Práce s řetězci

Michal Kvasnička

Práce s daty nezahrnuje jen práci s čísly, ale i s texty (řetězci). 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 (interface). Proto se místo těchto funkcí podíváme na funkce implementované v balíku **stringr**, 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**.)

Řetězce v R

R ukládá řetězce ve vektorech datového typu `character`.

Zadávání řetězců

Řetězec se zadává mezi dvěma uvozovkami nebo dvěma apostrofy (nejde je míchat, ale je možné uzavřít apostrof mezi uvozovky nebo naopak):

```
"Petr řekl: 'Už tě nemiluji.'"
'Agáta odpověděla: "Koho to zajímá?"'
```

Řetězce mohou v R obsahovat speciální znaky jako např. `"\n"` (konec řádku), `"\t"` (tabelátor) apod. Uvozovky je také možné zbavit jejich speciálního významu tak, že jim předřadíte zpětné lomítko. To slouží nejen k zadávání speciálních znaků, ale i k escapování speciálních znaků, které pak mají svůj doslovný význam. Pokud chcete v řetězci zadat zpětné lomítko, musíte je escapovat (tj. zdvojit); pokud chcete zadat uvozovky v řetězci ohraničeném uvozovkami, musíte je také escapovat zpětným lomítkem:

```
s <- "Petr na to odvětil: \"Je to proto, že vypadáš jako \\.\""
```

Tisk řetězců

Funkce `print()`, která se používá při implicitním vypsání obsahu proměnné, vypisuje na začátku řetězců uvozovky a všechny speciální a escapované 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 \."
```

Funkce, které slouží k tisku řetězců zahrnují:

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

Podívejte se na tyto funkce a jejich parametry do nápovědy.

Prázdné řetězce a vektory řetězců

Je potřeba rozlišovat mezi prázdnými řetězci (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.

Vektory písmen ASCII

R má dva speciální vektory, které obsahují písmena ASCII abecedy:

```
letters  # malá písmena anglické abecedy
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "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"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Porovnávání řetězců

Porovnat, zda jsou dva řetězce stejné, je možné pomocí `==`:

```
"abc" == "abc"
```

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

```
"abc" == "ABC"
```

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

Funguje i funkce `all.equal()`:

```
all.equal("abc", "abc", "abc")
```

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

Lze použít i `>`, `<` atd.

Kódování řetězců

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 daného počítače. Většinou se o kódování nemusíte nijak starat. Problém může nastat jen v případě přenosu dat mezi různými operačními systémy (v rámci Windows možná i mezi různými lokalizacemi). Naštěstí umí R změnit kódování i převést text z jednoho kódování do jiného. Základní dokumentaci ke kódování poskytuje `help("Encoding")`.

Ke konverzi kódování řetězců je možné použít funkci `iconv()`, viz dokumentace.

Základní operace

Spojení řetězců

Ke spojení více řetězců do jednoho slouží funkce `str_c()`. Funkce bere libovolný počet řetězců, vypustí z nich prázdné vektory řetězců a `NULL` (ale ne prázdné řetězce `""`) a zbylé neprázdné řetězce spojí je do jednoho vektoru:

```
library(stringr)
str_c("Jednou", "", "budem", character(0), "možná", NULL, "dál!")
```

```
## [1] "Jednoubudemmožnádál!"
```

Implicitně odděluje jednotlivé řetězce prázdným řetězcem. Oddělovací řetězec je možné nastavit pomocí pojmenovaného parametru `sep`:

```
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!"
```

(Všimněte si, že prázdný řetězec `""` funkce nevypustila.)

