2
## 3 China       212258 213766     3
```

Tabulku chceme transformovat tak, aby se sloupce s hodnotami (1999 a 2000) transformovaly do sloupců `year` a `value`. Je jasné, že chceme sáhnout po `pivot_longer()`:

```
table4a %>%
  # Tento řádek vytvoří nový sloupec s číslem řádku
  mutate(id = row_number()) %>%
  pivot_longer(-country, -id)
```

Toto nebude fungovat, protože parametr `cols` má jenom jednu pozici – v jejím rámci musíme identifikovat všechny sloupce, které se mají transformovat.

```
# Použití negativní identifikace
table4a %>%
  mutate(id = row_number()) %>%
  pivot_longer(-c(country,id))
```

```
## # A tibble: 6 x 4
##   country        id name   value
##   <chr>       <int> <chr>  <int>
## 1 Afghanistan     1 1999     745
## 2 Afghanistan     1 2000    2666
## 3 Brazil          2 1999   37737
## 4 Brazil          2 2000   80488
## 5 China           3 1999  212258
## 6 China           3 2000  213766
```

V tomto případě byl do jednomístného slotu vložen vektor vytvořený funkcí `c()`. Všimněte si, že i v něm jsou jména sloupců uvedena bez úvozovek.

Hint: V reálném nasazení je vždy užitečné zvážit použití negtivní identifikace sloupců. Představte si například situaci, kdy Vašim zdrojeme je databáze, do které každý rok přibude další sloupec. Pozitivní identifikace sloupců by způsobila, že Vaše skripty by po prvním updatu přestaly správně fungovat. Negativní identifikace tímto problémem netrpí.

Výše uvedená možnost není jediná možná. Následující příklady by vedly ke stejným výsledkům:

```
# Použití pozitivní identifikace
table4a %>%
  mutate(id = row_number()) %>%
  pivot_longer(c(`1999`,`2000`))

# Použití pozitivní identifikace a speciální funkce `:`
table4a %>%
  mutate(id = row_number()) %>%
  pivot_longer(`1999`:`2000`)

# Použití select helpers
table4a %>%
  mutate(id = row_number()) %>%
  pivot_longer(matches("\\d{4}"))
```

15.2.2 Transformace dvojice sloupců do mnoha sloupců s funkcí `pivot_wider()`

Funkce `pivot_wider()` je inverzní k funkci `pivot_longer()`. Použijeme ji v případě, že sloupec ve skutečnosti neobsahuje hodnoty jedné proměnné, ale hodnoty mnoha proměnných. Funkce `pivot_wider()` transforumje dvojici sloupců do mnoha nových sloupců. Hodnoty prvního z původních sloupců obsahují určení proměnné a v druhém sloupci jsou uloženy jejich hodnoty.

Příkladem takového datasetu může být tabulka `table2` z balíku *tidyr*:

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
```

```
## 11 China         2000 cases          213766
## 12 China         2000 population 1280428583
```

Pozorování je opět identifikováno hodnotami ve sloupcích `country` a `year`. Nicméně jedno pozorování je roztaženo do dvou řádků a hodnoty pro počet případů (`count`) a velikost populace (`population`) jsou obsaženy v jednom sloupci `count`.

Pro převední takové tabulky do tidy fromátu je potřeba provést operaci popsanou následujícím schématem:

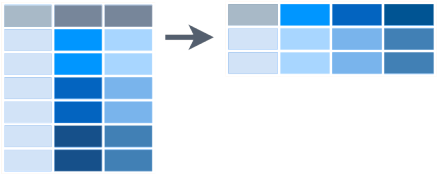


Figure 15.3: Fungování `pivot_wider()` (RStudio, 2015)

Funkce `pivot_wider()` použije hodnoty z prvního sloupce (`key`) jako jména nově vytvořených sloupců. Nově vytvořené buňky jsou potom vyplněny hodnotami z druhého sloupce (`value`) v původní tabulce:

```
table2 %>%
    pivot_wider(names_from = type, values_from = count)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Funkce `pivot_wider()` provedla transformaci ilustrovanou následujícím obrázkem:

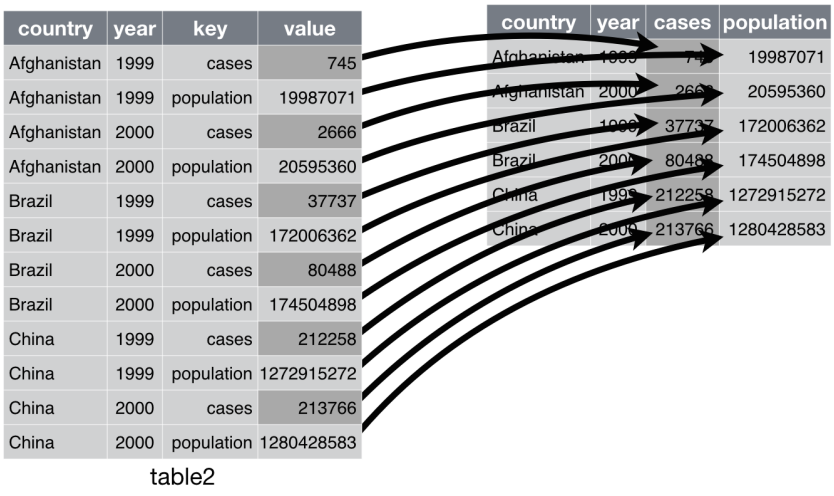


Figure 15.4: Fungování `pivot_wider()` (Wickham, 2016)

Funkce `pivot_wider()` má následující syntax a parametry (více viz `?pivot_wider`):

```
pivot_wider(data,
            id_cols = NULL,
            names_from = name,
            names_prefix = "",
            names_sep = "_",
            names_repair = "check_unique",
            values_from = value,
            values_fill = NULL,
            values_fn = NULL
            )
```

Klíčové parametry jsou následující:

- `data`...vstupní tabulka (data frame), která má projít transformací
- `id_cols`...sloupce, které identifikují jedno pozorování. V defaultní nastevením (tedy `NULL`) se k tomuto účelu použijí všechny sloupce, které neprocházejí transformací.
- `names_from`...sloupec (nebo sloupce), ze kterého se mají vytvořit jména nově vytvořených sloupců,
- `values_from`...sloupec (nebo sloupce), ze kterých se mají vzít hodnoty pro naplnění nově vytvořených sloupců.

Jak to funguje, pokud je parametr `names_from` delší než 1?

Předpokládejme, že chceme vytvořit tabulku, ve které budou sloupce definovány kombinací roku a proměnné:

```
table2 %>%
    pivot_wider(names_from = c(type,year), values_from = count)
```

```
## # A tibble: 3 x 5
##   country     cases_1999 population_1999 cases_2000 population_2000
##   <chr>            <int>           <int>      <int>           <int>
## 1 Afghanistan        745        19987071       2666        20595360
## 2 Brazil           37737       172006362      80488       174504898
## 3 China           212258      1272915272     213766      1280428583
```

Jméno sloupců je teď vytvořené z kombinace `type` a `year`. Výslednou podobu jmen upravuje parametr `names_sep` z `pivot_wider()`.

15.2.3 Praktické procvičení `pivot_longer()` a `pivot_wider()` I

Uvažujme tabulku vytvořenou v předchozím případě. Jak z ní vytvoříme tidy dataset?

Tabulka byla vytvořena s `pivot_wider()`. V prvním kroku tedy sáhneme tedy po inverzní funkci `pivot_longer()`.

V tomto příkladě využijeme řadu zatím nediskutovaných parametrů `pivot_longer()`:

```
table2 %>%
    pivot_wider(names_from = c(type,year), values_from = count) %>%
    pivot_longer(-country,
                 names_to = c("type","year"),
                 names_sep = "_",
                 names_transform = list(year = as.integer),
                 names_ptypes = list(type = character()),
                 values_to = "count"
                 )
```

```
## # A tibble: 12 x 4
##    country     type        year      count
##    <chr>       <chr>      <int>      <int>
##  1 Afghanistan cases       1999        745
##  2 Afghanistan population  1999   19987071
##  3 Afghanistan cases       2000       2666
##  4 Afghanistan population  2000   20595360
##  5 Brazil      cases       1999      37737
##  6 Brazil      population  1999  172006362
##  7 Brazil      cases       2000      80488
##  8 Brazil      population  2000  174504898
##  9 China       cases       1999     212258
## 10 China       population  1999 1272915272
## 11 China       cases       2000     213766
## 12 China       population  2000 1280428583
```

Co se v nově použitých parametrech stalo? Parametr `names_to` je nyní délky 2. To znamená, že jména transformovaných sloupců se rozpadnou do dvou sloupců se zadanými jmény. Jak se má tento rozpad provést určuje parametr `names_sep` (u složitějších případů `names_pattern`). V současném nastavení říká, že znaky před _ mají být přeneseny do sloupce `type` a znaky za _ do sloupce `year`. Bylo by také vhodné, aby hodnoty ve sloupci `type` byly character a ve sloupci `year` integer. Tato konverze se nastavuje parametrem `names_ptypes`. Ten obsahuje pojmenovaný list. Jména odpovídají jménům nově vytvořených sloupců ke ketrým je přiřazen prázdný vektor požadovaného datového typu.

Tato transformace zvrátila účinky `pivot_wider()`. Nicméně data stále nejsou tidy. Potřebujeme mít dva nové sloupce `cases` a `population` s hodnotami ze sloupce `count`. A job for `pivot_wider()`:

```
table2 %>%
    pivot_wider(names_from = c(type,year), values_from = count) %>%
    pivot_longer(-country,
                 names_to = c("type","year"),
                 names_sep = "_",
                 names_transform = list(year = as.integer),
                 names_ptypes = list(type = character()),
                 values_to = "count"
                 ) %>%
  pivot_wider(
    id_cols = c(country,year), # V tomto případě ekvivalenttní k NULL
    names_from = type,
    values_from = count
  )
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

. . . and here we go.

Výsledek odpovídá formátu tidy data.

15.3 Další funkce z *tidyr*

Kromě funkcí `pivot_*()`, které jsou bezesporu nejvíce používané při čistění a transformaci dat, obsahuje *tidyr* řadu dalších funkcí, které pomáhají s:

1. Odstraněním méně obvyklých případů při transformaci tabulek do tidy fromátu.
2. Nakládáním s chybějícími hodnotami.
3. Konstrukcí vlastních tabulek.

15.3.1 Různé typy pozorování v jedná tabulce: `nest()` a `unnest()`

Definice tidy formátu vyžaduje, aby byl každý typ pozorování uchován v oddělené tabulce.

Tabulka `population_world` obsahuje data, která toto kritérium nesplňují. Obsahuje pozorování jak za jednotlivá pozorování, tak jejich agregované hodnoty:

```
print(population_world)
```

```
## # A tibble: 9 x 5
##   country observation  year Female  Male
##   <chr>   <chr>       <int>  <dbl> <dbl>
## 1 Iceland unit         2005   148.  149.
## 2 Iceland unit         2010   158.  160.
## 3 Iceland unit         2015   164.  165.
## 4 Malta   unit         2005   201.  196.
## 5 Malta   unit         2010   207.  205.
## 6 Malta   unit         2015   210.  208.
## 7 World   aggregate    2005   348.  346.
## 8 World   aggregate    2010   365.  365.
## 9 World   aggregate    2015   375.  374.
```

Pomocí funkce `nest()` je možné vytvořit datovou strukturu, která tento problém vyřeší. `nest()` vytvoří "tabulku tabulek". Můžeme ji použít pro vytvoření tabulky, která bude obsahovat jednotlivé tabulky v tidy formátu:

```
population_world_nested <- population_world %>% nest(data = -observation)

print(population_world_nested)
```

```
## # A tibble: 2 x 2
##   observation data
##   <chr>       <list>
## 1 unit        <tibble [6 x 4]>
## 2 aggregate   <tibble [3 x 4]>
```

Tabulky v tidy formátu jsou obsaženy v nově vytvořeném sloupci `data`:

```
print(population_world_nested$data)
```

```
## [[1]]
## # A tibble: 6 x 4
##   country  year Female  Male
##   <chr>   <int>  <dbl> <dbl>
## 1 Iceland  2005   148.  149.
```

```
## 2 Iceland  2010   158.  160.
## 3 Iceland  2015   164.  165.
## 4 Malta    2005   201.  196.
## 5 Malta    2010   207.  205.
## 6 Malta    2015   210.  208.
##
## [[2]]
## # A tibble: 3 x 4
##   country  year Female  Male
##   <chr>   <int>  <dbl> <dbl>
## 1 World    2005   348.  346.
## 2 World    2010   365.  365.
## 3 World    2015   375.  374.
```

Fungování `nest()` ilustruje následující diagram:

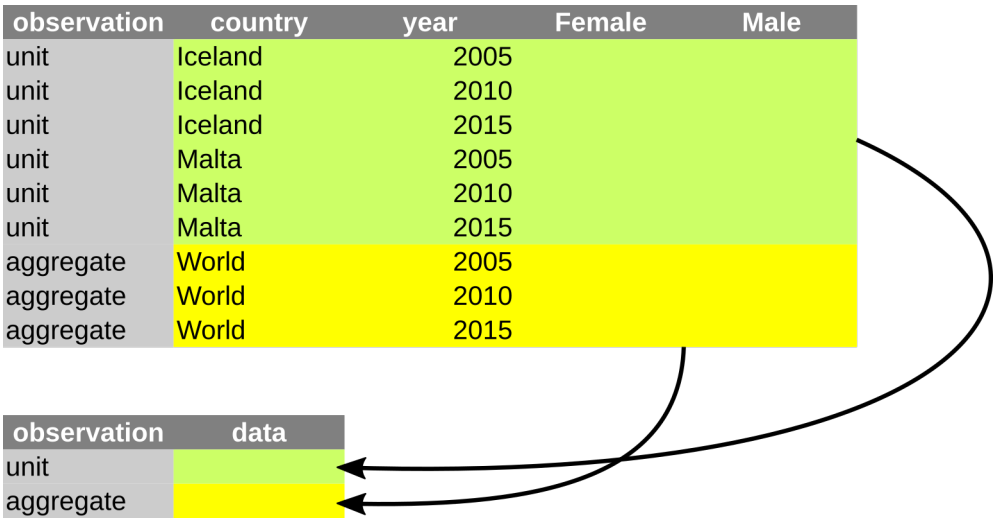


Figure 15.5: Fungování `nest()`

Hlavním praktickým využitím `nest()` je vytvoření datové struktury, která může sloužit pro následnou datovou analýzu s pomocí nástrojů, které nejsou plně kompatibilní s logikou *tidyverse*. Zejména u vysoce specializivaných aplikací je takových funkcí překvapivě mnoho. Na některé případy narazíme zejména na konci kurzu.

Syntaxe funkce `nest()` je velmi jednoduchá (viz `?nest`):

```
nest(data, ...)
```

- `data`...vstupní tabulka (data frame)
- `.....`identifikace sloupců, které mají být součástí nově vytvořených (pod)tabulek. Podobně jako v případě `pivot_longer()` lze využít více způsobů, jak sloupce specifikovat: select helpers, atp.

Základním způsobem identifikace sloupců je podle jejich jmen:

```
## # A tibble: 3 x 3
##   country observation data
##   <chr>   <chr>       <list>
## 1 Iceland unit        <tibble [3 x 3]>
## 2 Malta   unit        <tibble [3 x 3]>
## 3 World   aggregate   <tibble [3 x 3]>
```

Jméno vektoru `data` se použije jako název nově vytvořeného sloupce. Pokud vektor nijak nepojmenujete, vrátí Vám `nest()` varování a nový sloupec pojmenuje právě `data`.

Na příkladech výše bylo ukázán příklad výběru sloupců s pomocí speciální funkce `-`. Funkční by měly být všechny způsoby identifikace podle *tidyselect*.

Datovou strukturu vytvořenou funkcí `nest()` lze transformovat do původního stavu pomocí funkce `unnest()`:

```
population_world_nested %>% unnest(data)
```

```
## # A tibble: 9 x 5
##   observation country  year Female  Male
##   <chr>       <chr>   <int>  <dbl> <dbl>
## 1 unit        Iceland  2005   148.  149.
## 2 unit        Iceland  2010   158.  160.
## 3 unit        Iceland  2015   164.  165.
## 4 unit        Malta    2005   201.  196.
## 5 unit        Malta    2010   207.  205.
## 6 unit        Malta    2015   210.  208.
## 7 aggregate   World    2005   348.  346.
## 8 aggregate   World    2010   365.  365.
## 9 aggregate   World    2015   375.  374.
```

Syntaxe funkce `unnest()` je následující:

```
unnest(data,
       cols,
       ...,
       keep_empty = FALSE,
       ptype = NULL,
       names_sep = NULL,
       names_repair = "check_unique"
       )
```

Základní parametry jsou: - `data`...je vstupní tabulka, - `cols`...je parametr vymezující sloupce, které se mají transformovat (obsahujících tabulky, které se mají "rozbalit").

Funkce `nest()` a `unnest()` se od *tidyr* 1.0.0 významně změnily. Navíc k nim přibyly bratříčci `chop()` a `unchop()`. Ty se od `nest()`/`unnest()` liší v tom, že nevytvářejí nový sloupec tabulek, ale ponechávají původní sloupce, jen transformují jejich obsah na vektor:

```
## # A tibble: 3 x 5
##   country observation        year      Female        Male
##   <chr>   <chr>       <list<int>> <list<dbl>> <list<dbl>>
## 1 Iceland unit                [3]         [3]         [3]
## 2 Malta   unit                [3]         [3]         [3]
## 3 World   aggregate           [3]         [3]         [3]
```

Volba mezi `nest()`/`chop()` závisí čistě na potřebách následné analýzy.

15.3.2 Více hodnot v jedné buňce

Některé tabulky obsahují v jedné buňce hodnoty více proměnných. Jako příklad může sloužit tabulka `tidyr::table3`:

```
print(table3)
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

Ve sloupci `rate` je obsažen podíl počtu případů (cases) na celkové populaci (population). Proměnná `rate` je navíc nutně uložena jako text (`character`). S takovou proměnnou nelze rozumně pracovat. *tidyr* obsahuje nástroje, pomocí kterých je možné taková data převést do tidy formátu, který vyžaduje, aby obsahem jedné buňky byla vždy hodnota právě jedné proměnné.

15.3.2.1 Rozdělení jednoho sloupce do mnoha se `separate()`

Základní funkcí je `separate()`, která umožňuje jeden sloupec rozdělit do mnoha nových sloupců. V případě tabulky `table3` například rozdělit sloupec `rate` na nové sloupce `cases` (číslo před "/") a `population` (číslo za "/"):

```
table3 %>% separate(rate, c("cases","population"), sep="/")
```

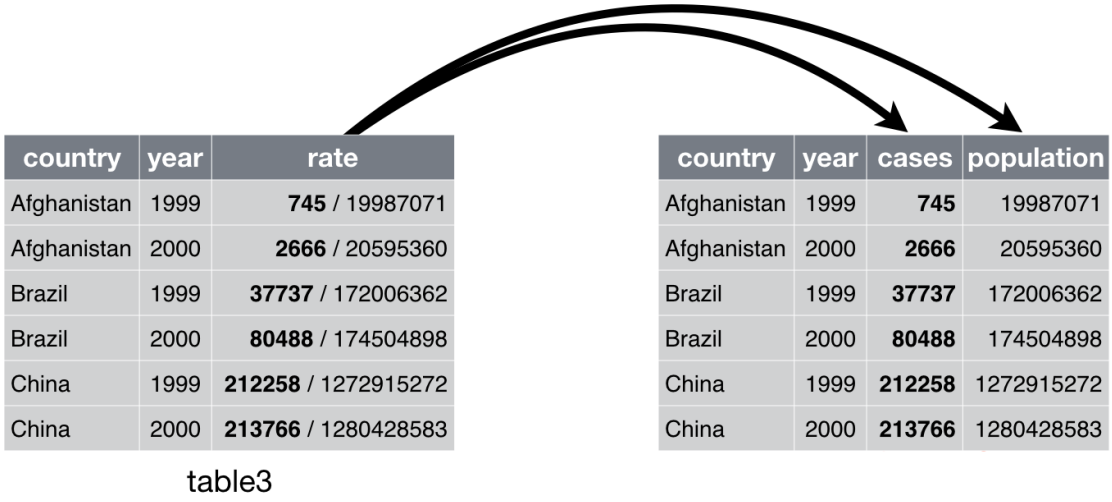
```
## # A tibble: 6 x 4
##   country      year cases  population
##   <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

`separate()` provádí operaci ilustrovanou následujícím digramem:

Funkce `separate()` má následující syntaxi a parametry (viz `?separate`):

```
separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE,
  convert = FALSE, extra = "warn", fill = "warn", ...)
```

- `data`... vstupní tabulka (data frame)
- `col`... specifikace sloupce, který se má rozdělit. Sloupec je specifikován jako jméno bez úvozovek.
- `into`... jména nově vytvářených sloupců specifikovaná jako vektor (character vector)
- `sep`... udává rozdělení vstupního sloupce na výstupní sloupce. Může být specifikován jako číslo (pozice, na které se hodnoty v buňce rozdělí) a zejména jako regulární výraz.
- `remove`... Má být vstupní sloupec zachován ve výstupní tabulce?
- `convert`... pokud je nastaveno na `TRUE`, potom se funkce pokusí o automatickou konverzi výstupních sloupců. Pokud by například byla tato možnost použita v případě `table3`, potom by výstupní sloupce byly konvertovány do celých čísel (integer).
- `extra`... udává, co se má stát, pokud vstupní řetezec obsahuje *více* hodnot, než je specifikováno výstupních sloupců.



country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Figure 15.6: Fungování `separate()` (Wickham, 2016)

- `fill`...udává, co se má stát, pokud vstupní řetězec obsahuje *méně* hodnot, než je specifikováno výstupních sloupců.

Při výchozím nastavení je parametr `extra` nastaven na hodnotu `warn`. To znamená, že v přítomnosti většího počtu hodnot než je specifikováno nově vytvářených sloupců vrátí `seprate()` varování a zahodí přebytečné hodnoty.

```
table3 %>% separate(rate, c("cases"), sep="/", remove=FALSE)
```

```
## Warning: Expected 1 pieces. Additional pieces discarded in 6 rows [1, 2, 3, 4,
## 5, 6].
```

```
## # A tibble: 6 x 4
##   country      year rate              cases
##   <chr>       <int> <chr>             <chr>
## 1 Afghanistan  1999 745/19987071      745
## 2 Afghanistan  2000 2666/20595360     2666
## 3 Brazil       1999 37737/172006362   37737
## 4 Brazil       2000 80488/174504898   80488
## 5 China        1999 212258/1272915272 212258
## 6 China        2000 213766/1280428583 213766
```

Toto chování lze změnit. V případě nastavení `extra` na `drop` provede ve výsledku stejnou opraci, ale nevypíše varování. V případě nastavení parametru na `merge` rozdělí vstupní hodnotu pouze na stejný počet skupin, jako je zadáno výstupních sloupců. V následujícím případě bude tedy výstup (ve sloupci `cases`) stejný jako vstup (ve sloupci `rate`):

```
table3 %>% separate(rate, c("cases"), sep="/", extra="merge", remove=FALSE)
```

```
## # A tibble: 6 x 4
##   country      year rate              cases
##   <chr>       <int> <chr>             <chr>
```

```
## 1 Afghanistan  1999 745/19987071      745/19987071
## 2 Afghanistan  2000 2666/20595360     2666/20595360
## 3 Brazil       1999 37737/172006362   37737/172006362
## 4 Brazil       2000 80488/174504898   80488/174504898
## 5 China        1999 212258/1272915272 212258/1272915272
## 6 China        2000 213766/1280428583 213766/1280428583
```

Může se také stát, že vstupní hondota obsahuje méně skupin, než je specifikováno výstupních sloupců. V následujícím příkladě se snaží `separate()` rozdělit údaje ze sloupce `rate` do tří nových sloupců:

```
table3 %>% separate(rate, c("cases","population","witches"), sep="/")
```

```
## Warning: Expected 3 pieces. Missing pieces filled with `NA` in 6 rows [1, 2, 3,
## 4, 5, 6].
```

```
## # A tibble: 6 x 5
##   country      year cases  population witches
##   <chr>       <int> <chr>  <chr>      <chr>
## 1 Afghanistan  1999 745    19987071   <NA>
## 2 Afghanistan  2000 2666   20595360   <NA>
## 3 Brazil       1999 37737  172006362  <NA>
## 4 Brazil       2000 80488  174504898  <NA>
## 5 China        1999 212258 1272915272 <NA>
## 6 China        2000 213766 1280428583 <NA>
```

Vstupní data však obsahují pouze dva bloky. Při výchozím nastavení parametru `fill` (`warn`) vrátí `separate()` varování a vyplní sloupce zprava. Alternativní nastavení `right` a `left` nevrací varování, a vyplňují sloupce zprava respektive zleva:

```
table3 %>% separate(rate, c("cases","population","witches"), sep="/", fill="left")
```

```
## # A tibble: 6 x 5
##   country      year cases population witches
##   <chr>       <int> <chr> <chr>      <chr>
## 1 Afghanistan  1999 <NA>  745        19987071
## 2 Afghanistan  2000 <NA>  2666       20595360
## 3 Brazil       1999 <NA>  37737      172006362
## 4 Brazil       2000 <NA>  80488      174504898
## 5 China        1999 <NA>  212258     1272915272
## 6 China        2000 <NA>  213766     1280428583
```

Funkce `separate()` má blízké příbuzné v podobě funkcí `extract()` (pozor na maskování `extract` balíkem `magrittr`) a `separate_rows()`.

`extract()` umožňuje specifikovat jednotlivé skupiny ve vstupním sloupci pomocí regulárních výrazů. Pokud některá skupina ve vstupním výrazu chybí, potom je ve výstupním sloupci kódována jako `NA`.

`separate_rows()` nerozkládá vstupní řetězec do sloupců, ale do řádků:

```
table3 %>% separate_rows(rate, sep="/")
```

```
## # A tibble: 12 x 3
##    country      year rate
##    <chr>       <int> <chr>
```

```
##  1 Afghanistan  1999 745
##  2 Afghanistan  1999 19987071
##  3 Afghanistan  2000 2666
##  4 Afghanistan  2000 20595360
##  5 Brazil       1999 37737
##  6 Brazil       1999 172006362
##  7 Brazil       2000 80488
##  8 Brazil       2000 174504898
##  9 China        1999 212258
## 10 China        1999 1272915272
## 11 China        2000 213766
## 12 China        2000 1280428583
```

Použití `separate_rows()` na `table3` nemá smysl. Hodí se například v situaci, kdy jsou v buňce obsaženy identifikátory více různých stavů. Praktickým příkladem může být údaj o zatržení různých checkboxů ve formuláři – takto je obsahují např. CSV exportované z Google Forms.

15.3.2.2 Sloučení mnoha sloupců do jednoho s `unite()`

tidyr obsahuje funkci, které umožňuje provádět inverzní operaci – tedy slučovat více sloupců do jednoho. Tabulka `tidyr::table5` obsahuje rok pozorování rozložený na století a rok:

```
print(table5)
```

```
## # A tibble: 6 x 4
##   country     century year  rate
## * <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

Kompletní letopočet můžeme složit pomocí funkce `unite()`:

```
table5 %>% unite(year,century,year, sep="")
```

```
## # A tibble: 6 x 3
##   country     year  rate
##   <chr>       <chr> <chr>
## 1 Afghanistan 1999  745/19987071
## 2 Afghanistan 2000  2666/20595360
## 3 Brazil      1999  37737/172006362
## 4 Brazil      2000  80488/174504898
## 5 China       1999  212258/1272915272
## 6 China       2000  213766/1280428583
```

`unite()` má obdobné rozhraní a parametry jako funkce určené k rozdělování hodnot (viz `?unite`):

```
unite(data, col, ..., sep = "_", remove = TRUE)
```

- `data`...vstupní tabulka (data frame)

- `col`... jméno nově vytvořeného sloupce (prosté jméno bez úvozovek)
- `.....` sloupce, ze kterých má být nově vytvořený sloupec vytvořen (viz `dplyr::select`)
- `sep`... znak oddělující hodnoty z jednotlivých sloupců
- `remove`... mají být původní sloupce odstraněny?

15.4 Implicitní a explicitní chybějící hodnoty

Tabulky často nejsou úplné – některá pozorování chybějí. Chybějící pozorování je účelné rozdělit na *implicitní* a *explicitní*.

Rozdíl mezi nimi je demonstrován na následujících tabulkách vytvořených z `table1`.

První tabulka `table1_expl` obsahuje explicitní chybějící hodnoty. Pozorování (Brazílie v roce 1999) je v tabulce přítomno, ale místo naměřených hodnot vidíme `NA`:

```
table1_expl <- table1
table1_expl[table1_expl$country == "Brazil" & table1_expl$year == 1999, c("cases","population")] <-
table1_expl[table1_expl$country == "Afghanistan" & table1_expl$year == 1999, "cases"] <- NA

print(table1_expl)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999     NA   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999     NA         NA
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Tabulka `table1_impl` obsahuje implicitní chybějící hodnoty. Pozorování s nenaměřenými chybami v tabulce vůbec není přítomno:

```
table1_impl <- table1
table1_impl <- table1_impl[!(table1_impl$country == "Brazil" & table1_impl$year == 1999),]

print(table1_impl)
```

```
## # A tibble: 5 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       2000  80488  174504898
## 4 China        1999 212258 1272915272
## 5 China        2000 213766 1280428583
```

Implicitní chybějící hodnoty jsou při analýze dat velmi zákeřné – nejsou viditelné "pouhým okem" a ani testem na přítomnost `NA`:

```
table1_impl[!complete.cases(table1_impl),]
```

```
## # A tibble: 0 x 4
## # ... with 4 variables: country <chr>, year <int>, cases <int>,
## #   population <int>
```

15.4.0.1 Odstranění implicitních chybějících hodnot s `complete()`

Při analýze dat je proto vhodné konvertovat implicitní chybějící hodnoty na explicitní. Pro tento účel je možné použít funkci `complete()`:

```
complete(table1_impl, country, year)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999     NA         NA
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Syntaxe funkce `complete()` je velmi přímočará:

```
complete(data, ..., fill = list())
```

- `data`...vstupní tabulka (data frame)
- `.....`sloupce, ze které definují (jednoznačně identifikují) pozorování (v příkladu výše `country` a `year`)

Ve výchozím nastavení jsou implicitní chybějící hodnoty nahrazeny `NA`. Toto chování lze změnit parametrem `fill`. Můžeme například vědět, že pokud není žádný případ zaznamenán, tak statistický úřad pozorování nezapisuje – i když by měl správně zapsat hodnotu 0. (Takto skutečně v některých případech postupuje ČSÚ.) Znalost dat nás tedy vede k tomu, že chybějící pozorování ve sloupci `cases` jsou ve skutečnosti nulová pozorování. Správně doplněné chybějící hodnoty je tedy možné získat nastavením parametru `fill`:

```
complete(table1_impl, country, year, fill = list(cases = 0))
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <dbl>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999      0         NA
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Do parametru `fill` se vkládá `list` s pojmenovanými položkami. Jména položek musí odpovídat jménům sloupců v tabulce a musí obsahovat právě jednu hodnotu. Ta je použita pro nahrazení chybějících hodnot.

V příkladu výše zůstalo chybějící pozorování v `population` nahrazeno `NA`. Pokud není pravidlo pro náhradu explicitně stanoveno ve `fill`, zůstává v platnosti výchozí nastavení.

15.4.0.2 Odstranění explicitních chybějících hodnot s `drop_na()`

V některých případech je naopak vhodné odstranit explicitní chybějící hodnoty a pracovat s tabulkou, ve které jsou implicitní chybějící hodnoty. Pro to je možné využít funkci `drop_na()`:

```
drop_na(table1_expl)
```

```
## # A tibble: 4 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  2000   2666   20595360
## 2 Brazil       2000  80488  174504898
## 3 China        1999 212258 1272915272
## 4 China        2000 213766 1280428583
```

Ve výchozím nastavení `drop_na()` zahazuje všechny řádky, na nichž se vyskytla chybějící hodnota – a to v libovolném sloupci. Toto chování lze změnit pomocí jediného dodatečného parametru funkce `...`, který umožňuje specifikovat, které sloupce mají být brány v potaz. Sloupce mohou být identifikovány všemi způsoby srozumitelnými pro `dplyr::select()`.

V následujícím příkladě je vypuštěn pouze řádek, ve kterém je chybějící hodnota ve sloupci `population`:

```
drop_na(table1_expl, population)
```

```
## # A tibble: 5 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999     NA   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       2000  80488  174504898
## 4 China        1999 212258 1272915272
## 5 China        2000 213766 1280428583
```

15.4.0.3 Nahrazení explicitních chybějících hodnot s `fill()`, `replace_na()`

První funkcí pro nahrazování explicitních chybějících pozorování je `replace_na()`. Její syntaxe a fungování je analogické k parametru `fill` funkce `complete()`. (`complete()` je nakonec pouze wrapper okolo `replace_na()` a několika dalších funkcí.)

Následující použití `replace_na()` nahradí chybějící pozorování ve sloupci `cases` nulami a v `population` nekonečnem (`Inf`):

```
replace_na(table1_expl, replace = list(cases = 0, population = Inf))
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <dbl>      <dbl>
## 1 Afghanistan  1999      0   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999      0        Inf
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

`replace_na()` je užitečná ve velmi omezeném množství případů (viz `complete()`). Častěji je pravděpoodbně v praxi využívána funkce `fill()`. `fill()` nahrazuje chybějící hodnotu hodnotou z předcházejícího (výchozí možnost) nebo následujícího řádku. Pozor, `fill()` pracuje pouze tehdy, pokud jsou chybějící hodnoty explicitní!

Funkci `fill()` je nutné v parametru `...` specifikovat sloupce, u kterých se má nahrazení chybějících hodnot provést. (`fill()` opět rozumí všem možnostem dostupným v `dplyr::select()`.) Následující ukázka demonstruje úskalí používání funkce `fill()`:

```
fill(table1_expl, cases)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999     NA   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999   2666         NA
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

1. Chybějící hodnota v prvním řádku nebyla nahrazena – neexistuje totiž žádný předcházející řádek.
2. Údaj pro Brazilii byl nahrazen údajem pro Afganistán.

První problém samozřejmě nemá řešení. Výzkumník může zvážit nahrazení hodnotou z následujícího řádku (pomocí parametru `.direction = "up"`). Pro druhý problém je řešení následující:

1. Rozdělit tabulku na mnoho dílčích tabulek podél proměnné `country`.
2. Provést nahrazení v každé z nich.
3. Tabulky složit zpátky.

Takový úkol je jistě proveditelný, ale velmi složitý. Naštěstí právě takovou funkcionalitu poskytuje `group_by()` z balíku `dplyr`. `group_by()` umožňuje každou definovanou skupinu zpracovat odděleně:

```
## # A tibble: 6 x 4
## # Groups:   country [3]
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999     NA   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999     NA         NA
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

V tomto případě je výsledek v pořádku – Brazílie "nedědí" hodnoty Afganistánu.

15.5 Konstrukce vlastních tabulek s `crossing()`

V některých případech je užitečné vytvořit si vlastní tabulku. Typickým příkladem může být vytvoření "kostry" tabulky, která bude obsahovat všechny možné varinty (identifikátory) pozorování, které se ve výsledné tabulce mohou vyskytnout.

Například můžeme chtít vytvořit panelová data pro země V4 a pro roky 2000–2001. Tento úkol je možné efektivně vyřešit pomocí `crossing()`:

```
## # A tibble: 8 x 2
##   country         year
##   <chr>          <int>
```

```
## 1 Czech Republic  2000
## 2 Czech Republic  2001
## 3 Hungary         2000
## 4 Hungary         2001
## 5 Poland          2000
## 6 Poland          2001
## 7 Slovakia        2000
## 8 Slovakia        2001
```

16 Manipulace s daty s nástroji z balíku *dplyr*

Balík *tidyr* obsahuje nástroje, které umožňují uživateli velmi jednoduše měnit organizační strukturu tabulky a provádět některé operace spojené s čištěním dat. Praktická práce s daty však vyžaduje více a to nejen v oblasti čištění dat. Typicky potřebujete subsetovat pozorování nebo proměnné, modifikovat nebo vytvářet nové proměnné a agregovat pozorování. Data také obvykle nejsou pouze v jediném zdroji – tabulce. Je tedy nutné tabulky různými způsoby slučovat. Pro tyto úkoly existuje v R balík *dplyr* – součást *tidyverse*.

