Data a proměnné

Michal Kvasnička

V této dlouhé lekci se podíváme na to, jak se v R pracuje s proměnnými, jaké máme k dispozici datové typy a datové struktury a jak s nim pracovat. Toto téma se může zdát nudné, ale pro další práci s daty je naprosto zásadní.

Proměnné

Proměnná je jméno, ke kterému se přiřadí nějaká oblast v paměti počítače, která obsahuje nějaké hodnoty. Každá proměnná obsahuje určitou datovou strukturu, např. vektor. Každý prvek této struktury má určitý typ (např. celé číslo) a nějakou hodnotu (např. číslo 1). Typ proměnné určuje, jak je hodnota proměnné reprezentovaná v paměti počítače a jaké může obsahovat hodnoty (např. jakékoli celé číslo, ale ne desetinné číslo nebo text). Typ proměnné však nezáleží jen na tom, jaká je současná hodnota proměnné. Např. hodnota 1 může být v paměti počítače reprezentovaná jako celé číslo i jako reálné číslo.

I když to technicky není přesné, můžete si proměnnou představit jako krabičku, do které vložíte nějakou hodnotu. Každá proměnná může v každém okamžiku obsahovat vždy jen jednu hodnotu. Pokud uložíme do proměnné novou hodnotu, stará hodnota se ztratí. Slovo "hodnota" je však třeba brát volně. "Hodnota" může být stejně dobře jedno číslo i složitá struktura složená z čísel, textů a jiných věcí.

Jména proměnných

Jména proměnných musí splňovat určité nároky. Jméno proměnné se může skládat jen z písmen, číslic, teček a podtržítek a musí začínat písmenem nebo tečkou, za kterou nenásleduje číslice. Jména `a`, `myNumber` nebo `my_number` je přípustná; jména jako `.2way` 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 velkých a malých písmenech, tj. proměnné `x` a `X` jsou dvě různé proměnné.

Jméno proměnné by ideálně mělo být stručné a mělo by výstižně popisovat, jaké hodnoty 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 dále. Místo toho se doporučuje používat podtržítka (`pv_of_bond`); někdo doporučuje "Camel Case" (`pvOfBond`), jiní lidé od toho odrazují.

Funkce `make.names()` převede řetězec na syntakticky platné jméno proměnné, ovšem bohužel s tečkami:

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

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

Funkce `make_names()` z balíku **lettercase** dělá totéž, ale s podtržítky.

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 jde použít, pokud se její jméno uzavře mezi dva "backticks":

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

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

Přiřazení dat do proměnných

Hodnoty se do proměnných přiřazují pomocí "šipky" `<-` nebo `->` (druhou možnost používá jen p. Mikula), 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-`.

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). Detaily najdete zde: http://blog.revolutionanalytics.com/2008/12/use-equals-or-arrow-for-assignment.html.

Příklad:

```
x <- 1  # x má nyní hodnotu 1
2 -> y  # y má nyní hodnotu 2
```

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 (R volá implicitně funkci `print()` za vás) nebo 3) tak, že celý výraz přiřazení zabalíte do závorek.

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

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

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

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

```
(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 funguje tak, že požádá R o vrácení hodnoty proměnné – a 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, co R vypíše do konzoly, se liší od skutečných hodnot proměnných. R totiž pro různé objekty volá různé funkce `print()` přizpůsobené těmto objektům a může vypsat více nebo méně informací, než je jich obsaženo v dané proměnné.

Někdy proměnná obsahuje mnoho hodnot (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`:

```
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
```

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

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

```
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
```

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
```

Základní datové typy

R má několik základních datových typů. Z nich se však většinou používají čtyři:

- **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** – kladné i záporné reálné číslo; když zadáte v konzoli nebo skriptu číslo bez příznaku `L`, bude v paměti reprezentované jako double, i když to bude shodou okolností celé číslo; na konzulu je však R vypíše celá čísla inteligentně bez desetinných míst
- **character** – řetězec (text); v R se zadává mezi dvěma uvozovkami nebo apostrofy; uvozovky a apostrofy nelze kombinovat, což umožňuje zadat apostrofy nebo uvozovky jako součást řetězce; jiná možnost, jak zadat zvláštní znaky, je "escapovat" je, tj. napsat je pomocí zpětného lomítka `\` a vybraného znaku (např. `\"` znamená uvozovku, `\n` konec řádku apod.)

```
x1 <- TRUE
x1
```

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

```r
x2 <- 1L
x2
```

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

```r
x3 <- 1
x3
```

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

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

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

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

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

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

```r
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):

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

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

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, tj. celočíselná i 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 vektory však vrací typ proměnných.