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:

```
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 vektor, 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 je 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 (sloupcový) 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.

Takže `str_c(s1, s2, sep = "-")` udělá toto:

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
```

Zajímavé příklady z dokumentace funkce:

```
str_c("Letter: ", letters)
```

```
##  [1] "Letter: a" "Letter: b" "Letter: c" "Letter: d" "Letter: e"
##  [6] "Letter: f" "Letter: g" "Letter: h" "Letter: i" "Letter: j"
## [11] "Letter: k" "Letter: l" "Letter: m" "Letter: n" "Letter: o"
## [16] "Letter: p" "Letter: q" "Letter: r" "Letter: s" "Letter: t"
## [21] "Letter: u" "Letter: v" "Letter: w" "Letter: x" "Letter: y"
## [26] "Letter: z"
```

```
str_c("Letter", letters, sep = ": ")
```

```
##  [1] "Letter: a" "Letter: b" "Letter: c" "Letter: d" "Letter: e"
##  [6] "Letter: f" "Letter: g" "Letter: h" "Letter: i" "Letter: j"
## [11] "Letter: k" "Letter: l" "Letter: m" "Letter: n" "Letter: o"
## [16] "Letter: p" "Letter: q" "Letter: r" "Letter: s" "Letter: t"
## [21] "Letter: u" "Letter: v" "Letter: w" "Letter: x" "Letter: y"
## [26] "Letter: z"
```

```
str_c(letters[-26], " comes before ", letters[-1])
```

```
##  [1] "a comes before b" "b comes before c" "c comes before d"
##  [4] "d comes before e" "e comes before f" "f comes before g"
##  [7] "g comes before h" "h comes before i" "i comes before j"
## [10] "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"
## [16] "p comes before q" "q comes before r" "r comes before s"
## [19] "s comes before t" "t comes before u" "u comes before v"
## [22] "v comes before w" "w comes before x" "x comes before y"
## [25] "y comes before z"
```

```
str_c(letters, collapse = "")
```

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

```
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"
```

Zjištění délky řetězce

Ke zjištění délky řetězce slouží funkce `str_length()`, která vrací vektor délek jednotlivých řetězců ve vektoru:

```
s1 <- "Ahoj!"
s2 <- c("A", "Bb", "Ccc", NA)
str_length(s1)
```

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

```
str_length(s2)
```

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

Pozor: funkce `length()` nevrací délku řetězce, ale vektoru řetězců. Pamatujte, že R nemá skalár: to, co vypadá jako jeden řetězec, je ve skutečnosti vektor řetězců o délce 1:

```
length(s1)
```

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

Pozor: technicky vzato vrací funkce `str_length()` počet "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 je v dokumentaci funkce.

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(s, decreasing = FALSE, na_last = TRUE, locale = "", ...)` vrací celé číslo, které odpovídá pořadí daného řetězce ve vektoru `s` (podobně jako `order()`).
- `str_sort(s, decreasing = FALSE, na_last = TRUE, locale = "", ...)` setřídí vektor řetězců `s`.

Přitom `s` 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); `...` 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] 24 25 26
```

```
str_sort(letters, locale = "en")
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
str_order(letters, locale = "haw")
```

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

```
str_sort(letters, locale = "haw")
```

```
##  [1] "a" "e" "i" "o" "u" "b" "c" "d" "f" "g" "h" "j" "k" "l" "m" "n" "p"
## [18] "q" "r" "s" "t" "v" "w" "x" "y" "z"
```

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(s, start, end)`, kde `s` je řetězec, ze kterého vybíráme, `start` je pozice znaku začátku výběru a `stop` je pozice konce výběru (včetně):

```
s1 <- "Sol 6: Katastrofa na Marsu."
str_sub(s1, start = 8, end = str_length(s1))
```

```
## [1] "Katastrofa na Marsu."
```

```
str_sub(s1, 8)
```

```
## [1] "Katastrofa na Marsu."
```

```
str_sub(s1, start = 8, end = -1)
```

```
## [1] "Katastrofa na Marsu."
```

```
str_sub(s1, end = 5)
```

```
## [1] "Sol 6"
```

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.

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"
```