```
library(tidyverse)
```

V této lekci se naučíte:

- subsetovat pozorování a proměnné
- vytvářet nové a modifikovat stávající proměnné
- vytvářet agregované hodnoty z více pozorování
- provádět jednotlivé operace zvlášť pro různé skupiny pozorování
- spojovat *(join)* a slučovat *(bind)* tabulky
- a další…

Funkcionalitu, kterou Vám poskytuje *dplyr* můžete získat i s nástroji *base* R. Nicméně používání *dplyr* vám přinese snadnější interaktivní práci, vyšší rychlost (zejména pro "středně velká" data) a srozumitelnost kódu. Výhodou je i kompatibilita API s ostatními částmi *tidyverse*.

Pro uživatele, kteří hodlají pracovat s velkými objemy dat (včetně Big Data), je důležitá i další možnost, kterou *dplyr* nabízí. Kód napsaný v *dplyr* může být vykonán s pomocí různých backendů – nástrojů, které provádí samotnou práci s daty. Vedle základního (defaultního) backendu je k dispozici balík *dtplyr*, který umožňuje vykonat kód s pomocí tříd a funkcí `data.table`, nebo balík *dbplyr*, který umožňuje kód vnitřně přeložit do SQL a nechat ho vykonat vzdálenou databázi. Uživatelé, kteří mají k dispozici speciální infrastrukturu pro analýzu Big Data, mohou podobně využít i backend pro Apache Spark z balíku *sparklyr*. Možnost změnit backend dělá z *dplyr* mocný nástroj, protože umožňuje jednoduché škálování úloh.

dplyr má implementováno mnoho dalších pokročilých funkcí. Obsahem této lekce jsou však spíše základy, které nicméně pokrývají vše, co je běžně potřeba pro interaktivní práci s daty. O *dplyr* platí více než o všech jiných balících zmíněných v celém kurzu, že je stále ve velmi aktivním vývoji. Dochází k častým a hlubokým změnám jak v API, tak v backendu (respektive backendech). Tato kapitola byla připravována s požitím verze 1.0.0. Udržujte svůj *dplyr* **aktualizovaný**. Určitě je také doporučeníhodné sledovat vývoj a čas od času si projít seznam zahrnutých funkcí.

16.0.1 Co je obsahem balíku *dplyr*?

Hadley Wickham používá pro označení skupin funkcí slovní druhy. Základním slovním druhem v balíku *dplyr* je sloveso. *dplyr* obsahuje slovesa (funkce), které pracují s jednou tabulkou (např. *vyber*, *seřaď*, nebo *agreguj*), nebo (primárně) dvěma tabulkami (*spoj* a *sluč*).

Samostatnou funkcionalitou je schopnost *dplyr* spouštět slovesa nejen nad celou jednou tabulkou, ale také nad jejími částmi (skupinami pozorování). Takové operace se nazývají "zgrupované".

Obsah přednášky:

- slovesa (funkce) pracující s jednou tabulkou
- zgrupované operace
- slovesa (funkce) pracující se dvěma (nebo více) tabulkami

Pro demonstraci funkcí z balíku *dplyr* jsou použita data z balíku *nycflights13* – údaje o letech z/do NYC v roce 2013:

```
library(nycflights13)
```

Ve většině příkladů budeme používat tabulku `planes` s údaji o letadlech, která do NYC létala:

```
planes %>% print
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type          manufacturer   model  engines seats speed engine
##    <chr>   <int> <chr>         <chr>          <chr>    <int> <int> <int> <chr>
##  1 N10156   2004 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  2 N102UW   1998 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  3 N103US   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  4 N104UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  5 N10575   2002 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  6 N105UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  7 N107US   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  8 N108UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  9 N109UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 10 N110UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## # ... with 3,312 more rows
```

16.0.1.1 Poznámka

dplyr je pokročilou evolucí balíku `plyr`. Balík `plyr` je stále v závislostech některých balíků, které využívají jeho služeb. To může (a pravděpodobně bude) vyvolávat konflikty, kterým se dá čelit jednoduchým způsobem – vždy nejprve načíst `plyr` a až následně *dplyr*. Pokud načtete balíky v opačném pořadí, tak Vám *dplyr* vypíše upovídané varování. (Tento problém se v čase zmenšuje. Stále však taková situace může reálně nastat.)

16.1 Slovesa pracující s jednou tabulkou

16.1.1 Výběr řádků (pozorování)

Základní funkcí, která umožňuje výběr řádků je `filter()`. Podobně jako většina funkcí z balíku *dplyr* má extrémně jednoduchou syntax:

```
filter(.data, ...)
```

Vstupem funkce je tabulka (`.data`) a jeden nebo více logických predikátů. Výstupem funkce je podmnožina řádků, která splňuje všechny zadané predikáty.

Například se můžeme chtít podívat na letadla, která vyrobil Airbus nebo Boeing a mají více než dva motory:

```
planes %>%
    filter(manufacturer %in% c("AIRBUS INDUSTRIE","BOEING"), engines > 2)
```

```
## # A tibble: 2 x 9
##   tailnum  year type          manufacturer   model  engines seats speed engine
```

```
##   <chr>   <int> <chr>          <chr>          <chr>    <int> <int> <int> <chr>
## 1 N281AT     NA Fixed wing mu~ AIRBUS INDUST~ A340-~       4   375    NA Turbo-~
## 2 N670US   1990 Fixed wing mu~ BOEING         747-4~       4   450    NA Turbo-~
```

Vybrány byly pouze řádky splňující všechny podmínky najednou. Následující volání `filter()`, které spojuje dvě výše použité podmínky do jedné logickým AND (`&`) proto vrátí stejný výsledek:

```
planes %>%
    filter(manufacturer %in% c("AIRBUS INDUSTRIE","BOEING") & engines > 2)
```

```
## # A tibble: 2 x 9
##   tailnum  year type           manufacturer   model  engines seats speed engine
##   <chr>   <int> <chr>          <chr>          <chr>    <int> <int> <int> <chr>
## 1 N281AT     NA Fixed wing mu~ AIRBUS INDUST~ A340-~       4   375    NA Turbo-~
## 2 N670US   1990 Fixed wing mu~ BOEING         747-4~       4   450    NA Turbo-~
```

Podmínky použité ve funkci `filter()` musí po vyhodnocení vracet logické hodnoty `TRUE`/`FALSE`. Ve `filter()` tedy můžeme používat funkce, které vracejí logickou hodnotu.

Například nás mohou zajímat všechna letadla z rodiny A340. Budeme tedy chtít vybrat všechny řádky, u nichž proměnná `model` začíná na "A340". Kromě balíku *dplyr* budeme potřebovat i *stringr*:

```
library(stringr)

planes %>%
    filter(str_detect(model,"^A340"))
```

```
## # A tibble: 1 x 9
##   tailnum  year type           manufacturer   model  engines seats speed engine
##   <chr>   <int> <chr>          <chr>          <chr>    <int> <int> <int> <chr>
## 1 N281AT     NA Fixed wing mu~ AIRBUS INDUST~ A340-~       4   375    NA Turbo-~
```

Je možné používat i funkce, které nevracejí logické hodnoty. V takové případě je však nutné jejich výsledek na logickou proměnou transformovat. Řekněme, že by nás zajímala letadla, kde na jeden motor připadá méně než 10 sedadel:

```
planes %>%
    filter(seats/engines < 10)
```

```
## # A tibble: 39 x 9
##    tailnum  year type          manufacturer model   engines seats speed engine
##    <chr>   <int> <chr>         <chr>        <chr>     <int> <int> <int> <chr>
##  1 N201AA   1959 Fixed wing s~ CESSNA       150           1     2    90 Recipro~
##  2 N202AA   1980 Fixed wing m~ CESSNA       421C          2     8    90 Recipro~
##  3 N315AT     NA Fixed wing s~ JOHN G HESS  AT-5          1     2    NA 4 Cycle
##  4 N347AA   1985 Rotorcraft   SIKORSKY     S-76A         2    14    NA Turbo-s~
##  5 N350AA   1980 Fixed wing m~ PIPER        PA-31-~       2     8   162 Recipro~
##  6 N364AA   1973 Fixed wing m~ CESSNA       310Q          2     6   167 Recipro~
##  7 N365AA   2001 Rotorcraft   AGUSTA SPA   A109E         2     8    NA Turbo-s~
##  8 N376AA   1978 Fixed wing s~ PIPER        PA-32R~       1     7    NA Recipro~
##  9 N377AA     NA Fixed wing s~ PAIR MIKE E  FALCON~       1     2    NA Recipro~
## 10 N378AA   1963 Fixed wing s~ CESSNA       172E          1     4   105 Recipro~
## # ... with 29 more rows
```

Ve speciálních případech je užitečné vybírat řádky ne podle splnění určitých podmínek, ale podle jiných kritérií. Pro tyto případy je v balíku *dplyr* obsažena funkce `slice()`. *(Poznámka: zde diskutované funkce `slice()` získala s verzí dplyr mnoho nových úloh a rolí. V předchozích verzích umožňovala pouze výběr řádku na základě jeho čísla.)*

V základní variantě přijímá `slice()` číslo řádku, nebo jejich rozsah, který má být vybrán:

```
planes %>%
    slice(1L)
```

```
## # A tibble: 1 x 9
##   tailnum  year type            manufacturer model   engines seats speed engine
##   <chr>   <int> <chr>           <chr>        <chr>     <int> <int> <int> <chr>
## 1 N10156   2004 Fixed wing mul~ EMBRAER      EMB-14~       2    55    NA Turbo-~
```

```
planes %>%
    slice(1L:5L)
```

```
## # A tibble: 5 x 9
##   tailnum  year type           manufacturer   model  engines seats speed engine
##   <chr>   <int> <chr>          <chr>          <chr>    <int> <int> <int> <chr>
## 1 N10156   2004 Fixed wing mu~ EMBRAER        EMB-1~       2    55    NA Turbo-~
## 2 N102UW   1998 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 3 N103US   1999 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 4 N104UW   1999 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 5 N10575   2002 Fixed wing mu~ EMBRAER        EMB-1~       2    55    NA Turbo-~
```

```
planes %>%
    slice(c(1,2:5))
```

```
## # A tibble: 5 x 9
##   tailnum  year type           manufacturer   model  engines seats speed engine
##   <chr>   <int> <chr>          <chr>          <chr>    <int> <int> <int> <chr>
## 1 N10156   2004 Fixed wing mu~ EMBRAER        EMB-1~       2    55    NA Turbo-~
## 2 N102UW   1998 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 3 N103US   1999 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 4 N104UW   1999 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 5 N10575   2002 Fixed wing mu~ EMBRAER        EMB-1~       2    55    NA Turbo-~
```

Poslední příklad ukazuje, že (a) číslo řádku nemusí být nutně zadáno jako integer (`L`) a (b) vstupem může být vektor vytvořený funkcí `c()`.

Speciální variantou `slice()` jsou varianty `slice_head()`, `slice_tail()`, `slice_min()` a `slice_max()`. Ty vrací `n` prvních/posledních řádků respektive `n` řádků s nejnižší/nejvyšší hodnotou:

```
planes %>%
    slice_head(n = 5)
```

```
## # A tibble: 5 x 9
##   tailnum  year type           manufacturer   model  engines seats speed engine
##   <chr>   <int> <chr>          <chr>          <chr>    <int> <int> <int> <chr>
## 1 N10156   2004 Fixed wing mu~ EMBRAER        EMB-1~       2    55    NA Turbo-~
## 2 N102UW   1998 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 3 N103US   1999 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 4 N104UW   1999 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 5 N10575   2002 Fixed wing mu~ EMBRAER        EMB-1~       2    55    NA Turbo-~
```

```
planes %>%
    slice_max(seats, n = 1)
```

```
## # A tibble: 1 x 9
##   tailnum  year type              manufacturer model  engines seats speed engine
##   <chr>   <int> <chr>             <chr>        <chr>    <int> <int> <int> <chr>
## 1 N670US   1990 Fixed wing mult~ BOEING       747-4~       4   450    NA Turbo-~
```

Počet řádků nemusí být určen jako absolutní číslo, ale je možné ho specifikovat v parametru `prop` jako podíl všech pozorování:

```
planes %>%
    slice_head(prop = 0.001)
```

```
## # A tibble: 3 x 9
##   tailnum  year type           manufacturer   model  engines seats speed engine
##   <chr>   <int> <chr>          <chr>          <chr>    <int> <int> <int> <chr>
## 1 N10156   2004 Fixed wing mu~ EMBRAER        EMB-1~       2    55    NA Turbo-~
## 2 N102UW   1998 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 3 N103US   1999 Fixed wing mu~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
```

Toto volání vrátí 0,1 % pozorování ze začátku tabulky.

Poslední speciální variantou `slice()` je `slice_sample()`, která umožňuje provést náhodný výběr pozorování z tabulky. (V předchozích verzích `dplyr` k tomu sloužily funkce `sample_*()`.) Počet řádků v náhodném výběru může být stanoveno opět absolutním číslem (parametr `n`) nebo podílem (parametr `prop`). Funkce umožňuje používat ve výběru váhy (parametr `weight_by`) a zvolit výběr s nahrazením (parametr `replace`).

16.1.2 Výběr sloupců (proměnných)

Pro výběr sloupců slouží funkce `select()`. Syntax je podobná jako v případě `filter()`:

```
select(.data, ...)
```

Do `select()` vstupuje tabulka a identifikace sloupců, které mají být vybrány. Například:

```
planes %>%
    select(tailnum, manufacturer)
```

```
## # A tibble: 3,322 x 2
##    tailnum manufacturer
##    <chr>   <chr>
##  1 N10156  EMBRAER
##  2 N102UW  AIRBUS INDUSTRIE
##  3 N103US  AIRBUS INDUSTRIE
##  4 N104UW  AIRBUS INDUSTRIE
##  5 N10575  EMBRAER
##  6 N105UW  AIRBUS INDUSTRIE
##  7 N107US  AIRBUS INDUSTRIE
##  8 N108UW  AIRBUS INDUSTRIE
##  9 N109UW  AIRBUS INDUSTRIE
## 10 N110UW  AIRBUS INDUSTRIE
## # ... with 3,312 more rows
```

Příklad ukazuje první a základní možnost, jak identifikovat sloupec – a to jeho jménem. `select()` však umožňuje specifikovat sloupec i pomocí čísla pozice.

Následující volání funkce `select()` tak vrací stejný výsledek, jako tomu byl v případě identifikace sloupců jejich jménem.

```
planes %>%
    select(1,4)
```

```
## # A tibble: 3,322 x 2
##    tailnum manufacturer
##    <chr>   <chr>
##  1 N10156  EMBRAER
##  2 N102UW  AIRBUS INDUSTRIE
##  3 N103US  AIRBUS INDUSTRIE
##  4 N104UW  AIRBUS INDUSTRIE
##  5 N10575  EMBRAER
##  6 N105UW  AIRBUS INDUSTRIE
##  7 N107US  AIRBUS INDUSTRIE
##  8 N108UW  AIRBUS INDUSTRIE
##  9 N109UW  AIRBUS INDUSTRIE
## 10 N110UW  AIRBUS INDUSTRIE
## # ... with 3,312 more rows
```

```
planes %>% names
```

```
## [1] "tailnum"      "year"         "type"         "manufacturer" "model"
## [6] "engines"      "seats"        "speed"        "engine"
```

16.1.2.1 Funkce `select()` a speciální funkce

Při identifikaci sloupců je možné využít speciální funkce. Některé fungují pouze "uvnitř" `select()` a některých dalších funkcí z *tidyverse*.

První taková funkce je `:`. Umožňuje specifikovat rozsah sloupců, místo vypisování všech prvků. Všechny následující volání tak vrací stejný výsledek:

```
planes %>%
    select(1,2,3,4,8)

planes %>%
    select(tailnum, year, type, manufacturer, speed)

planes %>%
    select(1:4, 8)

planes %>%
    select(tailnum:manufacturer, speed)
```

Další speciální funkcí je `-` (mínus). Tato funkce umožňuje "negativní" výběr. Při jejím použití není sloupec zahrnut, ale naopak vypuštěn:

```
planes %>%
    select(-tailnum, -year, -type, -manufacturer, -speed)
```

```
## # A tibble: 3,322 x 4
##    model     engines seats engine
##    <chr>       <int> <int> <chr>
##  1 EMB-145XR       2    55 Turbo-fan
##  2 A320-214        2   182 Turbo-fan
##  3 A320-214        2   182 Turbo-fan
##  4 A320-214        2   182 Turbo-fan
##  5 EMB-145LR       2    55 Turbo-fan
##  6 A320-214        2   182 Turbo-fan
##  7 A320-214        2   182 Turbo-fan
##  8 A320-214        2   182 Turbo-fan
##  9 A320-214        2   182 Turbo-fan
## 10 A320-214        2   182 Turbo-fan
## # ... with 3,312 more rows
```

Speciální funkce je možné kombinovat – je například možné vypustit sloupce identifikované rozsahem (`:`):

```
planes %>%
    select(-tailnum:-manufacturer, -speed)
```

```
## # A tibble: 3,322 x 4
##    model     engines seats engine
##    <chr>       <int> <int> <chr>
##  1 EMB-145XR       2    55 Turbo-fan
##  2 A320-214        2   182 Turbo-fan
##  3 A320-214        2   182 Turbo-fan
##  4 A320-214        2   182 Turbo-fan
##  5 EMB-145LR       2    55 Turbo-fan
##  6 A320-214        2   182 Turbo-fan
##  7 A320-214        2   182 Turbo-fan
##  8 A320-214        2   182 Turbo-fan
##  9 A320-214        2   182 Turbo-fan
## 10 A320-214        2   182 Turbo-fan
## # ... with 3,312 more rows
```

Výsledné tabulky jsou pochopitelně shodné.

Obě tyto speciální funkce vyžadují přesnou specifikaci jména nebo pozice sloupce. V reálném životě občas pracujeme s poněkud vágnějším zadáním. Mohli bychom chtít například vybrat všechny sloupce, které obsahují informace o motorech. Ty jsou v tabulce `planes` dva `engine` (typ motoru) a `engines` (počet motorů).

První možností je samozřejmě možné použít následující volání a vybrat sloupce jejich výčtem:

```
planes %>%
    select(engine,engines)
```

To však není praktické v případě, že pracujeme s větším množstvím sloupců, jejichž názvy jsou systematické. V tom případě je užitečné sáhnout po *select helpers* (funkcích pomocníčcích chcete-li). *dplyr* jich nabízí hned několik:

- `starts_with()` vybírá sloupce, jejichž jméno začíná na řetězec, který je argumentem funkce `starts_with()`
- `ends_with()` vybírá sloupce, jejichž jméno končí na řetězec, který je argumentem funkce `ends_with()`
- `contains()` vybírá sloupce, jejichž jméno obsahuje řetězec, který je argumentem funkce `contains()`
- `matches()` vybírá sloupce, jejichž jméno odpovídá zadanému regulárnímu výrazu

- `num_range()` slouží pro výběr sloupců, jejichž jméno je tvořeno kombinací řetězce a čísla – například `trial_1`, `trial_2`,...
- `everything()` vrací všechny sloupce
- `last_col()` vrací poslední sloupec

Pro výběr proměnných se vztahem k motorům lze použít hned tři funkce:

```
planes %>%
    select(starts_with("engine"))

planes %>%
    select(contains("engine"))

planes %>%
    select(matches("^engine"))
```

První a třetí varianta vybere všechny sloupce, které začínají na "engine". Druhé variantě postačí k výběru, že řetězec "engine" se vyskytuje kdekoliv ve jméně sloupce.

Další *select helpers* umožňují vybrat sloupce podle jmen ze vstupního vektoru.

- `all_of()` vrátí tabulku s vybranými sloupci pouze tehdy, pokud se ji podaří najít všechna jména obsažená ve vstupním vektoru. V opačném případě vrátí chybu.
- `any_of()` vrátí prostě jenom ty sloupce, které v tabulce najde.

```
planes %>%
    select(all_of("engine")) # Vrátí jeden sloupec

planes %>%
    select(all_of(c("engine","Engine"))) # Vrátí chybu

planes %>%
    select(any_of(c("engine","Engine"))) # Vrátí jeden sloupec

planes %>%
    select(any_of(c("Engine"))) # Nevrátí řádný sloupec.
```

Select helpers mohou být kombinováni se všemi ostatními způsoby identifikace sloupců:

```
planes %>%
    select(tailnum, starts_with("engine"))
```

```
## # A tibble: 3,322 x 3
##    tailnum engines engine
##    <chr>     <int> <chr>
##  1 N10156        2 Turbo-fan
##  2 N102UW        2 Turbo-fan
##  3 N103US        2 Turbo-fan
##  4 N104UW        2 Turbo-fan
##  5 N10575        2 Turbo-fan
##  6 N105UW        2 Turbo-fan
##  7 N107US        2 Turbo-fan
##  8 N108UW        2 Turbo-fan
##  9 N109UW        2 Turbo-fan
## 10 N110UW        2 Turbo-fan
## # ... with 3,312 more rows
```

Posledním *select helper* je `where()`. Tato funkce umožňuje vybrat sloupce s pomocí funkce vracející logickou hdonotu. Je tak možné například vybrat pouze sloupce, které obahují celá čísla:

```
planes %>%
  select(where(is.integer))
```

```
## # A tibble: 3,322 x 4
##     year engines seats speed
##    <int>   <int> <int> <int>
##  1  2004       2    55    NA
##  2  1998       2   182    NA
##  3  1999       2   182    NA
##  4  1999       2   182    NA
##  5  2002       2    55    NA
##  6  1999       2   182    NA
##  7  1999       2   182    NA
##  8  1999       2   182    NA
##  9  1999       2   182    NA
## 10  1999       2   182    NA
## # ... with 3,312 more rows
```

Všiměte si, že samotná funkce `is.integer` je parametrem `where()` a nikoliv její výstup. Je proto nutné do funkce zadat `is.integer` a nikoliv `is.integer()`.

Funkce `where()` je jedna z novinek v *dplyr* 1.0.0 a společně s dalšími funkcemi nahradila tzv. scoped varianty základních funkcí (`select_if()` atp.).

Select helper s velmi specifickým využitím je `everything()`, které slouží k vybrání všeho. Nebo lépe všeho ostatního. Pokud z nějakého důvodu chceme změnit pořadí sloupců v tabulce, potom se hodí právě `everything()`.

```
planes %>%
    select(engine, engines, everything())
```

```
## # A tibble: 3,322 x 9
##    engine    engines tailnum  year type         manufacturer  model  seats speed
##    <chr>       <int> <chr>   <int> <chr>        <chr>         <chr>  <int> <int>
##  1 Turbo-fan       2 N10156   2004 Fixed wing ~ EMBRAER       EMB-1~    55    NA
##  2 Turbo-fan       2 N102UW   1998 Fixed wing ~ AIRBUS INDUS~ A320-~   182    NA
##  3 Turbo-fan       2 N103US   1999 Fixed wing ~ AIRBUS INDUS~ A320-~   182    NA
##  4 Turbo-fan       2 N104UW   1999 Fixed wing ~ AIRBUS INDUS~ A320-~   182    NA
##  5 Turbo-fan       2 N10575   2002 Fixed wing ~ EMBRAER       EMB-1~    55    NA
##  6 Turbo-fan       2 N105UW   1999 Fixed wing ~ AIRBUS INDUS~ A320-~   182    NA
##  7 Turbo-fan       2 N107US   1999 Fixed wing ~ AIRBUS INDUS~ A320-~   182    NA
##  8 Turbo-fan       2 N108UW   1999 Fixed wing ~ AIRBUS INDUS~ A320-~   182    NA
##  9 Turbo-fan       2 N109UW   1999 Fixed wing ~ AIRBUS INDUS~ A320-~   182    NA
## 10 Turbo-fan       2 N110UW   1999 Fixed wing ~ AIRBUS INDUS~ A320-~   182    NA
## # ... with 3,312 more rows
```

Změní pořadí sloupců tak, že na první pozici přesune `engine` a `engines` a následně do tabulky vyskládá všechny ostatní sloupce. Díky `everything()` není nutné jejich jména vypisovat. Do verze 1.0.0 bylo použití `select()` prakticky jedinou možností jak změnit pořadí sloupců. O dtéto verze v *dplyr* existuje specializovaná funkce `relocate()`, která také umí pracovat se *select helpers*.

16.1.2.2 Další speciální funkce

Při práci s výběry, které jsou v podstatě jen logickým vektorem nad jmény sloupců, je možné používat logické operátory `!` (negace), `|` (OR) a `&` (AND):

```
planes %>%
  select(!starts_with("engi")) # Vrátí sloupce, které nezačínají na "engi"
```

```
## # A tibble: 3,322 x 7
##    tailnum  year type                    manufacturer     model     seats speed
##    <chr>   <int> <chr>                   <chr>            <chr>     <int> <int>
##  1 N10156   2004 Fixed wing multi engine EMBRAER          EMB-145XR    55    NA
##  2 N102UW   1998 Fixed wing multi engine AIRBUS INDUSTRIE A320-214    182    NA
##  3 N103US   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214    182    NA
##  4 N104UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214    182    NA
##  5 N10575   2002 Fixed wing multi engine EMBRAER          EMB-145LR    55    NA
##  6 N105UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214    182    NA
##  7 N107US   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214    182    NA
##  8 N108UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214    182    NA
##  9 N109UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214    182    NA
## 10 N110UW   1999 Fixed wing multi engine AIRBUS INDUSTRIE A320-214    182    NA
## # ... with 3,312 more rows
```

```
planes %>%
  select(starts_with("e") & ends_with("s")) # Vrátí sloupce začínající na "e" a zároveň končící na "
```

```
## # A tibble: 3,322 x 1
##    engines
##      <int>
##  1       2
##  2       2
##  3       2
##  4       2
##  5       2
##  6       2
##  7       2
##  8       2
##  9       2
## 10       2
## # ... with 3,312 more rows
```

```
planes %>%
  select(starts_with("e") | ends_with("s")) # Vrátí sloupce začínající na "e" nebo končící na "s"
```

```
## # A tibble: 3,322 x 3
##    engines engine    seats
##      <int> <chr>     <int>
##  1       2 Turbo-fan    55
##  2       2 Turbo-fan   182
##  3       2 Turbo-fan   182
##  4       2 Turbo-fan   182
##  5       2 Turbo-fan    55
##  6       2 Turbo-fan   182
##  7       2 Turbo-fan   182
```

```
##  8       2 Turbo-fan   182
##  9       2 Turbo-fan   182
## 10       2 Turbo-fan   182
## # ... with 3,312 more rows
```

Podobně je možné výběry kombinovat pomocí funkce `c()`:

```
planes %>%
  select(starts_with(c("e","s"))) # Vrátí sloupce začínající na "e" nebo na "s"
```

```
## # A tibble: 3,322 x 4
##    engines engine    seats speed
##      <int> <chr>     <int> <int>
##  1       2 Turbo-fan    55    NA
##  2       2 Turbo-fan   182    NA
##  3       2 Turbo-fan   182    NA
##  4       2 Turbo-fan   182    NA
##  5       2 Turbo-fan    55    NA
##  6       2 Turbo-fan   182    NA
##  7       2 Turbo-fan   182    NA
##  8       2 Turbo-fan   182    NA
##  9       2 Turbo-fan   182    NA
## 10       2 Turbo-fan   182    NA
## # ... with 3,312 more rows
```

16.1.2.3 Výběr a přejmenování sloupce

Jednou ze speciálních funkcí je i =. To slouží v `select()` pro přejmenování. Například volání

```
planes %>%
    select(tailnum, company = manufacturer)
```

```
## # A tibble: 3,322 x 2
##    tailnum company
##    <chr>   <chr>
##  1 N10156  EMBRAER
##  2 N102UW  AIRBUS INDUSTRIE
##  3 N103US  AIRBUS INDUSTRIE
##  4 N104UW  AIRBUS INDUSTRIE
##  5 N10575  EMBRAER
##  6 N105UW  AIRBUS INDUSTRIE
##  7 N107US  AIRBUS INDUSTRIE
##  8 N108UW  AIRBUS INDUSTRIE
##  9 N109UW  AIRBUS INDUSTRIE
## 10 N110UW  AIRBUS INDUSTRIE
## # ... with 3,312 more rows
```

vybere sloupce `tailnum` a `manufacturer`. Sloupec `manufacturer` však zároveň přejmenuje na `company`.

Speciálně pro přejmenovávání sloupců je v *dplyr* obsažena funkce `rename()` (fakticky jde jen o lehkou mutaci `select()`). Ta sloupce nevybírá, ale jen přejmenovává. Použití = je v ní povinné:

```
planes %>%
    rename(tailnum, company = manufacturer)
```

```
## Error: All renaming inputs must be named.
```

Po opravě získáme správný výsledek:

```
planes %>%
    rename(company = manufacturer)
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type                    company model engines seats speed engine
##    <chr>   <int> <chr>                   <chr>   <chr>   <int> <int> <int> <chr>
##  1 N10156   2004 Fixed wing multi engine EMBRAER EMB-~       2    55    NA Turbo~
##  2 N102UW   1998 Fixed wing multi engine AIRBUS~ A320~       2   182    NA Turbo~
##  3 N103US   1999 Fixed wing multi engine AIRBUS~ A320~       2   182    NA Turbo~
##  4 N104UW   1999 Fixed wing multi engine AIRBUS~ A320~       2   182    NA Turbo~
##  5 N10575   2002 Fixed wing multi engine EMBRAER EMB-~       2    55    NA Turbo~
##  6 N105UW   1999 Fixed wing multi engine AIRBUS~ A320~       2   182    NA Turbo~
##  7 N107US   1999 Fixed wing multi engine AIRBUS~ A320~       2   182    NA Turbo~
##  8 N108UW   1999 Fixed wing multi engine AIRBUS~ A320~       2   182    NA Turbo~
##  9 N109UW   1999 Fixed wing multi engine AIRBUS~ A320~       2   182    NA Turbo~
## 10 N110UW   1999 Fixed wing multi engine AIRBUS~ A320~       2   182    NA Turbo~
## # ... with 3,312 more rows
```

Tabulka obsahuje všechny sloupce, ale jeden z nich byl přejmenován.

16.2 Tvorba a úprava obsahu

Balík *dplyr* obsahuje dvě základní funkce pro vytváření a agregaci obsahu v tabulkách: `mutate()` a `summarise()`

16.2.1 Tvorba nových sloupců s `mutate()`

Funkce `mutate()` vytváří nové sloupce, proměnné, v tabulce. Zachovává tedy počet řádků v tabulce a přidává nové sloupce. Syntax `mutate()` je podobně jako u dalších funkcí z *tidyverse* poměrně střídmá:

```
mutate(.data, ...)
```

Funkce přijímá vstupní tabulku a specifikaci sloupců, které se mají vytvořit v `...`. Fungování `mutate()` může být ilustrováno následujícím (mírně zjednodušujícím) schématem:

Figure 16.1: Tvorba nových sloupců s `mutate()`

`mutate()` může být použito i pro modifikaci stávajících sloupců. V tomto případě však `mutate()` interně nejprve vytvoří nový sloupec a až následně jím nahradí sloupec původní. Při modifikaci sloupce na opravdu velkých tabulkách tak může `mutate()` spotřebovávat nečekané množství systémových zdrojů.

Praktické využití `mutate()` je možné ilustrovat na příkladu. Například můžeme chtít pro každé pozorování (řádek, letadlo) v tabulce `planes` spočítat, kolik sedadel připadá na jeden motor a zjistit, zda se jedná o vrtulové letadlo:

```
planes %>%
    mutate(
        seats_per_engine = (seats/engines) %>% round(),
        turbo_prop_plane = engine == "Turbo-prop"
    ) #%>%
```

```
## # A tibble: 3,322 x 11
##    tailnum  year type          manufacturer   model  engines seats speed engine
##    <chr>   <int> <chr>         <chr>          <chr>    <int> <int> <int> <chr>
##  1 N10156   2004 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  2 N102UW   1998 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  3 N103US   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  4 N104UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  5 N10575   2002 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  6 N105UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  7 N107US   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  8 N108UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  9 N109UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 10 N110UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## # ... with 3,312 more rows, and 2 more variables: seats_per_engine <dbl>,
## #   turbo_prop_plane <lgl>
```

```
    #select(seats_per_engine, turbo_prop_plane, everything())
```

`mutate()` vytvořilo dva nové sloupce. Sloupec `seats_per_engine` obsahuje zaokrouhlený počet sedadel na motor. Za povšimnutí stojí způsob, jakým byl jeho výpočet ve funkci `mutate()` specifikován. Na levé straně je jméno nově vytvářeného sloupce. Na pravé straně od "=" je postup, který se má použít pro vytvoření jejího obsahu. Jména sloupců z tabulky se přitom používají jako proměnné. Příklad také ukazuje, že v `mutate()` je možné používat komplikované výrazy včetně trubek %>%.

V jednom volání `mutate()` je možné vytvořit více nových sloupců. Jednotlivé specifikace jsou ve volání odděleny čárkou. Druhý vytvořený sloupec ukazuje příklad vytvoření logické proměnné. Ohledně typu zpracovávaných nebo výsledných proměnných nemá `mutate()` žádné omezení.

`mutate()` přidává nově vytvořené sloupce na konec tabulky. Proto je v příkladu použita funkce `select()`, která je přesunuje na začátek tabulky. Verze 1.0.0 umožňuje nastavit, kde se v tabulce nové sloupce vytvoří, nicméně tato funkcionalita je stále ve fázi vývoje.

U popisu fungování `mutate()` je výše zmíněná možnost modifikace stávajících sloupců. V praxi se taková operace provede jednoduše. Předpokládejme, že chceme sloupec `year` nahradit jeho vlastním logaritmem:

```
planes %>%
    mutate(
        year = log(year)
    )
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type          manufacturer   model  engines seats speed engine
##    <chr>   <dbl> <chr>         <chr>          <chr>    <int> <int> <int> <chr>
##  1 N10156   7.60 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  2 N102UW   7.60 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  3 N103US   7.60 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  4 N104UW   7.60 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  5 N10575   7.60 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  6 N105UW   7.60 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
```

```
##  7 N107US   7.60 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  8 N108UW   7.60 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  9 N109UW   7.60 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 10 N110UW   7.60 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## # ... with 3,312 more rows
```

Pokud jméno nového sloupce odpovídá některému sloupci, který již je v tabulce obsažen, je tento novým sloupcem nahrazen.

`mutate()` umí pracovat i s proměnnými, které nejsou součástí tabulky. V následujícím případě je nově vytvořený sloupec `this_is_true` naplněn konstantou přiřazenou do proměnné `x`.

```
x <- TRUE

planes %>%
    mutate(
        this_is_true = x
    ) %>%
    select(this_is_true, everything())
```

```
## # A tibble: 3,322 x 10
##    this_is_true tailnum  year type       manufacturer  model engines seats speed
##    <lgl>        <chr>   <int> <chr>      <chr>         <chr>   <int> <int> <int>
##  1 TRUE         N10156   2004 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  2 TRUE         N102UW   1998 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  3 TRUE         N103US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  4 TRUE         N104UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  5 TRUE         N10575   2002 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  6 TRUE         N105UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  7 TRUE         N107US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  8 TRUE         N108UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  9 TRUE         N109UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## 10 TRUE         N110UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## # ... with 3,312 more rows, and 1 more variable: engine <chr>
```

Stejného výsledku by bylo dosaženo, kdyby byla konstanta definována přímo v `mutate()` tj. `this_is_true = TRUE`.

Na proměnnou, která takto do `mutate()` vstupuje zvnějšku je uvaleno omezení: musí mít délku jedna, nebo délku odpovídající počtu řádků tabulky. Tato podmínka není v následujícím příkladu splněna (vektor `x` má délku 3):

```
x <- c(TRUE, TRUE, TRUE)

planes %>%
    mutate(
        this_is_true = x
    ) %>%
    select(this_is_true, everything())
```

```
## Error: Problem with `mutate()` column `this_is_true`.
## i `this_is_true = x`.
## i `this_is_true` must be size 3322 or 1, not 3.
```

Pokud má vektor `x` délku 1, potom je tato jedna hodnota přiřazena ke každému řádku. Pokud je délka `x` právě rovna počtu řádků, potom je ke každému řádku přiřazena hodnota na odpovídající pozici:

```
x <- 1:nrow(planes)

planes %>%
    mutate(
        new_variable = x
    ) %>%
    select(new_variable, everything())
```

```
## # A tibble: 3,322 x 10
##    new_variable tailnum  year type       manufacturer  model engines seats speed
##           <int> <chr>   <int> <chr>      <chr>         <chr>   <int> <int> <int>
##  1            1 N10156   2004 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  2            2 N102UW   1998 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  3            3 N103US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  4            4 N104UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  5            5 N10575   2002 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  6            6 N105UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  7            7 N107US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  8            8 N108UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  9            9 N109UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## 10           10 N110UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## # ... with 3,312 more rows, and 1 more variable: engine <chr>
```

Naprosto stejná pravidla platí pro funkce. V příkladu je použita funkce `rnorm(n)`, která vrací `n` výběrů z normálního rozdělení. První dva příklady jsou vyhodnoceny bez problémů. Poslední je nekorektní a skončí chybou, protože `rnorm(3)` vrací vektor o délce 3.