```r
typeof(x1)
```

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

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

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

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

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

```r
typeof(x2)
```

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

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

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

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

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

```r
typeof(x3)
```

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

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

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

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

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

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

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

```r
typeof(x4)
```

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

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

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

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:

- chybějící hodnota `NA` (“not available”)
- hodnota není číslo – je chybná `NaN` (“not a number”)
- hodnota je nekonečná `Inf` (nebo samozřejmě `-Inf`)

Nekonečná hodnota vznikne např. při dělení nenulového čísla nulou:

```
1 / 0
```

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

Chybná hodnota vznikne při různých nepovolených 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.

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

```
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 (tj. vlastní datový typ) a zároveň rezervované slovo, které R vrací v situaci, kdy nějaká hodnota není definovaná. K otestování, zda je hodnota objektu `NULL` slouží funkce `is.null()`.

Převody mezi datovými typy

V případě, že R potřebuje nějakým způsobem sladit dva základní datové typy (např. je spojit do jednoho 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
c(TRUE, 1L, 1, "1")
```

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

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

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

```r
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:

```r
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.

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

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

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

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

```r
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
```

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

```r
x <- c("a", "b", "c")
as.numeric(x)
```

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

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

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.

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

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

```r
attributes(x)
```

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

```r
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
```

```r
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):

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

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

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

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

```r
x
```

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

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

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

```
## NULL
```

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

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

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

Operace se základními typy

Základní aritmetické operace

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

```r
9 %/% 2
```

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

```r
9 %% 2
```

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

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

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

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

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

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

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

Srovnání čísel

Ke srovnání aritmetických hodnot slouží následující operátory: porovnání shody celých čísel (`==`), různosti celých čísel (`!=`), větší (`<`), větší rovno (`<=`), menší (`>`) a menší rovno (`>=`). Pokud jsou použité na dva vektory, vrací vektor odpovídajících logických hodnot:

```r
c(1L, 2L, 3L) == c(3L, 2L, 1L)
```

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

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

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

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

Důvod je ten, že necelá čísla z desítkové soustavy není vždy možné vyjádřit dobře ve dvojkové soustavě a výsledek se 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 everywhere
```

```
## [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).

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:

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

Všimněte si, že `!(V1 & V2)` = `!V1 | !V2` a `!(V1 | V2)` = `!V1 & !V2`.

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

```
 c(T, F) & c(T,T)
```

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

Zkratující forma se používá v podmínkách (bude dále). 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(T, F) && c(T,T)
```

```
## [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()` viz výše.

Speciální datové typy

Kromě základních datových typů existují v R i další datové typy, které jsou implementované jako třídy objektů. Z nich nejdůležitější jsou faktory a třídy pro uchovávání datumů a času.

Faktory

Faktory slouží k uchovávání kategoriálních proměnných, ať už ordinálních (záleží na pořadí), nebo neordinálních. Příkladem ordinální kategoriální proměnné je kvalita služby: může např. nabývat hodnot "velmi špatná", "špatná", "průměrná", "dobrá", "výborná". Příkladem neordinální kategoriální proměnné je 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 ideální hned z několika důvodů: 1) je obtížné pamatovat si, co která hodnota znamená a 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. R také zná platné úrovně faktorů, tj. může hlídat, 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
```

To však není nejbezpečnější způsob, jak zadat hodnoty faktoru. Obecně je bezpečnější říct funkci `factor()` i to, jakých hodnot může faktor nabývat (pomocí parametru `levels`). To navíc umožní zadat i hodnoty, které nyní faktor neobsahuje, ale obsahovat by je mohl, a také určit pořadí faktorů (jinak je R řadí podle abecedy); pořadí je důležité pro ordinální faktory a také při některých statistických metodách určuje, která hodnota bude brána jako kontrast. Parametr `labels` navíc umožňuje změnit to, jak jsou hodnoty faktoru vypisované, pokud se mají lišit od úrovní (`levels`) – v paměti počítače jsou pak jako úrovně uložené hodnoty `labels`:

```
factor(c("male", "female", "female", "male", "female"),  # hodnoty vektoru
       levels = c("female", "male", "asexual"),           # možné úrovně
       labels = c("žena", "muž", "asexulní"))             # co se bude vypisovat
```

```
## [1] muž  žena žena muž  žena
## Levels: žena muž asexulní
```

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 nahradí hodnotou `NA`:

```
factor(c("male", "female", "female", "male", "beaver"),  # hodnoty vektoru
       levels = c("female", "male", "asexual"),           # možné úrovně
       labels = c("žena", "muž", "asexulní"))             # co se bude vypisovat
```

```
## [1] muž  žena žena muž  <NA>
## Levels: žena muž asexulní
```

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.

Poznámka: pokud opravdu dobře nevíte, co děláte, používejte raději ne-ordinální faktory. Ordinální faktory se ve formulích (tj. např. v ekonometrické analýze) chovají jinak, než byste možná čekali, viz https://stat.ethz.ch/pipermail/r-help/2008-January/150930.html a https://cran.r-project.org/doc/manuals/R-intro.pdf, 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
```

Někdy je užitečné rozdělit spojitou škálu do diskrétních hodnot. 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 funkce viz dokumentace.

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, existuje však jedna zrádná výjimka:

```
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
```

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 balíků, každý z nich pouze jednou a vektory řetězců jsou implementovány jako odkazy do tohoto skladu. Prakticky to znamená, že převedením řetězců na faktor se žádná paměť neušetří.

Z těchto historických důvodů se R snaží řetězce převádět na faktory např. při použití funkce `data.frame()` i při načítání tabulek ze souboru (viz dále). Pokud tomu chcete zabránit, musíte o to R explicitně požádat.

Datum a čas

Datum a čas lze v R ukládat do mnoha různých tříd objektů. Z nich jsou nejzákladnější tři: třída `Date` pro reprezentaci datumů (celých dnů bez času) a třídy `POSIXct` a `POSIXlt` pro reprezentaci času. První dvě třídy ukládají datum a čas jako počet dnů nebo sekund od 1. ledna 1970.

Datum

Třídu `Date` dostaneme explicitní konverzí řetězce, který obsahuje datum:

```r
as.Date("2016-11-25")  # 25. listopadu 2016
```

```
## [1] "2016-11-25"
```

Pokud je datum zadáno v jiném formátu, musíte tento formát popsat parametrem `format`:

```r
as.Date(c("1led1960", "2led1960", "31bře1960", "30čec1960"), format = "%d%b%Y")
```

```
## [1] "1960-01-01" "1960-01-02" "1960-03-31" "1960-07-30"
```

Formát se zadává pomocí formátovacích řetězců, které jsou popsané v dokumentaci funkce `strptime()`. Jména měsíců a dnů v týdnu 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:

```r
# nastav locales, aby jména dnů byla anglicky
# (zde se tento řádek neprovede, aby byl zbytek textu česky)
Sys.setlocale("LC_TIME", "C")
```

Víc detailů je v dokumentaci k `locales` (`?locales`).

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).

Čas

K uložení času slouží třídy `POSIXct` a `POSIXlt`. Třída `POSIXct` ukládá počet sekund od referenčního data, jedná se tedy o celé číslo. Třída `POSIXlt` je seznam o mnoha položkách. Pro ukládání času do datasetu se tedy více hodí třída `POSIXct`, protože je paměťově úspornější.

K převodu na třídy času slouží funkce `as.POSIXct()` a `as.POSIXlt()`. Užitečné jsou i funkce `strftime()` a `strptime()`, které slouží k převodu řetězce na třídu `POSIXlt` a opačným směrem. Detaily jsou v dokumentaci.