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."
```

Replikace řetězců

Někdy je potřeba nějaký řetězec "zmnožit". K tomu slouží funkce `str_dup(s, n)`, který vezme vektor `s`, zopakuje jej `n`-krát a výsledek spojí. To se hodí např. při načítání mnoha textových sloupců pomocí balíku **readr**. Při tom je užitečné říct funkci `read_csv()`, že všechny sloupce tabulky mají typ `character`. K tomu slouží řetězec mnoha "c":

```
str_dup("c", 28)
```

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

Funkce `str_dup()` také recykluje všechny své parametry:

```
str_dup(c("a", "b", "c"), 1:3)
```

```
## [1] "a"   "bb"  "ccc"
```

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(s, side)`, kde `s` 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?"
```

Zarovnání řetězců na stejnou délku

Někdy je užitečné zarovnat řetězce na stejnou délku přidáním bílých (nebo jiných) znaků. K tomu slouží funkce `str_pad(s, w, side, pad)`, kde `s` je vektor zarovnávaných řetězců, `w` 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"
```

```r
str_pad(c("Ahoj", "lidi"), 9, side = "both", pad = "-")
```

```
## [1] "--Ahoj---" "--lidi---"
```

Delší řetězce funkce nemění:

```r
str_pad(c("Ahoj", "malé zelené bytosti z Viltvoldu 7"), width = 7)
```

```
## [1] "   Ahoj"                          "malé zelené bytosti z Viltvoldu 7"
```

Zarovnání do odstavce

Zarovnat řetězec do odstavce umožňuje funkce `str_wrap(s, w, indent, exdent)`, kde `s` je zarovnávaný řetězec, `w` je cílová šířka sloupce (implicitně 80 znaků), `indent` je odsazení prvního řádku a `exdent` je odsazení následujících řádků (oboje implicitně 0 znaků):

```r
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.
```

Konverze malých a velkých písmen

Ke konverzi velikosti písmen slouží funkce:

funkce	význam
`str_to_upper(s, locale)`	konvertuje řetězec `s` na velká písmena
`str_to_lower(s, locale)`	konvertuje řetězec `s` na malá písmena
`str_to_title(s, locale)`	slova začínají velkým písmenem, zbytek jsou malá písmena

Každá z nich převede řetězec `s`. Pokud není nepovinný parametr `locale` zadán, použije se aktuální locale počítače.

```r
dog <- "The quick brown dog"
str_to_upper(dog)
```

```
## [1] "THE QUICK BROWN DOG"
```

```r
str_to_lower(dog)
```

```
## [1] "the quick brown dog"
```

```r
str_to_title(dog)
```

```
## [1] "The Quick Brown Dog"
```

```r
# Locale matters!
str_to_upper("i", "en") # English
```

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