```
planes %>%
    mutate(
        new_variable = rnorm(1)
    ) %>%
    select(new_variable, everything())
```

```
## # A tibble: 3,322 x 10
##    new_variable tailnum  year type       manufacturer  model engines seats speed
##           <dbl> <chr>   <int> <chr>      <chr>         <chr>   <int> <int> <int>
##  1      -0.0155 N10156   2004 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  2      -0.0155 N102UW   1998 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  3      -0.0155 N103US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  4      -0.0155 N104UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  5      -0.0155 N10575   2002 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  6      -0.0155 N105UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  7      -0.0155 N107US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  8      -0.0155 N108UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  9      -0.0155 N109UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## 10      -0.0155 N110UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## # ... with 3,312 more rows, and 1 more variable: engine <chr>
```

```
planes %>%
    mutate(
        new_variable = rnorm(nrow(planes))
    ) %>%
    select(new_variable, everything())
```

```
## # A tibble: 3,322 x 10
##    new_variable tailnum  year type       manufacturer  model engines seats speed
##           <dbl> <chr>   <int> <chr>      <chr>         <chr>   <int> <int> <int>
##  1       -0.876 N10156   2004 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  2       -1.68  N102UW   1998 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  3       -0.520 N103US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  4        0.595 N104UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  5       -0.862 N10575   2002 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  6        0.316 N105UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  7       -0.679 N107US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  8       -0.224 N108UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  9        1.92  N109UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## 10       -0.628 N110UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## # ... with 3,312 more rows, and 1 more variable: engine <chr>
```

```
planes %>%
    mutate(
        new_variable = rnorm(3)
    ) %>%
    select(new_variable, everything())
```

```
## Error: Problem with `mutate()` column `new_variable`.
## i `new_variable = rnorm(3)`.
## i `new_variable` must be size 3322 or 1, not 3.
```

16.2.1.1 Úskalí `mutate()`

Výše byl použit příklad, ve kterém byla při stanovení hodnoty použita proměnná definovaná mimo tabulku. Při troše smůly se může stát, že jméno této proměnné se bude shodovat se jménem některého sloupce. V souladu s logikou R dostane přednost obsah sloupce:

```
tailnum <- TRUE

planes %>%
    mutate(
        this_is_true = tailnum
    ) %>%
    select(this_is_true, everything())
```

```
## # A tibble: 3,322 x 10
##    this_is_true tailnum  year type       manufacturer  model engines seats speed
##    <chr>        <chr>   <int> <chr>      <chr>         <chr>   <int> <int> <int>
##  1 N10156       N10156   2004 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  2 N102UW       N102UW   1998 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  3 N103US       N103US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  4 N104UW       N104UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  5 N10575       N10575   2002 Fixed win~ EMBRAER       EMB-~       2    55    NA
##  6 N105UW       N105UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  7 N107US       N107US   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  8 N108UW       N108UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
##  9 N109UW       N109UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## 10 N110UW       N110UW   1999 Fixed win~ AIRBUS INDUS~ A320~       2   182    NA
## # ... with 3,312 more rows, and 1 more variable: engine <chr>
```

Další úskalí v použití spočívá v tom, že `mutate()` pracuje nad celou tabulkou a ne nad jednotlivými řádky. Toto chování lze změnit pomocí vhodného zgrupování, ale je potřeba ho mít na paměti. Co to znamená v praxi:

```
planes %>%
    mutate(
        mean_year = mean(year, na.rm = TRUE)
    ) %>%
    select(mean_year, everything())
```

```
## # A tibble: 3,322 x 10
##    mean_year tailnum  year type    manufacturer model engines seats speed engine
##        <dbl> <chr>   <int> <chr>   <chr>        <chr>   <int> <int> <int> <chr>
##  1     2000. N10156   2004 Fixed ~ EMBRAER      EMB-~       2    55    NA Turbo~
##  2     2000. N102UW   1998 Fixed ~ AIRBUS INDU~ A320~       2   182    NA Turbo~
##  3     2000. N103US   1999 Fixed ~ AIRBUS INDU~ A320~       2   182    NA Turbo~
##  4     2000. N104UW   1999 Fixed ~ AIRBUS INDU~ A320~       2   182    NA Turbo~
##  5     2000. N10575   2002 Fixed ~ EMBRAER      EMB-~       2    55    NA Turbo~
##  6     2000. N105UW   1999 Fixed ~ AIRBUS INDU~ A320~       2   182    NA Turbo~
##  7     2000. N107US   1999 Fixed ~ AIRBUS INDU~ A320~       2   182    NA Turbo~
##  8     2000. N108UW   1999 Fixed ~ AIRBUS INDU~ A320~       2   182    NA Turbo~
##  9     2000. N109UW   1999 Fixed ~ AIRBUS INDU~ A320~       2   182    NA Turbo~
## 10     2000. N110UW   1999 Fixed ~ AIRBUS INDU~ A320~       2   182    NA Turbo~
## # ... with 3,312 more rows
```

`mutate()` v tomto případě vypočítal průměrnou hodnotu ze všech roků a tu přiřadil ke všem sloupcům. Opět je to dáno tím, že `mean` neprodukuje vektor o délce odpovídající počtu řádků, ale vektor o délce 1.

16.2.2 Agregace proměnných se `summarise()`

Podstatou agregace je shrnutí obsahu tabulky (jednoho nebo více sloupců) a vytvoření nové tabulky, která obsahuje tyto agregované hodnoty (typicky statistiky jako průměr, medián, minimum, atp.).

Figure 16.2: Agregace obsahu tabulky se `summarise()`

Pro tyto účely slouží v *dplyr* funkce `summarise()`. Její použití se v logice velmi podobá `mutate()`. To ilustruje následující příklad:

```
planes %>%
    summarise(
        min_year = min(year, na.rm = TRUE),
        max_year = max(year, na.rm = TRUE),
        min_engines = min(engines, na.rm = TRUE),
        max_engines = max(engines, na.rm = TRUE)
    )
```

```
## # A tibble: 1 x 4
##   min_year max_year min_engines max_engines
##      <int>    <int>       <int>       <int>
## 1     1956     2013           1           4
```

V tomto volání funkce `summarise()` jsou vytvořeny 4 agregované hodnoty: maxima a minima ze sloupců `year` a `engines`. Výsledkem je tabulka, která podle stanovených pravidel shrnuje celou tabulku do jediného řádku.

16.3 Další užitečné funkce z balíku *dplyr*

Balík *dplyr* obsahuje opravdu velmi mnoho funkcí, které pracují nad jednou tabulkou. V této kapitole je představen lehký výběr těch, které se v praxi datové analýzy používají opravdu často.

Funkce `distinct()` je ekvivalentem `unique()` – vrací tabulku, která obsahuje pouze unikátní pozorování. V případě shody více řádků zachovává v nové tabulce první z nich. Proti `unique()` je rychlejší a hlavně umožňuje specifikovat sloupce, podle kterých se má unikátnost pozorování posuzovat:

```
planes %>%
    distinct(manufacturer, type)
```

```
## # A tibble: 37 x 2
##    type                      manufacturer
##    <chr>                     <chr>
##  1 Fixed wing multi engine   EMBRAER
##  2 Fixed wing multi engine   AIRBUS INDUSTRIE
##  3 Fixed wing multi engine   BOEING
##  4 Fixed wing multi engine   AIRBUS
##  5 Fixed wing multi engine   BOMBARDIER INC
##  6 Fixed wing single engine  CESSNA
##  7 Fixed wing multi engine   CESSNA
##  8 Fixed wing single engine  JOHN G HESS
##  9 Fixed wing multi engine   GULFSTREAM AEROSPACE
## 10 Rotorcraft                SIKORSKY
## # ... with 27 more rows
```

V základním nastavení je výstupní tabulka omezena pouze na proměnné, které byly použity k posouzení unikátnosti. Toto chování se dá změnit pomocí parametru `.keep_all`:

```
planes %>%
    distinct(manufacturer, type, .keep_all = TRUE)
```

```
## # A tibble: 37 x 9
##    tailnum  year type        manufacturer    model  engines seats speed engine
##    <chr>   <int> <chr>       <chr>           <chr>    <int> <int> <int> <chr>
##  1 N10156   2004 Fixed wing~ EMBRAER         EMB-1~       2    55    NA Turbo-f~
##  2 N102UW   1998 Fixed wing~ AIRBUS INDUSTR~ A320-~       2   182    NA Turbo-f~
##  3 N11206   2000 Fixed wing~ BOEING          737-8~       2   149    NA Turbo-f~
##  4 N125UW   2009 Fixed wing~ AIRBUS          A320-~       2   182    NA Turbo-f~
##  5 N131EV   2009 Fixed wing~ BOMBARDIER INC  CL-60~       2    95    NA Turbo-f~
##  6 N201AA   1959 Fixed wing~ CESSNA          150          1     2    90 Recipro~
##  7 N202AA   1980 Fixed wing~ CESSNA          421C         2     8    90 Recipro~
##  8 N315AT     NA Fixed wing~ JOHN G HESS     AT-5         1     2    NA 4 Cycle
##  9 N344AA   1992 Fixed wing~ GULFSTREAM AER~ G-IV         2    22    NA Turbo-f~
## 10 N347AA   1985 Rotorcraft  SIKORSKY        S-76A        2    14    NA Turbo-s~
## # ... with 27 more rows
```

Užitečnou funkcí je řazení pozorování. To má v *dplyr* na starosti funkce `arrange()`. `arrange()` přijímá jako parametry vstupní tabulku a jména sloupců, podle kterých má tabulku seřadit:

```
planes %>%
    arrange(manufacturer,engines)
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type           manufacturer model  engines seats speed engine
##    <chr>   <int> <chr>          <chr>        <chr>    <int> <int> <int> <chr>
##  1 N365AA   2001 Rotorcraft     AGUSTA SPA   A109E        2     8    NA Turbo-s~
##  2 N125UW   2009 Fixed wing mu~ AIRBUS       A320-~       2   182    NA Turbo-f~
##  3 N126UW   2009 Fixed wing mu~ AIRBUS       A320-~       2   182    NA Turbo-f~
##  4 N127UW   2010 Fixed wing mu~ AIRBUS       A320-~       2   182    NA Turbo-f~
##  5 N128UW   2010 Fixed wing mu~ AIRBUS       A320-~       2   182    NA Turbo-f~
##  6 N150UW   2013 Fixed wing mu~ AIRBUS       A321-~       2   199    NA Turbo-f~
##  7 N151UW   2013 Fixed wing mu~ AIRBUS       A321-~       2   199    NA Turbo-f~
##  8 N152UW   2013 Fixed wing mu~ AIRBUS       A321-~       2   199    NA Turbo-f~
##  9 N153UW   2013 Fixed wing mu~ AIRBUS       A321-~       2   199    NA Turbo-f~
## 10 N154UW   2013 Fixed wing mu~ AIRBUS       A321-~       2   199    NA Turbo-f~
## # ... with 3,312 more rows
```

`arrange()` nejprve řadí tabulku podle první zadaného sloupce, následně podle druhého, atp. Směr řazení je možné změnit pomocí speciální funkce `desc()`:

```
planes %>%
    arrange(desc(manufacturer),engines)
```

```
## # A tibble: 3,322 x 9
##    tailnum  year type        manufacturer    model  engines seats speed engine
##    <chr>   <int> <chr>       <chr>           <chr>    <int> <int> <int> <chr>
##  1 N397AA   1985 Fixed wing~ STEWART MACO    FALCO~       1     2    NA Recipro~
##  2 N521AA     NA Fixed wing~ STEWART MACO    FALCO~       1     2    NA Recipro~
##  3 N347AA   1985 Rotorcraft  SIKORSKY        S-76A        2    14    NA Turbo-s~
##  4 N537JB   2012 Rotorcraft  ROBINSON HELIC~ R66          1     5    NA Turbo-s~
##  5 N376AA   1978 Fixed wing~ PIPER           PA-32~       1     7    NA Recipro~
##  6 N425AA   1968 Fixed wing~ PIPER           PA-28~       1     4   107 Recipro~
##  7 N545AA   1976 Fixed wing~ PIPER           PA-32~       1     7   126 Recipro~
##  8 N350AA   1980 Fixed wing~ PIPER           PA-31~       2     8   162 Recipro~
##  9 N525AA   1980 Fixed wing~ PIPER           PA-31~       2     8   162 Recipro~
## 10 N377AA     NA Fixed wing~ PAIR MIKE E     FALCO~       1     2    NA Recipro~
## # ... with 3,312 more rows
```

16.4 Operace nad sloupci

Funkce z balíku *dplyr* umožňují spouštět funkce nad specifikovanými sloupci tabulky. Prvním příkladem užití takové funkcionality může být výběr sloupců určitého datového typu:

```
planes %>%
  select(where(is.character))
```

```
## # A tibble: 3,322 x 5
##    tailnum type                    manufacturer     model     engine
##    <chr>   <chr>                   <chr>            <chr>     <chr>
##  1 N10156  Fixed wing multi engine EMBRAER          EMB-145XR Turbo-fan
##  2 N102UW  Fixed wing multi engine AIRBUS INDUSTRIE A320-214  Turbo-fan
##  3 N103US  Fixed wing multi engine AIRBUS INDUSTRIE A320-214  Turbo-fan
```

```
##  4 N104UW  Fixed wing multi engine AIRBUS INDUSTRIE A320-214  Turbo-fan
##  5 N10575  Fixed wing multi engine EMBRAER          EMB-145LR Turbo-fan
##  6 N105UW  Fixed wing multi engine AIRBUS INDUSTRIE A320-214  Turbo-fan
##  7 N107US  Fixed wing multi engine AIRBUS INDUSTRIE A320-214  Turbo-fan
##  8 N108UW  Fixed wing multi engine AIRBUS INDUSTRIE A320-214  Turbo-fan
##  9 N109UW  Fixed wing multi engine AIRBUS INDUSTRIE A320-214  Turbo-fan
## 10 N110UW  Fixed wing multi engine AIRBUS INDUSTRIE A320-214  Turbo-fan
## # ... with 3,312 more rows
```

Výběr byl proveden s použitím dodatečné select-helper funce `where()`. Do té byl vložen výraz, který byl vyhodnocen nad sloupci tabulky. Výsledek, logický vektor, byl potom použit pro výběr funkcí `select()`.

Všiměte si, že funkce `is.character()` je v příkladu použita bez závorek. Parametrem `where()` je totiž funkce samotná a nikoliv její výstup.

dplyr umožňuje uživateli provádět i sofistikovanější operace – typicky modifikovat sloupce, které splňují určitou podmínku. Můžeme například chtít konvertovat číselné sloupce na `character`. Pro tyto účely slouží funkce `across()`. Ta má dva vstupy: (a) výraz, který identifikuje sloupce, které se mají modifikovat, a (b) výraz, který se má pro samotnou modifikaci použít. řešení by mohlo vypadat následujícím způsobem:

```
planes %>%
  mutate(
    across(
      where(is.numeric),
      as.character
    )
  )
```

```
## # A tibble: 3,322 x 9
##    tailnum year  type           manufacturer   model  engines seats speed engine
##    <chr>   <chr> <chr>          <chr>          <chr>  <chr>   <chr> <chr> <chr>
##  1 N10156  2004  Fixed wing m~ EMBRAER        EMB-1~ 2       55    <NA>  Turbo-~
##  2 N102UW  1998  Fixed wing m~ AIRBUS INDUST~ A320-~ 2       182   <NA>  Turbo-~
##  3 N103US  1999  Fixed wing m~ AIRBUS INDUST~ A320-~ 2       182   <NA>  Turbo-~
##  4 N104UW  1999  Fixed wing m~ AIRBUS INDUST~ A320-~ 2       182   <NA>  Turbo-~
##  5 N10575  2002  Fixed wing m~ EMBRAER        EMB-1~ 2       55    <NA>  Turbo-~
##  6 N105UW  1999  Fixed wing m~ AIRBUS INDUST~ A320-~ 2       182   <NA>  Turbo-~
##  7 N107US  1999  Fixed wing m~ AIRBUS INDUST~ A320-~ 2       182   <NA>  Turbo-~
##  8 N108UW  1999  Fixed wing m~ AIRBUS INDUST~ A320-~ 2       182   <NA>  Turbo-~
##  9 N109UW  1999  Fixed wing m~ AIRBUS INDUST~ A320-~ 2       182   <NA>  Turbo-~
## 10 N110UW  1999  Fixed wing m~ AIRBUS INDUST~ A320-~ 2       182   <NA>  Turbo-~
## # ... with 3,312 more rows
```

Funkce `across()` funguje výborně s funkcemi jako je `mutate()` nebo `summarise()`. V *dplyr* 1.0.4 přibyly nové funkce, které jsou navženy tak, aby podobným způsobem umožnili pracovat s `filter()`. Jde o `if_any()` a `if_all()`. Jejich syntaxe a parametry jsou stejné jako u `across()`, ale na ro rozdíl od `across()` vracejí logický vektor, který je požadovaným vstupem funkce `filter()`.

Následující příklad ukazuje, jak řešit obyvklý problém v datové analýze – někde nám utíkají pozorování, protože některé řádky v nějakém sloupci obsahují `NA`. Funkce `if_any()` nám umožňuje takové řádky lehce najít:

```
planes %>%
  filter(
    if_any(
      everything(), # tj. hledej ve všech sloupcích
```

```
      is.na
      )
    )
```

```
## # A tibble: 3,299 x 9
##    tailnum  year type          manufacturer   model  engines seats speed engine
##    <chr>   <int> <chr>         <chr>          <chr>    <int> <int> <int> <chr>
##  1 N10156   2004 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  2 N102UW   1998 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  3 N103US   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  4 N104UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  5 N10575   2002 Fixed wing m~ EMBRAER        EMB-1~       2    55    NA Turbo-~
##  6 N105UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  7 N107US   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  8 N108UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
##  9 N109UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## 10 N110UW   1999 Fixed wing m~ AIRBUS INDUST~ A320-~       2   182    NA Turbo-~
## # ... with 3,289 more rows
```

Funkce `across()`, `where()`, `if_any()` a `if_all()` nahrazují tzv. scoped varianty základních funkcí ze starších verzí *dplyr*.

16.5 Operace nad skupinami řádků

Všechny předchozí funkce lze s různou mírou elegance nahradit funkcemi ze základního R. Grupované operace však lze nahradit jen obtížně a v žádném případě ne elegantně. Podstatou zgrupované operace je vyhodnocení funkce nad jednotlivými segmenty tabulky. Na obrázku je zgrupovaná operace provedena funkce `summarise()`:

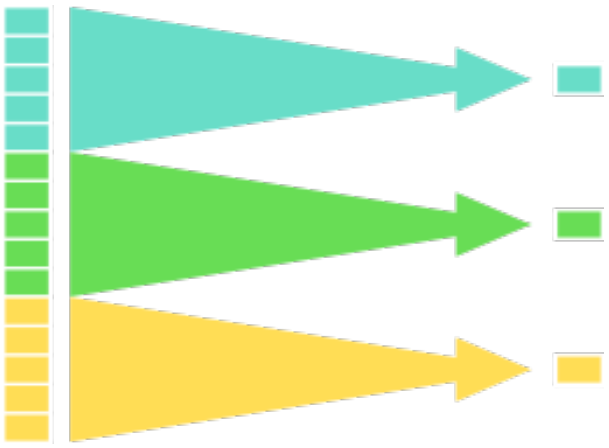


Figure 16.3: Zgrupované operace, příklad `summarise()`

`summarise()` je vykonáno nad jednotlivými barvenými grupami. Výsledky za jednotlivé grupy jsou následně složeny do nové tabulky.

V praktickém nasazení nás například může zajímat minimální, maximální a průměrný počet sedadel v letadlech jednotlivých výrobců.

V prvním kroku je potřeba pomocí funkce `group_by()` vytvořit grupování – tj. identifikovat řádky, které tvoří jednu grupu. Následně je možné volat funkci `summarise()`:

```
planes %>%
    group_by(manufacturer) %>%
    summarise(
        min_seats  = min(seats, na.rm  = TRUE),
        mean_seats = mean(seats, na.rm = TRUE),
        max_seats  = max(seats, na.rm = TRUE)
    )
```

```
## # A tibble: 35 x 4
##    manufacturer           min_seats mean_seats max_seats
##    <chr>                      <int>      <dbl>     <int>
##  1 AGUSTA SPA                     8        8           8
##  2 AIRBUS                       100      221.        379
##  3 AIRBUS INDUSTRIE             145      187.        379
##  4 AMERICAN AIRCRAFT INC          2        2           2
##  5 AVIAT AIRCRAFT INC             2        2           2
##  6 AVIONS MARCEL DASSAULT        12       12          12
##  7 BARKER JACK L                  2        2           2
##  8 BEECH                          9        9.5        10
##  9 BELL                           5        8          11
## 10 BOEING                       100      175.        450
## # ... with 25 more rows
```

Protože nás zajímají počty sedadel v letadlech *"jednotlivých výrobců"* je pro zgrupování použita proměnná `manufacturer`. `group_by` však umí vytvořit i grupy tvořené kombinací více proměnných. Například by bylo možné zjistit počty sedadel pro skupinu vymezenou výrobcem a typem letounu:

```
planes %>%
    group_by(manufacturer, type) %>%
    summarise(
        min_seats  = min(seats, na.rm  = TRUE),
        mean_seats = mean(seats, na.rm = TRUE),
        max_seats  = max(seats, na.rm = TRUE)
    )
```

```
## `summarise()` has grouped output by 'manufacturer'. You can override using the `.groups` argument
```

```
## # A tibble: 37 x 5
## # Groups:   manufacturer [35]
##    manufacturer           type                      min_seats mean_seats max_seats
##    <chr>                  <chr>                         <int>      <dbl>     <int>
##  1 AGUSTA SPA             Rotorcraft                        8        8           8
##  2 AIRBUS                 Fixed wing multi engine         100      221.        379
##  3 AIRBUS INDUSTRIE       Fixed wing multi engine         145      187.        379
##  4 AMERICAN AIRCRAFT INC  Fixed wing single engine          2        2           2
##  5 AVIAT AIRCRAFT INC     Fixed wing single engine          2        2           2
##  6 AVIONS MARCEL DASSAULT Fixed wing multi engine          12       12          12
##  7 BARKER JACK L          Fixed wing single engine          2        2           2
##  8 BEECH                  Fixed wing multi engine           9        9.5        10
##  9 BELL                   Rotorcraft                        5        8          11
## 10 BOEING                 Fixed wing multi engine         100      175.        450
## # ... with 27 more rows
```

V `group_by()` je možné použít proměnné všech typů (jakkoliv u *double* to asi příliš často nedává smysl).

Grupované operace pochopitelně nejsou omezeny pouze na `summarise()`. Následující příklad ukazuje použití s `mutate()`. Jako cvičení můžete kód analyzovat a zjistit, co dělá.

```
planes %>%
    group_by(manufacturer) %>%
    mutate(
        year_diff = year - mean(year, na.rm = TRUE)
    ) %>%
    select(tailnum, manufacturer, year, year_diff) %>%
    arrange(manufacturer, year)
```

```
## # A tibble: 3,322 x 4
## # Groups:   manufacturer [35]
##    tailnum manufacturer  year year_diff
##    <chr>   <chr>        <int>     <dbl>
##  1 N365AA  AGUSTA SPA    2001      0
##  2 N186US  AIRBUS        2002     -5.20
##  3 N187US  AIRBUS        2002     -5.20
##  4 N188US  AIRBUS        2002     -5.20
##  5 N338NB  AIRBUS        2002     -5.20
##  6 N339NB  AIRBUS        2002     -5.20
##  7 N340NB  AIRBUS        2002     -5.20
##  8 N341NB  AIRBUS        2002     -5.20
##  9 N342NB  AIRBUS        2002     -5.20
## 10 N343NB  AIRBUS        2002     -5.20
## # ... with 3,312 more rows
```

V prvním kroku kód přidá zgrupování k tabulce planes. Výsledek této operace můžeme vidět pomocí funkce `class()`:

```
planes %>%
    group_by(manufacturer) %>%
    class()
```

```
## [1] "grouped_df" "tbl_df"     "tbl"        "data.frame"
```

Třída tabulky `planes` byla rozšířena o `grouped_df`. To umožní kompatibilním metodám nakládat s tabulkou speciálním způsobem: provést operaci zgrupovaně. Pokud pro danou funkci není "zgrupovaná" metoda dostupná, provede se funkce jako obvykle nad celou tabulkou:

```
planes %>%
    group_by(manufacturer) %>%
    summary()
```

```
##    tailnum               year         type           manufacturer
##  Length:3322        Min.   :1956   Length:3322        Length:3322
##  Class :character   1st Qu.:1997   Class :character   Class :character
##  Mode  :character   Median :2001   Mode  :character   Mode  :character
##                     Mean   :2000
##                     3rd Qu.:2005
##                     Max.   :2013
##                     NA's   :70
##     model              engines          seats           speed
```

```
##  Length:3322        Min.   :1.000   Min.   :  2.0   Min.   : 90.0
##  Class :character   1st Qu.:2.000   1st Qu.:140.0   1st Qu.:107.5
##  Mode  :character   Median :2.000   Median :149.0   Median :162.0
##                     Mean   :1.995   Mean   :154.3   Mean   :236.8
##                     3rd Qu.:2.000   3rd Qu.:182.0   3rd Qu.:432.0
##                     Max.   :4.000   Max.   :450.0   Max.   :432.0
##                                                     NA's   :3299
##     engine
##  Length:3322
##  Class :character
##  Mode  :character
##
##
##
##
```

Třída `grouped_df()` zůstává u tabulky zachována, dokud není jinou funkcí odstraněna. *dplyr* umožňuje grupování odstranit funkcí `ungroup()`:

```
planes %>%
    group_by(manufacturer) %>%
    ungroup() %>%
    class()
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

Částečnou výjimkou z tohoto pravidla je funkce `summarise()`, která získala parametr `.groups`, který umožňuje nastavit, jak má být nastaveno grupování u výstupní tabulky.

V dalším kroku příkladu je volána funkce `mutate()`. Ta zvlášť pro každou grupu vypočítá průměrný rok výroby (`mean(year, na.rm = TRUE)`) a tuto hodnotu (vektor o délce 1) odečte od všech hodnot v proměnné `year`. Výsledkem je tak vektor o délce identické s počtem řádků v grupě. Tento vektor je přidán jako sloupec `year_diff`. Tabulka je následně zpřehledněna voláním `select()` a `mutate()`.

16.5.1 Bezpečné grupování

Zgrupované operace představují mimořádně mocný nástroj, který zásadně zjednodušuje a zpřehledňuje datovou analýzu. Mají však svá rizika a to zejména mezi židlí a klávesnicí. Pokud uživatel zapomene, že ve skutečnosti pracuje se zgrupovanou tabulkou může dostat bez varování zásadně odlišné výsledky. Proto je rozumné tabulku "na konci trubky" odgrupovat.

Potenciální riziko si uvědomují i tvůrci *dplyr*. Například od verze 1.0.0 vrací `summarise()` po operaci nad zgrupovanou tabulkou varování. Od této verze je také možné také parametrem funkce nastavit, zda a jak má `summarise()` grupování zachovat. (V případě nastavení tohoto parametru už žádné varování nevrací.)

16.6 Slovesa pracující se dvěma (nebo více) tabulkami

Data bývají často dostupná ve více tabulkách, které mohou například mohou pocházet z různých zdrojů: HDP ze Světové banky, migrace z Eurostatu, atp. Pro účely datové analýzy je nutné takové tabulky spojovat do jednoho celku.

dplyr podporuje dva druhy spojovacích operací:

- *bind* spojuje tabulky, které mají stejnou strukturu – v podstatě přidává sloupce (`bind_cols()`) nebo řádky (`bind_rows()`)
- *join* slučuje tabulky podle určitého definovaného klíče – například sloučí k sobě údaje o jednom člověku z více tabulek, které mohou mít naprosto odlišnou strukturu (více funkcí `*_join()`)

16.6.1 Spojování tabulek s `bind_*()`

Funkcí `bind_rows()` a `bind_cols()` je spojovat tabulky se stejnou strukturou. V případě, že jsou pozorování se stejnými proměnnými rozděleny do více tabulek, je potřeba tabulky poskládat "pod sebe". Jinými slovy přidávat další a další řádky s dodatečnými pozorováními. V tomto případě se hodí použít funkci `bind_rows(...)`. Ta jako argument přijímá jména (neomezeného počtu) tabulek a nebo seznam (list) tabulek. Fungování `bind_rows()` můžeme demonstrovat na tabulce `dplyr::band_members`:

```
band_members
```

```
## # A tibble: 3 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

Do `bind_rows()` můžeme vložit více tabulek:

```
bind_rows(band_members, band_members, band_members)
```

```
## # A tibble: 9 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
## 4 Mick  Stones
## 5 John  Beatles
## 6 Paul  Beatles
## 7 Mick  Stones
## 8 John  Beatles
## 9 Paul  Beatles
```

... nebo list tabulek:

```
bind_rows(
    list(band_members, band_members, band_members)
)
```

```
## # A tibble: 9 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
## 4 Mick  Stones
## 5 John  Beatles
## 6 Paul  Beatles
## 7 Mick  Stones
## 8 John  Beatles
## 9 Paul  Beatles
```

Výsledky jsou pochopitelně stejné. Schopnost spojit tabulky uložené v seznamu je zvláště užitečná v případě, že pracujeme například s výstupem funkce `map()` z balíku **purrr**.

Předchozí příklady byly bezproblémové, protože tabulky měly stejnou strukturu – tedy stejně pojmenované sloupce se stejnými datovými typy. Co se stane v případě volání `bind_rows()` na nekonzistentní tabulky ukazuje následující příklad:

```
band_members %>%
    rename(NAME = name) %>%
    bind_rows(., band_members)
```

```
## # A tibble: 6 x 3
##   NAME  band    name
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles <NA>
## 3 Paul  Beatles <NA>
## 4 <NA>  Stones  Mick
## 5 <NA>  Beatles John
## 6 <NA>  Beatles Paul
```

`bind_rows()` pod sebe složil hodnoty ze sloupců stejného jména. Sloupce s neshodujícími se jmény zachoval, ale do tabulky doplnil `NA`.

```
band_members %>%
    rename(NAME = name) %>%
    bind_rows(., band_members)
```

```
## # A tibble: 6 x 3
##   NAME  band    name
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles <NA>
## 3 Paul  Beatles <NA>
## 4 <NA>  Stones  Mick
## 5 <NA>  Beatles John
## 6 <NA>  Beatles Paul
```

V případě nekonzistentních datových typů je situace zajímavější. V následujícím příkladu je sloupec name konvertován z *character* na *factor* a následně je tabulka spojena s nezměněnou tabulkou `band_members`:

```
band_members %>%
    mutate(
        name = as.factor(name)
        ) %>%
    bind_rows(., band_members)
```

```
## # A tibble: 6 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
## 4 Mick  Stones
## 5 John  Beatles
## 6 Paul  Beatles
```

R umí provést automatickou konverzi faktorů na znaky. Provede ji, ale vypíše i upovídané varování. Výsledek je nicméně perfektně použitelný a ve většině případů bude odpovídat přání uživatele.

Větší problém nastane, pokud se ve spojovaných tabulkách vyskytnou sloupce stejného jména a rozdílných datových typů, u kterých R neumí provést automatickou konverzi. V tomto případě jde o *double* a *character*:

```
band_members %>%
    mutate(
        name = rnorm1)
        ) %>%
    bind_rows(., band_members)
```

```
## Error: <text>:4:9: unexpected ')'
## 3:         name = rnorm1)
## 4:         )
##            ^
```

Tato operace se neprovede a R vrátí chybu.

Podobně jako `bind_rows()` funguje funkce `bind_cols()`. Tabulky ovšem neskládá “pod sebe”“, ale”vedle sebe". Předpokladem jejího použití je opět shodná struktura tabulek. To v tomto případě znamená zejména to, že jedno pozorování je vždy na stejném řádku. Pozorování je tak vlastně identifikováno číslem řádku. Syntaxe je stejná jako u `bind_rows()`:

```
bind_cols(band_members, band_members, band_members)
```

```
## New names:
## * name -> name...1
## * band -> band...2
## * name -> name...3
## * band -> band...4
## * name -> name...5
## * ...
```

```
## # A tibble: 3 x 6
##   name...1 band...2 name...3 band...4 name...5 band...6
##   <chr>    <chr>    <chr>    <chr>    <chr>    <chr>
## 1 Mick     Stones   Mick     Stones   Mick     Stones
## 2 John     Beatles  John     Beatles  John     Beatles
## 3 Paul     Beatles  Paul     Beatles  Paul     Beatles
```

Za povšimnutí stojí, že pokud jsou ve spojovaných tabulkách shodná jména sloupců, `bind_cols()` je do výsledné tabulky přidá všechny. Aby se však zabránilo nepřípustné duplicitě ve jménech sloupců, rozšíří duplicitní jména o příponu.

Z logiky věci nejsou problém odlišné datové typy ve spojovaných tabulkách, ale problém můžou představovat tabulky s různým počtem řádků:

```
band_members %>%
    sample_n(2) %>%
    bind_cols(band_members,.)
```

```
## Error: Can't recycle `..1` (size 3) to match `..2` (size 2).
```

Protože u `bind_cols()` jsou pozorování fakticky identifikována číslem řádku, tak není možné takovou operaci smysluplně provést. `bind_cols()` v takovém případě nic nehádá, nic nepředpokládá ani nerecykluje, ale poctivě vyvěsí bílou vlajku a vrátí chybu.

16.6.2 Slučování tabulek s `*_join()`

16.6.2.1 Mutating joins

Slučování tabulek lze provádět pomocí různých slučovacích funkcí. Ty jsou v *dplyr* jednotně pojmenovány tak, že končí řetězcem *_join. *Jejich základní skupina – tzv.* mutating joins*, tj. slučovací funkce, které přidávají sloupce, se v zásadě chová jako "inteligentní" varianta `bind_cols()`. Pozorování však není definováno číslem řádku, ale proměnnou nebo kombinací více proměnných.

Pro ilustraci slučování jsou potřeba dvě tabulky. Vedle `band_members` využijeme `dplyr::band_instruments`:

```
band_instruments
```

```
## # A tibble: 3 x 2
##   name  plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

Pravděpodobně nejčastěji používanou slučovací funkcí je `left_join()`:

```
left_join(x, y, by = NULL, copy = FALSE, suffix = c(".x", ".y"), ...)
```

Jako argument přijímá právě dvě tabulky `x` a `y`. Slučovací funkce v *dplyr* obecně umí pracovat pouze se dvěma tabulkami. Toto omezení je však možné obejít – viz dále.

Dalším důležitým parametrem je `by` jeho hodnota určuje, podle kterých sloupců se má provést slučování – tedy které sloupce definují pozorování. V případě, že je hodnota parametru `NULL`, potom se sloučení provede na základě všech sloupců, jejich jméno je v obou tabulkách. V tomto případě vrátí *dplyr* informaci o tom, které sloupce použil.

Příklad použití `left_join()`:

```
left_join(band_members, band_instruments)
```

```
## Joining, by = "name"
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

`left_join()` funguje tak, že vrací všechny řádky z `x` a všechny sloupce z tabulky `x` i `y`. Řádky z `x`, pro které neexistuje shoda v tabulce `y` mají v připojených sloupcích `NA`.

V tomto příkladu nebyl vyplněn parametr `by`. `left_join()` tedy jako klíč pro slučování použil sloupec `name`, který se jako jediný vyskytoval v obou slučovaných tabulkách. Volání

```
left_join(band_members, band_instruments, by = "name")
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

by pochopitelně vedlo ke stejným výsledkům.

V případě slučování tabulek, ve kterých se vyskytují shodná jména sloupců, která ovšem neidentifikují pozorování je nutné parametr `by` specifikovat. Sloučení tabulek se stejnými názvy sloupců by jinak dopadlo například takto:

```
left_join(band_members, band_members)
```

```
## Joining, by = c("name", "band")
```

```
## # A tibble: 3 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

Výsledkem by byla vstupní tabulka `x`. Pří specifikování identifikačního sloupce (například `name`) bude výsledek odlišný:

```
left_join(band_members, band_members, by = "name")
```

```
## # A tibble: 3 x 3
##   name  band.x  band.y
##   <chr> <chr>   <chr>
## 1 Mick  Stones  Stones
## 2 John  Beatles Beatles
## 3 Paul  Beatles Beatles
```

Slučované tabulky v tomto případě obsahují sloupec se shodným jménem, který není použit ke slučování. V tom případě je tento sloupec (podobně jako u `bind_cols()`) přejmenován připojením přípony. Podobu přípony je možné specifikovat parametrem `suffix`:

```
left_join(band_members, band_members, by = "name", suffix = c(".prvni",".druhy"))
```

```
## # A tibble: 3 x 3
##   name  band.prvni band.druhy
##   <chr> <chr>      <chr>
## 1 Mick  Stones     Stones
## 2 John  Beatles    Beatles
## 3 Paul  Beatles    Beatles
```

V předchozích příkladech byl parametr `by` použit pouze pro specifikaci jednoho sloupce, jehož jméno bylo přítomno v obou slučovaných tabulkách. Možností nastavení je však více:

- by může obsahovat jména více sloupců zadaných jako nepojmenovaný vektor, například by = c("name","band")
- by může obsahovat i pojmenovaný vektor. Ten má ale zvláštní interpretaci. Jméno každého prvku odpovídá v takovém případě jménu sloupce z tabulky x a hodnota jménu sloupce z tabulky y. To umožňuje slučování tabulek, i když tyto neobsahují ani jeden sloupec se shodným jménem. Praktickou ukázkou je následující příklad, který využívá tabulku dplyr::band_instruments2:

```
band_instruments2
```

```
## # A tibble: 3 x 2
##   artist plays
##   <chr>  <chr>
## 1 John   guitar
## 2 Paul   bass
## 3 Keith  guitar
```

Cílem je sloučit tabulky band_members a band_instruments2 podle jména hudebníka. Tato proměnná se však jmenuje name v band_members a artist v band_instruments2:

```
left_join(band_members, band_instruments2, by = c("name" = "artist"))
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

Používání slučovacích funkcí však má svoje úskalí. Představme si situaci, kdy identifikace pozorování není jednoznačná – tedy situaci, kdy identifikátor pozorování odpovídá více řádkům. Vytvoříme tabulku band_instruments3, která bude právě tuto podmínku splňovat:

```
band_instruments3 <- bind_rows(band_instruments, band_instruments)

print(band_instruments3)
```

```
## # A tibble: 6 x 2
##   name  plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
## 4 John  guitar
## 5 Paul  bass
## 6 Keith guitar
```

Identifikátor – jméno hudebníka – teď není unikátní. Přináleží mu právě dva řádky. Následně tuto novou tabulku sloučíme s band_members:

```
left_join(band_members, band_instruments3, by = "name")
```

```
## # A tibble: 5 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 John  Beatles guitar
## 4 Paul  Beatles bass
## 5 Paul  Beatles bass
```

Zde je patrné, že výsledek může být problematický. Výsledná tabulka má větší počet řádků, než vstupní tabulka `x` – řádky, pro které bylo v tabulce `y` více záznamů se namnožily. *dplyr* tuto operaci provedl **bez jakéhokoliv varování. Je proto kritické kontrolovat v průběhu sestavování datasetu konzistenci dat.** Jinak se lehko může stát, že výsledná tabulka a analýza na ni postavená bude bezcenná.