Pokud např. chceme zjistit jaký den odpovídá nějakému datu, můžeme použít funkci `strftime()`:

```r
d <- as.Date("2016-11-05")
strftime(d, format = "%A")
```

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

Operace s datem a časem

Pro třídy `Date`, `POSIXct` a `POSIXlt` fungují některé aritmetický a logické operace:

```r
d <- as.Date(c("2016-01-10", "2016-03-11"))
d
```

```
## [1] "2016-01-10" "2016-03-11"
```

```r
d[2] - d[1]              # kolik dnů je mezi druhým a prvním dnem ve vektoru?
```

```
## Time difference of 61 days
```

```r
diff(d)                    # funguje i funkce diferencí
```

```
## Time difference of 61 days
```

```r
as.numeric(d[2] - d[1])  # převod na celé číslo
```

```
## [1] 61
```

```r
d > "2016-02-05"           # které dny ve vektoru jsou po 5. únoru 2016?
```

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

K získání základních informací o datech slouží funkce `weekdays()`, `months()`, `days()` a `quarters()`:

```r
weekdays(d)  # den v týdnu
```

```
## [1] "Neděle" "Pátek"
```

```r
months(d)    # měsíc v roce
```

```
## [1] "leden"  "březen"
```

```r
quarters(d)  # čtvrtletí
```

```
## [1] "Q1" "Q1"
```

Funkce `difftime()` vrací vzdálenost mezi dvěma dny v zadaných jednotkách (parametr `units`), které mohou mít jednotky `"auto"`, `"secs"`, `"mins"`, `"hours"`, `"days"` nebo `"weeks"`:

```r
difftime(d[2], d[1], units = "weeks")
```

```
## Time difference of 8.714286 weeks
```

Funkce `seq()` může vytvářet časové vektory. Parametr `by` se nastaví buď na 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:

```r
seq(as.Date("2015-1-1"), by = "2 days", length = 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"
```

Systémový čas

Aktuální (systémový) čas lze zjistit funkcí `Sys.time()`, která vrací třídu `POSIXct`:

```r
Sys.time()
```

```
## [1] "2016-09-29 13:37:19 CEST"
```

Další užitečné balíky pro práci s datem a časem

Balík **lubridate** definuje mnoho užitečných funkcí, které zjednodušují práci s datem a časem.

Balíky **chron** a **zoo** implementují pokročilejší třídy pro reprezentaci data a času v R.

Základní datové struktury

Základní datové struktury lze roztřídit podle dvou charakteristik: 1) podle jejich dimensionality (jednorozměrné, dvourozměrné atd.) a 2) zda jsou všechny jejich položky stejného datového typu (tj. struktura je homogenní), nebo zda je heterogenní.

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

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).

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

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

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

```
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:

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

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

Jednotlivé prvky atomických vektorů mohou mít jména. Jména jsou uložena jako atribut `names`. Je možné je zadat třemi způsoby: přímo ve funkci `c()`, pomocí funkce `attr()` nebo pomocí speciální funkce `names()`:

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

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

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

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

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

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

```
x
```

```
## aa bb cc
##  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
y <- x[x > 10]  # vybereme prvky větší než 10, viz dále
length(y)
```

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

```
y
```

```
## integer(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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```r
# 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
```

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

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

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

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

```r
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
## [36] 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 později). 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 `apply()`, viz dále.

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. Každý řádek je 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.

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.

Veškeré aritmetické a logické operace se na vektorech provádí po prvcích:

```
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
```

K otestování, zda je proměnná atomický vektor slouží funkce `is.atomic()`:

```
is.atomic(x)
```

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

Pozor: existuje i funkce `is.vector()`. Ta však vrací logickou hodnotu `TRUE` nejen pro atomický vektor, ale i pro neatomický vektor (seznam), viz dále:

```r
is.vector(x)
```

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

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 je jasné, jaké budou.

```r
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
```

```r
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.

```r
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`.)

```r
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 spojuje matice po řádcích (tj. skládá je pod sebe), druhá 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
```

```r
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
```

```r
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:

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

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

```r
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
```

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

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

Podobně lze zrušením atributu `dim` převést matici zpět na vektor (matice se vektorizuje po sloupcích).

```r
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 vrací překvapivě `FALSE`, protože testuje, zda je v proměnné uložen vektor.

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

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

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

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

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).

```r
A * B
```

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

```r
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:

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

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

```r
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:

```r
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
```

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

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

Atomická pole

Atomické pole je vícerozměrná tabulka (tabulka, která má víc než dva rozměry), jehož všechny prvky mají stejný datový typ.