```r
str_to_upper("i", "tr") # Turkish
```

```
## [1] "İ"
```

Konverze kódování

Funkce `str_conv(string, encoding)` nastaví nebo změní aktuální kódování řetězce, viz dokumentace.

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` i `+420 123 456 789` nebo `+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), který popisuje určitý objem textu. 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.)

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 sebe se spojí do jednoho výrazu; např. `abcd` je spojení čtyř znaků, které fungují jako celek, jako řetězec "abcd"
- logické "nebo" označené symbolem `|` vybírá jednu z možností; např. `ab|cd` znamená řetězec "ab" nebo řetězec "cd"
- opakování – umožňuje říct, kolikrát se má nějaký řetězec v textu vyskytovat; k tomu poskytují regulární výrazy *kvantifikátory*, viz níže
- seskupování – znaky zapsané do obyčejných závorek `( )` tvoří skupinu, tj. jeden celek, který se může např. opakovat, měnit priority vyhodnocování výrazů apod.; skupiny mají i speciální význam při vybírání částí regulárních výrazů, viz níže

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!").

Jakýkoli symbol

Tečka `.` označuje libovolný znak. Výraz `ma.ka` najde slova jako "matka", "maska" nebo "marka", ale ne "maka".

Rozsahy

Hranaté závorky určují výčet platných znaků (character classes). 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 např. 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:

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	

Regulární výrazy v R mohou 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.

rozsah	význam
`[:lower:]`	malá písmena v locale počítače
`[:upper:]`	velká písmena v locale počítač
`[:alpha:]`	malá i velká písmena v locale počítač
`[:digit:]`	číslice, tj. 0, 1, . . . , 9
`[:alnum:]`	alfanumerické znaky, tj. `[:alpha:]` a `[:digit:]`
`[:blank:]`	mezera a tabelátor
`[:cntrl:]`	řídící znaky
`[:punct:]`	`!`, `"`, `#`, `%`, `&`, `'`, `(`, `)`, `*`, `+`, `,`, `-`, `.`, `/`, `:`, `;`, `<`, `>`, `?`, `@`, `[`, `\`, `]`, `^`, `_`, `‘`, `{`,

Rozdíl mezi `\w` a `[:alnum:]` spočívá v tom, že `[:alnum:]` obsahuje v českém locale i písmena s háčky a čárkami, zatímco `\w` obsahuje jen písmena ASCII. Pokud tedy uvažujete písmena, doporučuji používat vždy POSIXové rozsahy.

Kvantifikátory

Znaky `*+?{` jsou kvantifikátory, které řídí počet opakování předchozího znaku, rozsahu nebo skupiny.

znak	význam
`*`	libovolný počet opakování (tj. vůbec 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

Příklady:

- `.*` jakýkoli znak opakovaný libovolně krát, tj. jakýkoli řetězec včetně `""`
- `\w+` aspoň jedno písmeno
- `[+-]?\d+` celé číslo (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.\""
```

Začátek a konec řetězce

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).

Výraz `^-+$` splní pouze řádky, které obsahují pouze pomlčky.

Skupiny

Skupiny ovlivňují priority 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.)

Stejně tak ovlivňují priority 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) stačí regulární výraz `(.)(.).\2\1`.

Doslovný význam speciálních znaků

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 `\d` je tak třeba v R zapsat `"\\d"`, místo `\.` je třeba zapsat `"\\."` atd.

Příklad: telefonní číslo

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 musí být zapsán jako

```
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)
```

Příklad: datum

Regulární výraz, který pozná všechny výše uvedené formáty data:

```
r <- "\\d{1,2}\\.\\s*(\\d{1,2}\\.|leden|únor|březen|duben|červen|červenec|srpen|září|říjen|listopad|
dat <- c("1. 6. 2001", "1.1.2012", "1. červen 2016", "ahoj")
str_detect(dat, r)
```

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

Další zdroje k regulárním výrazům

Regulární výrazy obsahují mnohem více než to, co jsem zde ukázal. Plný výčet detailů je na těchto (podle mého mínění velmi nepřehledných) stránkách: http://www.regular-expressions.info/reference.html.

Vyzkoušet si, co regulární výraz najde a co ne, si můžete interaktivně na stránce http://regexr.com/.

Rady Hadleyho Wickhama

"When writing regular expressions, I strongly recommend generating a list of positive (pattern should match) and negative (pattern shouldn't match) test cases to ensure that you are matching the correct components."

Funkce pro práci s regulárními výrazy

Všechny funkce pro práci s regulárními výrazy v balíku **stringr** mají stejný interface:

```
str_jméno(řetězec, vzor)
```

kde `řetězec` je vektor zpracovávaných řetězců a `vzor` je použitý regulární výraz (některé funkce mají i další parametry). Jména všech těchto funkcí začínají na `str_`. Pokud některá funkce vrací je první výskyt hledaného vzoru, pak odpovídají funkce končící na `_all` vrací všechny výskytu tohoto vzoru.