`left_join()` není jediný *mutating join* implementovaný v *dplyr*, další jsou následující:

- `right_join()` je bratr `left_join()`. Vrací sloupce z `x` i `y`, ale řádky z `y`.
- `inner_join()` vrací sloupce z `x` i `y`, ale pouze řádky, která jsou jak v `x`, tak v `y`.
- `full_join()` vrací všechny sloupce a všechny řádky z `x` a `y`.

16.6.2.2 Filtering joins

Druhou skupinou slučovacích funkcí jsou tzv. *filtering joins*. Tyto funkce opět pracují nad dvěma tabulkami `x` a `y`, ale vždy vrací sloupce pouze z tabulky `x`.

První takovou funkcí je `semi_join()`, který vrací pouze ty řádky, které existují v obou tabulkách `x` i `y`. Je to vlastně blízký příbuzný `inner_join()`.

Pro ilustraci fungování *filtering joins* můžeme porovnat výsledky těchto funkcí:

```
inner_join(band_members, band_instruments, by = "name")
```

```
## # A tibble: 2 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
```

```
semi_join(band_members, band_instruments, by = "name")
```

```
## # A tibble: 2 x 2
##   name  band
##   <chr> <chr>
## 1 John  Beatles
## 2 Paul  Beatles
```

`inner_join()` vrací sloupce z obou tabulek – skutečně "slučuje". `semi_join()` vrací sloupce pouze z první tabulky – spíše tedy filtruje na základě informací z druhé tabulky.

Druhou funkcí z této skupiny slučovacích funkcí je `anti_join()`, která je svým způsobem inverzní k `semi_join()`. Vrací řádky z `x`, které nejsou obsaženy v `y`:

```
anti_join(band_members, band_instruments, by = "name")
```

```
## # A tibble: 1 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
```

16.6.2.3 Slučování více tabulek

Všechny funkce `*_join()` pracují s dvěma tabulkami. V drtivé většině případů je to zcela postačující, nicméně najdou se i výjimky. V takovém případě je možné využit funkci `reduce()` z balíku *purrr* (součást *tidyverse*).

```
reduce(.x, .f, ..., .init)
```

Funkce má dva základní argumenty:

- `.x` ... je seznam (*list*) nebo atomický vektor - `.f` ... je funkce přijímající 2 vstupy - `...` ... dodatečné argumenty pro funkci `.f`

`reduce()` funguje tak, že provádí "akumulaci". Nejprve aplikuje funkci `.f` na první dva prvky `.x`. V druhé iteraci aplikuje `.f` na výstup první iterace a na třetí prvek `.x` a tak dále.

```
list(band_members, band_members, band_members, band_instruments) %>%
    reduce(., left_join, by = "name")
```

```
## # A tibble: 3 x 5
##   name  band.x  band.y  band    plays
##   <chr> <chr>   <chr>   <chr>   <chr>
## 1 Mick  Stones  Stones  Stones  <NA>
## 2 John  Beatles Beatles Beatles guitar
## 3 Paul  Beatles Beatles Beatles bass
```

Za pozornost stojí varianty jména sloupce `band`, které přesně odpovídají mechanismu fungování `reduce()`. V první iteraci nastal konflikt jmen sloupců. Obě slučované tabulky obsahovaly sloupec pojmenovaný `band` a proto ve výsledné tabulce dostaly příponu. V druhé iteraci již ke konfliktu nedošlo. Stará tabulka totiž obsahovala jména `band.x` a `band.y` a nová `band`. Sloupec z nové tabulky tak byl připojen bez změny jména.

Příklad by mohl pokračovat:

```
list(band_members, band_members, band_members, band_members) %>%
    reduce(., left_join, by = "name")
```

```
## # A tibble: 3 x 5
##   name  band.x  band.y  band.x.x band.y.y
##   <chr> <chr>   <chr>   <chr>    <chr>
## 1 Mick  Stones  Stones  Stones   Stones
## 2 John  Beatles Beatles Beatles  Beatles
## 3 Paul  Beatles Beatles Beatles  Beatles
```

Part IV

Vizualizace dat

17 Vizualizace dat s balíkem ggplot2

V R existuje celá řada nástrojů pro vizualizaci dat, které jsou opravdu velmi heterogenní. Z úvodních lekcí znáte příklady tzv. `base` grafiky, která přichází s každou instalací R. Příklad může být Vám známá funkce `plot()`. Tato základní implementace nástrojů pro vizualizaci dat sleduje podobnou logiku jako je obvyklá v ostatních systémech: uživatel explicitně sděluje systému co a jak má vykreslit: "Na souřadnice [X,Y] nakresli zelený křížek".

Součástí *tidyverse* je balík *ggplot2*, který přistupuje k vizualizaci zcela jiným způsobem – a to na mnoha úrovních. *ggplot2* je založen na teorii *"grammar of graphics"* (Wilkinson 2005, 2010) a pracuje na mnohem vyšší úrovni abstrakce. Uživatel vlastně předává funkci jako vstup popis dat a konkrétní provedení vizualizace nechává na ni. Tento způsob práce je poměrně neobvyklý, ale po jeho zvládnutí už neexistuje cesta zpět.

ggplot2 má jedno základní omezení – kreslí pouze 2D obrázky. Na konci kapitoly však uvidíte, že nic jiného pravděpodobně ani nepotřebujete.

Co se v této lekci naučíte?

- základní logiku fungování *ggplot2*
- základní logiku ovládání *ggplot2*
- provést jednoduché vizualizace dat a modifikovat jejich vzhled

Složitější úkoly přijdou později.

Let's lock and load...

```
library(tidyverse)
```

17.0.0.1 Poznámka

ggplot2 je tak populární, že existují i jeho porty pro jiné jazyky – například pro Python. I v samotném světě R existuje balík, který s *ggplot2* sdílí nejen filosofii, ale i tvůrce. Balík *ggvis* obsahuje oproti *ggplot2* některá koncepční vylepšení. Celkově je však orientován spíše na interaktivní grafiku a proto pro naše účely není tak vhodný. Nicméně kdo pochopí *ggplot2* zvládne lehce i *ggvis*.

Výhodou popularity *ggplot2* je i obrovské množství balíků, které možnosti *ggplot2* doplňují nebo rozšiřují. Pokud tedy něco neumí samotný *ggplot2* je rozumné porozhlédnout se po balících, které hledanou funkcionalitu doplňují. Neúplnou galerii "rozšíření"" můžete najít například zde: http://www.ggplot2-exts.org/gallery/

17.0.0.1.1 Poznámka 2 POdobně jako ostatní součásti *tidyverse* se i *ggplot2* dramaticky vyvíjí. Tento materiál byl například původně vytvořen s verzí 2.x. Nyní (září 2019) je aktuální verze 3.2.1. Nové verze přinesly zásadní změny, které se však v drtivě většině odehrály "pod povrchem" a pro uživatele nejsou patrné.

Velkou změnou pro uživatele je uvedení `geom_sf()`, který radikálním způsobem zjednodušuje kreslení map.

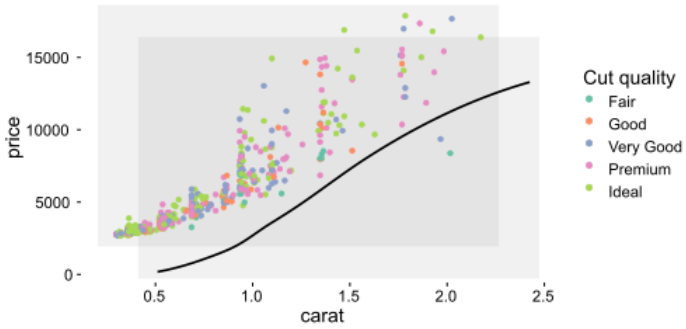


Figure 17.1: Schéma obrázku vytvořeného v *ggplot2*

17.1 Logika fungování *ggplot2*

Pro vysvětlení fungování *ggplot2* použijeme schématický obrázek:

ggplot2 je postaven na teorii *layered grammar of graphics*, každý obrázek vytvořený za jeho pomoci se skládá z několika prvků:

17.1.0.1 Od dat k vizualizaci

Vizualizace dat, která se skládá z jedné, nebo mnoha překrývajících se vrstev (*layers*). Každá vrstva přidává do výsledného obrázku jednu dodatečnou vizualizaci. Schématický obrázek má dvě vrstvy obsahující vizualizaci dat. První vrstva (bodový graf) obsahuje zobrazení surových dat – tedy dat, jak jsou. Druhá vrstva vykresluje statistickou transformaci surových dat – proloženou křivku. Obrázek je tedy konstruován podobně, jako byste přes sebe překládaly průsvitné fólie pokreslené fixem. Výsledný obrázek by se postupně rozšiřoval o další a další prvky. Nicméně prvky přidané později překrývají prvky přidané dříve.

Jako příklad vizualizace dat můžeme zkonstruovat následující bodový graf, který obsahuje dvě pozorování:

```
## Warning: `data_frame()` was deprecated in tibble 1.1.0.
## Please use `tibble()` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_warnings()` to see where this warning was generated.
```

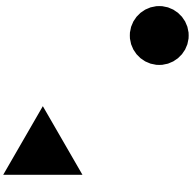


Figure 17.2: Vizualizace dat s *ggplot2*

Z tohoto obrázku vidíme, že pozorování jsou dvě a že nejsou identická. Nic dalšího říci nemůžeme.

17.1.0.2 Zpět od vizualizace k datům

Aby byl obrázek srozumitelný, musí být k vizualizaci dat připojeny prvky, které ji umožňují porozumět. Tedy umožňují uživateli převést vizualizaci zpět do dat. Takové prvky jsou škály, legendy, osy, atp. Viz obrázek:

17.1.0.3 Vzhled obrázku

Poslední skupina prvků upravuje celkový vzhled obrázku. Nemá žádný vztah ke dvěma předchozím kategoriím. Tyto prvky upravují pouze celkový vzhled obrázku (velikost písma, barvu pozadí,...).

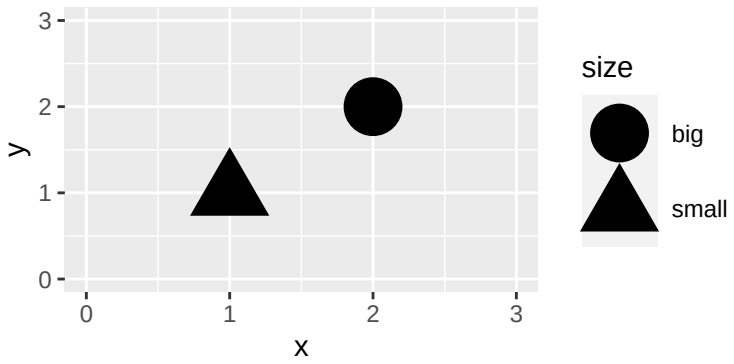


Figure 17.3: Prvky umožňující převod vizualizace zpět do dat

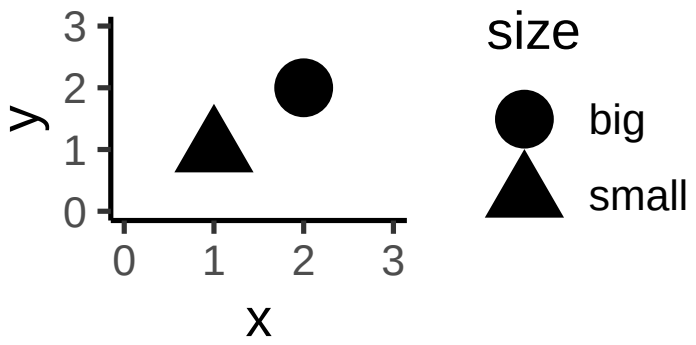


Figure 17.4: Změny vzhledu obrázků

Následující příklad ukazuje možný vzhled obrázku:

V *ggplot2* se všechny tři skupiny prvků ovládají nezávisle na sobě. Pro vytvoření každého prvku existují speciální funkce.

17.2 Základní vizualizace: vrstva po vrstvě

Celkovou konstrukci obrázku můžeme ilustrovat na tabulce `diamonds`, která je součástí balíku `ggplot2`. Tabulka obsahuje informace o jednotlivých kamenech (viz `?diamonds`). Tabulka obsahuje přes 50 tisíc pozorování. Jejich vykreslování by zejména na pomalejších počítačích trvalo velmi dlouho. Proto budeme pracovat pouze s 500 náhodně vybranými řádky:

```
diamonds <- diamonds %>% sample_n(500)
print(diamonds)
```

```
## # A tibble: 500 x 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  1.73 Ideal     J     SI1      62.6    57  6927  7.69  7.65  4.8
##  2  2.01 Premium   H     VS2      61.1    61 15908  8.14  8.09  4.96
##  3  1.72 Premium   G     VS1      59.4    60 17765  7.86  7.82  4.66
##  4  0.86 Ideal     E     SI1      60.6    56  4228  6.16  6.18  3.74
##  5  0.35 Ideal     E     VS1      61.6    57   829  4.5   4.53  2.78
##  6  1.27 Premium   G     VS1      61.1    60  9371  6.99  7.03  4.28
##  7  0.65 Ideal     D     VVS1     61.8    57  4022  5.54  5.56  3.43
##  8  1.55 Premium   H     VS2      60.7    59 11846  7.5   7.46  4.54
##  9  0.51 Very Good G     VS1      62.7    58  1599  5.05  5.09  3.18
```

```
## 10  0.72 Premium   F     VVS2     61.6    59  3105  5.78  5.72  3.54
## # ... with 490 more rows
```

Našim cílem je vytvořit podobný obrázek, který byl použit pro schématickou ilustraci fungování *ggplot2*: Vykreslit závislost ceny (`price`), váhy (`carat`) a kvality řezu (`cut`).

Základ každého obrázku vytvoříme pomocí funkce `ggplot()`:

```
ggplot(data = NULL, mapping = aes())
```

Funkce má dva základní argumenty: `data` obsahují tabulku s daty, která se mají vizualizovat. Parametr `mapping` pak přijímá základní pravidla, podle kterých se má vizualizace řídit. Přesněji řečeno pravidla vytvořená funkcí `aes()`, která určují, jaká proměnná se má *mapovat* na kterou *estetiku*.

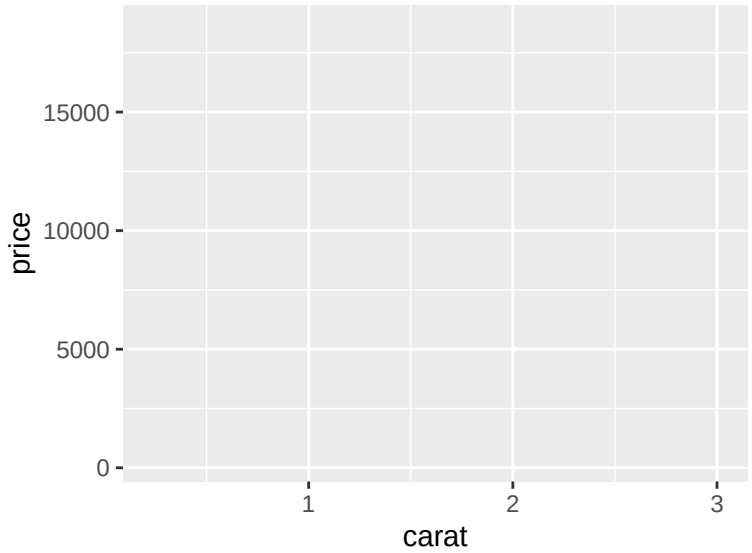
Pro pochopení těchto pojmů je užitečné zamyslet se nad tím, jak vlastně funguje bodový graf. Jeho podstatou je to, že jedno pozorování, je reprezentováno jedním bodem. Jaké vlastnosti (estetiky) však tento bod může mít?

1. Pozici v rovině obrázku: souřadnice na ose x (`x`) a y (`y`)
2. Tvar (`shape`): kolečko, čtvereček, srdíčko, listy, káry,...
3. Velikost (`size`)
4. Barvu (`color/colour`)
5. Barvu výplně (`fill`)
6. Typ ohraničující linky (`stroke`)
7. Průhlednost (`alpha`)

Všechny estetiky je možné nastavit (*set*) nebo namapovat (*map*). V případě nastavení bude estetika u všech pozorování stejná. V případě namapování se hodnota estetiky bude řídit podle hodnoty přiřazené proměnné.

Pokud tedy chceme vykreslit bodový graf závislosti, ceny a váhy, potom tyto proměnné chceme namapovat na vybrané estetiky. Stejného výsledku jako u úvodního obrázku dosáhneme pomocí `aes(x = carat, y = price)`. Základní volání `ggplot()` tedy může vypadat následovně:

```
diamonds %>%
    ggplot(
        aes(x = carat, y = price)
    )
```



Výsledkem je prázdná formátované plocha. Volání `ggplot()` totiž pouze vytvoří prázdné "plátno", na které je potřeba přidávat jednotlivé vrstvy s obsahem.

Nové vrstvy se typicky přidávají voláním funkcí `geom_*()`. Tyto funkce nejsou nic jiného, než aliasy pro různé nastavení funkce `layer()`:

```
layer(geom = NULL, stat = NULL, data = NULL, mapping = NULL,
  position = NULL, params = list(), inherit.aes = TRUE,
  check.aes = TRUE, check.param = TRUE, subset = NULL, show.legend = NA)
```

Funkce `layer()` má mnoho parametrů. Zásadní jsou tyto:

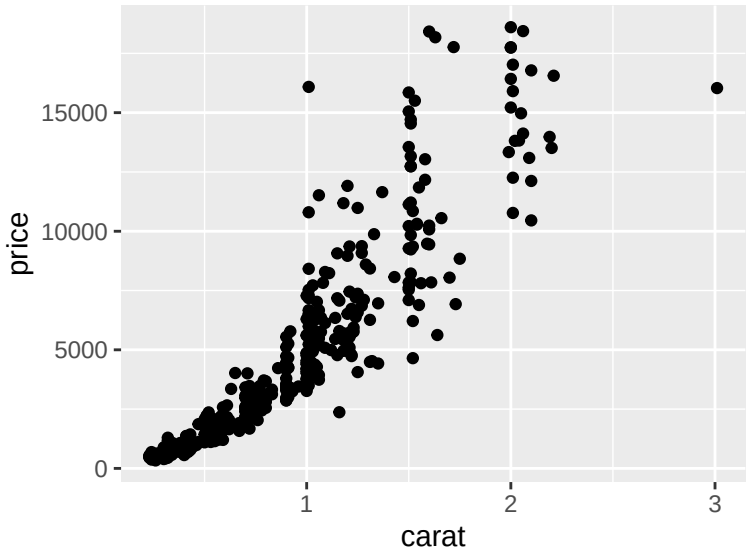
- `data`... tabulka s daty, která se má použít v dané vrstvě. Pokud tento parametr není zadán, potom se použije tabulka dat specifikované ve volání `ggpplot()`. Každá vrstva tak může pracovat s jinými daty. (A také `ggplot()` nemusí obsahovat specifikaci dat.)
- `geom`... typ geometrického objektu, který se má použít pro vizualizaci dat. Může se jednat o bod (`point`), polygon (`polygon`), úsečku (`line`) a mnoho dalších.
- `mapping`... obsahuje namapování proměnných na estetiky pomocí funkce `aes()`. Pokud není parametr specifikován, potom se použije specifikace z `ggplot()`. Pokud je specifikace v konfliktu se specifikací v `ggplot()`, potom se použije specifikace z `layer()`. Pokud je specifikována estetika, která není přiřazena v `ggplot()`, potom se použije jako doplnění ke specifikaci z `ggplot()`.
- `stat`... statistická transformace dat, která se má provést před vykreslením. Vykreslují se až transformované data.
- `position`... umožňuje upravit pozici hrubých dat před jejich vykreslením

Funkce `layer()` není zpravidla nikdy volána přímo. V podstatě vždy se používají funkce `geom_*()`. Pokud například chceme přidat vrstvu s bodovým grafem, potom použijeme `geom_point()`. Tato funkce je přesným ekvivalentem volání:

```
layer(
  data = NULL,
  mapping = NULL,
  geom = "point",
  stat = "identity",
  position = "identity"
)
```

Pokračujme v příkladu konstrukcí bodového grafu:

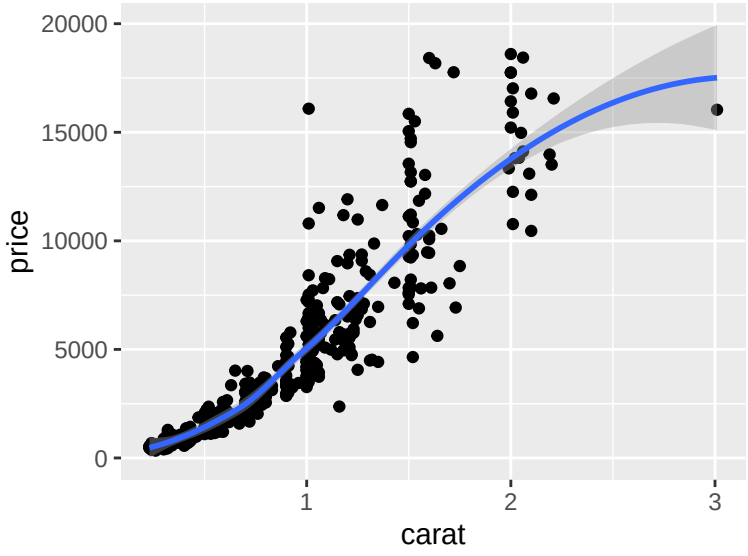
```
diamonds %>%
    ggplot(
        aes(x = carat, y = price)
    ) +
    geom_point()
```



Vrstvu obsahující body jsme vytvořili voláním funkce `geom_point()`. Tato nová vrstva je k původnímu volání `ggplot()` připojena funkcí `+`. Pomocí této funkce můžeme do obrázku přidávat další vrstvy – například vrstvu, která vykreslí proloženou křivku:

```
diamonds %>%
    ggplot(
        aes(x = carat, y = price)
    ) +
    geom_point() +
    geom_smooth()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Jak vlastně funguje `ggplot2` vevnitř? Základní volání `ggplot()` vytvoří "prázdnou" datovou strukturu. Funkce `+` potom R říká, že má tuto datovou strukturu modifikovat. Jakým způsobem se to má stát určuje funkce volaná za `+`. Například `+ geom_point()` přidá k původní datové struktuře vrstvu s body, atd.

Primárním výsledkem volání `ggplot()` je tedy datová struktura. Tu můžeme přiřadit do proměnné:

```
p <- diamonds %>%
    ggplot(
        aes(x = carat, y = price)
    ) +
    geom_point()
```

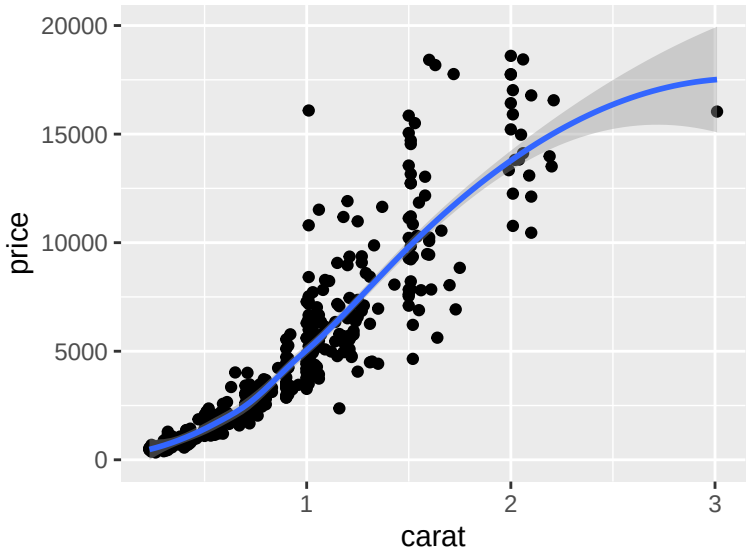
...a dále modifikovat...

```
p <- p + geom_smooth()
```

...nebo "vytisknout":

```
p
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



17.3 Mapování a nastavování estetik

V předchozím textu byl vysvětlen pojem estetika a mapování. Nyní se tomuto tématu budeme věnovat podrobněji. Každý `geom` má několik estetik – dimenzí, jejichž podobu lze řídit podle určité proměnné. V případě bodového grafu (`point`) jde o: `x`, `y`, `shape`, `size`, `color`/`colour`, `fill`, `stroke` a `alpha`. Například `geom line` (spojnicový graf) má estetiky jiné – nerozumí `stroke`, `shape` a `fill`. Namísto toho umí pracovat s `linetype`.

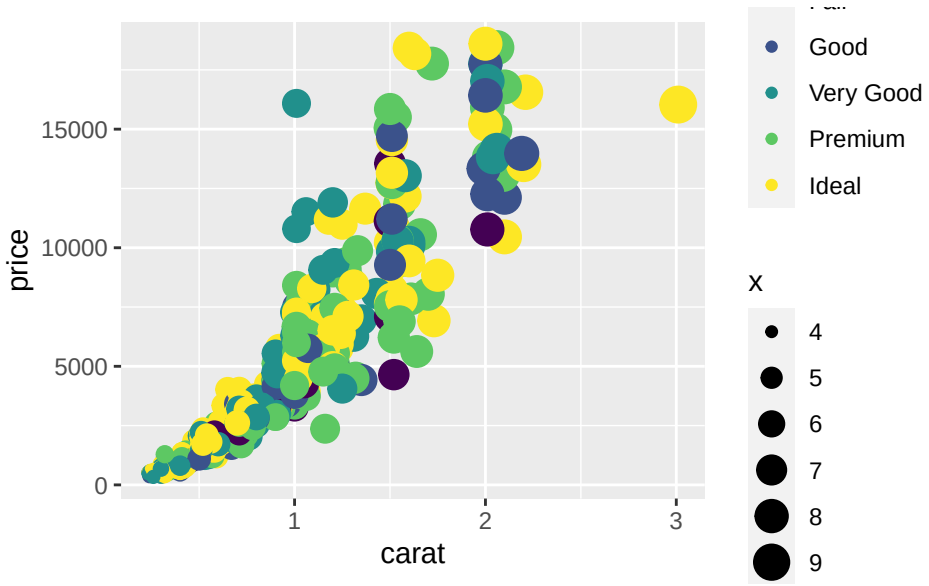
Výše uvedený příklad demonstruje mechanismus mapování. S pomocí funkce `aes()` je estetikám přiřazena proměnná. Některé estetiky je v závislosti na geomu nutné přiřadit. V případě `geom_point()` je to například `x` a `y`:

```
diamonds %>%
    ggplot(
        aes(x = carat)
    ) +
    geom_point()
```

```
## Error: geom_point requires the following missing aesthetics: y
```

Pomocí `aes()` je možné mapovat i další estetiky:

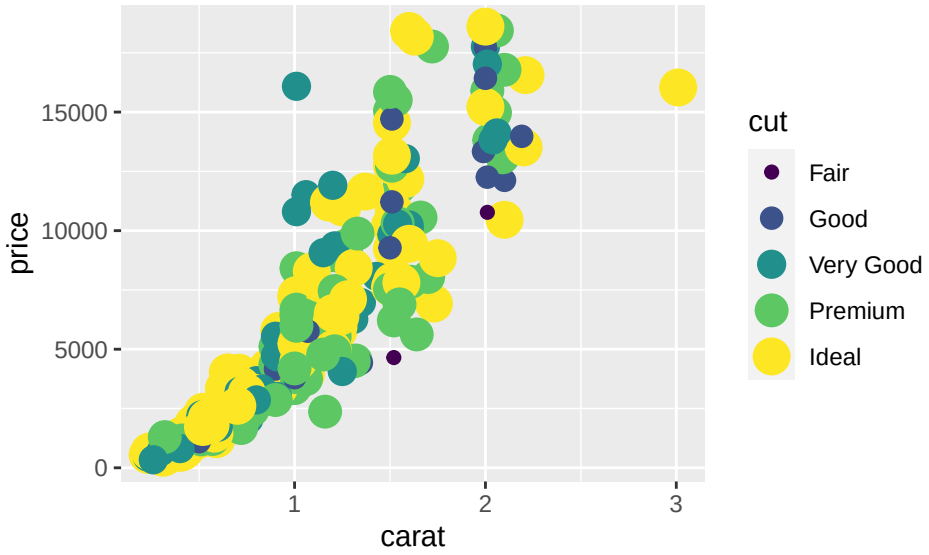
```
diamonds %>%
    ggplot(
        aes(x = carat, y = price, size = x, color = cut)
    ) +
    geom_point()
```



Pro každou namapovanou estetiku `ggplot2` vytvoří vodítko, které čtenáři umožní překlad z vizualizace do dat. V případě `x` a `y` vykreslí osy grafu. U zbývajících estetik vykreslil legendu.

Jednu proměnnou jde mapovat i na více estetik:

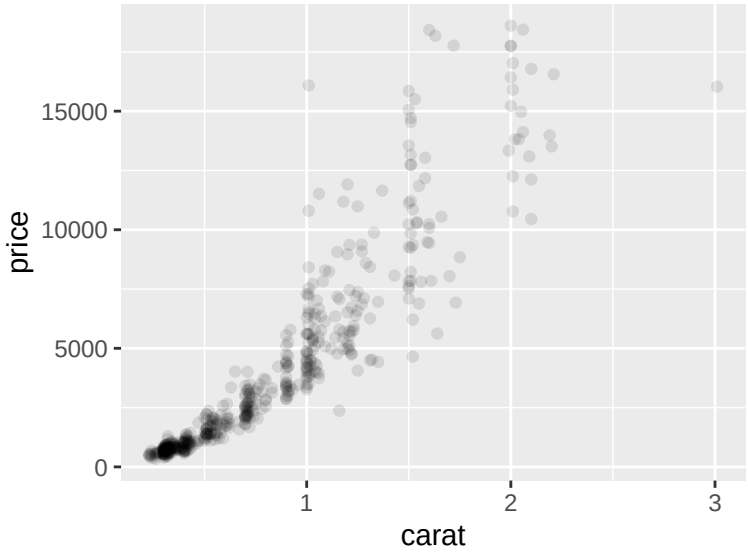
```
diamonds %>%
    ggplot(
        aes(x = carat, y = price, size = cut, color = cut)
    ) +
    geom_point()
```



ggplot2 takovou kombinaci zohlední i v legendě. Tento příklad ukazuje i další vlastnost ggplot2. Pokud uživatel požaduje provedení něčeho, co Tvůrce (H. Wickham) nepovažuje za dobrý nápad, vrátí ggplot2 upozornění (via message()).

ggplot2 poskytuje obrovské množství možností a částečně uživateli radí. Nicméně je na uživateli, aby vše nastavil tak, aby výsledné obrázky splňovaly svůj účel – jasně čtenáři předávaly určitou informaci. Obrázek použitý v příkladu například trpí zásadní vadou, která se říká "overplotting". Pozorování se vzájemně překrývají a ve výsledný obrázek čtenáři mnoho neřekne. Nemůže vědět, zda se v některém místě vzájemně překrývá málo, nebo mnoho pozorování. Jednoduchou a často dostačující cestou jak bojovat s tímto problémem je nastavení průhlednosti (estetiky alpha). Tuto estetiku lze samozřejmě mapovat na proměnnou, ale v tomto případě ji použijeme pro ilustraci nastavení estetiky:

```
diamonds %>%
    ggplot(
        aes(x = carat, y = price)
    ) +
    geom_point(alpha = 0.1)
```



Pokud chce uživatel estetiku nastavit, je potřeba ji deklarovat v příslušné geom_*() funkci a mimo funkci aes(). Příklad ukazuje, že nastavení průhlednosti problém s overplottingem minimálně zmírnilo.

17.3.1 Kontrola mapování

Ve všech příkladech jsme dosud pouze mapovali estetiku na určité proměnné. Mapování předává ggplot2 informaci typu: "barvu puntíků urči podle proměnné cut". Jak konkrétně to ggplot2 provede (t.j. bude mít špatný řez červenou, modrou, zelenou nebo jinou barvu) jsme zatím nijak neovlivňovali.

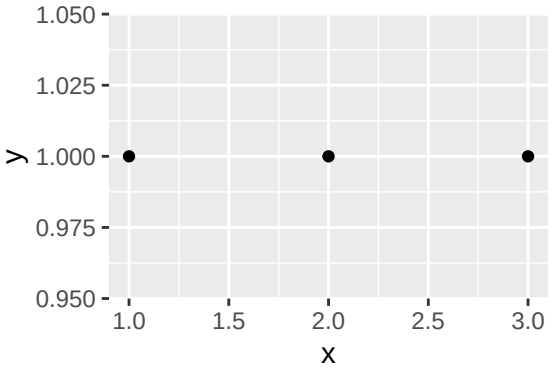
V této kapitole si ukážeme, jak lze s pomocí funkcí scale_*_*() měnit škálování a prvky, které pomáhají čtenáři převést vizualizaci zpět do dat.

Pro tyto účely budeme používat jednoduchou datovou tabulku se 3 pozorováními a 4 sloupci x, y, z a w...

```
xtable <- data_frame(
    x = c(1,2,3),
    y = c(1,1,1),
    z = c("A","B","C"),
    w = c(-1,0,1)
)
```

...a jednoduchý bodový graf:

```
xtable %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_point()
```



ggplot2 namapoval data na estetiky (pozice) a vykreslil osy, které čtenáři umožňují číst data. Obrázek je takový, jaký asi všichni očekávají. Nicméně to nemusí být to, co uživatel chce. Co může chtít změnit? Například:

- jméno škály (`name`)
- popisky na ose (`labels`)
- pozice "hlavních" popisků na ose (`breaks`)
- pozice "pomocných" popisků/čar na ose (`minor_breaks`)
- interval zobrazený na obrázku – "jak dlouhá" má být osa (`limits`)
- pozici osy (`position`)
- transformaci hodnot na ose – například zobrazení logaritmu pozorovaných hodnot (`trans`)
- zobrazení sekundární osy (`sec.axis`)

Úplný výčet možností se samozřejmě liší podle estetiky.

Mapování se řídí pomocí funkcí `scale_*_*()`, která se podobně jako `geom_()` připojují k volání `ggplot()` pomocí funkce `+`. Jména funkcí `scale_*_*()` se mohou zdá komplikovaná, ale opak je pravdou. Ve jejích jménech je totiž systém. Jméno funkce se sestává ze tří částí, spojených znakem "_":

1. Všechna jména začínají `scale`
2. Druhá část jména je jméno estetiky, kterou chceme kontrolovat
3. Poslední část jména je jméno konkrétní škály (to je potřeba si občas pamatovat nebo najít)

Pokud bychom například chtěli změnit jméno osy x (estetika `x`) na našem obrázku, potom budeme chtít použít funkci:

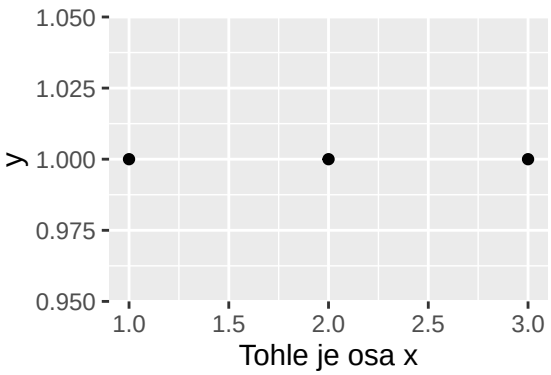
```
scale_x_continuous()
```

První část jména je daná, druhá je jasná a třetí odkazuje na povahu proměnná `x` v tabulce `xtable`, která je spojitá. (Takto pěkně ale jména škály nefungují vždy. Občas je třeba hledat.)