Atomická pole se konstruují pomocí funkce `array()`. Zbytek viz dokumentace.

```r
array(1:27, dim = c(3, 3, 3))
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   10   13   16
## [2,]   11   14   17
## [3,]   12   15   18
##
## , , 3
##
##      [,1] [,2] [,3]
## [1,]   19   22   25
## [2,]   20   23   26
## [3,]   21   24   27
```

Neatomické vektory (seznamy)

Neatomické vektory (častěji nazývané seznamy) jsou vektory, jejichž jednotlivé prvky mohou mít různé datové typy, třeba i jiné seznamy. Seznamy se vytvářejí pomocí funkce `list()`:

```r
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(l)`.

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

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

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

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

Pozor: Funkce `is.vector()` vrací hodnotu `TRUE` i pro seznamy.

Seznamy jsou někdy užitečné samy o sobě; jejich hlavní význam však spočívá v tom, že se používají jako základ pro tvoru většiny objektů (v rámci objektů tříd typu `S3`).

Neatomické matice a pole

Pokud přiřadíme seznamu atribut `dim`, pak se seznam změní na neatomickou matici nebo pole. Je to obskurní struktura, ale někdy by se mohla hodit:

```
dim(l) <- c(2, 2)
l
```

```
##      [,1]   [,2]
## [1,] 1      Integer,3
## [2,] "ahoj" List,2
```

Při výpisu na obrazovku jsou vektory vypsány jako typ a délka (např. `Integer,3` znamená vektor celých čísel délky 3).

Datasety

Pro analýzu dat jsou nejdůležitější datovou strukturou datasety. R má několik implementací datasetů, základní se nazývá `data.frame`. Datasety jsou tabulky, jejichž řádky představují jednotlivá pozorování a sloupce jednotlivé proměnné. Fakticky se jedná o seznam atomických vektorů o stejné délce, které jsou spojené vedle sebe. Každý vektor (sloupec tabulky) může obsahovat proměnné jiného typu.

Řekněme, že chceme zaznamenat údaje o subjektech, které se zúčastnily nějakého experimentu. Pro každý subjekt pozorujeme jeho id, výšku a váhu. Pokud máme čtyři subjekty, můžeme vytvořit dataset 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 datasetu můžeme zadat jejich jména, která pak R vypíše. R samo přidá jména řádků (automaticky jim dá přirozená čísla od 1 do počtu pozorování).

Počet řádků datasetu 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] 3
```

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

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

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

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

I při konstrukci datasetů 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),
                         stringsAsFactors = FALSE)
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
```

Při zadání dat do dataset pomocí funkce `data.frame()` R převede všechny řetězce na faktory. Této konverzi zabrání parametr `stringsAsFactors = FALSE`.

Datasety 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"
##
## $row.names
## [1] 1 2 3 4
##
## $class
## [1] "data.frame"
```

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 datasetu (do tabulky). Je to filosoficky správnější a i praktičtější: některé funkce, které se používají ke zpracování datasetů 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` (bud pomocí funkce `attr()` nebo funkce `class()`). Pak se však budou pro daný objekt volat jiné funkce a výsledek může být podivný. Nedělejte to, pokud nevíte, co děláte.

Někdy je užitečné moci převést dataset na matici a matici na dataset. K převodu datasetu na matici slouží funkce `as.matrix()` a `data.matrix()`. První převede všechny sloupce datasetu automatickou konverzí na stejný typ, a pak na matici. Druhá převede explicitní konverzí všechny sloupce na reálná čísla, a pak na matici. Při automatické konverzi můžeme skončit s řetězci, což nemusí být žádoucí; se 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
```

```
## Warning in data.matrix(experiment): NAs introduced by coercion
```

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

Matici lze převést na dataset pomocí funkcí `as.data.frame()` i `data.frame()`. Pokud má matice pojmenované sloupce, jejich jména jsou v datasetu 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
```

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

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

```r
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
```

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

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

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

```r
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
```

Zjištění obsahu datové strukury

Ke zjištění struktury a obsahu datové struktury slouží funkce `str()`. Funkce vypíše třídu proměnné a zobrazí její strukturu.

```r
n <- 1:12
str(n)
```

```
##  int [1:12] 1 2 3 4 5 6 7 8 9 10 ...
```

```r
l <- list(a = 1, b = 1:3, "ahoj", d = list(1, 1:2, TRUE))
str(l)
```

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

```
d <- data.frame(x = 1:12, y = c(TRUE, FALSE))
str(d)
```

```
## 'data.frame':    12 obs. of  2 variables:
##  $ x: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ y: logi  TRUE FALSE TRUE FALSE TRUE FALSE ...
```

Výběry částí datových struktur

Většina proměnných obsahuje více než jednu hodnotu: vektory obsahují prvky, matice a datasety obsahují řádky a sloupce atd. Někdy je potřeba z těchto hodnot vybrat jen některé. K tomu slouží subsetování. 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.