Detekce vzoru

K detekci, zda řetězec obsahuje zvolený regulární výraz, slouží funkce `str_detect(s, p)`, kde `s` je prohledávaný vektor řetězců a `p` je hledaný vzor. Funkce vrací logický vektor stejné délky jako `s` s hodnotou `TRUE`, pokud byl vzor nalezen, a hodnotou `FALSE` v opačném případě. Příklady viz výše.

Výběr řetězců, které odpovídají vzoru

Funkce `str_subset(s, p)` vrací ty prvky vektoru `s`, které obsahují vzor `p`. Je vlastně ekvivalentní výrazu

```
x[str_detect(x, p)]
```

Řekněme, že chceme vybrat ty řádky, 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"
```

Počet výskytů vzoru v řetězci

Funkce `str_count(s, p)` spočítá počet výskytů regulárního výrazu `p` v řetězci `s`. Pracuje vektorově přes zadaný vektor řetězců `s` i přes regulární výraz `p`. Implicitní hodnota `p` je prázdný řetězec `""`. V tomto případě vrací funkce `str_count()` počet znaků v řetězci (počítaných podle aktuálního locale).

```
ovoce <- c("jablo", "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
```

```
## [1] 4 8
```

Získání částí řetězců, které splnují vzor

Funkce `str_extract(s, p)` získá z každého řetězce ve vektoru `s` tu jeho část, která odpovídá prvnímu výskytu vzoru `p`. Funkce vrací vektor stejné délky jako `s`; pokud není vzor nalezen, vrací `NA`.

Výběr klíčových slov z tweetů:

```
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_all(s, p, simplify)` vrátí všechny výskytu tohoto vzoru. Pokud není parametr `simplify` zadán nebo je `FALSE`, vrací funkce seznam, jehož prvky odpovídají prvkům vektoru `s`; nenalezené výskyty jsou pak prázdný vektor řetězců (`character(0)`). Pokud je `simplify = TRUE`, pak vrací matici, jejíž řádky odpovídají prvkům vektoru `s`; 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(s, p)` a `str_match_all(s, p)`, kde `s` je vektor prohledávaných řetězců a `p` regulární výraz. K rozdělení regulárního výrazu do částí se používají skupiny.

Funkce `str_match()` hledá první výskyt regulárního výrazu `p` v řetězci a vrací matici, jejíž řádky odpovídají prvkům vektoru `s`. První sloupec je celý regulární výraz, druhý sloupec první skupina v regulárním výrazu, třetí sloupec druhá skupina atd. (Skupiny mohou být zanořené. Jejich pořadí se počítá podle pořadí levé závorky.) Nenalezené prvky mají hodnotu `NA`.

Pokud např. chceme získat jméno odkazovaného předmětu bez hashtagu, můžeme vzít druhý sloupec následující matice:

```
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
```

Funkce `str_match_all()` vrací všechny výskyty regulárního výrazu. Jejím výsledkem je seznam matic. Rádky těchto matic odpovídají jednotlivým nálezům. Sloupce mají význam jako výše. Nenalezené prvky mají v tomto případě hodnotu `character(0)`.

```
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]
```

Příklad: převod českého telefonního čísla do standardního tvaru (budeme brát jen první číslo pro každého člověka):

```
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]
apply(cisla2, 1, function(x) str_c(x, collapse = "-"))
```

```
## [1] "777-666-555" "734-123-456" NA
```

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(s, p)` najde indexy prvního výskytu regulárního výrazu `p` v řetězci `s`. Výsledek je matice, jejíž řádky odpovídají prvkům vektoru `s`. První sloupce je index začátku prvního výskytu vzoru, druhý sloupec je index konce prvního výskytu vzoru. Pokud není vzor v řetězci nalezen, vrátí `NA`:

```
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 `s` slouží funkce `str_locate_all(s, p)`, která vrací seznam matic indexů. Prvky seznamu odpovídají prvkům vektoru `s`. Rádky každé matice odpovídají jednotlivým výskytům vzoru v jednom prvku vektoru `s`. První sloupec matice je index začátku výskytu, druhý sloupec je index konce výskytu. Pokud není vzor nalezen, vrací matici s nula řádky:

```
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 viz dokumentace.