Pokud známe jméno, je z poloviny vyhráno:

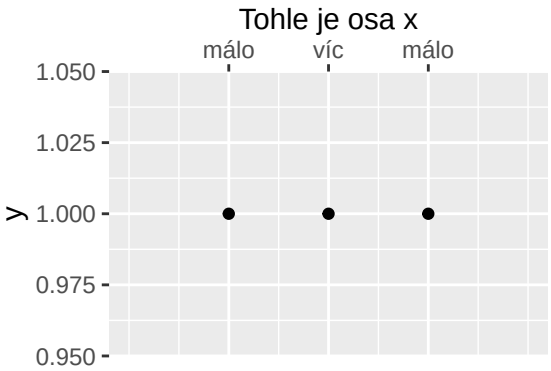
```
xtable %>%
    ggplot(
        aes(x = x, y = y)
```

```
    ) +
    geom_point() +
    scale_x_continuous(name = "Tohle je osa x")
```



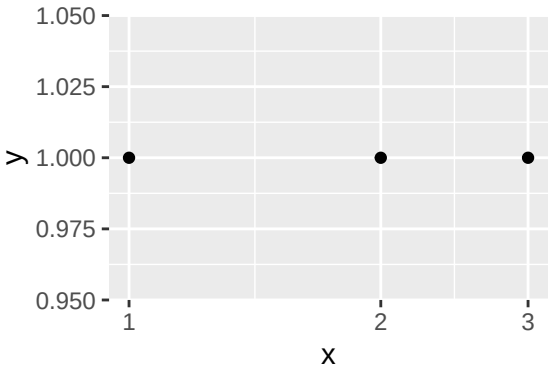
Jak bylo zmíněno výše, pomocí škál je možné nastavit velké množství parametrů:

```
xtable %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_point() +
    scale_x_continuous(
        name = "Tohle je osa x",
        breaks = c(1,2,3),
        labels = c("málo","víc","málo"),
        limits = c(0,4),
        position = "top"
        )
```



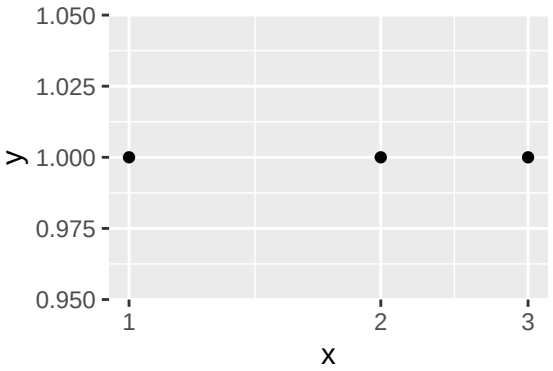
Poněkud specifické je nastavení transformace. V praxi se často používá logaritmická transformace:

```
xtable %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_point() +
    scale_x_continuous(trans = "log10")
```



Řada transformací je pro uživatele připravená (viz `?scale_x_continuous()`). Uživatel má navíc možnost vytvořit vlastní transformace. Pro nejčastější transformace má `ggplot2` předpřipravené škály – například `log10` nebo `reverse`:

```
xtable %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_point() +
    scale_x_log10()
```



V předchozích příkladech jsme volali funkci `scale_x_*()` pro nastavení mapování estetiky `x`. Nicméně ani jednou jsme neupravovali estetiku y a taky to fungovalo. Jak je to možné? Pokud není funkce `scale_*_*()` explicitně volána uživatelem, `ggplot2` odhadne, která funkce je nejvhodnější a zavolá ji interně sám. Volání:

```
xtable %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_point()
```

Je tak ve výsledku ekvivalentní k:

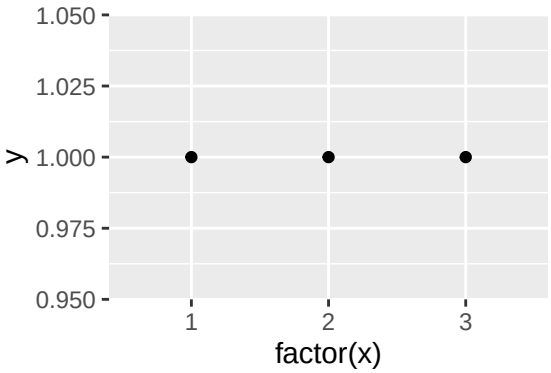
```
xtable %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_point() +
    scale_x_continuous() +
    scale_y_continuous()
```

Pokud se změní povaha podkladových dat, potom se i ggplot2 rozhodne volat jinou funkci scale_*_*(). V případě volání:

```
xtable %>%
    ggplot(
        aes(x = factor(x), y = y)
    ) +
    geom_point()
```

kde proměnná x je konvertována na faktor. Vrátí ggplot2 obrázek ekvivalentní volání:

```
xtable %>%
    ggplot(
        aes(x = factor(x), y = y)
    ) +
    geom_point() +
    scale_x_discrete() +
    scale_y_continuous()
```



ggplot2 v tomto případě volá škálu vytvořenou pro mapování diskrétních proměnných.

17.3.1.1 Mapování barev (estetiky colour a fill)

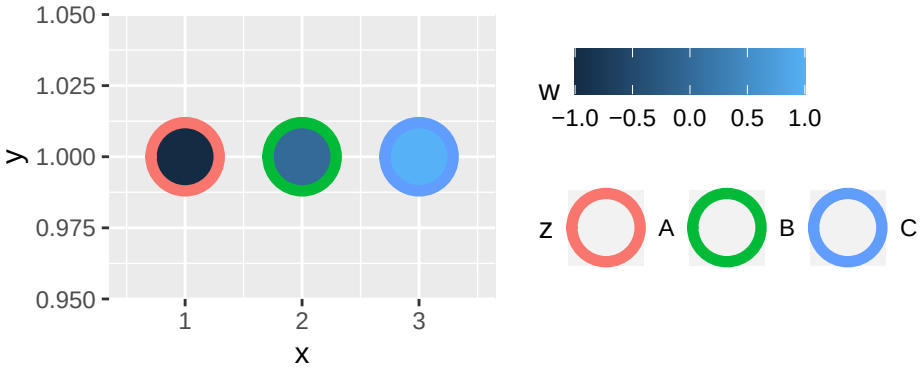
Pomocí funkcí scale_*_*() lze měnit mapování všech estetik. Mezi nejčastější upravované škály patří bezesporu barvy, které se mapují v estetikách color/colour a fill.

Poznámka: ggplot2 neumí a podle jeho tvůrců nikdy nebude umět vybarvovat texturou (“šrafováním”).

Pro ilustraci těchto estetik použijeme následující podobu bodového grafu:

```
xtable %>%
    ggplot(
        aes(x = x, y = y, color = z, fill = w)
    ) +
    geom_point(
        shape = 21,
        stroke = 3,
        size = 10
        ) +
    scale_x_continuous(
        limits = c(0.5,3.5)
    ) +
    theme(
        legend.direction = "horizontal"
```

```
    ) -> p

p
```



Výsledná podoba obrázku je ovlivněna funkcí `theme()`. Její fungování je popsáno níže. Celá struktura je přiřazená do proměnné `p`. S tou budeme nadále pracovat.

Tento graf mapuje proměnné `z` a `w` na estetiky `color` a `fill`. Ostatní estetiky jsou nastaveny a jsou pro všechny body v grafu shodné.

Proměnná `z` a `w` se liší ve své povaze. `w`je spojitá proměnná a `z` je kategoriální. Práce s barvami by se obecně měla řídit podle povahy zobrazovaných dat:

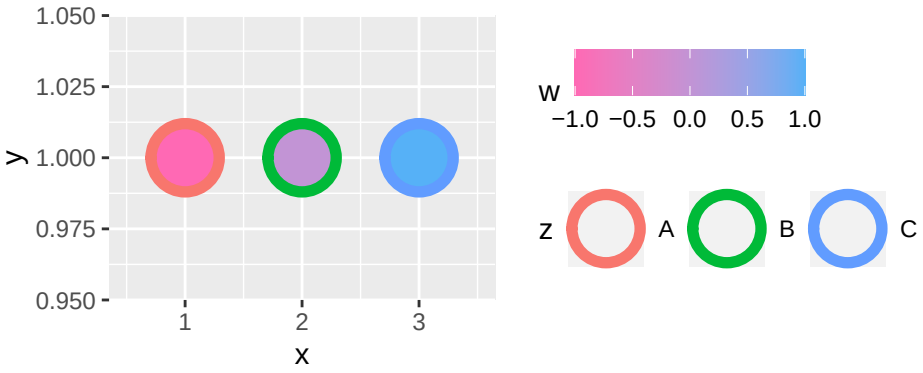
- Spojité proměnné se typicky zobrazují barevným přechodem – jak vidíte na obrázku, základní volbou `ggplot2` je přechod z tmavě do světle modré.
- Kategoriální proměnné, jejichž hodnoty jsou "na stejné úrovni". Příkladem takové proměnné mohou být například značky automobilů. V takovém případě je zvolit barvy, které jsou pro vnímání člověka "stejně daleko" od sebe. Příkladem takového výběru barev může být proměnná `z` v ukázkovém obrázku.
- Posledním typem jsou kategoriální proměnné, které vyjadřují sekvenci (např.: malá, střední, velká). V takovém případě je vhodné volit barevný přechod, který je také rozdělen do kroků, které musí být tak velké, aby byly pro člověka jasně odlišitelné.

Při volbě barev je také užitečné myslet na barvoslepé a na tisk na černobílé tiskárně. Jiné barvy jsou také vhodné pro plochy a jiné pro body. Ve výsledku je výběr barev nesmírně složitý. Najít sekvenci barev pro 10 úrovní je těžké, pro 20 úrovní je to nemožné. Moudrým postupem je proto používat uměřený počet barev (úrovní) a v jejich volbě vycházet z doporučení odborníků.

17.3.1.1.1 Spojité proměnné Typickým postupem zobrazování spojité proměnné v barvě je použití barevného přechodu (gradientu) mezi dvěma nebo více barvami. `ggplot2` nabízí hned 4 škály založené na gradientech:

`scale_*_gradient()` (identické se `scale_*_continuous()`) je základní volbou, po které u spojitých proměnných `ggplot2` sáhne, pokud uživatel neporoučí jinak. Vykresluje přechod mezi dvěma barvami, které jsou nastavené parametry `low` a `high`:

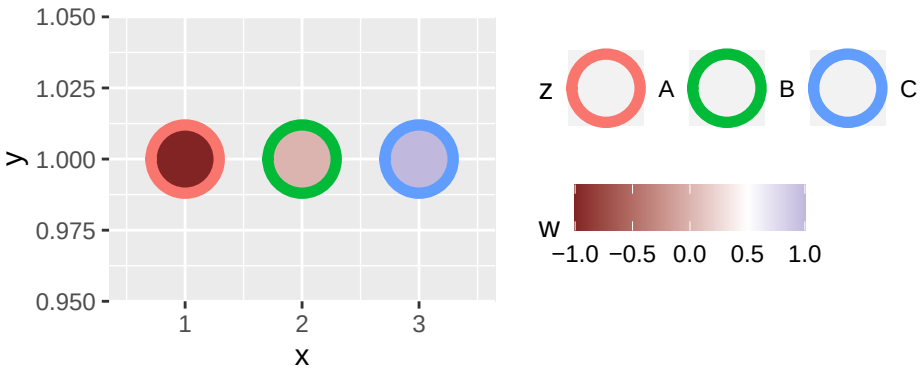
```
p + scale_fill_gradient(low = "hotpink", high = "#56B1F7")
```



V příkladu je nastaven přechod z tmavě poníkové do modré. Barva je vždy definována jako řetězec (nebo funkcí, která řetězec nebo jejich vektor vytvoří) – a to buď svým jménem (`"hotpink"`) nebo RGB kódem (`"#56B1F7"`).

`scale_*_gradient2()` je derivátem `scale_*_gradient()` umožňuje vytvořit přechod přes tři barvy (`low`, `mid`, `high`) a specifikovat pozici středního bodu parametrem `midpoint`. (Parametr `midpoint` je defaultně nastaven na hodnotu 0.)

```
p + scale_fill_gradient2(midpoint = 0.5)
```



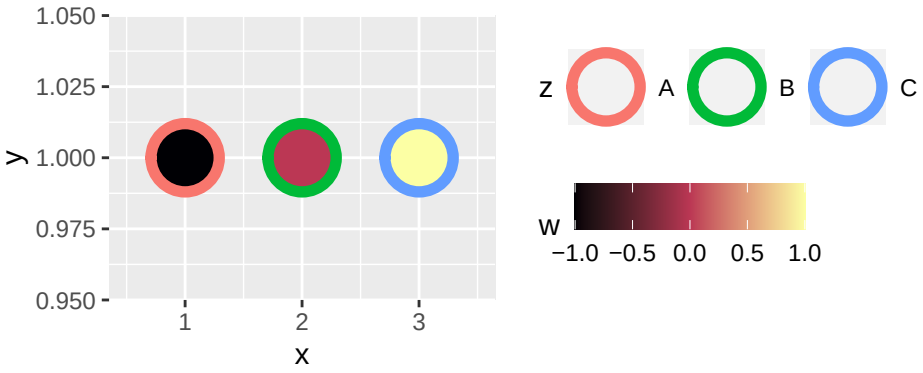
`scale_*_gradientn()` umožňuje použít přechod přes tolik barev, kolik si jen srdce uživatele přeje. Z tohoto důvodu neobsahuje žádné základní nastavení:

```
p + scale_fill_gradientn()
```

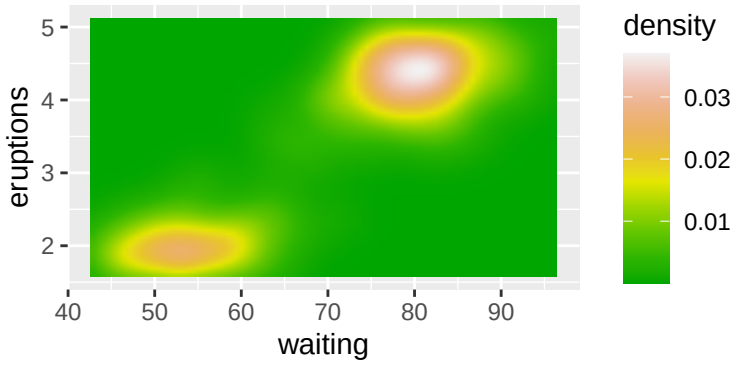
```
## Error in scale_fill_gradientn(): argument "colors" is missing, with no default
```

Uživatel musí prostřednictvím parametru `colours` zadat vektor barev, které se mají použít. Domnívám se, že pro uživatele – neodborníka – se jedná o nemožný úkol. Je však možné použít předpřipravené palety, které jsou součástí mnoha balíků. Následující příklad využívá paletu `inferno` z balíku **viridis**:

```
p + scale_fill_gradientn(colours = viridis::inferno(3))
```

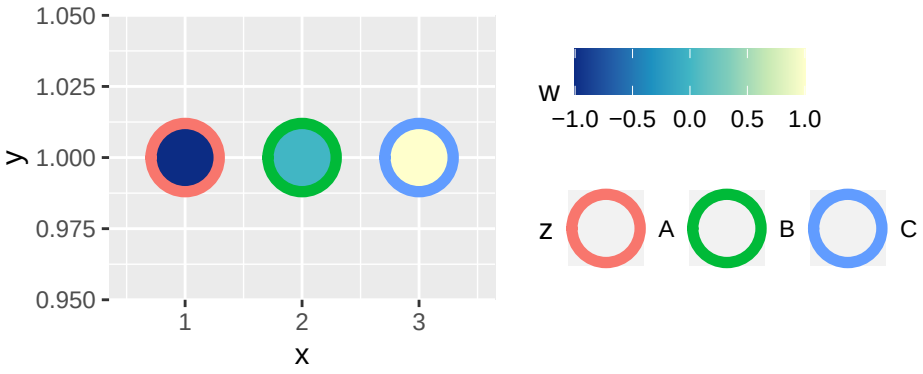


Využití `scale_*_gradientn()` je vhodné spíše v plochách. Nicméně i tam je dobré spolehnout na rady odborníků. Příklad "plošného" obrázku:



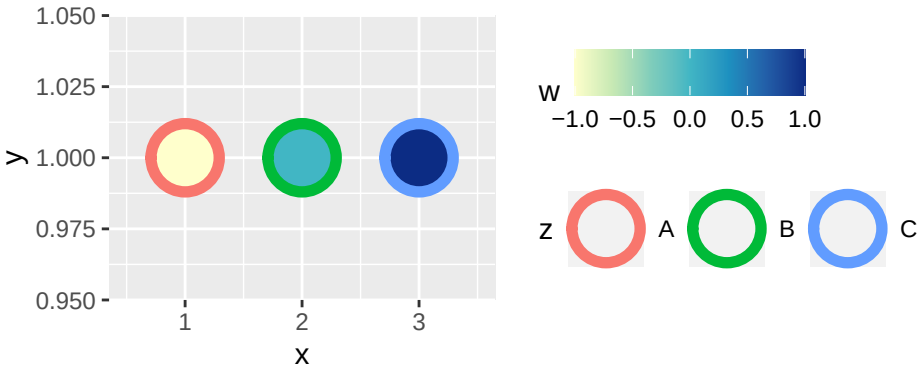
Na radách odborníků je zcela postaven `scale_*_distiller()`. Pracuje s ručně vybranými diskrétními paletami, které jsou dostupné na http://colorbrewer2.org/. `scale_*_distiller()` tyto diskrétní palety adaptuje i pro spojité proměnné. Rozumný způsob práce je na webu zvolit vhodnou barevnou škálu a její jméno specifikovat v parametru `palette`. Ten um pracovat s číslem palety i s jejím jménem:

```
p + scale_fill_distiller(palette = "YlGnBu")
```



Parametrem `direction`, který nabývá hodnot `1` nebo `-1`, lze pořadí barev obrátit:

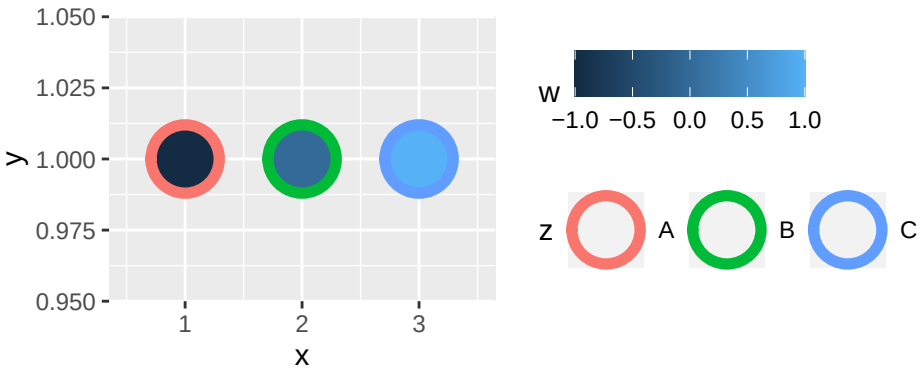
```
p + scale_fill_distiller(palette = "YlGnBu", direction = 1)
```



Používání http://colorbrewer2.org/ lze doporučit v maximální rozumné míře.

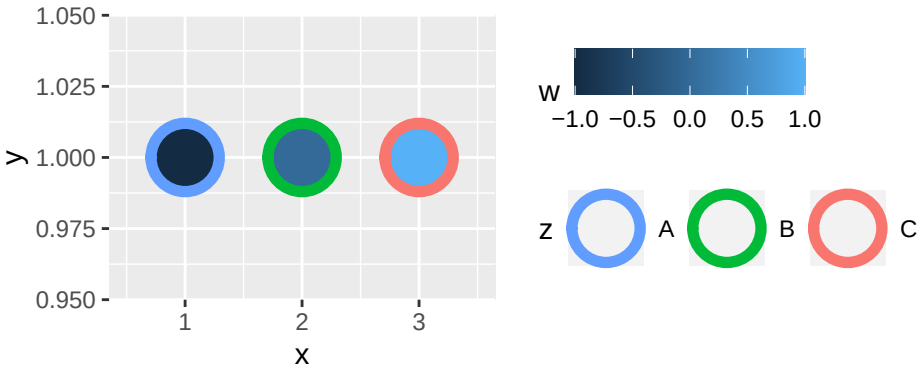
17.3.1.1.2 Diskrétní proměnné Pro diskrétní proměnné je výchozí volbou `scale_*_hue()`. Tato škála vybírá od sebe stejně vzdálené barvy na "HCL wheel" (viz např. Wikipedie). Teoreticky je tak schopna vytvořit barevné schéma pro prakticky neomezený počet kategorií. V praxi od sebe budou barvy při vyšším počtu kategorií nerozpoznatelné.

```
p + scale_color_hue()
```



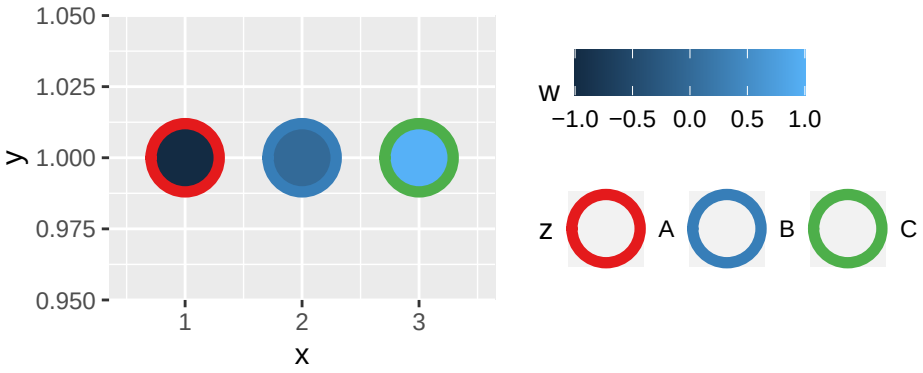
`scale_*_hue()` umožňuje vybrat výchozí bod na HCL wheel a parametrem `direction` směr, jakým se na něm výběr pohybuje:

```
p + scale_color_hue(direction = -1)
```



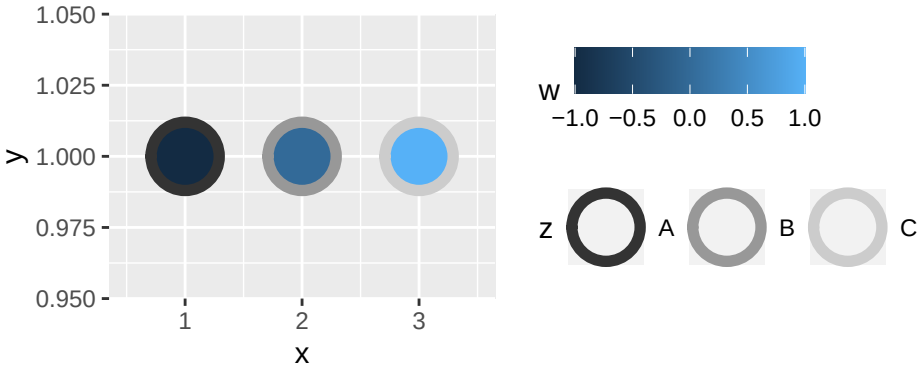
Zejména při finalizaci grafických výstupů je vhodné poohlédnout se po předpřipravených paletách. V samotném ggplot2 jde o `scale_*_brewer()` – bratra `scale_*_distiller()` pro diskrétní veličiny.

```
p + scale_color_brewer(palette = "Set1")
```



Speciální škálou je `scale_*_grey()`. Ta, jak název napovídá, mapuje do odstínů šedé:

```
p + scale_color_grey()
```



Speciálními případy škál jsou `scale_*_manual()` a `scale_*_identity()`. `scale_*_manual()` umožňuje uživateli vytvořit si vlastní škálu. U barev bude tato možnost využívána asi jen zřídka. Za určitých okolností se však může hodit u jiných estetik – například u `shape`.

`scale_*_identity()` dokáže vzít proměnnou z datové tabulky a použít ji "tak jak je" pro vykreslení barev, tvarů a podobně.

17.3.1.2 Co jsme zamlčeli...

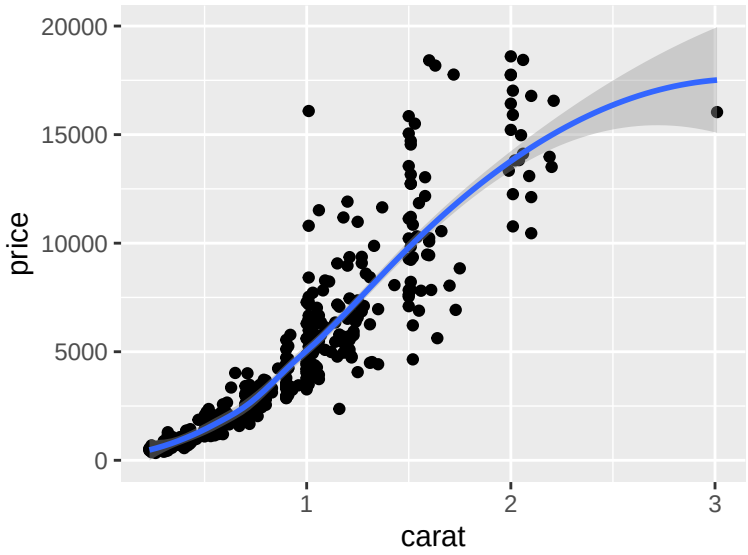
Všechny krásné škálovací funkce `scale_*_*()` v posledku využívají služeb konstruktorových funkcí `discrete_scale()` a `continuous_scale()`. S těmito funkcemi komunikujete ze `scale_*_*()` přes argument `...`. Pokud tedy budete chtít provádět něco složitějšího, tak se vyplatí projít si nápovědu právě k těmto konstruktorovým funkcím.

17.4 Grupování

U některých funkcí `geom_*()` je mezi estetikami uvedena jedna s názvem `group`. Lehko si můžete ověřit, že se nijak nepřekládá do vizuální podoby. Její funkce je totiž jiná. Sděluje `ggplot2`, že určitá skupina pozorování "patří k sobě" – tvoří jednu skupinu.

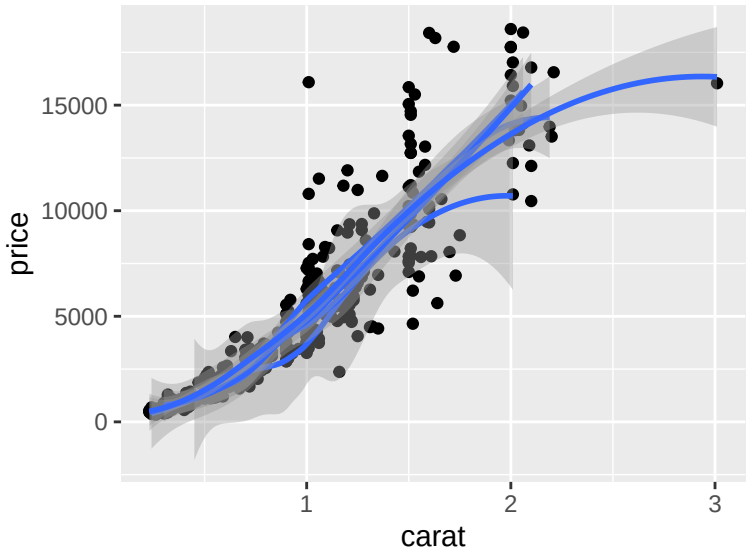
Praktické využití lze ilustrovat na proložení křivky v tabulce `diamonds`:

```
diamonds %>%
    ggplot(
        aes(x = carat, y = price)
    ) +
    geom_point() +
    geom_smooth()
```



V tomto případě se proložila křivka přes všechny pozorování. Můžeme se ale chtít podívat, jestli je závislost stejná pro různé *druhy* kamenů – například pro skupiny definované kvalitou řezu (`cut`). V tomto případě použijeme estetiku `group`:

```
diamonds %>%
    ggplot(
        aes(x = carat, y = price, group = cut)
    ) +
    geom_point() +
    geom_smooth()
```

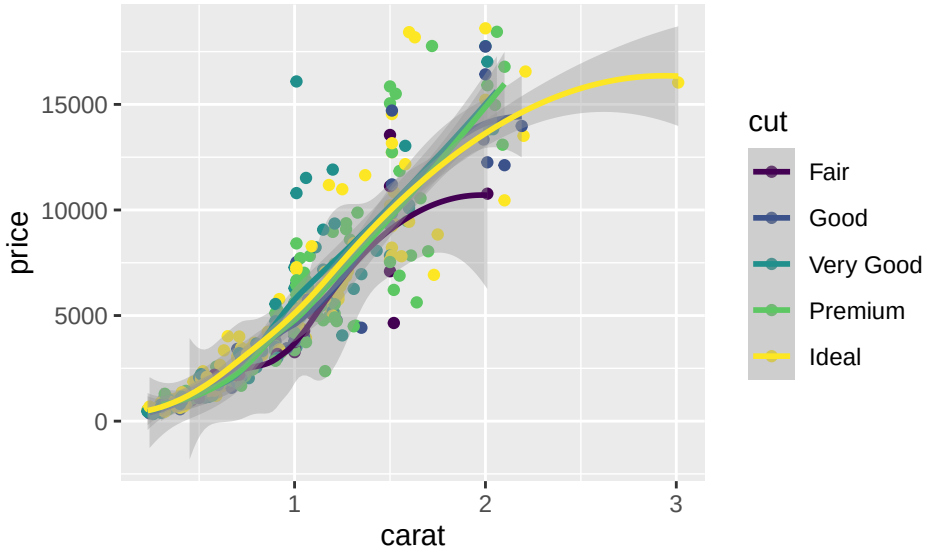


V tomto případě `geom_smooth()`, který estetice `group` rozumí, pracuje s každou skupinou vymezenou hodnotou kategoriální proměnné `cut` zvlášť. Fakticky každou skupinu proloží její vlastní křivkou.

V praxi je často užitečné pro zpřehlednění využít možnosti mapování jedné proměnné na více estetik:

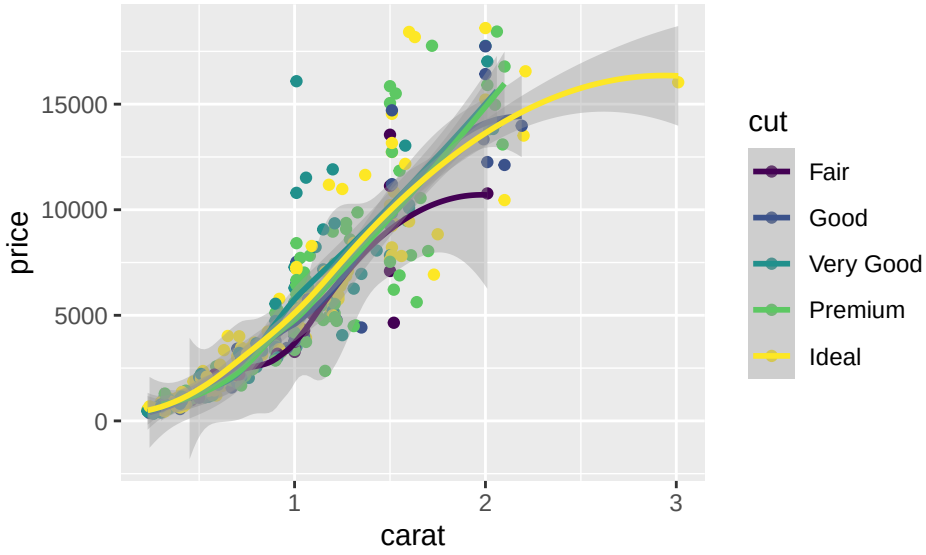
```
diamonds %>%
    ggplot(
        aes(x = carat, y = price, group = cut, color = cut)
    ) +
```

```
    geom_point() +
    geom_smooth()
```



Všimněte si že v tomto případě dostanete stejný výsledek i při namapování kategorické proměnné na estetiku `color`:

```
diamonds %>%
    ggplot(
        aes(x = carat, y = price, color = cut)
    ) +
    geom_point() +
    geom_smooth()
```

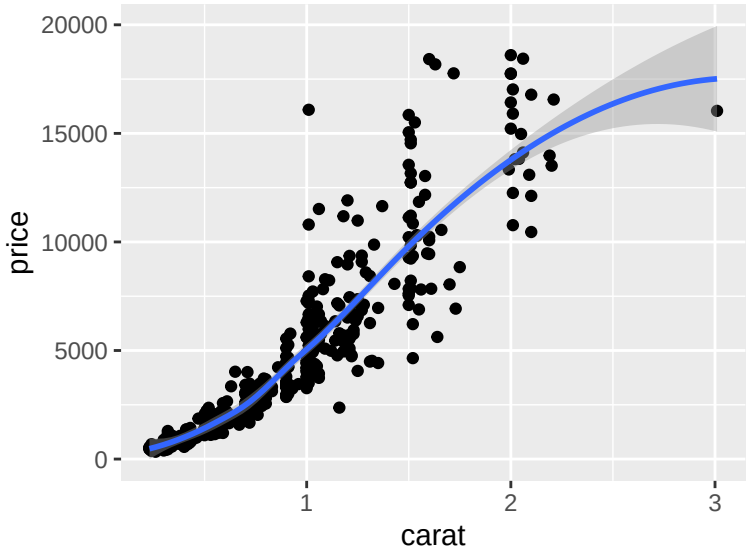


17.5 Statistické transformace

`ggplot2` umožňuje nejen vykreslovat samotná data, ale i jejich interně vytvořené statistické transformace. Příkladem může být například proložení křivkou (`geom_smooth()`). Jejich tvorbu mají na starosti funkce `stat_*()`. Vztah mezi funkcemi `geom_*()` a `stat_*()` není úplně jasný. Každá transformace (`stat`) má svůj

výchozí (přednastavený) geometrický tvar (`geom`) a každý `geom` má svůj výchozí `stat`. V některých případech tak může být jedno, jestli uživatel volá `geom` nebo ekvivalentní `stat`, Viz například použití `stat_smooth()`:

```
diamonds %>%
    ggplot(
        aes(x = carat, y = price)
    ) +
    geom_point() +
    stat_smooth()
```



Tato nejednoznačnost vymezení toho, co vlastně dělá `stat_*()` a co `geom_*()` je velmi otravnou nedokonalostí v návrhu `ggplot2` (a v `ggvis` je již vyřešena). Pokud není nevyhnutelné volat přímo `stat_*()` nutné, je lepší vždy používat funkce `geom_*()`.

Pro přímé volání `stat_*()` existovaly v podstatě dva důvody:

1. potřeba specifických nastavení statistických transformací
2. potřeba použití jiného než standardního geomu pro vykreslení statistické transformace

Pohledem do nápovědy zjistíte, že `stat_smooth()` umožňuje mnohem více nastavení vyhlazování:

```
geom_smooth(mapping = NULL, data = NULL, stat = "smooth",
  position = "identity", ..., method = "auto", formula = y ~ x,
  se = TRUE, na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)

stat_smooth(mapping = NULL, data = NULL, geom = "smooth",
  position = "identity", ..., method = "auto", formula = y ~ x,
  se = TRUE, n = 80, span = 0.75, fullrange = FALSE, level = 0.95,
  method.args = list(), na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE)
```

V novějších verzích `ggplot2` se však již dají zpravidla předávat hodnoty parametrů pro `stat_*()` prostřednictvím `...` v `geom_*()`. Tato možnost prakticky eliminuje potřebu volat `stat_*()` funkce přímo kvůli nastavení statistických transformací.

Druhý důvod pro přímé použití `stat_*()` funkce však stále trvá. V některých případech můžete potřebovat použít pro vyreslení transformace netypický geom. Příkladem může být například zobrazování jádrové hustoty rozdělení dvou proměnných nikoliv pomocí vrstevnic (standardní zobrazení), ale pomocí barevných polygonů. (Výsledkem by byly například barevné plochy s barvou sytější tam, kde je odhadnuta vyšší hustota.) Tyto úlohy jsou však bezpochyby z kapitoly pro pokročilé.

17.6 Pozicování

V ggplotu2 existuje hned několik způsobů, jak je možné ovlivnit pozici vykresleného geomu (tj. geometrického objektu reprezentujícího data) na stránce. První možností bylo použití škál – například jejich logaritmická transformace.

ggplot2 však v tomto ohledu nabízí mnohem více možností. Místem kde začít hledat je parametr `position` ve funkci `layer()`. Smyslem jeho použití je předejít překryvu pozorování ve výsledném obrázku. Parametr může nabývat následujících hodnot:

- `identity`...je základní volba pro většinu funkcí `geom_*()`. Prakticky znamená, že z hlediska modifikace pozice geomu předává instrukci "nedělej nic a kresli na skutečnou pozici".
- `jitter`...ke skutečné pozici přidává náhodnou chybu
- `dodge`...geomy skládá vedle sebe
- `stack`...staví komínky – skládá geomy nad sebe
- `nudge`...u překrývajících se geomů přidává ke skutečné pozici definovanou vzdálenost na ose `x` a `y`
- `jitterdodge`...kombinuje `jitter` a `dodge`

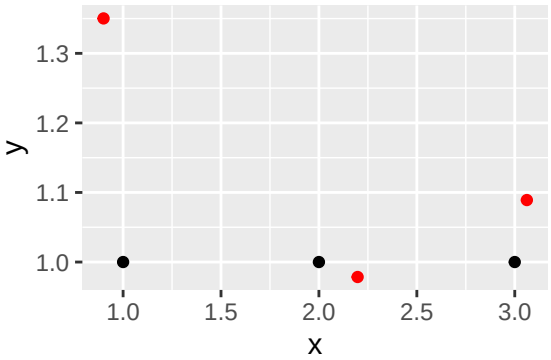
V následujícím textu jsou blíže ukázány nejčastěji používané varianty změny pozic: `jitter`, `dodge`, `stack` a navíc poněkud specifická varianta `fill`. Varianty `nudge` a `jitterdodge` jsou specifické pro určité méně geomy a jejich přímé volání je v podstatě výjimečné.

17.6.1 `jitter`

`position = "jitter"` je populární volba zejména u bodových grafů, ve kterých je velký problém s overplottingem. Toto použití je tak typické, že existuje dokonce speciální `geom_*()` funkce: `geom_jitter()`. Ta je ekvivalentní k `geom_point(position = "jitter")`.