Hranaté závorky vrací object stejné třídy, jako je původní objekt. To se vždy nehodí, proto R nabízí i operátor dvojitých hranatých závorek (`[[]]`), který extrahuje prvky ze seznamů, datasetů a podobných struktur. Podobnou funkci plní i operátor dolar (`$`).

Subsetování lze použít nejen k získání vybraných prvků z datové struktury, ale také k jejich nahrazení nebo doplnění.

Atomické vektory

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"
```

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

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

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

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

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

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

```r
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):

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

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

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

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

```r
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!

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

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

```r
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:

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

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

```r
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:

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

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

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

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

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

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

```r
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:

```r
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.

Matice

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:

```r
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
```

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

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

```r
M[1:2, c(1,4)]  # pruky na pruních dvou řádcích a v pruním a čtvrtém sloupci
```

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

```r
M[-1, -1]  # matice bez pruního 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:

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

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

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

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

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

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

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

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

```r
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:

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

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

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

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

Subsetování může nejen vybírat hodnoty, ale také měnit jejich pořadí. Funkce `order()` vrací indexy uspořádané podle velikosti původního vektoru. Funkce např. umožňuje setřídit hodnoty všech sloupců matice podle jednoho sloupce:

```r
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
```

```r
# 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
```

```r
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:

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

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

```
M[, o]
```

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

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
```

Seznamy

Subsetování pomocí hranatých závorek zachovává mód proměnné. V případě atomických vektorů a matic je to vhodné chování – výsledkem subsetování je atomický vektor nebo matice. V případě seznamů znamená zachování módu proměnné, že výsledkem subsetování je opět seznam, který obsahuje dané prvky, nikoli prvek jako takový:

```
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:

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

```
## NULL
```

```
l$d
```

```
## NULL
```

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!

Datasety

Datasety 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 dataset, ve druhém může vrátit dataset (pokud se vybere více sloupců), nebo vektor (pokud se vybere jen jeden sloupec). Dolar 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"]   # dataset s jediným sloupcem
```

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

```
d[1]     # opět dataset 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
```

```r
d[1:2, 1, drop = FALSE]  # dataset složený z prvních dvou hodnot vektoru x
```

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

```r
d[1:2, 1:2]              # dataset složený z prvních dvou řádků
```

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

```r
d[1:2, c("x", "y")]      # dataset složený z prvních dvou řádků
```

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

```r
d[1:2, ]                 # dataset složený z prvních dvou řádků
```

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

Samozřejmě je možné použít i indexování pomocí logických hodnot:

```r
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
```

```r
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:

```r
# 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 datasetu a podobných struktur slouží funkce `complete.cases()`. V případě datasetu vrací vektor logických hodnot, který je `TRUE` pro každý řádek datasetu, 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 datasetů existuje funkce `subset()`. Té však nebudeme věnovat pozornost, protože se později naučíte mnohem příjemnější a rychlejší funkce implementované v balíku **dplyr**.

Do existujícího datasetu 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 datasetu:

```
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 datasetu.

Tibble

Kromě `data.frame` existuje v ještě několik typů datasetů. Později budete potřebovat ještě typ `tibble`, do kterého převádí datasety balíky **dplyr** a **tidyr**. Tento typ datasetu má některé velmi příjemné vlastnosti. Detailní popis této třídy najdete zde: https://blog.rstudio.org/2016/03/24/tibble-1-0-0/ a ve vinětě k balíku **tibble**.