Nahrazení vzoru

K nahrazení vzoru ve vektoru řetězců slouží funkce `str_replace()` a `str_replace_all()`. Funkce `str_replace(s, p, r)` nahradí ve vektoru řetězců `s` první výskyt vzoru `p` řetězcem `r`. Funkce `str_replace_all()` má stejnou syntaxi, ale nahradí všechny výskyty vzoru ve vektoru `s`.

Řekněme, že chceme v každém tweetu nahradit všechna klíčová slova řetězcem "'XXX":

```
r <- "#[[:alpha:]]+"
str_replace_all(tweet, r, "#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."
```

Klíčové slovo vymažeme tak, že je nahradíme prázdným řetězcem `""`:

```
str_replace_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 `p` a nahrazující řetězec `r` vektorem řetězců: jména prvků vektorů určují, co se nahrazuje, hodnoty určují, čím se nahrazuje:

```
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"
```

Obě funkce 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 zdvojit klíčová slova v tweetech upravit tak, že slovo bude nejdříve uvedeno bez hashtagu, a pak v závorce s hashtagem:

```
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."
```

Příklad: máte vektor datumů v anglosaském formátu, tj. “MM-DD-YYYY”, a chcete je převést do českého formátu “DD. MM. YYYY”:

```
datumy <- c("born: 06-01-2001", "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. 2001"    "died: 01. 02. 2017"    "no information at all"
```

Rozdělení řetězců podle vzoru

Často je potřeba rozdělit řetězec do několika částí oddělených nějakým vzorem. K tomu slouží funkce `str_split(s, p, n)` a `str_split_fixed(s, p, n)`, které rozdělí řetězec `s` v bodech, kde je nalezen vzor `p`. 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 `s`. Funkce `str_split_fixed()` musí mít zadaný počet `n` a vrací matici, jejíž řádky odpovídají prvkům vektoru `s` a sloupce jednotlivým nálezům (přebytečné sloupce jsou naplněné prázdným řetězcem `""`).

```
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"
```

Pokud je počet `n` nedostatečný, je nerozdělený zbytek řetězce vložen do posledního zadaného sloupce.

Příklad: rozdělit dobře formátované datum v české konvenci na den, měsíc a rok:

```
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" ""   ""
```

Příklad: rozdělit datum ve tvaru “DD.MM.YYYY” nebo “DD. MM. YYYY” nebo “DD-MM-YYYY” na den, měsíc a rok:

```
datum <- c("1-5-1999", "1.6.2001", "1. 2. 2003", "blábol")
str_split_fixed(datum, "(\\.\\s*|-)", 3)
```

```
##      [,1]     [,2] [,3]
## [1,] "1"      "5"  "1999"
## [2,] "1"      "6"  "2001"
## [3,] "1"      "2"  "2003"
## [4,] "blábol" ""   ""
```

Extrakce slov

Funkce `word()` rozdělí řetězec na slova a vrátí slova se zadanými indexy. Použití je