`geom_jitter()` přidává ke skutečné pozici pozorování náhodnou složku, která je generovaná z uniformního rozdělení:

```
xtable %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_jitter(color = "red") +
    geom_point()
```

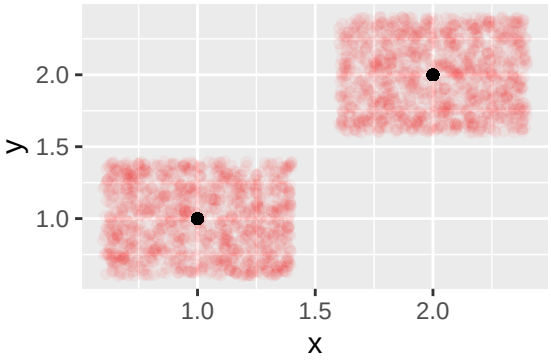


V obrátku vidíte skutečnou pozici pozorování vykreslenou pomocí `geom_point()` a červené tečky ukazující posunuté hodnoty vykreslené pomocí `geom_jitter()`. Protože je chyba náhodná, bude obrázek při každém vykreslení vypadat jinak.

Pomocí parametrů `width` a `height` je možná nastavit šířku a výšku obdélníku, ve kterém se posunutá pozorování budou nacházet. Protože se náhodná složka generuje z uniformního rozdělení, je každý posun stejně pravděpodobný. Skutečná pozorovaná hodnota je pak právě ve středu obdélníku. To ilustruje následující obrázek:

```
xtable2 <- data_frame(
    x = rep(1:2,1000),
    y = rep(1:2,1000)
)
```

```
xtable2 %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_jitter(
        alpha = 0.05,
        color = "red"
        ) +
    geom_point()
```



Posunutá pozorování (červené tečky) tvoří mračno rovnoměrně vyplňující celý obdélník za pozorovanou hodnotou (černá tečka), která je přesně v jeho středu.

17.6.2 `stack`

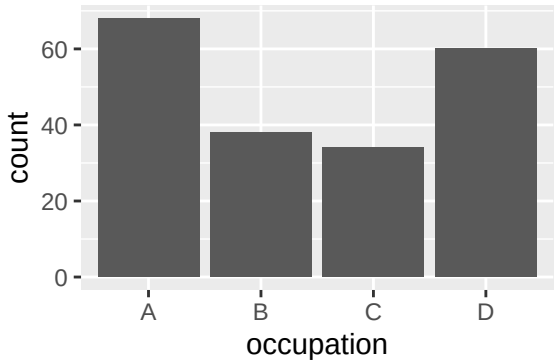
Typické použití této změny této modifikace pozice je u sloupcových grafů. Ty se v `ggplot2` tvoří pomocí `geom_bar()` a jejich základní nastavení je, že mají být použity k vykreslení rozdělení diskrétní proměnné.

Pro použití `geom_bar()` vytvoříme novou tabulku `occupation`:

```
occupation <- data_frame(
    sex = rep(c("male","female"),100),
    occupation = sample(c("A","B","C","D"), 200, replace = TRUE, prob = abs(rnorm(4))),
    age = sample(c("old","young"), 200, replace = TRUE)
)
```

Tabulka obsahuje povolání (A, B, C nebo D), pohlaví a věkovou kohortu 200 lidí. Zajímá nás rozdělení povolání ve vzorku:

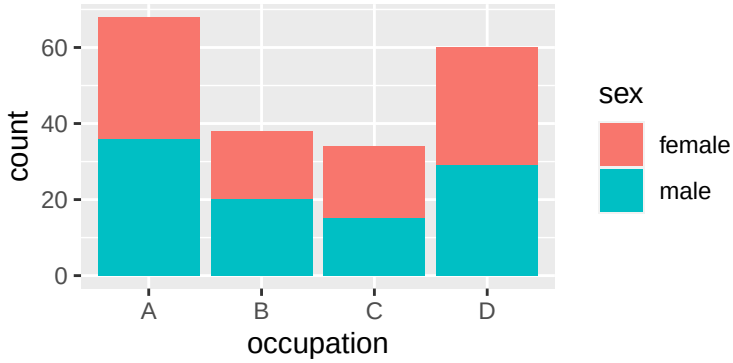
```
occupation %>%
    ggplot(
        aes(x = occupation)
    ) +
    geom_bar()
```



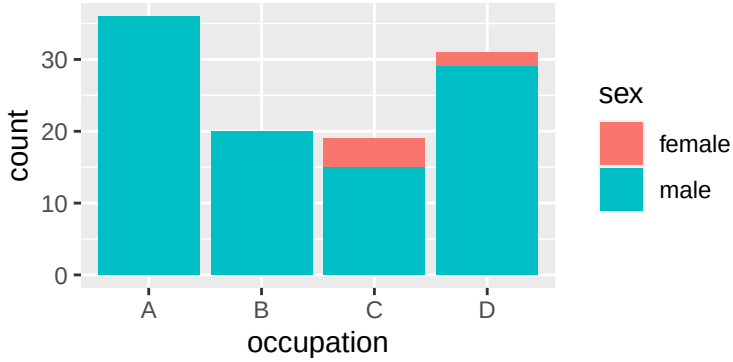
`geom_bar()` v základním nastavení, tedy se `stat = "count"`, provádí statistickou transformaci dat. Zjistí absolutní četnost jednotlivých variant proměnné v estetice `x` a tyto četnosti následně vykreslí formou sloupcového grafu.

Obrázek momentálně nijak nezohledňuje pohlaví, ale to se dá změnit. Lze ho namapovat například (a typicky) na estetiku `fill`:

```
occupation %>%
    ggplot(
        aes(x = occupation, fill = sex)
    ) +
    geom_bar()
```



Ve výsledku dodá statistická transformace provedená `stat_count()` pro každé povolání dvě hodnoty: počet mužů a počet žen. Ty by se v logicky měly kreslit přes sebe – patří v obrázku na jedno místo. Očekávali bychom tedy tento obrázek:



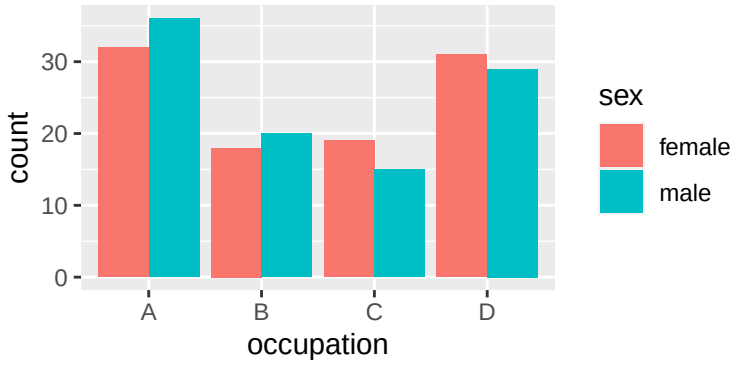
Takový obrázek však není příliš užitečný, protože poskytuje spolehlivou informaci pouze o maximálních hodnotách. Proto již v základním nastavení `geom_bar()` používá `position = "stack`, která počet žen a mužů postaví nad sebe – do jednoho komínku.

Poznámka: "Špatný" obrázek můžete vykreslit s `geom_bar(position = "identity)`.

17.6.3 `dodge`

Dalším typickým řešením u sloupcových grafů je vykreslení hodnot vedle sebe. Tento způsob čtenáři lépe předává informaci o počtu mužů a žen napříč povoláními. Horizontální posunutí sloupců zajišťuje `position = "dodge"`:

```
occupation %>%
    ggplot(
        aes(x = occupation, fill = sex)
    ) +
    geom_bar(position = "dodge")
```



17.6.4 `fill`

Poněkud specifickou úpravou pozice je `fill`, která upravuje nejen pozici, ale vlastně provádí i další statistickou transformaci. Nevrací totiž absolutní, ale relativní četnost:

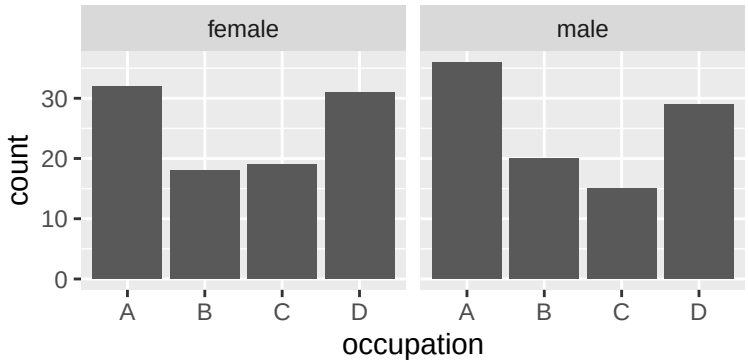
```
occupation %>%
    ggplot(
        aes(x = occupation, fill = sex)
    ) +
    geom_bar(position = "fill")
```

17.6.5 Facetování: na půli cesty mezi grupováním a pozicováním

Výše popsané pozicování mělo pozici geomů vlastně v rámci jednoho obrázku. Facetování k problému přistupuje jinak – rozlamuje původně jeden obrázek na více dílčích obrázků (facet). `ggplot2` umožňuje vytvářet buď matice dílčích obrázků pomocí `facet_grid()` nebo prosté rozlomení do více obrázků pomocí `facet_wrap()`.

Nejjednodušší příklad je použití `facet_wrap()`. Řekněme, že budeme chtít rozdělit obrázek s rozdělením povolání podle pohlaví:

```
occupation %>%
    ggplot(
        aes(x = occupation)
    ) +
    geom_bar() -> p

p + facet_wrap("sex")
```

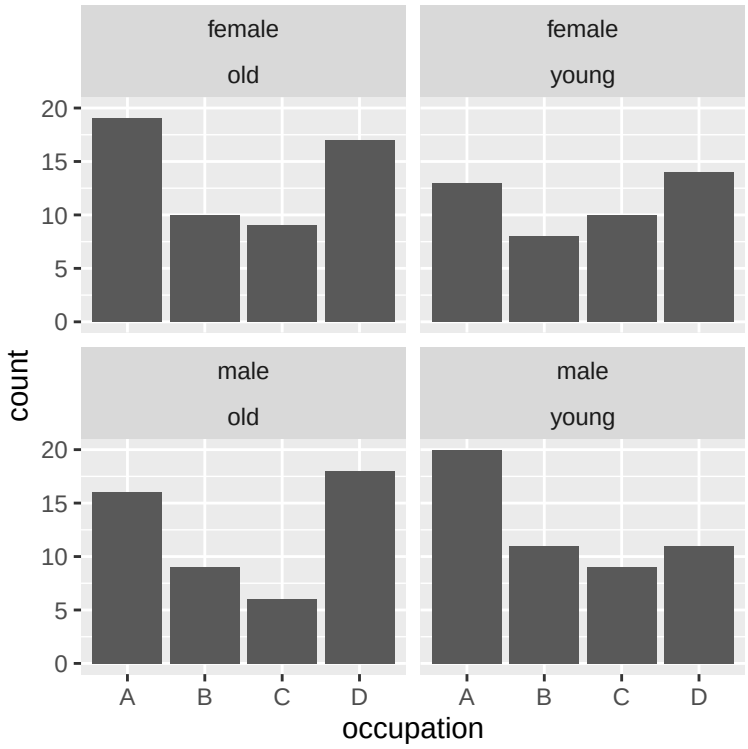


`facet_wrap()` rozdělil celkový obrázek na dva dílčí podle hodnot proměnné `sex`. Ta je v příkladu určena svým jménem (*character*). Další možností je její určení pomocí jednostranné rovnice (*formula*, blíže viz přednáška o ekonometrii v R):

```
p + facet_wrap(~ sex)
```

Vytvářet dílčí obrázky lze i podle kombinace více proměnných – můžeme chtít například samostatné dílčí obrázky pro kombinaci pohlaví a věku. V tomto případě se pro specifikaci proměnných použije vektor jejich jmen:

```
p + facet_wrap(c("sex","age"))
```

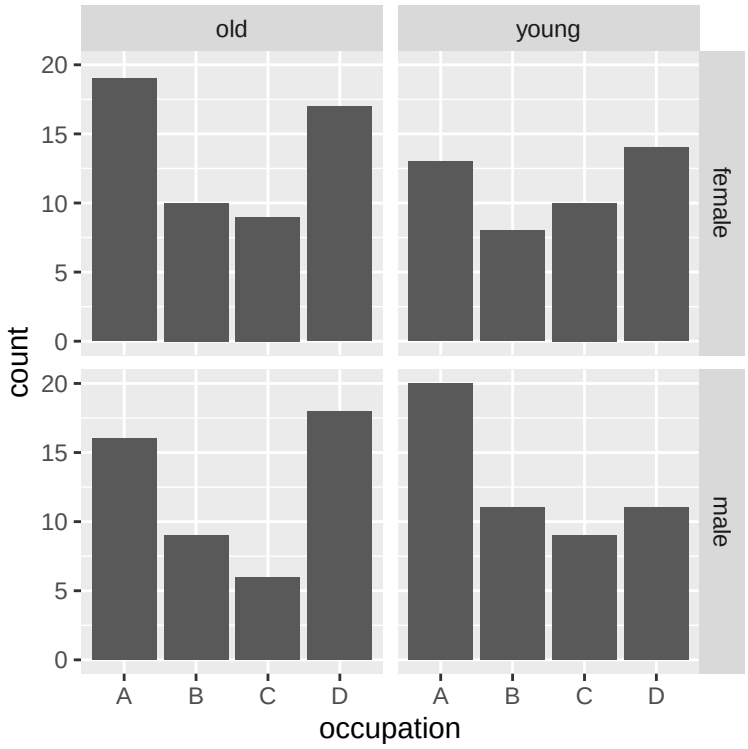


Nebo alternativně:

```
p + facet_wrap(~ sex + age)
```

Použít pro tyto aplikace `facet_wrap()` je možné, ale často je vhodnější sáhnout po `facet_grid()`, který je přímo navržen pro rozlomení obrázku podle kombinace více proměnných. `facet_grid()` vyžaduje specifikaci facetování pomocí objektu třídy `formula`:

```
p + facet_grid(sex ~ age)
```



Na levé straně rovnice (na levé straně ~) je proměnná (nebo jejich kombinace spojená +), která má tvořit řádky a na levé straně je proměnná nebo jejich kombinace, která má tvořit sloupce.

Funkce vytvářející facetování mají i další užitečné parametry. Pro `facet_wrap()` jsou specifické parametry `ncol` a `nrow`, které určují formát výstupní tabulky dílčích obrázků – kolik má mít sloupců a kolik řádků. Stačí přitom zadat jen jeden z těchto parametrů:

```
p + facet_wrap(c("sex","age"), nrow = 1)
```

Užitečný, ale pro správné čtení dat velmi ošemetný, parametr je `scales`. Ten umožňuje uvolnit škály v jednotlivých dílčích obrázcích. Základní hodnotou je `fixed`. V tomto případě jsou měřítka na všech dílčích

obrázcích stejná. Varianty `free`, `free_y` a `free_x` umožňují uvolnit měřítka na všech osách nebo jen na ose `x` popřípadě `y`:

```
p + facet_wrap(c("sex","age"), nrow = 1, scales = "free_y")
```

Příklad ilustruje, že uvolnění os může být pro čtenáře velmi zavádějící. Stejná výška sloupce totiž v různých dílčích obrázcích znamená jinou pozorovanou hodnotu.

17.7 Souřadnicové systémy

ggplot2 podporuje řadu souřadnicových systémů. Základní volbou je lineární `coord_cartesian()`, ale podporovány jsou i nelineární souřadnicové systémy, které se ovšem používají pouze ve velmi specifických případech jako je například vykreslování map (`coord_map()`). Dalším příkladem nelineárního systému je `coord_polar()`.

Základní volbou aplikovanou v téměř všech voláních *ggplot2* je bezpochyby lineární systém, ve kterém je pozice prvků určena jejich souřadnicemi na ose `x` a `y`. Základní funkcí je `coord_cartesian()`, ale *ggplot2* obsahuje i její různé mutace a transformace:

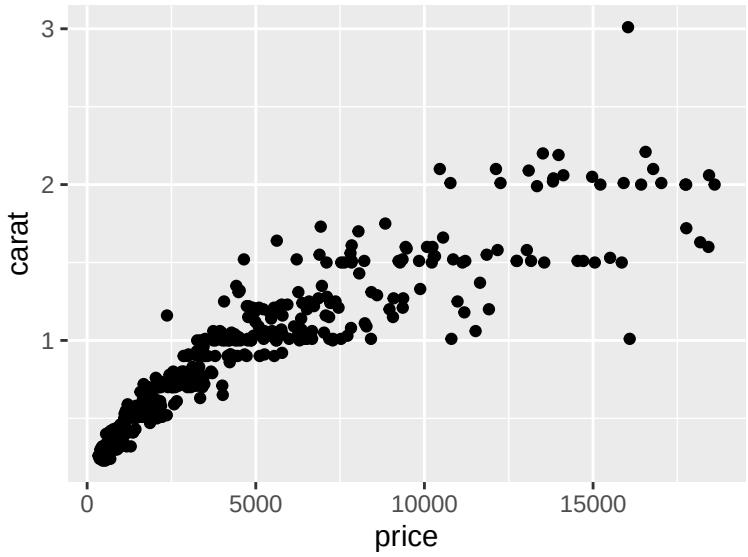
- `coord_fixed()` udržuje konstantní zadaný poměr stran.
- `coord_equal()` je zkratka pro `coord_fixed()` s poměrem stran 1.
- `coord_flip()` prohazuje osy `x` a `y`.
- `coord_trans()` umožňuje provést transformaci os.
- `coord_sf()` je speciální funkce pro práci s mapami.

Fungování `coord_flip()` je možné ilustrovat pomocí jednoduchého bodového grafu. V definici mapování je váha (`carat`) namapována na osu `x` a cena na osu `y`:

```
diamonds %>%
    ggplot(
        aes(x = carat, y = price)
    ) +
    geom_point() -> p
```

Použití `coord_flip()` prohodí osy `x` a `y` – výsledek tedy odpovídá použití `aes(y = carat, x = price)`:

```
p + coord_flip()
```

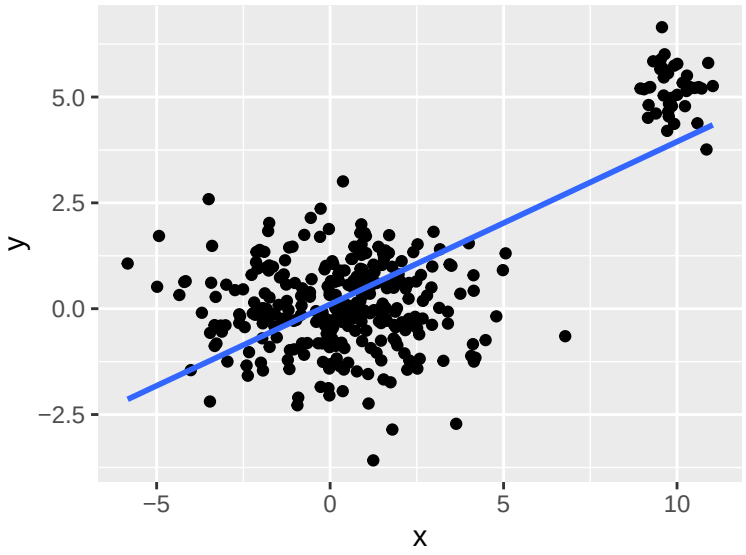


`coord_flip()` je užitečné u některých specifických geomů – například u boxplotů, která zobrazují základní charakteristiky rozdělení spojité proměnné.

`coord_trans()` umožňuje transformovat osy podobným způsobem jako `scale_*_*()` funkce. Stejně jako ostatní funkce odvozené od `coord_cartesian()` umožňuje nastavit interval zobrazený na osách. Podobnou možnost mají opět i `scale_*_*()` funkce.

Nabízí se tedy otázka, na co jsou vlastně funkce `coord_*()` užitečné. Pro demonstraci odpovědi nasimulujeme velmi specifickou tabulku:

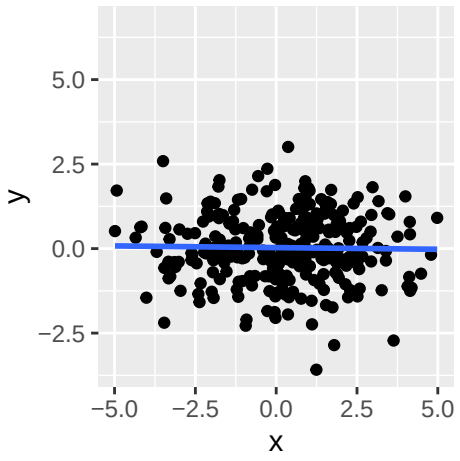
```
cltable <- data_frame(
    x = c(
        rnorm(300, sd = 2),
        rnorm(40, sd = 0.5) + 10
        ),
    y = c(
        rnorm(300, sd = 1),
        rnorm(40, sd = 0.5) + 5
        )
)

cltable %>%
    ggplot(
        aes(x = x, y = y)
    ) +
    geom_point() +
    geom_smooth(method = "lm", se = FALSE) -> p

p
```



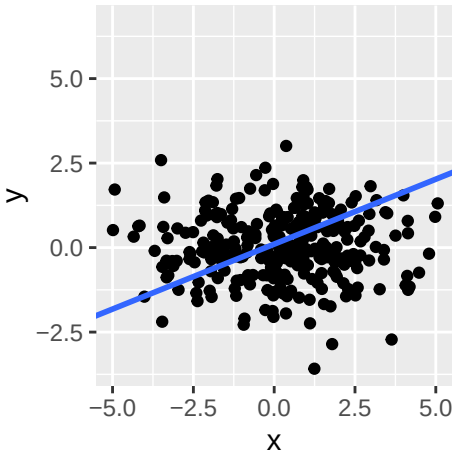
Simulovaná data obsahují dva shluky pozorování. Pokud je proložíme regresní přímkou (tj. vykreslíme statistickou transformaci dat) získáme (falešné) zdání existence rostoucího vztahu mezi veličinou na ose `x` a na ose `y`.

Řekněme, že z nějakých důvodů budeme chtít omezit zobrazené hodnoty na ose `x` na interval $[-5, 5]$. To lez provést buď pomocí funkcí `scale_x_*()` nebo `coord_*()`:

```
p + scale_x_continuous(limits = c(-5,5))
```




```
p + coord_cartesian(xlim = c(-5,5))
```



Rozdíl je zjevný. Při použití `scale_x_*()` je regresní přímka prakticky vodorovná. Naopak při použití `coord_*()` je stále rostoucí. Rozdíl je v tom, kdy je aplikované statistická transformace (v tomto případě vyhlazení). Při použití `scale_*_*()` funkce je nejprve omezen rozsah dat a následně je aplikována transformace. Výsledek tak může být zavádějící. Naproti tomu při použití `coord_*()` je nejprve aplikována transformace a následně je omezen rozsah zobrazených dat. Transformace tak reflektuje i ta pozorování, která nejsou zobrazená.

`coord_*()` tak nabízí možnost skutečně "zazoomovat" část obrázku, aniž by to ovlivnilo obrázek samotný.

17.8 Vzhled obrázků

Vzhled obrázků, nebo přesněji prvky, které nemají vztah k datům se v *ggplot2* ovládají pomocí funkce `theme()`. Ta umožňuje změnit vzhled obrázků k nepoznání: můžete mít obrázek, který vypadá, jako by byl vytvořen v Excelu, Stata, nebo jako by vyšel v The Economist.

Funkce `theme()` má obrovské množství parametrů a nebylo by praktické je nastavovat vždy u každého obrázku. Proto v *ggplot2* existuje řada předpřipravených kompletních témat (vzhledů). Například `theme_bw()`, `theme_classic()` nebo `theme_void()`. Základní vzhled obrázků potom odpovídá `theme_gray()`. Nápověda *ggplot2* obsahuje hrubá doporučení, kdy je vhodné použít které téma.

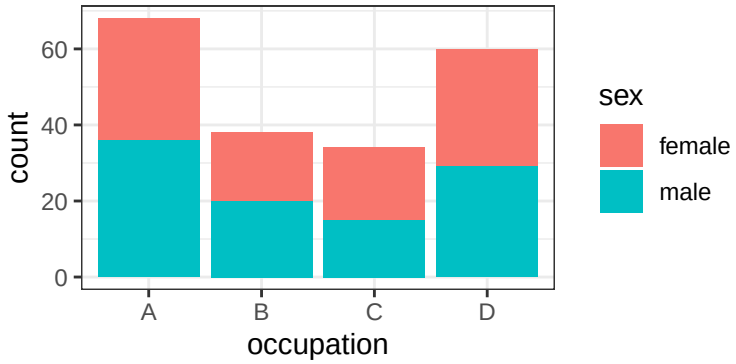
17.8.1 Ukázky předpřipravených témat

Pro ukázky použijeme již známý sloupcový graf:

```
occupation %>%
    ggplot(
        aes(x = occupation, fill = sex)
    ) +
    geom_bar() -> p
```

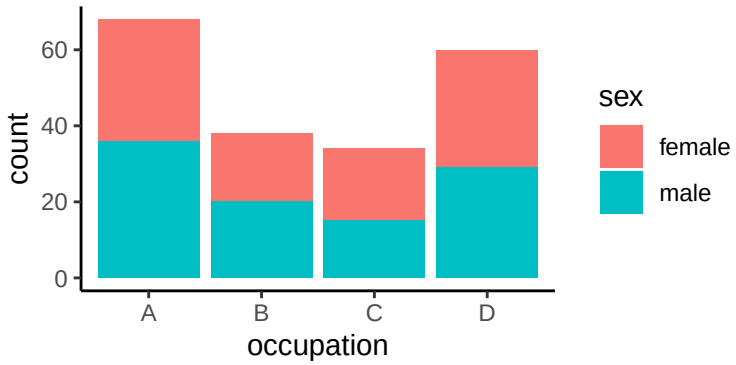
Populární téma je `theme_bw()`, které *ggplot2* doporučuje například pro prezentace:

```
p + theme_bw()
```



Klasicky vypadající téma s minimem čar je `theme_classic()`:

```
p + theme_classic()
```



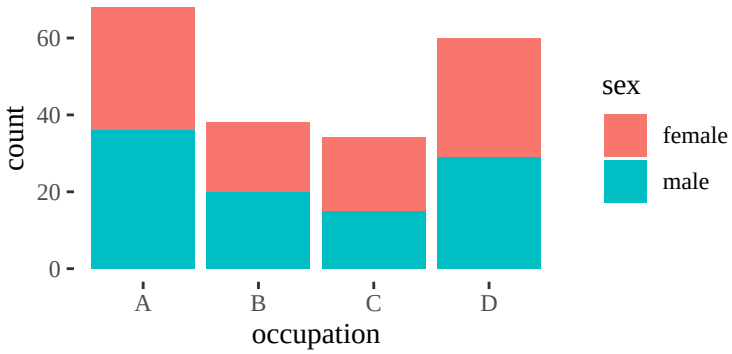
Například pro kreslení map oceníte velmi speciální téma `theme_void()`, které zahodí všechny čáry:

```
p + theme_void()
```

Sbírku předpřipravených témat obsahuje například balík *ggthemes*. Kromě různých `theme_*()` obsahuje balík i řadu `scale_*_*()` a několik `geom_*()` funkcí.

Například minimalistické téma vytvořené podle Tufteho doporučení obsahuje `ggthemes::theme_tufte()`:

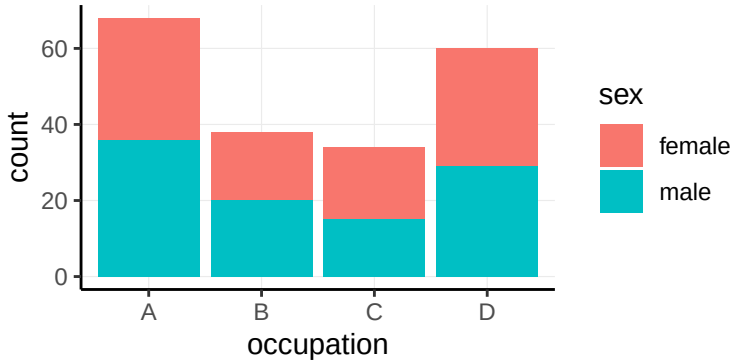
```
p + ggthemes::theme_tufte()
```



17.8.2 Modifikace témat

ggplot2 umožňuje vytvářet vlastní předpřipravená témata, ale většinou bohatě postačuje použít předpřipravené a upravit ho, to lze zařídit následujícím způsobem:

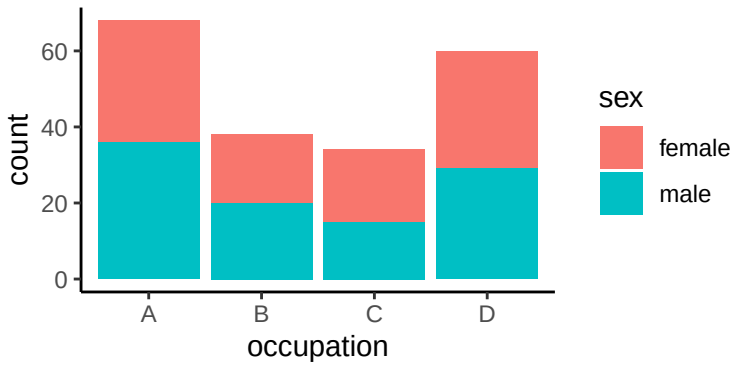
```
p +
    theme_classic() +
    theme(
        panel.grid.major = element_line(size = 0.2)
    )
```



U `theme()` funguje vrstvení stejně, jako u všeho ostatního. Volání jednotlivých funkcí modifikuje podkladovou datovou strukturu. Zavolání funkce `theme_classic()` tedy nastaví všechny parametry tak, jak to odpovídá `theme_classic` – to znamená, žádné linky na pozadí. Následné volání `theme()` přepíše to, co před tím nastavilo volání `theme_classic()` a linky přidá. Žádný další parametr nebyl v `theme()` zadán a proto se nic jiného nezmění.

Pokud by funkce byly volány v opačném pořadí, potom by byl výsledek následující:

```
p +
    theme(
        panel.grid.major = element_line(size = 0.2)
    ) +
    theme_classic()
```



Výsledek by přesně odpovídal `theme_classic`. `theme_classic()` totiž bez milosti přepíše veškeré formátování vytvořené v předchozích voláních `theme()` nebo `theme_*()`.

Samotné volání `theme()` ve výše uvedeném příkladu vypadá skutečně velmi krypticky, ale není to tak, protože, podobně jako (skoro) ve všem v `ggplot2`, za ním stojí promyšlený systém.

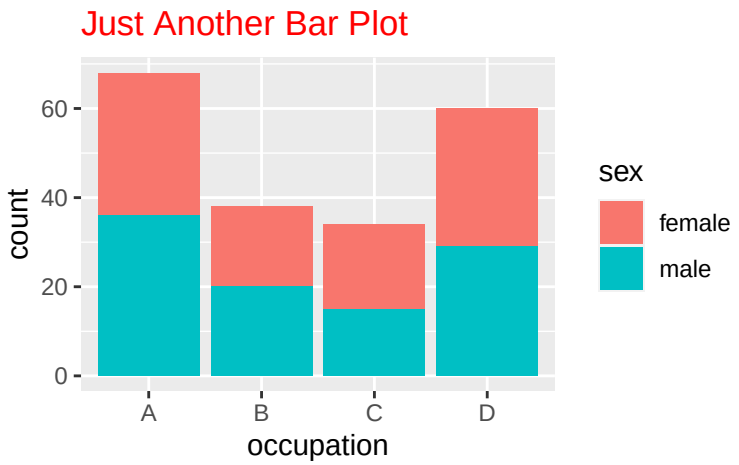
Funkce `theme()` má opravdu mnoho parametrů (viz `?theme`), jejich jména odpovídají jménům jednotlivých grafických prvků. Jména jsou naštěstí často velmi výstižná, intuitivní, a navíc je RStudio zvládá (u novějších verzí *ggplot2*) našeptávat.

Formátování grafických prvků se ve většině případů provádí pomocí volání jedné z následujících funkcí:

```
element_text(family = NULL, face = NULL, colour = NULL, size = NULL,
  hjust = NULL, vjust = NULL, angle = NULL, lineheight = NULL,
  color = NULL, margin = NULL, debug = NULL)

element_line(colour = NULL, size = NULL, linetype = NULL,
  lineend = NULL, color = NULL)

element_rect(fill = NULL, colour = NULL, size = NULL, linetype = NULL,
  color = NULL)

element_blank()
```

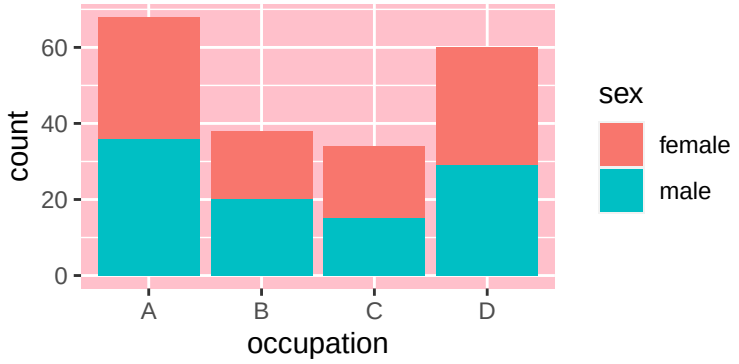
Tu správnou lze uhodnout. Pokud chcete upravit formátování prvku, který je ve své podstatě text, potom chcete použít `element_text()`. Příkladem může být modifikace titulku (`plot.title`) obrázku:

```
p +
    labs(
       title = "Just Another Bar Plot"
    ) +
    theme(
        plot.title = element_text(colour = "red")
    )
```



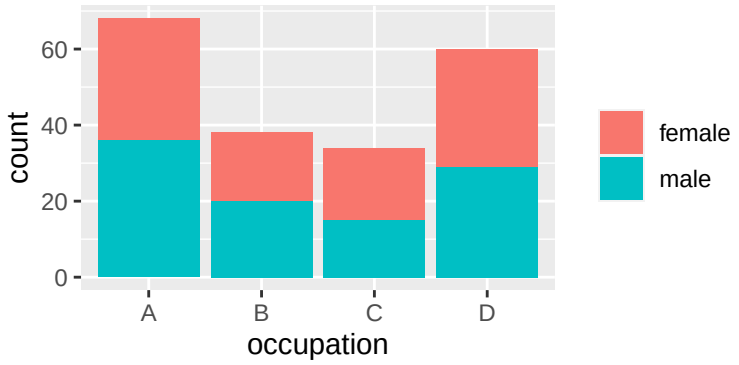
Pokud je prvek svým charakterem čára, potom se jeho formátování nastavuje pomocí `element_line()` (viz modifikace čar na pozadí výše). Prvky, které jsou ve své podstatě čtyřúhelník se potom modifikují pomocí `element_rect()`. Příkladem může být změna barvy pozadí grafu na poníkovou:

```
p +
    theme(
        panel.background = element_rect(fill = "pink")
    )
```



Speciální funkce je potom `element_blank()`, která způsobí, že se daný prvek z obrázku kompletně vypustí:

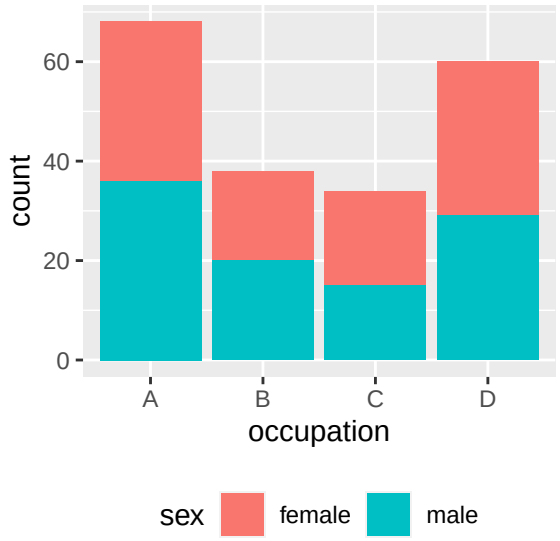
```
p +
    theme(
        legend.title = element_blank()
    )
```



Například toto volání vypustilo jméno legendy a to zcela. Špinavé triky typu `scale_fill_discrete(name = "")` sice způsobí, že jméno "nebude vidět". Respektive se vykreslí prázdný řetězec. Místo pro titulek však bude alokováno a bude tak rozhazovat uspořádání obrázku. `element_blank()` je správná cesta.

Některé parametry v `theme()` se nastavují pomocí speciálních funkcí – typicky jde o nastavení velikostí pomocí funkce `unit()`. Další parametry se nastavují prostým zadáním hodnoty. Příkladem může být pozice a orientace legendy:

```
p +
    theme(
        legend.position = "bottom",
        legend.direction = "horizontal"
    )
```



17.9 Ukládání obrázků

Vytvořené obrázky je možné ukládat s použitím funkce `ggsave()`:

```
ggsave(filename, plot = last_plot(), device = NULL, path = NULL,
  scale = 1, width = NA, height = NA, units = c("in", "cm", "mm"),
  dpi = 300, limitsize = TRUE, ...)
```

V základním nastavení `plot = last_plot()` funkce `ggsave()` ukládá poslední vykreslený obrázek. Do parametru `plot` je však možné přiřadit i obrázek (datovou strukturu) uloženou v proměnné. To je velmi užitečné například při tvorbě obrázků, jejichž vykreslování je velmi náročné – komplikovaných map, bodových grafů s opravdu velkým počtem pozorování a podobně.

Jméno souboru, do kterého se má obrázek uložit se definuje v parametru `filename` jako řetězec. `ggsave()` řetězec analyzuje a podle přípony zvolí exportní formát obrázku. `ggsave()` umožňuje formáty do vektorových formátů i bitmap:

- vektorové formáty: eps, ps, tex (pictex), pdf, svg a wmf (pouze ve Windows)
- bitmapy: jpeg, tiff, png, bmp

Formát (výstupní zařízení) lze zadat přímo pomocí parametru `device`.