Dataset `tibble` vytvoříte pomocí funkce `tibble()` nebo `data_frame()` (jedná se o synonyma):

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

```
## # A tibble: 1,000,000 × 3
##        x     y        zed
##    <int> <dbl>      <dbl>
## 1      1     2 -3.6666667
## 2      2     4 -0.3333333
## 3      3     6  3.0000000
## 4      4     8  6.3333333
## 5      5    10  9.6666667
## 6      6    12 13.0000000
## 7      7    14 16.3333333
## 8      8    16 19.6666667
## 9      9    18 23.0000000
## 10    10    20 26.3333333
## # ... with 999,990 more rows
```

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

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

1. nepřevádí řetězce na faktory
2. nemění "nepovolená" jména sloupců na povolená nahrazením divných znaků tečkami
3. vyhodnocuje své argumenty postupně, takže můžete později zadaný argument použít při tvorbě dříve zadaného argumentu (jako v příkladu výše)
4. nepoužívá jména řádků (která jsou ostatně nebezpečná k uchovávání dat)
5. převod na `tibble` pomocí `as_tibble()` je rychlejší než převod na `data.frame` pomocí `as.data.frame()`

Liší se také tisk `tibble` od tisku `data.frame`: oproti `data.frame` zobrazí `tibble` navíc rozměr datasetu 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 bud ve funkci `print()`, nebo v `options()`, viz viněta:

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

```
## # A tibble: 1,000,000 × 3
##       x     y        zed
##   <int> <dbl>      <dbl>
## 1     1     2 -3.6666667
## 2     2     4 -0.3333333
## 3     3     6  3.0000000
## 4     4     8  6.3333333
## 5     5    10  9.6666667
## # ... with 1e+06 more rows
```

Liší se také subsetování. `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 × 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 column 'z'
```

```
## NULL
```

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)
```

```
##   x  y        zed
## 1 1  2 -3.6666667
## 2 2  4 -0.3333333
## 3 3  6  3.0000000
## 4 4  8  6.3333333
## 5 5 10  9.6666667
## 6 6 12 13.0000000
```

Volba datové struktury

Vstupní data pro jakoukoli analýzu budou mít ve většině případů formát datasetu. 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 datasety (`data.frame`, `tibble` apod.). 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 datasetů (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.

Domácí úkol

Máte k dispozici dvě matice (`pmat1` a `pmat2`) se záznamy ze dvou lékařských studií. Obě matice mají stejný formát: jednotlivé řádky odpovídají jednotlivým pozorováním (lidem), sloupce postupně jejich výšce v cm, váze v kg, jménu (jednoznačnému) a výsledku jejich reakce na nový lék (nedokumentované). Vašimi úkoly je:

1. obě matice spojit a převést na `data.frame`, který pojmenujete `people1`; všechny sloupce by měly zůstat jako řetězce
2. nastavit jména sloupců datasetu postupně na "height", "weight", "name" a "coef" a výsledný dataset uložit do proměnné `people2`
3. změnit pořadí sloupců na "name", "height" a "weight" (sloupec "coef" vyhodíte – dále se s ním nebude pracovat a musí zůstat utajen); výsledný dataset uložíte do proměnné `people3`
4. sloupec s výškou a váhou převedete z řetězce na reálná čísla; výsledný dataset uložíte do proměnné `people4`
5. přidáte nový sloupec s BMI jako poslední sloupec (BMI vypočítáte jako w/h^2, kde w je váha v kg a h je výška v metrech, a zaokrouhlíte na jedno desetinné místo pomocí funkce `round()`); výsledný dataset uložíte do proměnné `people5`
6. do datasetu `people6` vyberete všechna pozorování, kde je BMI vyšší roven 30
7. výsledek setřídíte podle jména podle abecedy vzestupně (od A do Z) a výsledný dataset uložíte do proměnné `people7`
8. výsledek setřídíte podle BMI sestupně (od nejvyššího po nejnižší) a uložíte do datasetu `people8`
9. do proměnné `num_obese` uložíte počet obézních lidí (lidí s BMI 30 a více)
10. spočítáte medián BMI *všech* lidí přítomných v původním vzorku dat (`pmat1` i `pmat2`); výsledný dataset uložíte do proměnné `med_bmi`

Upravte soubor `hw_data_a_promenne.R` pouze na označených místech. Výsledek každého dílčího úkolu uložíte do nové proměnné předepsané v zadání. Hodnoty předchozích výpočtů neměňte, jinak za daný úkol získáte nula bodů. Skript průběžně ukládá výsledky vašich výpočtů proto, abyste v případě pádu skriptu získali úkoly, které jste splnili před pádem skriptu.