```
word(s, start, end, sep)
```

kde `s` 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 `fixed(" ")`, tj. 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.

Modifikace chování regulárních výrazů

Chování regulárních výrazů je možné ve funkcích z balíku **stringr** modifikovat pomocí čtyř funkcí:

- `fixed(pattern, ignore_case = FALSE)` porovnává řetězec doslovně, takže znaky bere doslovně jako byty. Je to rychlé, ale nemusí to vždy fungovat pro ne-ASCII znaky. Pokud se má např. tečka chápat doslovně jako tečka a ne jako jakýkoli znak, stačí ji zabalit: `fixed(".")`.
- `coll(pattern, ignore_case = FALSE, locale = NULL, ...)` porovnává řetězce tak, že respektuje standard collation rules.
- `regex(pattern, ignore_case = FALSE, multiline = FALSE, comments = FALSE, dotall = FALSE, ...)` je použit implicitně. Používá regulární výrazy podle standardu ICU a umožňuje je mírně modifikovat.
- `boundary(type = c("character", "line_break", "sentence", "word"), skip_word_none = TRUE, ...)` hledá hranice mezi věcmi. Např. `boundary("word")` hledá hranice mezi slovy.

Argumenty funkcí mají následující významy:

- `pattern` je vzor (regulární výraz), jehož chování chceme modifikovat
- `ignore_case` zda se má ignorovat rozdíl mezi malými a velkými písmeny (`TRUE` chápe malá a velká písmena jako zaměnitelná)
- `locale` určí locale, které se má použít pro porovnávání, viz `stri_locale_list()`; implicitně systémové locale
- `multiline` – pokud je `TRUE`, pak `$` a `^` matchují začátek a konec řádku; pokud je `FALSE` (což je implicitní hodnota), pak mačují začátek a konec celého řetězce
- `comments` – pokud je `TRUE`, pak se ignorují mezerové znaky a komentáře začínající `#`; doslovné mezery je třeba escapovat pomocí zpětného lomítka
- `dotall` – pokud je `TRUE`, tečka znamená nejen znaky, ale i konec řádku
- `type` označí typ hranice, která se hledá; může nabývat hodnot `"character"`, `"line_break"`, `"sentence"`, `"word"`
- `skip_word_none` ignoruje "slova", která neobsahují žádná písmena nebo číslice, tj. punctuation.

Příklady z dokumentace:

```
pattern <- "a.b"
strings <- c("abb", "a.b")
str_detect(strings, pattern)
```

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

```r
str_detect(strings, fixed(pattern))
```

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

```r
str_detect(strings, coll(pattern))
```

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

```r
# coll() is useful for locale-aware case-insensitive matching
i <- c("I", "\u0130", "i")
i
```

```
## [1] "I" "İ" "i"
```

```r
str_detect(i, fixed("i", TRUE))
```

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

```r
str_detect(i, coll("i", TRUE))
```

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

```r
str_detect(i, coll("i", TRUE, locale = "tr"))
```

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

```r
# Word boundaries
words <- c("These are   some words.")
str_count(words, boundary("word"))
```

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

```r
str_split(words, " ")[[1]]
```

```
## [1] "These"  "are"    ""       ""       "some"   "words."
```

```r
str_split(words, boundary("word"))[[1]]
```

```
## [1] "These" "are"   "some"  "words"
```

```r
# Regular expression variations
str_extract_all("The Cat in the Hat", "[a-z]+")
```

```
## [[1]]
## [1] "he"  "at"  "in"  "the" "at"
```

```r
str_extract_all("The Cat in the Hat", regex("[a-z]+", TRUE))
```

```
## [[1]]
## [1] "The" "Cat" "in"  "the" "Hat"
```

```r
str_extract_all("a\nb\nc", "^.")
```

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

```r
str_extract_all("a\nb\nc", regex("^.", multiline = TRUE))
```

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