První rozhodnutí je, zda použít vektorový formát nebo bitmapu. Uložení do bitmapy znamená uložení "fotografie" obrázku. Takto uložený obrázek může být menší (přesněji nikdy nebude velký) a lze očekávat, že

půjde vložit do libovolného dokumentu. Na druhou stranu nepůjde zvětšovat bez různých zkreslení a šumů a výstup nikdy nebude kvalitnější než v případě použití vektorové grafiky.

Vektorové formáty ukládají data, která popisují, kde co na obrázku je. Na rozdíl od bitmap je lze libovolně škálovat bez ztráty kvality a občas přijde vhod i možnost je upravovat (např. posunovat popisky u `geom_label()`/`geom_text`). Schopnost různých WYSIWYG editorů (např. MS Word atp.) pracovat s vektorovými formáty je navíc omezená a je lepší ji řádně otestovat. Nevýhodou je také jejich velká velikost v případě, že objektů v obrázku je mnoho nebo mají složité tvary.

Příkladem obrázku, u kterého se dramaticky liší velikost podle použitého formátu je následující mapa ČR s katastrálními územími obcí. Zobrazená bitmapa (png) má přibližně 0,1 MB. Stejná mapa ve vektorovém formátu má téměř 11,6 MB.

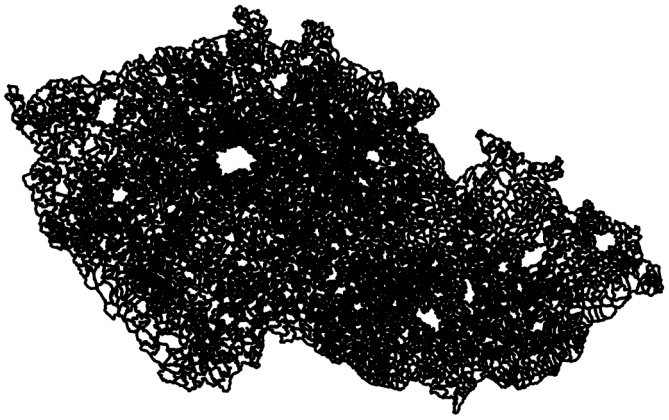


Figure 17.5: Mapa ČR s katastrálními územími obcí

Další parametry `ggsave()` umožňují přesné nastavení výstupu co do rozměrů a rozlišení (pouze pro bitmapy). Při vhodném nastavení těchto parametrů vzhledem k výsledné publikaci lze dosáhnout dobrých výsledků i při používání bitmap. Ve většině případů je však doporučeníhodné používání vektorových formátů.

17.10 Co dělat a co nedělat

ggplot2 a jeho rozšíření nabízí ohromnou škálu možností. Před tím než začnete kreslit je dobré se rozmyslet, zda je vůbec rozumné kreslit. Často je užitečnější a pro čtenáře srozumitelnější prezentovat data v podobě tabulky – a to zejména pokud je jich málo. Pokud se rozhodnete kreslit, potom se vždy musíte snažit, aby obrázek čtenáři srozumitelně komunikoval nezkreslenou informaci.

Tufte (2001) formuluje jednoduché doporučení, které byste měli mít vždy v paměti:

Graphical excellence is that which gives to the viewer the greatest number of ideas in the shortest time with the least ink in the smallest space.

Part V

Statistické metody v R

18 Ekonometrie v R

Skupina balíků sdružená v `tidyverse` představuje kompaktní sadu nástrojů pro práci s daty. Sdílejí datové struktury (tabulky v tidy formátu) i logiku ovládání. Bohužel podobná unifikace chybí u nástrojů, které výzkumník potřebuje pro ekonometrickou (statistickou) analýzu dat, která následuje po zpracování dat. Ekonometrické nástroje jsou v R rozptýleny v mnoha balících různé kvality, které nejsou žádným způsobem koordinovány nebo unifikovány. To velmi znepříjemňuje ekonometrickou analýzu dat v R. I přesto R poskytuje všechny základní a mnoho pokročilých nástrojů a v zásadě pokrývá všechny běžné potřeby výzkumníků.

Účelem této kapitoly není vysvětlovat teorii ekonometrické analýzy nebo diskutovat vhodnost nasazení jednotlivých metod nebo designu regresní analýzy (identifikační strategie), ale seznámit čtenáře se základní logikou fungování ekonometrických nástrojů v R. Právě základní logika fungování je totiž něco, co drtivá většina ekonometrických nástrojů v R sdílí. Tuto základní logiku lze popsat pomocí obrázku XXX:

Na počátku ekonometrické analýzy má výzkumník data a představu o tom, jak svět funguje (o data generujícím procesu). Pro naprostou většinu ekonometrických nástrojů (a všechny základní) je vhodné připravit si data do podoby tabulky v tidy formátu. Představa o fungování světa (tj. o vztazích mezi proměnnými v datové tabulce) se formalizuje do podoby modelu. Jeho podoba se předává R v podobě speciální datové třídy `formula`.

Data a specifikovaný model jsou nezbytnými vstupy pro estimátor. Estimátor je typicky funkce, která provede odhad parametrů modelu a vytvoří výstupní objekt. Tento objekt obsahuje typicky pouze samotný odhad parametrů modelu, tabulku dat použitých pro odhad, rezidua, vyrovnané hodnoty a další doplňující údaje. Typicky neobsahuje žádné testy, robustní směrodatné chyby a podobně. Analýza a zobrazení odhadu (výsledků) se v R typicky provádí pomocí specializovaných funkcí, pro které je výstup estimační funkce vstupem.

V této kapitole se naučíte:

- Specifikovat model ve třídě `formula`
- Použít základní estimační funkce
- Provést základní diagnostické testy odhadu
- Vypočítat robustní směrodatné chyby
- Provádět hromadně odhady více modelových specifikací na stejném vzorku dat
- Provádět hromadně odhady jedné modelové specifikace na různých vzorcích dat
- Exportovat výsledky do přehledných tabulek

18.1 R a ostatní

R představuje spolu s Pythonem špičkový software ohledně práce s daty. Tomu odpovídá i to, že právě tyto dva jazyky jsou mezi datovými analytiky pravděpodobně nejvíce používané. Pozice R ohledně ekonometrické analýzy však zdaleka není tak jasná. V ekonomii se rozhodně nejedná o nejpoužívanější nástroj. Tím je stále bezesporu Stata, která se logikou svého fungování velmi podobá Gretlu. (Gretl je navržen podle Stata.) Stata pracuje nad jednou tabulkou dat, která je nahraná v paměti. V rámci každého odhadu specifikujete například i typ robustních chyb, které chcete použít. Své rozhodnutí však později nemůžete změnit. Výsledkem takového návrhu je velmi obtížná práce s daty ve Stata (např. slučování dvou tabulek). Nutnost specifikovat všechny parametry již v rámci odhadu modelu může být také velmi omezující v případě odhadů, které jsou časově náročné. Stata také dává oproti R výzkumníkovi často menší svobodu v detailním nastavení jednotlivých procedur. V neposlední řadě je Stata také velmi nákladná (levné verze jsou omezené tak, že nejsou v reálném boji použitelné). Omezené možnosti jsou ovšem lehce vyvažovány jednoduchostí ovládání. Hlavní výhodou

Stata je však implementace obrovského množství nejrůznějších estimátorů, které mají velmi konzistentní rozhraní.

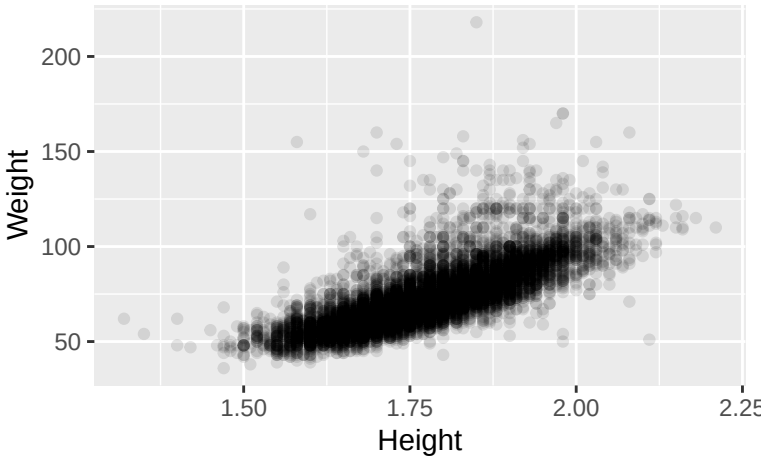
18.2 Specifikace modelu pomocí `formula`

Fungování ekonometrických nástrojů můžeme ilustrovat na příkladu datasetu s výškou, váhou, věkem, sportem, a pohlavím sportovců, kteří se zúčastnili olympijských her v Londýně v roce 2012 (dostupný v `VGAMdata::oly12`):

```
VGAMdata::oly12 %>%
  mutate(
    Sport = str_extract(Sport, "[:alnum:]*")
  ) %>%
  select(
    Height, Weight, Sex, Age, Sport
  ) %>%
  as_tibble %>%
  drop_na() %>%
  mutate_if(is.factor,as.character) -> oly12

print(oly12, n=5)
```

```
## # A tibble: 9,038 x 5
##   Height Weight Sex     Age Sport
##    <dbl>  <int> <chr> <int> <chr>
## 1   1.7      60 M        23 Judo
## 2   1.93    125 M        33 Athletics
## 3   1.87     76 M        30 Athletics
## 4   1.78     85 F        26 Athletics
## 5   1.82     80 M        27 Handball
## # ... with 9,033 more rows
```

Nejprve nás bude zajímat vztah mezi váhou a výškou sportovců. Ten si můžeme vykreslit pomocí jednoduchého bodového grafu:



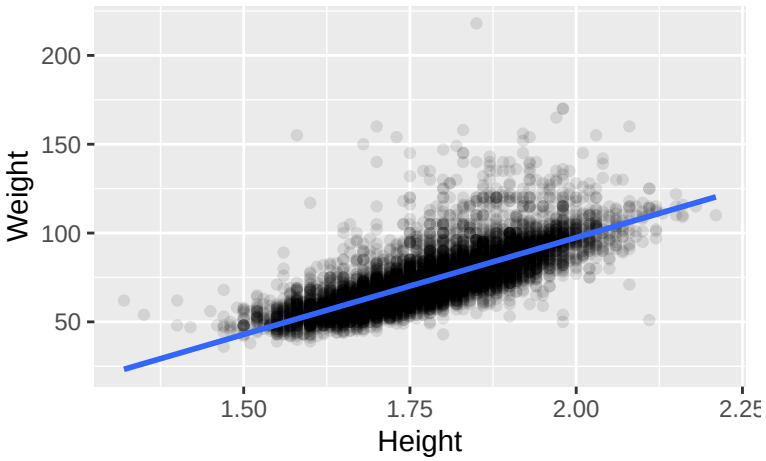
Zdá se, že existuje lineární vztah mezi váhou a výškou sportovců, který můžeme popsat rovnicí:

$$Weight = \alpha + \beta Height + \varepsilon$$

kde α a β jsou neznámé parametry, které chceme odhadnout a ε je náhodná složka.

Tuto rovnici si můžeme představit jako přímku, která je proložena mrakem pozorování:

```
## `geom_smooth()` using formula 'y ~ x'
```



Rovnici však musíme zapsat tak, aby byla srozumitelná i pro R. K tomu slouží objekty třídy `formula`. Jejich formulace následuje syntax:

```
LHS ~ RHS
```

Tedy výrazy na levé straně (LHS) = (zastoupené znakem ~) výrazy na pravé straně (RHS). Rovnici vysvětlující váhu bychom tedy mohli popsat jako:

```
model <- Weight ~ Height
class(model)
```

```
## [1] "formula"
```

V ekonomii je závislá (vysvětlovaná) proměnná (tedy proměnná na levé straně rovnice) funkcí více než jedné nezávislé (vysvětlující) proměnné. Pokud bychom chtěli vysvětlovat váhu nejen výškou, ale i věkem, potom bychom rozšířili specifikaci následujícím způsobem:

```
model <- Weight ~ Height + Age
```

Dodatečná proměnné jsou přidávány pomocí +.

Za povšimnutí stojí, že při použití takovéto specifikace by byl model odhadnut s úrovňovou konstantou α – i když není explicitně v rovnici přítomna. Explicitně se naopak musí zadat její nepřítomnost připojením `-1` nebo `0`:

```
model <- Weight ~ Height - 1
model <- Weight ~ Height + 0
```

Jména proměnných v rovnici musí odpovídat jménům sloupců v datové tabulce. Zásadní výhodou R je, že do rovnice je možné zadat i transformace dat:

```
model <- Weight ~ log(Height)
model <- Weight ~ Height >= 1.8
```

První varianta například odpovídá specifikaci:

$$Weight = \alpha + \beta \log(Height) + \varepsilon$$

Ve druhé je váha vysvětlována dummy (umělou proměnnou). Výraz `Height >= 1.8` vytvoří logickou proměnnou, která je při volání estimační funkce transformována na vektor s hodnotami 1 a 0.

Není tedy potřeba vytvářet nové transformované sloupce v datové tabulce. To je velký rozdíl mezi R a Statou, Gretlem,... (Jakkoliv je v některých případech vytvoření transformovaných proměnných do tabulky doporučeníhodné.)

V rovnici může být přímo přítomna celá řada funkcí – s výjimkou těch, jejichž interpretace by nebyla jasná. Pokud bychom například chtěli odhadnout modelovou specifikaci

$$Weight = \alpha + \beta(Height + Age) + \varepsilon$$

potom je nutná sdělit R, že má nejdříve sečíst `Height` a `Age` a pro vysvětlení `Weight` použít až výsledný součet. Pro tyto účely se používá funkce `I()`. Ta sděluje, že se mají nejprve provést operace definované uvnitř funkce a pro odhad použít až výsledek.

Výše uvedený příklad by se tak R tlumočil jako

```
model <- Weight ~ I(Height + Age)
```

Třída formula umožňuje jednoduše jednoduše definovat i interakční členy pomocí operátorů dvojtečka (`:`) a hvězdička (`*`):

```
model <- Weight ~ Height:Sex
```

Chápe R jako

$$Weight = \alpha + \beta Height \times Sex_{male} + \gamma Height \times Sex_{female} + \varepsilon$$

Tato rovnice vyjadřuje možný odlišný mezní efekt výšky na váhu u mužů a žen. Výsledné proložení mrakem pozorování by se tak v bodovém grafu výše vlastně rozpadlo na dvě přímky s odlišným sklonem (parametry β a γ) a shodnou úrovňovou konstantou α.

Další možností jak popsat interakci proměnných je `*`:

```
model <- Weight ~ Height*Sex
```

Tato `formula` odpovídá jiné rovnici:

$$Weight = \alpha + \beta Height + \epsilon Sex_{male} + \gamma Height \times Sex_{male} + \varepsilon$$

Tato specifikace umožňuje odlišnost regresní křivky pro muže a ženy jak ve sklonu, tak v úrovňové konstantě.

Zvláštní význam má i symbol stříšky (`^`). Rovnici

$$Weight = \alpha + \beta_1 Height + \beta_2 Age + \beta_3 Sex_{male} + \gamma_1 Height \times Sex_{male} + \gamma_2 Height \times Age + \gamma_3 Sex_{male} \times Age + \varepsilon$$

můžeme zapsat jako:

```
model <- Weight ~ (Height + Age + Sex)^2
```

Výraz se přeloží tak, že výsledná rovnice obsahuje jednotlivé proměnné a také jejich interakce do, v tomto případě, druhého řádu.

Pro třídu formula jsou implementovány i některé zajímavé metody. Velmi užitečná je například funkce `update()` s velmi prostou syntaxí:

```
update(old,new,...)
```

Kde `old` je původní rovnice a `new` pravidla, která stanovují, jak se původní rovnice má upravit.

```
model <- Weight ~ Height + Age
print(model)
```

```
## Weight ~ Height + Age
```

Takto můžeme zachovat všechny proměnné – reprezentované pomocí `.` a jen některou přidat nebo odebrat:

```
model %>%
  update(. ~ . -Age)
```

```
## Weight ~ Height
```

Nebo je možné nahradit celou stranu rovnice:

```
model %>%
  update(. ~ Age)
```

```
## Weight ~ Age
```

18.3 Odhad modelu

Při volání estimační funkce se dohromady spojí všechny tři komponenty: modelová specifikace, data a estimátor.

Základní estimační funkcí je `lm()`, určená pro odhad lineárních modelů. Funkce přijímá celou řadu parametrů:

```
lm(formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
```

Mezi základní patří:

- `formula` je objekt třídy "formula", který obsahuje symbolický popis rovnice, jejíž parametry se mají odhadnout.
- `data` je tabulka (`data.frame`) která obsahuje data, která se mají použít. Jména sloupců v tabulce musí odpovídat proměnným použitým v parametru `formula`. Datová tabulka nemusí být specifikována. Pokud je tento parametr prázdný, potom R interpretuje jména v modelu jako jména proměnných z prostředí, ze kterého je funkce `lm()` volána.
- `subset` je nepovinný parametr, který určuje, které pozorování (řádky) se mají použít pro odhad
- `weights` je vektor vah. Pokud není zadán je pro odhad použit obyčejný OLS estimátor, ve kterém mají všechna pozorování stejnou váhu. V případě zadání vah je vráceno WLS.

Struktura a forma těchto základních vstupů je typicky shodná právě s `lm()`.

18.3.1 Vytvoření matice plánu

Pro odhad použijeme model v následující specifikaci:

```
model <- Weight ~ Height + Age + I(Age^2) + Sex
```

Proměnné `Weight`, `Height` a `Age` jsou numerické spojité proměnné. Proměnná `Sex` je zcela jiný případ. Jsou to kategoriální proměnná, se dvěma úrovněmi (`M` – muž, `F` – žena). Taková proměnná nemůže vstoupit do regrese přímo. Je nutné ji transformovat na matici nul a jedniček. Transformovaná matice bude mít vždy o jeden sloupec méně, než kolik má transformovaná proměnná úrovní.

Estimační funkce si všechny vstupní faktory transformuje sama pomocí interního volání funkce `model.matrix()`. V rámci volání tento funkce se provedou i požadované transformace dat. V případě našeho modelu vytvoření druhé mocniny věku. Výsledkem volání `model.matrix()` je tedy matice plánu:

```
model.matrix(model, data = oly12) %>% head()
```

```
##   (Intercept) Height Age I(Age^2) SexM
## 1           1   1.70  23      529    1
## 2           1   1.93  33     1089    1
## 3           1   1.87  30      900    1
## 4           1   1.78  26      676    0
## 5           1   1.82  27      729    1
## 6           1   1.82  30      900    0
```

Funkce `model.matrix()` provede požadované transformace numerických proměnných (přidán sloupec `I(Age^2)`) a pokud najde jinou než numerickou proměnnou (`factor`) provede jejich transformaci do matice binárních proměnných. Při této transformaci postupuje následovně. Prohlédne si faktor a určí referenční úroveň (*level*). Ta nebude ve výsledné matici obsažena, pokud by byla, model by nešel odhadnout. Pro každou zbývající úroveň vytvoří vektor, který nabývá hodnoty 1 pro pozorování s danou úrovní faktoru. V příkladu výše si `model.frame()` vybral jako referenční úroveň hodnotu `F` a vytvořil vektor `SexM`, který nabývá hodnoty 1 pro muže a 0 ve všech ostatních případech.

Pokud by byla vstupní kategoriální proměnná třídy `character`, potom by ji `model.frame()`nejprve transformoval na `factor` a pokračoval tak, jak je popsáno výše. Toto chování může být motivem pro to, aby uživatel sám provedl konverzi na faktor. Třída faktor totiž umožňuje nastavit, která úroveň má být použita jako referenční. Toho můžeme docílit pomocí několika cest.

První variantou je konverze do `factor` s pomocí `base::factor()`. Tato funkce umožňuje uživateli zadat vektor možných úrovní. První z nich se potom nastaví jako referenční:

```
factor(Sex$oly12, c("M","F"))
```

Tento postup lze aplikovat i na proměnné, které již jsou `factor`. Nevýhodou je nutnost zadávat všechny úrovně. Podobnou variantou je použití funkce `forcats::as_factor()`. Ta má vlastnost, že pořadí úrovní stanoví podle podle pořadí jejich výskytu v transformovaném vektoru. Problém této funkce je, že pracuje zvláštně s chybějícími hodnotami. (To se pravděpodobně bude během vývoje balíku ještě měnit.)

Další principiálně odlišnou variantou je nastavení referenční úrovně pomocí funkce `stats::relevel()` (alternativně `forcats::fct_relevel()`). Ta umožňuje prosté nastavení referenční úrovně:

```
factor(Sex$oly12) %>% relevel("M")
```

Po změně referenční úrovně bude výsledek následující:

```
oly12 %>%
  mutate(
    Sex = factor(Sex) %>% relevel("M")
  ) %>%
  model.matrix(model,.) %>%
  head()
```

```
##   (Intercept) Height Age I(Age^2) SexF
## 1           1   1.70  23      529    0
## 2           1   1.93  33     1089    0
## 3           1   1.87  30      900    0
## 4           1   1.78  26      676    1
## 5           1   1.82  27      729    0
## 6           1   1.82  30      900    1
```

Zejména v rámci průzkumu dat je zajímavá varianta provést konverzi na faktor v rámci modelu (`formula`) – všimněte si, že může být aplikována i na numerickou proměnnou:

```
Weight ~ Height + factor(Age) + Sex
```

Konverze numerických hodnot do faktoru je poměrně častá zvláště u dotazníkových dat, kde čísla mohou pouze kódovat hodnoty kategoriální proměnné. Dalším případem je například vytvoření fixních efektů pro roky.

Funkce `model.matrix()`, kterou si estimační funkce volá interně tedy vytvoří matici plánu, která je následně použité při samotném odhadu parametrů modelu. Odhadová procedura si liší podle použité estimační funkce. V případě `lm()` je to QR dekompozice (https://en.wikipedia.org/wiki/QR_decomposition), při použití `glm()` jsou to například různé numerické optimalizační algoritmy (defaultně IWLS:https://en.wikipedia.org/wiki/Iteratively_reweighted_least_squares).

18.3.2 Výstupy estimační funkce

```
lm(model, data = oly12) -> est_model

class(est_model)
```

```
## [1] "lm"
```

Funkce `lm()` vrací S3 objekt třídy `lm`. Jiné estimační funkce vrací jiné objekty, které jsou ovšem často potomky `lm`. Objekt třídy `lm` má následující strukturu (viz `?lm`):

```
str(est_model, max.level = 1)
```

```
## List of 13
##  $ coefficients : Named num [1:5] -1.06e+02 9.46e+01 4.10e-01 -4.09e-03 5.59
##   ..- attr(*, "names")= chr [1:5] "(Intercept)" "Height" "Age" "I(Age^2)" ...
##  $ residuals    : Named num [1:9038] -7.71 33.72 -9.15 14.68 0.11 ...
##   ..- attr(*, "names")= chr [1:9038] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:9038] -6926.6 -1161.4 -96.3 -15.7 -221.1 ...
##   ..- attr(*, "names")= chr [1:9038] "(Intercept)" "Height" "Age" "I(Age^2)" ...
##  $ rank         : int 5
##  $ fitted.values: Named num [1:9038] 67.7 91.3 85.2 70.3 79.9 ...
```

```
##   ..- attr(*, "names")= chr [1:9038] "1" "2" "3" "4" ...
##  $ assign       : int [1:5] 0 1 2 3 4
##  $ qr           :List of 5
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 9033
##  $ contrasts    :List of 1
##  $ xlevels      :List of 1
##  $ call         : language lm(formula = model, data = oly12)
##  $ terms        :Classes 'terms', 'formula'  language Weight ~ Height + Age + I(Age^2) + Sex
##   .. ..- attr(*, "variables")= language list(Weight, Height, Age, I(Age^2), Sex)
##   .. ..- attr(*, "factors")= int [1:5, 1:4] 0 1 0 0 0 0 0 1 0 0 ...
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. ..- attr(*, "term.labels")= chr [1:4] "Height" "Age" "I(Age^2)" "Sex"
##   .. ..- attr(*, "order")= int [1:4] 1 1 1 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(Weight, Height, Age, I(Age^2), Sex)
##   .. ..- attr(*, "dataClasses")= Named chr [1:5] "numeric" "numeric" "numeric" "numeric" ...
##   .. .. ..- attr(*, "names")= chr [1:5] "Weight" "Height" "Age" "I(Age^2)" ...
##  $ model        :'data.frame':   9038 obs. of  5 variables:
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language Weight ~ Height + Age + I(Age^2) + Se
##   .. .. ..- attr(*, "variables")= language list(Weight, Height, Age, I(Age^2), Sex)
##   .. .. ..- attr(*, "factors")= int [1:5, 1:4] 0 1 0 0 0 0 0 1 0 0 ...
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..- attr(*, "term.labels")= chr [1:4] "Height" "Age" "I(Age^2)" "Sex"
##   .. .. ..- attr(*, "order")= int [1:4] 1 1 1 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(Weight, Height, Age, I(Age^2), Sex)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:5] "numeric" "numeric" "numeric" "numeric" ...
##   .. .. .. ..- attr(*, "names")= chr [1:5] "Weight" "Height" "Age" "I(Age^2)" ...
##  - attr(*, "class")= chr "lm"
```

Důležité jsou zejména následující položky:

- `coefficients`…pojmenovaný vektor odhadnutých koeficientů

```
est_model$coefficients
```

```
##   (Intercept)        Height           Age      I(Age^2)          SexM
## -1.060127e+02  9.463456e+01  4.096565e-01 -4.094392e-03  5.591553e+00
```

- `residuals`…vektor reziduí – tj. odhadů náhodné složky ε

```
est_model$residuals %>% head
```

```
##          1          2          3          4          5          6
## -7.7137568 33.7165890 -9.1502080 14.6799373  0.1103484 -1.8269273
```

- `fitted.values`…vyrovnané hodnoty

```
est_model$fitted.values %>% head
```

```
##        1        2        3        4        5        6
## 67.71376 91.28341 85.15021 70.32006 79.88965 74.82693
```

- `model`... modifikovanou tabulku dat, která byla skutečně použita při odhadu (tj, například s vypuštěnými nepoužitými proměnnými nebo neúplnými řádky).

```
est_model$fitted.values %>% head
```

```
##        1        2        3        4        5        6
## 67.71376 91.28341 85.15021 70.32006 79.88965 74.82693
```

Za povšimnutí stojí, že objekt obsahuje skutečně pouze odhad – data, odhadnuté koeficienty, rezidua, vyrovnané hodnoty a dodatečné informace. Neobsahuje žádné dodatečné výpočty jako například diagnostické testy.

Pro objekty třídy `lm` existuje metoda generické funkce `print()`. Její výstupy však nejsou nijak zvlášť informativní:

```
print(est_model)
```

```
##
## Call:
## lm(formula = model, data = oly12)
##
## Coefficients:
## (Intercept)       Height          Age     I(Age^2)         SexM
##  -1.060e+02    9.463e+01    4.097e-01   -4.094e-03    5.592e+00
```

Obsahuje jen použité volání a odhadnuté hodnoty koeficientů. Více se dozvíme pomocí funkce `summary()`:

```
summary(est_model)
```

```
##
## Call:
## lm(formula = model, data = oly12)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -56.334  -5.717  -1.336   3.756 135.665
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.060e+02  2.385e+00 -44.444  < 2e-16 ***
## Height       9.463e+01  1.141e+00  82.918  < 2e-16 ***
## Age          4.097e-01  1.096e-01   3.738 0.000187 ***
## I(Age^2)    -4.094e-03  1.819e-03  -2.250 0.024442 *
## SexM         5.592e+00  2.560e-01  21.840  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.13 on 9033 degrees of freedom
## Multiple R-squared:  0.6031, Adjusted R-squared:  0.6029
## F-statistic:  3431 on 4 and 9033 DF,  p-value: < 2.2e-16
```

V tomto případě vidíme kromě odhadů koeficientů i směrodatné chyby a výsledky testů statistické významnosti koeficientů (v tomto případě t-testů). Výstup je také doplněn několika dalšími údaji jako je výsledek F-testu a (adjustovaný) koeficient determinace. Funkce `summary()` poskytuje rychlý pohled na výsledky, nicméně ten může být zavádějící. Funkce totiž vrací pouze standardní výstupy. Nijak například nereflektuje možné zkreslení odhadu směrodatných odchylek kvůli heteroskedasticitě a podobně. Všimněte si, že `summary()` používá méně obvyklé – a přísnější – hvězdičkové značení!

18.3.3 Práce s výsledkem odhadů

V R najdete řadu specializovaných funkcí, které dokážou z výsledného modelu získávat, nebo na jeho základě dopočítávat užitečné informace. Tabulka obsahuje jejich základní přehled:

Funkce	Popis
`coefficients()`, `coefs()`	Vrací odhadnuté koeficienty – obsah `.$coefficients`
`residuals()`, `resid()`	Vrací vektor reziduí – pro objekty `lm` je výstup identický s `.$residuals`
`fitted.values()`	Vrací očekávané hodnoty – obsah `.$fitted.values`
`predict()`	Umožňuje dopočítat predikce pro jiná data, než na kterých byl model odhadnut
`plot()`	Vrací diagnostické grafy vytvořené pomocí základní grafiky (ne v `ggplot2`)
`confint()`	Vrací konfidenční intervaly pro odhady koeficientů
`deviance()`	Vrací MSE – průměr čtverců reziduí
`vcov()`	Vrací odhad kovarianční matice
`logLik()`	Vrací log-likelihood
`AIC()`	Vrací Akaikovo informační krítérium (AIC)
`BIC()`	Vrací Schwarzovo Bayesian kritérium (BIC)
`nobs()`	Vrací počet pozorování

Pro některé účely je vhodné seznámit se i se sofistikovanějšími funkcemi. Pro účely diagnostiky nebo okometrického posouzení kvality vyrovnání může být užitečné výsledky odhadu vizualizovat. Výsledky jsou však schované ve zvláštním objektu a ne v tabulce, kterou potřebujeme pro použití s `ggplot2`. Pro transformaci různých objektů do tabulky existuje v `ggplot2` funkce `fortify()`. Stejnou funkcionalitu poskytuje i balík **broom**. Pro objekty třídy `lm` je ekvivalentem funkce `broom::augment()`:

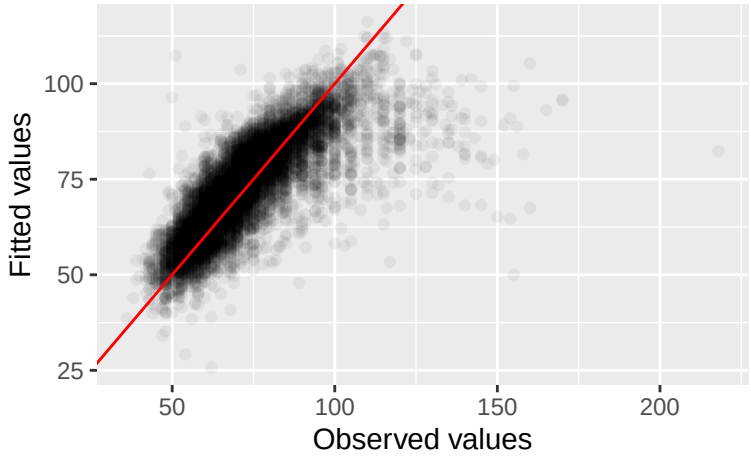
```
library(broom)
augment(est_model) %>% as_tibble %>% print(n = 5)
```

```
## # A tibble: 9,038 x 11
##   Weight Height   Age `I(Age^2)` Sex   .fitted .resid     .hat .sigma    .cooksd
##    <int>  <dbl> <int>   <I<dbl>> <chr>   <dbl>  <dbl>    <dbl>  <dbl>      <dbl>
## 1     60   1.7     23        529 M        67.7 -7.71  0.000453   10.1    5.26e-5
## 2    125   1.93    33       1089 M        91.3 33.7   0.000474   10.1    1.05e-3
## 3     76   1.87    30        900 M        85.2 -9.15  0.000299   10.1    4.89e-5
## 4     85   1.78    26        676 F        70.3 14.7   0.000336   10.1    1.41e-4
## 5     80   1.82    27        729 M        79.9  0.110 0.000235   10.1    5.58e-9
## # ... with 9,033 more rows, and 1 more variable: .std.resid <dbl>
```

Pozor, **broom** navrací `tibble`. Proto je užitečné výstup do `tibble` konvertovat. Výsledek je již jednoduše použitelný pro vizualizaci pomocí `ggplot2`. Následující obrázek ukazuje korelaci pozorovaných (*observed*) a vyrovnaných (*fitted*) hodnot. Do obrázku je přidána pomocí `geom_abline()` i osa kvadrantu na která se pozorované a vyrovnané hodnoty rovnají.

```
est_model %>%
  augment() %>%
  ggplot(
```

```
    aes(x = Weight, y = .fitted)
  ) +
  geom_point(alpha = 0.05) +
  geom_abline(slope = 1, intercept = 0, color = "red") +
  scale_x_continuous(name = "Observed values") +
  scale_y_continuous(name = "Fitted values")
```



Balík **broom** obsahuje i další užitečné funkce: `tidy()` a `glance()`. Funkce `tidy()` vrací tabulku s odhady parametrů, směrodatných odchylek, testových statistik a p-hodnot:

```
tidy(est_model)
```

```
## # A tibble: 5 x 5
##   term          estimate std.error statistic   p.value
##   <chr>            <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept) -106.        2.39       -44.4  0
## 2 Height        94.6       1.14        82.9  0
## 3 Age            0.410     0.110        3.74 1.87e-  4
## 4 I(Age^2)      -0.00409   0.00182     -2.25 2.44e-  2
## 5 SexM           5.59      0.256       21.8  4.38e-103
```

Funkce `glance()` vrací opět tabulku. Jejím obsahem ovšem nejsou údaje týkající se jednotlivých proměnných, ale modelu jako celku:

```
glance(est_model)
```

```
## # A tibble: 1 x 12
##   r.squared adj.r.squared sigma statistic p.value    df  logLik    AIC    BIC
##       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>   <dbl>  <dbl>  <dbl>
## 1     0.603         0.603  10.1     3431.       0     4 -33746. 67504. 67546.
## # ... with 3 more variables: deviance <dbl>, df.residual <int>, nobs <int>
```

Tyto funkce jsou mimořádně užitečné při strojovém zpracovávání velkého množství odhadnutých modelů. Funkce z balíku **broom** přitom neslouží jenom pro převádění objektů `lm` do tabulky. Jejich záběr je mnohem obecnější a obsahují metody pro mnohem více tříd objektů.

Zvláštní pozornost se vyplatí věnovat funkci `predict()`. Ty v základní konfiguraci vrací prostě vyrovnané hodnoty:

```
all.equal(predict(est_model),fitted(est_model))
```

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

Při mnoha aplikacích nás spíše zajímá schopnost modelu predikovat out-of-sample. Tedy predikovat hodnoty vysvětlované proměnné na jiném vzorku dat, než který byl použit k odhadu parametrů. I pro tento účel se hodí funkce `predict()`.

Nejprve si tabulku `oly12` rozdělíme na dvě dílčí tabulky `oly12_a` a `oly12_b`:

```
idx <- sample(nrow(oly12), round(nrow(oly12)/2))

oly12[idx,]  -> oly12_a
oly12[-idx,]  -> oly12_b
```

Pro odhad parametrů použijeme tabulku `oly12_a`:

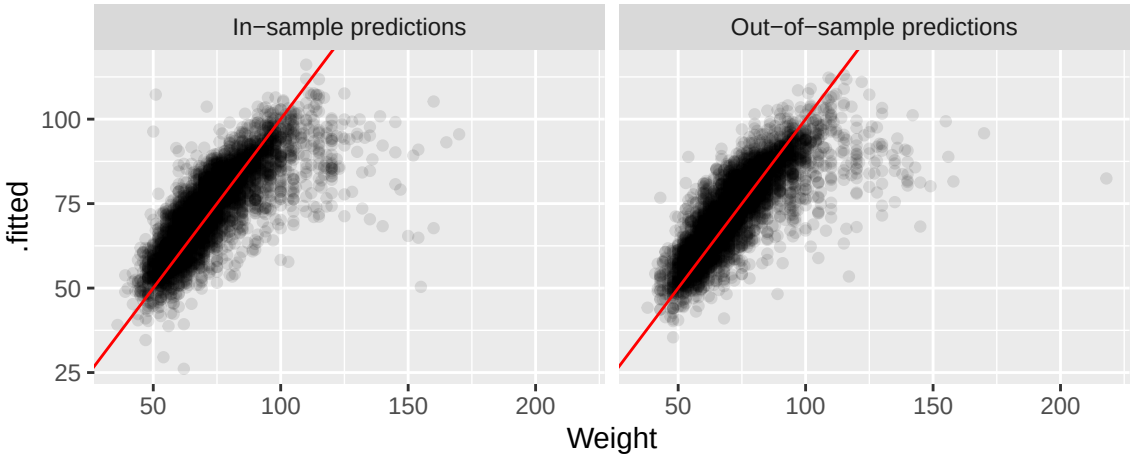
```
oly12_a %>%
    lm(model, data=.) -> est_model_a
```

Na základě odhadnutého modelu vypočítáme očekávané hodnoty `Weight` pro pozorování ze vzorku `oly12_b`:

```
predict(est_model_a, newdata = oly12_b) -> prediction_oly12_b
```

Výsledkem je vektor predikovaných hodnot. Podívejme se, jak si model vedl při predikci out-of-sample:

```
oly12_b %>%
    bind_cols(.fitted = prediction_oly12_b) %>%
    mutate(
        sample = "B"
    ) -> oly12_b

est_model_a %>%
  augment() %>%
  mutate(
    sample = "A"
  ) %>%
  bind_rows(oly12_b) %>%
     ggplot(
        aes(
            x = Weight,
            y = .fitted
        )
    ) +
    geom_point(
        alpha = 0.1
    ) +
  geom_abline(slope = 1, intercept = 0, color = "red") +
  facet_wrap("sample", labeller = as_labeller(
    c("A"="In-sample predictions","B"="Out-of-sample predictions")
    ))
```



Obrázky zhruba ukazují, že odhadnutý model si vede dobře i v out-of-sample predikcích. tedy predikcích vypočítaných na jiných datech, než jaké byly použity pro odhad modelu. Vzhledem k tomu, že vzorek byl rozdělen náhodně to asi není žádné velké překvapení.