```r
str_extract_all("a\nb\nc", "a.")
```

```
## [[1]]
## character(0)
```

```r
str_extract_all("a\nb\nc", regex("a.", dotall = TRUE))
```

```
## [[1]]
## [1] "a\n"
```

Funkce, které vrací seznamy

Mnoho funkcí z balíku **stringr** vrací seznamy. Oddíl "Functions that return lists" ve vinětě k balíku **stringr** ukazuje, jak dál jak dál zpracovávat výstupy těchto funkcí.

Domácí úkol

Tento domácí úkol je inspirovaný skutečným problémem. Máte dataset `shipping_data`, který obsahuje dva sloupce: `shop` s jednoznačným id internetového obchodu a `shipping` s dopravními náklady. Potíž je v tom, že dopravní náklady jsou kódované jako poněkud volně formátovaný řetězec. V tomto řetězci mohou být dopravní náklady za libovolný počet dopravních společností a jiná slova a čísla. Údaj pro jednu dopravní společnost má tvar `jméno_společnosti: částka Kč` nebo `jméno_společnosti: zdarma`, ovšem počet mezer může být libovolný a mezery mohou i úplně chybět. Údaje pro jednotlivé společnosti jsou odděleny mezerami nebo čárkami. Jméno každé dopravní společnosti se skládá vždy z několika velkých písmen, např. CP, DPD apod. Dataset obsahuje dopravní společnosti CP, DPD, PPL a ještě jednu dopravní společnost, jejíž jméno musíte zjistit z dat (bude jiné ve cvičných datech a jiné při testování vašeho skriptu). Původní data tedy vypadají takto (vypisují se jen vybrané řádky):

```
##   shop                          shipping
## 1  190                             100 Kč
## 2  416              doprava PPL:   30Kč
## 3  420
## 4  749 CP:50 Kč, PPL: 30 Kč,  ABC:60Kč
## 5  813     PPL:zdarma CP:80Kč DPD:40Kč
```

Vaším úkolem je

- zjistit jména všech dopravních společností a uložit je jako vektor řetězců do proměnné `delivery_names`
- přidat do datasetu numerické sloupce s dopravními náklady pro jednotlivé dopravní společnosti; každý sloupec se bude jmenovat `s_jméno_dopravní_společnosti` (např. `s_PPL`); slovo "zdarma" nahradí 0; pokud daný obchod dopravní společnost nevyužívá, bude hodnota `NA`

- přidat do datasetu sloupec `min_shipping`, který bude obsahovat minimální dopravní náklady, tj. minimum z dopravních nákladů pro daný obchod bez `NA`, nebo `NA`, pokud jsou všechny dopravní náklady `NA`

Na pořadí sloupců v datasetu nezáleží. Očekávaný výstup je tedy následující:

```
##   shop                          shipping s_PPL s_CP s_DPD s_ABC min_shipping
## 1  190                             100 Kč    NA   NA    NA    NA           NA
## 2  416              doprava PPL:   30Kč    30   NA    NA    NA           30
## 3  420                                    NA   NA    NA    NA           NA
## 4  749 CP:50 Kč, PPL: 30 Kč,  ABC:60Kč    30   50    NA    60           30
## 5  813     PPL:zdarma CP:80Kč DPD:40Kč     0   80    40    NA            0
```

Upravte skript `hw07.R`. Data načtěte ze souboru `shipping_data.RData`. Tento soubor obsahuje nejen dataset `shipping_data`, ale také dataset `shipping_data_check`. Tento druhý dataset ukazuje, jak by měl vypadat výsledek vaší práce – svůj skript tedy můžete pomocí něj otestovat. Ale pozor: tento druhý dataset nebude přítomný při testování vašeho skriptu, takže se na něj ve skriptu `hw07.R` nijak neodkazujte. (Pro testování si napište jiný skript.)

Rady (nejdříve to zkuste bez nich):

- budete potřebovat pracovat s regulárními výrazy, s funkcemi typu apply a konvertovat datové typy (a možná i struktury)
- když získáte vektor se jmény dopravních společností, můžete nad ním iterovat pomocí `lapply()` nebo jiné podobné funkce
- pokud nebudete umět získat jména dopravních společností, iterujte nad vektorem `c("CP", "DPD", "PPL")`; sice tím ztratíte některá data, ale aspoň některá získáte
- nepanikařte; toto je zatím nejtěžší domácí úkol, ale stačí na něj cca 15 řádků kódu (váš kód ale může být klidně kratší nebo delší a stále fungovat)