18.3.4 Další estimační funkce a kde je najít

Následující tabulka představuje pár estimačních funkcí dostupných v R.

Model/estimátor	Funkce	Poznámka
Logit/probit	`glm()`	Modely diskrétní volby
Poisson	`glm()`	Count data
Poisson pseudo-ML (PPML)	`glm()`	Např. gravitační modely
Lineární panelová data	`plm::plm()`	pooled, FE, RE,...
Arellano-Bond	`plm::pgmmm()`	Dynamické panely
Common Correlated Effects	`plm::pcce()`	Např. gravitační modely
Poisson, logit, ordered probit,... (na panelu)	`pglm::pglm()`	
Tobit	`AER::tobit()`	
2SLS	`AER::ivreg()`	
Heckit (Tobit 2)	`sampleSelection::heckit()`	

...a mnoho, mnoho dalších.

18.4 Diagnostika

18.4.1 Vlivná pozorování

Populárním způsobem, jak najít pozorování, která významně ovlivňují výsledky modelu, je použití leave-one-out strategií. V nich se sleduje efekt vyloučení jednoho pozorování na odhady koeficientů, popřípadě na vyrovnaná pozorování. V R jsou implementovány v `influence.measures()`:

```
influence.measures(est_model) -> infl_obs

infl_obs %>% class
```

```
## [1] "infl"
```

`influence.measures()` najednou volá celou řadu diagnostických testů – mj. `dfbetas()` (vliv jednoho pozorování na odhadnuté koeficienty) a `dffits()` (vliv jednoho pozorování na vyrovnané hodnoty).

Výstupní objekt třídy `infl` obsahuje dva prvky (matice). `.$infmat` obsahuje hodnoty jednotlivých statistik a `.$is.inf` rozhodnutí (o podezření), zda se jedná o vlivné pozorování.

Viz obrázek:

```
augment(est_model) %>%
    bind_cols(.,as.data.frame(infl_obs$is.inf)) %>%
    ggplot(
        aes(x=Weight,
            y=.fitted)
    ) +
    geom_point(
        aes(
            color = dffit
            ),
        alpha = 0.2
    ) +
    xlab("Observed values") +
    ylab("Fitted values") +
    scale_color_discrete(name="Influential\nobservation\n(DFFIT)")
```

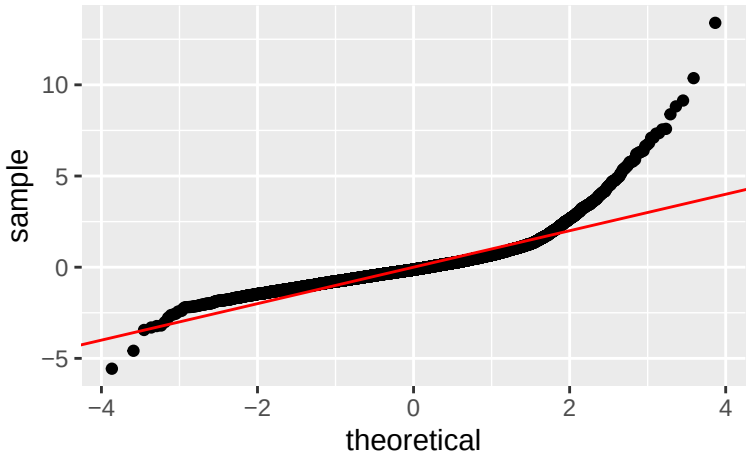
Hodnoty, u kterých se predikce velmi odlišuje od pozorovaných hodnot jsou podle statistiky DFFIT jsou navrženy jako vlivné.

Tyto výsledky naznačují, že se v datech může dít něco zvláštního, nebo že nemáme k dispozici dostatečný počet pozorování z určité části prostoru – v tomto případě například u sportovců s vyšší hmotností.

18.4.2 Normalita reziduí

Prvním způsobem, jak ověřit normalitu reziduí je pohledem na jejich rozdělení. Pomoci může například Q-Q plot, který porovnává kvantily normálního rozdělení a standardizovaných reziduí. V případě, že je rozdělení reziduí normální, by pozorování měla ležet na ose kvadrantu:

```
augment(est_model) %>%
    ggplot(
        aes(sample = .std.resid)
    ) +
    geom_qq() +
    geom_abline(
        slope = 1,
        intercept = 0,
        color = "red"
        )
```



Ani jeden okometrický test pro normalitu příliš nehovoří. Nicméně je potřeba nepodlehnout zdání. R poskytuje v balíku `normtest` celou baterii statistických testů.

Například můžeme zkusit otestovat normalitu pomocí Jarque-Bera testu:

```
library(normtest)

residuals(est_model) %>%
    jb.norm.test() -> jb

class(jb)
```

```
## [1] "htest"
```

```
print(jb)
```

```
##
##  Jarque-Bera test for normality
##
## data:  .
## JB = 82199, p-value < 2.2e-16
```

Výstupem `jb.norm.test()` je objekt typu `htest`. Tato třída je oblíbená pro výsledky statistických testů. Pro tuto třídu existují metody v balíku `broom`, např.:

```
tidy(jb)
```

```
## # A tibble: 1 x 3
##   statistic p.value method
##       <dbl>   <dbl> <chr>
## 1    82199.       0 Jarque-Bera test for normality
```

Nulovou hypotézou Jarque-Bera testu je normalita, kterou tento test zamítá.

18.4.3 Testy linearity

Z testů linearity je dostupný například Rainbow test, který srovnává kvalitu vyrovnání v modelech odhadnutých "uprostřed" pozorování a na celém vzorku.

V R je Rainbow test dostupný v balíku `lmtest` ve funkci `raintest`:

```
raintest(formula, fraction = 0.5, order.by = NULL, center = NULL,
   data=list())
```

Parametr `fraction` udává velikost samplu pro odhad modelu "uprostřed". V parametru `order.by` se udává proměnná, podél které se mají pozorování řadit.

Parametr `formula` může být naplněn i odhadnutým `lm` modelem. V tom případě není nutné zabývat se parametrem `data`.

```
library(lmtest)
raintest(model, order.by = ~ Height, data = oly12)
```

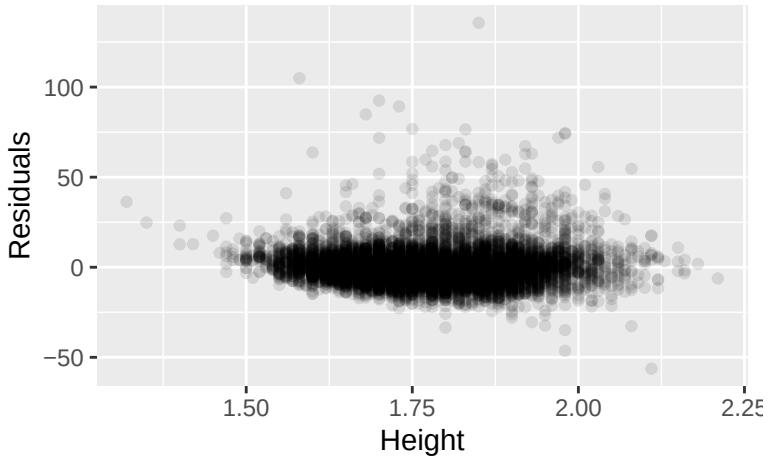
```
##
##  Rainbow test
##
## data:  model
## Rain = 1.1287, df1 = 4519, df2 = 4514, p-value = 2.397e-05
```

Rain test v tomto případě zamítá nulovou hypotézu. To znamená, že fit na neomezeném vzorku je významně horší, než na podmnožině "prostředních" pozorování. Tento výsledek indikuje možný problém s linearitou.

18.4.4 Heteroskedasticita a odhad robustních chyb

Pokud je porušena homoskedasticita – tzn. konstantní rozptyl náhodné složky – potom není OLS estimátor BLUE (Best linear unbiased estimator), nicméně odhady parametrů jsou stále nevychýlené (unbiased). To bohužel neplatí pro odhady jejich rozptylu – ty naopak vychýlené jsou a v důsledku jsou nedůvěryhodné i testy statistické významnosti parametrů.

```
est_model %>%
    augment() %>%
    arrange(Height) %>%
    ggplot(
        aes(
            x = Height,
            y = .resid
            )
    ) +
    geom_point(
        alpha = 0.1
    ) +
    ylab("Residuals")
```



V reálném světě je podmínka homoskedasticity porušena velmi často. Jestli tomu tak je můžeme testovat pomocí mnoha testů. Jako příklad mohou sloužit příbuzné testy `bptest()` z balíku `lmtest` a `ncvTest` z balíku `car`:

```
lmtest::bptest(est_model)
```

```
##
##  studentized Breusch-Pagan test
##
## data:  est_model
## BP = 73.288, df = 4, p-value = 4.585e-15
```

```
car::ncvTest(est_model)
```

```
## Non-constant Variance Score Test
## Variance formula: ~ fitted.values
## Chisquare = 535.0632, Df = 1, p = < 2.22e-16
```

Oba testy testují nulovou hypotézu o homoskedasticitě oba ji suverénně zamítají – p-hodnoty obou testů jsou v podstatě nerozlišitelné od nuly.

Odhady směrodatných chyb lze korigovat použitím tzv. HC *(heteroskedasticity consistent)* estimátorů kovarianční matice. Ty jsou implementovány například v balících `car` (`car::hccm`), ale zejména v balíku `sandwich`.

Balík `sandwich` obsahuje celou řadu nástrojů pro odhad kovarianční matice pro průřezová i panelová data. Bližší pozornost budeme věnovat dvěma. Základní funkci `vcovHC()` a také relativně nově implementované funkci pro výpočet klastrovaných robustních chyb `vcovCL()`.

Obě dvě funkce mají podobnou syntax:

```
vcovHC(x,
  type = c("HC3", "const", "HC", "HC0", "HC1", "HC2", "HC4", "HC4m", "HC5"),
  omega = NULL, sandwich = TRUE, ...)
```

```
vcovCL(x, cluster = NULL, type = NULL, sandwich = TRUE, fix = FALSE, ...)
```

Vstupem funkce je odhadnutý model (`x`). V parametru `type` se potom specifikuje způsob korekce heteroskedasticity (viz `?vcovHC()`). V případě funkce `vcovCL()` je navíc nutné specifikovat proměnnou, nebo v případě více dimenzí proměnné, podle kterých se mají standardní chyby klastrovat. Klastrování se typicky používá v

situaci, kdy určitá pozorování na sobě nejsou nezávislá. Klasický příklad je prospěch dětí z jedné třídy. Podstata dat sportovců naznačuje, že i v tomto případě je klastrování namístě. Jako specifické skupiny můžeme chápat sportovce věnující se jedné disciplině. Po `vcovCL()` tedy bude chtít spočítat kovarianční maticí klastrovanou podél proměnné `Sport`. Tu musíme dodat jako samostatný vektor. V objektu totiž není zahrnuta:

```
library(sandwich)
vcovCL(est_model, cluster = oly12$Sport)
```

```
##                (Intercept)       Height          Age      I(Age^2)         SexM
## (Intercept) 144.744530958 -87.73269188  0.544857828 -7.405519e-03  9.080987791
## Height      -87.732691881  57.87470724 -0.914600493  1.437195e-02 -6.275686441
## Age           0.544857828  -0.91460049  0.079432487 -1.394717e-03  0.137949915
## I(Age^2)     -0.007405519   0.01437195 -0.001394717  2.566568e-05 -0.002251259
## SexM          9.080987791  -6.27568644  0.137949915 -2.251259e-03  0.979011051
```

Výsledkem je odhad kovarianční matice. Zpravidla nás ovšem více zajímá odhad významnosti parametrů – provedení statistických testů s použitím robustní kovarianční matice jako vstupu. Pro tento účel použijeme funkci `coeftest()` z balíku `lmtest`:

```
coeftest(x, vcov. = NULL, df = NULL, ...)
```

Prvním vstupem funkce `coeftest()` je odhadnutý model. (Funkce má implementovány metody pro modely třídy `lm` a `glm`.) Druhým parametrem je kovarianční matice nebo funkce, která se má použít pro její výpočet. Pokud je tento parametr ponechán prázdný, potom `coeftest()` použije nekorigovanou kovarianční matici:

```
library(lmtest)
coeftest(est_model)
```

```
##
## t test of coefficients:
##
##                Estimate  Std. Error  t value  Pr(>|t|)
## (Intercept) -1.0601e+02  2.3853e+00 -44.4438 < 2.2e-16 ***
## Height       9.4635e+01  1.1413e+00  82.9183 < 2.2e-16 ***
## Age          4.0966e-01  1.0960e-01   3.7378 0.0001868 ***
## I(Age^2)    -4.0944e-03  1.8193e-03  -2.2505 0.0244416 *
## SexM         5.5916e+00  2.5603e-01  21.8397 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Zadaní jména funkce jako parametru `vcov.` je vhodné tehdy, pokud nepotřebujete provádět žádné další nastavení parametrů výpočtu kovarianční matice:

```
coeftest(est_model, vcov. = vcovHC)
```

```
##
## t test of coefficients:
##
##                Estimate  Std. Error  t value  Pr(>|t|)
## (Intercept) -1.0601e+02  2.4526e+00 -43.2250 < 2.2e-16 ***
## Height       9.4635e+01  1.2120e+00  78.0841 < 2.2e-16 ***
## Age          4.0966e-01  9.2268e-02   4.4398 9.109e-06 ***
## I(Age^2)    -4.0944e-03  1.5164e-03  -2.7002  0.006944 **
```

```
## SexM         5.5916e+00  2.4404e-01  22.9123 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

V případě klastrovaných chyb však potřebuje takové nastavení provést – minimálně potřebujeme zadat vektor podél kterého se má klastrovat. Do `vcov.` v tomto případě potřebujeme zadat celou kovarianční matici – tedy výstup funkce `vcovCL`:

```
coeftest(est_model, vcov. = vcovCL(est_model, cluster = oly12$Sport))
```

```
##
## t test of coefficients:
##
##                Estimate  Std. Error t value  Pr(>|t|)
## (Intercept) -1.0601e+02  1.2031e+01 -8.8116 < 2.2e-16 ***
## Height       9.4635e+01  7.6075e+00 12.4396 < 2.2e-16 ***
## Age          4.0966e-01  2.8184e-01  1.4535    0.1461
## I(Age^2)    -4.0944e-03  5.0661e-03 -0.8082    0.4190
## SexM         5.5916e+00  9.8945e-01  5.6512 1.642e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Klastrovat lze podél více proměnných. V tomto případě je potřeba do parametru `cluster` vložit list vektorů nebo `data.frame`. Nyní zkusíme cvičně klastrovat nejen podle sportovního odvětví, ale i podle pohlaví:

```
coeftest(est_model, vcov. = vcovCL(est_model, cluster = list(oly12$Sport, oly12$Sex)))
```

```
##
## t test of coefficients:
##
##                Estimate  Std. Error t value  Pr(>|t|)
## (Intercept) -1.0601e+02  2.6150e+01 -4.0540 5.076e-05 ***
## Height       9.4635e+01  1.2569e+01  7.5295 5.584e-14 ***
## Age          4.0966e-01  3.3167e-01  1.2351    0.2168
## I(Age^2)    -4.0944e-03  4.2600e-03 -0.9611    0.3365
## SexM         5.5916e+00  1.1367e+00  4.9189 8.857e-07 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Výstupem funkce `coeftest()` je objekt speciální třídy `coeftest`. Pokud ho chceme převést na tabulku můžeme použít funkci `broom::tidy()`:

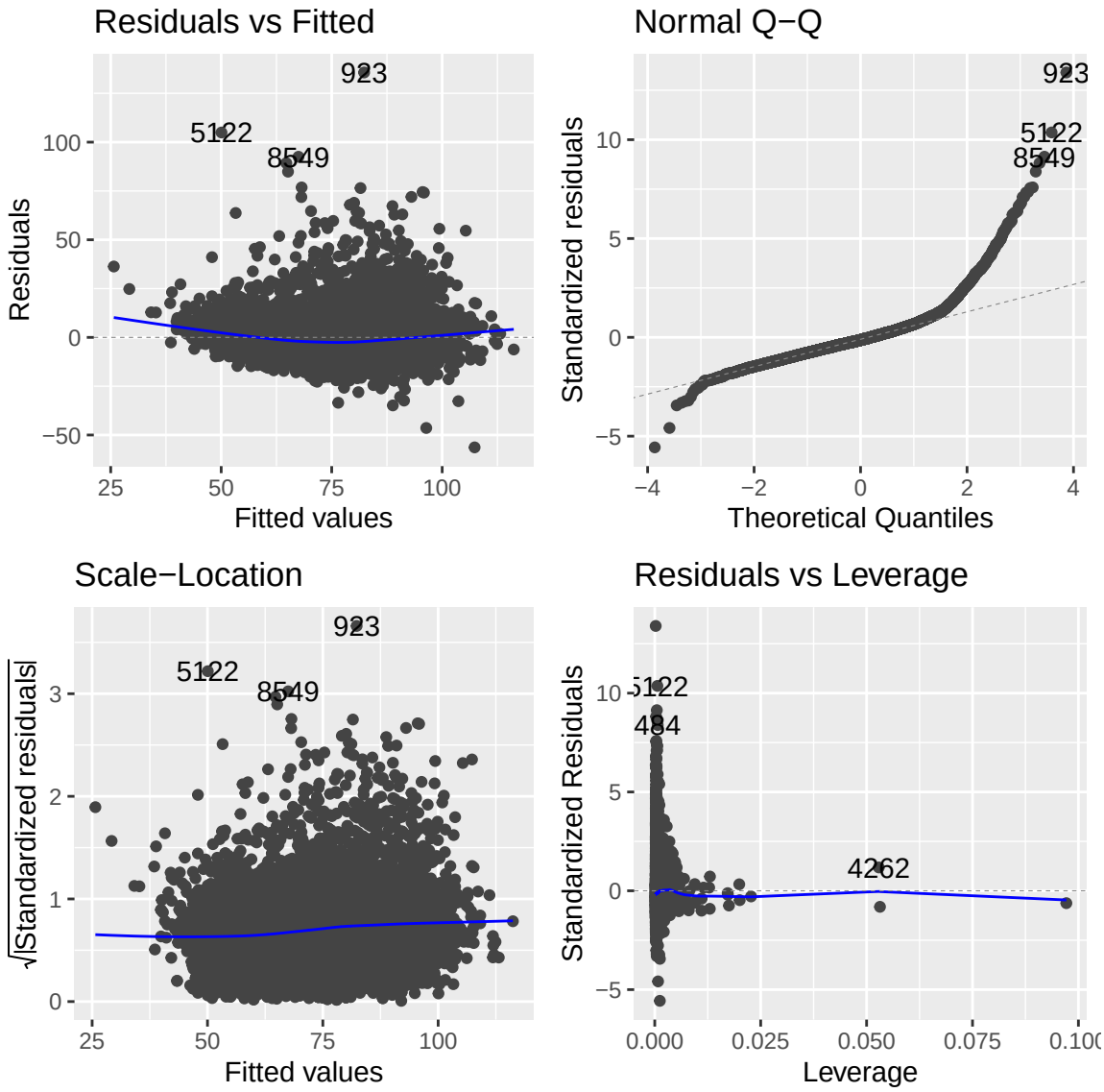
```
coeftest(est_model, vcov. = vcovCL(est_model, cluster = list(oly12$Sport, oly12$Sex))) %>% tidy()
```

```
## # A tibble: 5 x 5
##   term          estimate std.error statistic  p.value
##   <chr>            <dbl>     <dbl>     <dbl>    <dbl>
## 1 (Intercept) -106.       26.1        -4.05  5.08e- 5
## 2 Height        94.6      12.6         7.53  5.58e-14
## 3 Age            0.410     0.332       1.24  2.17e- 1
## 4 I(Age^2)      -0.00409   0.00426    -0.961 3.37e- 1
## 5 SexM           5.59      1.14        4.92  8.86e- 7
```

18.4.5 Dignostika graficky

V mnoha případech je užitečné se na diagnostiku podívat prostřednictvím vizualizací. Tuto možnost nabízí například funkce `plot()`, která má pro tento účel implementovanou metodu. Její fungování můžete vyzkoušet zadáním `plot(est_model)`. Tento postup logicky pracuje se základní grafikou. Existují však i řešení, které podobně uživatelsky přítulně vykreslí diagnostické grafy s pomocí `ggplot2`. Jedním z nich je funkce `ggplot2::autoplot()`, která využívá služeb balíku **ggfortify**:

```
library(ggfortify)
autoplot(est_model)
```



18.5 Odhad více modelů

Často můžeme chtít odhadnout model v různých specifikacích, nebo na různých vzorcích dat.

18.5.1 Odhad různých specifikací na stejných datech

Častá úloha je odhad různých modelových specifikací na stejném vzorku pozorování. Pro tyto účely doporučuji následující postup:

1. Vytvořit si `list` se všemi rovnicemi (specifikacemi)
2. Odhadnout model pro všechny specifikace pomocí funkce `map()` z balíku **purrr**

Nejprve vytvoříme různé modelové specifikace s pomocí funkce `update()`:

```
Models <- list(
    # Baseline model
    model,
    # Přidána interakce Sex*Height
    model %>% update(.~. + Sex*Height),
    # Přidán fixní efekt na zemi původu atleta
    model %>% update(.~. + factor(Country))
)
```

Nyní pomocí `purrr::map()` zavoláme `lm()` na všechny prvky listu `Models` – tedy na všechny modelové specifikace:

```
Spec_models <- purrr::map(Models, lm, data = VGAMdata::oly12)
Spec_models <- purrr::map(Models, function(x) lm(x, data = VGAMdata::oly12))
Spec_models <- purrr::map(Models, ~lm(., data = VGAMdata::oly12))
```

Všechny tři výše uvedené způsoby použití `map()` jsou v pořádku. Z nějakého důvodu však balík **stargazer**, který budeme používat pro tvorbu výstupních tabulek, neumí pracovat se seznamem odhadnutých modelů, který byl vytvořen prvním způsobem.

Výsledné objekty jsou nyní uložené jako položky listu `Spec_models`. K nim můžeme přistupovat pomocí subsetování:

```
Spec_models[[2]] %>% summary
```

```
##
## Call:
## lm(formula = ., data = VGAMdata::oly12)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -59.122  -5.646  -1.235   3.622 135.458
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -78.890968   3.402356 -23.187  < 2e-16 ***
## Height       78.901728   1.814233  43.490  < 2e-16 ***
## Age           0.392204   0.108875   3.602 0.000317 ***
## I(Age^2)     -0.003842   0.001807  -2.126 0.033527 *
## SexM        -39.474467   4.065283  -9.710  < 2e-16 ***
## Height:SexM  25.695023   2.313338  11.107  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.06 on 9032 degrees of freedom
##   (1346 observations deleted due to missingness)
## Multiple R-squared:  0.6084, Adjusted R-squared:  0.6082
## F-statistic:  2807 on 5 and 9032 DF,  p-value: < 2.2e-16
```

18.5.2 Odhad modelu na dílčích datech

Základní model můžeme odhadnout na různých vzorcích dat – v tomto případě pro různé sporty (proměnná `Sport`). Pro podobnou operaci existuje více přístupů. Historicky byla pro takové operace určena funkce `dplyr::do()`. Ta však bude v budoucnosti v balíku *dplyr* nahrazena a není ještě zcela jasné jak.

Dobrou variantou je postup analogický k odhadu více modelů na stejných datech. Nyní chceme odhadnout stejný model na různých datech. V předchozím případě jsme přes list rovnic iterovali pomocí funkce `map()`. Nyní bychom chtěli iterovat přes list tibblů. Ten vytvoříme pomocí funkce `split()` z base R.

Vstupy `split()` bude úplný tibble a pravostranná formule se jménem vektoru, podle kterého se má tento tibble rozdělit na menší tibblíky (bylo by možné použít přímo i tento vektor ve tvaru `.$Sport`). Typicky je tento vektor přímo součástí vstupního tibble. To není problém – můžeme využí placeholder `.`. Dejme tomu, že chceme odhadnout parametry pro každý sport zvlášť:

```
Sport_models <- oly12 %>%
    split(~ Sport) %>%
    map(
        # Zde si pomáháme anonymní funkcí, která nám pomůže doručit data
        # do správného parametru funkce lm()
        function(x) lm(Models[[1]], data = x)
    )
```

Výsledná proměnná je list, který obsahuje odhadnuté modely:

```
Sport_models[1:3] %>% print()
```

```
## $Archery
##
## Call:
## lm(formula = Models[[1]], data = x)
##
## Coefficients:
## (Intercept)       Height          Age     I(Age^2)         SexM
##  -45.146045    56.714558     0.459166     0.002153    10.191336
##
##
## $Athletics
##
## Call:
## lm(formula = Models[[1]], data = x)
##
## Coefficients:
## (Intercept)       Height          Age     I(Age^2)         SexM
##  -145.12558    123.04350     -0.40467      0.01024      2.15605
##
##
## $Badminton
##
## Call:
## lm(formula = Models[[1]], data = x)
##
## Coefficients:
## (Intercept)       Height          Age     I(Age^2)         SexM
##  -70.285717    75.702484     0.093450     0.001788     3.787492
```

S modely uloženými v seznamu. Následující kód například opět za využívá `map()`. Pomocí `broom::glance()` vytvoří tabulku modelů seřazenou sestupně podle adjustovaného koeficientu determinace:

```
Sport_models %>%
  # Varinata map_dfr() je ekvivalentní k map() %>% bind_rows()
    map_dfr(glance) %>%
    arrange(desc(adj.r.squared)) %>%
    print()
```

```
## # A tibble: 26 x 12
##    r.squared adj.r.squared sigma statistic   p.value    df logLik   AIC   BIC
##        <dbl>         <dbl> <dbl>     <dbl>     <dbl> <dbl>  <dbl> <dbl> <dbl>
##  1     0.879         0.874  2.98      179. 4.32e- 44     4  -256.  524.  540.
##  2     0.878         0.870  3.81      110. 3.90e- 27     4  -179.  371.  384.
##  3     0.874         0.869  4.56      153. 9.52e- 39     4  -271.  553.  568.
##  4     0.850         0.848  6.01      423. 9.22e-122     4  -974. 1960. 1983.
##  5     0.837         0.835  4.29      485. 2.43e-147     4 -1099. 2209. 2233.
##  6     0.833         0.830  6.50      315. 6.52e- 97     4  -846. 1705. 1726.
##  7     0.819         0.815  4.49      171. 4.96e- 55     4  -453.  918.  936.
##  8     0.815         0.813  5.84      557. 5.13e-184     4 -1627. 3267. 3292.
##  9     0.806         0.805  5.04      887. 2.58e-302     4 -2605. 5223. 5251.
## 10     0.797         0.794  5.08      293. 3.55e-102     4  -923. 1858. 1880.
## # ... with 16 more rows, and 3 more variables: deviance <dbl>,
## #   df.residual <int>, nobs <int>
```

(Pozn. takové řazení je užitečné jenom ve velmi omezeném počtu případů.)

18.6 Tvorba pěkně formátovaných výsledků s balíkem `stargazer`

Pro porovnání výsledků odhadů je užitečné srovnat si je vedle sebe do tabulky, jakou znáte z empirických článků. Tento úkol umí splnit balík Marka Hlaváče `stargazer`.

```
library(stargazer)
```

Stargazer dokáže vytvářet a do ASCII, HTML a LaTeXu exportovat:

- regresní tabulky
- tabulky popisných statistik
- exportovat vstupní `data.frame` bez další transformace (to je užitečné například pro export korelačních tabulek)

Balík `stargazer` obsahuje jedinou funkci `stargazer()`, která má velké množství parametrů – srozumitelný tutoriál najdete například zde: http://jakeruss.com/cheatsheets/stargazer.html

Vstupem do funkce může být:

- jeden nebo více odhadnutých modelů různých tříd (viz seznam v helpu)
- data.framy, vektory nebo matice pro tvorbu popisných statistik nebo přímý export

Vstupy mohou být vloženy jednotlivě nebo jako `list`, popřípadě `list` v `listu`.

18.6.1 Tabulka s popisnými statistikami

Tvorbu tabulky s popisnými statistikami je možné ilustrovat na minimalistickém příkladu. Vstupem je v tomto případě `data.frame`. Klíčovým parametrem je `summary = TRUE`, který říká `stargazeru`, že má vytvořit popisné statistiky. V případě `FALSE` by exportoval vstupní tabulku tak jak je.

Parametr `type` udává formát, do kterého `stargazer` exportuje. Možná nastavení jsou `html`, `text` a `latex` (default). Zde nevyužitým parametrem je `out` – do něj se vkládá jméno výstupního souboru.

```
stargazer(
    VGAMdata::oly12,
    summary = TRUE,
    type = "latex"
)
```

% Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fas.harvard.edu
% Date and time: Pá, zář 10, 2021 - 16:05:22

Table 18.3:

Statistic	N	Mean	St. Dev.	Min	Pctl(25)	Pctl(75)	Max
Age	10,384	26.069	5.441	13	22	29	71
Height	9,823	1.769	0.113	1.320	1.690	1.850	2.210
Weight	9,104	72.853	16.067	36.000	61.000	81.000	218.000
Gold	10,384	0.017	0.136	0	0	0	2
Silver	10,384	0.017	0.133	0	0	0	2
Bronze	10,384	0.018	0.136	0	0	0	2
Total	10,384	0.052	0.250	0	0	0	5

V základním nastavení `stargazer` vrací počet pozorování, průměr, směrodatnou odchylku, minimum a maximum. Výčet statistik, stejná jako formátování tabulky, lze měnit pomocí parametrů funkce `stargazer()`.

Pozor, stargazer neumí zpracovat tibble! Tabulka `oly12` je tibble:

```
stargazer(
    oly12,
    summary = TRUE,
    type = "latex"
)
```

Výstup bude vypadat následovně:

% Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fas.harvard.edu
% Date and time: Pá, zář 10, 2021 - 16:05:22

Table 18.4:

Statistic	N	Mean	St. Dev.	Min	Pctl(25)	Pctl(75)	Max

Pro správnou funkci je potřeba tabulku odtibblovat – například pomocí `as.data.frame(oly12)`.

18.6.2 Tabulka s regresními modely

Pokud jsou vstupem stagazeru odhadnuté modely nebo jejich list, potom `stargazer` automaticky vytvoří regresní tabulku. Pro ilustraci můžeme použít list `Spec_models`. Pro vytvoření regresní tabulky ze `Spec_models` je vhodné použít další z mnoha parametrů funkce `stargazer()`. Jeden z modelů ve `Spec_models` obsahuje

velké množství fixních efektů. Jejich odhady nás v podstatě nezajímají, ale jejich výpis by tabulku zvětšil do obrovských rozměrů. Použijeme proto parametr `omit`, který umožňuje potlačit výpis parametrů, jejichž jména odpovídají zadanému regulárnímu výrazu.

V parametru `omit.labels` je možné nastavit jméno, které se má použít pro signalizaci přítomnosti nevytisknutých proměnných v regresi.

```
stargazer(
    Spec_models,
    type = pandoc.output.format(),
    omit = "factor\\(Country\\)",
    omit.labels = "Country FE"
)
```

% Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fas.harvard.edu
% Date and time: Pá, zář 10, 2021 - 16:05:22

Table 18.5:

	Dependent variable:		
	Weight		
	(1)	(2)	(3)
Height	94.635***	78.902***	94.887***
	(1.141)	(1.814)	(1.234)
Age	0.410***	0.392***	0.491***
	(0.110)	(0.109)	(0.110)
I(Age^2)	−0.004**	−0.004**	−0.005***
	(0.002)	(0.002)	(0.002)
SexM	5.592***	−39.474***	5.502***
	(0.256)	(4.065)	(0.266)
Height:SexM		25.695***	
		(2.313)	
Constant	−106.013***	−78.891***	−112.691***
	(2.385)	(3.402)	(6.240)
Country FE	No	No	No
Observations	9,038	9,038	9,038
R^2	0.603	0.608	0.628
Adjusted R^2	0.603	0.608	0.620
Residual Std. Error	10.126 (df = 9033)	10.058 (df = 9032)	9.909 (df = 8837)
F Statistic	3,430.928*** (df = 4; 9033)	2,806.601*** (df = 5; 9032)	74.631*** (df = 200; 8837)
Note:			*p<0.1; **p<0.05; ***p<0.01

Za povšimnutí stojí, že `stargazer` defaultně používá jinou hvězdičkovou konvenci, než je tomu ve zbytku R. Ve v tabulce vytvořené stargazerem vidíte stejnou konvenci, na kterou jste zvyklí z Gretlu. R defaultně hvězdičkami více šetří – viz `summary` výše.

> Stargazer poskytuje extrémně užitečnou funkcionalitu, nicméně celkově se jedná o dost nemoderní balík s velmi složitým kódem, který je náchylný k chybám a divnému chování.

Výše ukázané tabulky neobsahují robustní chyby. Protože z testů víme, že rezidua jsou heteroskedastická, musíme je do tabulek dostat.

V defaultním nastavení `stargazer` získává odhad standardních chyb vnitřním voláním `summary`. Uživatel však může vložit vlastní odhady robustních chyb do argumentu `se`. Ten očekává `list` numerických vektorů. (V nápovědě k parametru `se` je v aktuální verzi chyba.)

Postup je následující:

1. Nejprve vytvoříme funkci `get.se()`, která odhadne robustní chyby a vrátí je jako vektor.
2. Pomocí `purrr::map()` aplikujeme funkci `get.se()` na všechny modely v listu `Spec_models`.
3. Výsledný seznam vektorů vložíme do parametru `se` funkce `stargazer()`.

Zároveň totéž provedeme pro p-hodnoty.

```
get.se <- function(x) coeftest(x, vcov. = vcovHC) %>%
    tidy %>%
    pull(std.error)

get.pval <- function(x) coeftest(x, vcov. = vcovHC) %>%
    tidy %>%
    pull(p.value)

se.list <- purrr::map(Spec_models, get.se)
pval.list <- purrr::map(Spec_models, get.pval)

stargazer(
    Spec_models,
    type = pandoc.output.format(),
    omit = "Country",
    omit.labels = "Country FE",
    se = se.list,
    p = pval.list
)
```

% Table created by stargazer v.5.2.2 by Marek Hlavac, Harvard University. E-mail: hlavac at fas.harvard.edu
% Date and time: Pá, zář 10, 2021 - 16:05:25

Pokud bychom chtěli vytvořit regresní tabulku z modelů odhadnutých pro jednotlivé sporty, můžeme využít toho, že `data.frame` je ve své podstatě `list`:

```
stargazer(
    Sport_models$est_model,
    type = "html"
)
```

18.7 Alternativy k balíku *stargazer*

Table 18.6:

	Dependent variable:		
		Weight	
	(1)	(2)	(3)
Height	94.635***	78.902***	94.887
	(1.212)	(1.682)	
Age	0.410***	0.392***	0.491
	(0.092)	(0.091)	
I(Age^2)	−0.004***	−0.004**	−0.005
	(0.002)	(0.002)	
SexM	5.592***	−39.474***	5.502
	(0.244)	(4.125)	
Height:SexM		25.695***	
		(2.352)	
Constant	−106.013***	−78.891***	−112.691
	(2.453)	(3.149)	
Country FE	No	No	No
Observations	9,038	9,038	9,038
R^2	0.603	0.608	0.628
Adjusted R^2	0.603	0.608	0.620
Residual Std. Error	10.126 (df = 9033)	10.058 (df = 9032)	9.909 (df = 8837)
F Statistic	3,430.928*** (df = 4; 9033)	2,806.601*** (df = 5; 9032)	74.631*** (df = 200; 8837)
Note:			*p<0.1; **p<0.05; ***p<0.01

Part VI

Reproducible research

Doporučená literatura

Burns, P. (2011). The R Inferno.

Grolemund, G. and Wickham, H. (2011). Dates and times made easy with lubridate. *Journal of Statistical Software*, 40(3):1–25.

Peng, R. D. (2016). *R Programming for Data Science*. LeanPub.

Spector, P. (2008). *Data Manipulation with R*. Use R! Springer, 1st edition. ISBN 978-0-387-74730-9.

Wickham, H. (2014). *Advanced R*. Chapman and Hall/CRC, Boca Raton, Florida, USA, 1st edition. ISBN 978-1-4664-8696-3.

Wickham, H. and Grolemund, G. (2017). *R forData Science*. O'Reilly, Sebastopol, California, USA, 1st edition. ISBN 978-1-491-91039-9.