

Úvod do analýzy dat v R

Aleš Vomáčka

Jaromír Mazák

13. 1. 2023

Obsah

Předmluva	7
R, Rstudio, Tidyverse	8
R	8
Rstudio	9
Tidyverse	9
Proč ne grafická rozhraní?	9
Struktura knihy	11
Data	12
Countries	12
Dogs	12
Ukraine	13
I Trocha teorie	14
1 Instalace softwaru	15
1.1 Instalace R	15
1.1.1 Windows	15
1.1.2 MacOS	16
1.1.3 Linux	16
1.2 Instalace Rstudia	17
1.3 Instalace Tidyverse	17
1.4 Základní nastavení Rstudia	18
2 První pohled na Rstudio	20
2.1 Orientace v Rstudiu	20
2.2 Nastavení	21
3 Jak si organizovat práci	23
3.1 Rstudio projekty	23
3.2 Organizace projektů	24
3.3 Pojmenovávání souborů	25
3.4 Kódovací styly	26

4	Typy objektů	27
4.1	Atomové vektory	27
4.2	Faktory	29
4.3	Matice a tabulky	30
4.4	Listy	31
4.5	Dataframy	32
5	Jména objektů	33
5.1	Přiřazování jmen	33
5.2	Pravidla pojmenovávání	35
6	Funkce	36
6.1	Používání funkcí	36
6.2	Argumenty funkcí	37
6.3	Funkce a vnořené objekty	38
6.4	Řetězení funkcí	41
6.5	Dokumentace funkcí	43
6.6	Vytváření vlastních funkcí	43
7	R balíčky	46
7.1	Instalace balíčků	46
7.2	Nahrání balíčku	46
7.3	Konflikty mezi balíčky	47
7.4	Kde hledat balíčky	48
II	Manipulace s dataframy	49
8	Import a export dat	50
8.1	Pracovní adresář	50
8.2	Import dat	51
	8.2.1 Comma seperated values	51
	8.2.2 RDS	53
	8.2.3 SPSS a Stata	53
8.3	Export dat	53
9	První pohled na dataframe	55
9.1	Pohled na dataframe	55
9.2	Sumarizace dataframu	56
10	Práce se sloupci	59
10.1	Výběr sloupců	59
10.2	Pomocné funkce	60
10.3	Přejmenovávání proměnných	61

10.4	Pořadí proměnných	62
11	Práce s řádky	64
11.1	Filtrování řádků	64
11.2	Řezání dataframů	66
11.3	Group_by()	67
11.4	Pořadí řádků	68
12	Široký a dlouhý formát	70
12.1	Z širokého do dlouhého formátu	70
12.2	Z dlouhého do širokého formátu	72
13	Spojování dataframů	74
13.1	Spojovací funkce	74
13.2	Kterou spojovací funkci použít?	77
III	Manipulace s proměnnými	79
14	Transformace proměnných	80
14.1	Jednoduché transformace	80
14.2	Transformace po skupinách	82
14.3	Řádkové operace	83
14.4	Podmíněné transformace	84
15	Sumarizace proměnných	87
15.1	Jednoduchá sumarizace	87
15.2	Sumarizace po skupinách	87
16	Transformace a sumarizace více proměnných	89
16.1	Transformace většího množství proměnných	89
16.2	Sumarizace většího množství proměnných	91
16.3	Analýza po skupinách	93
16.4	Sumarizace více proměnných bez použití across()	95
17	Práce s faktory	97
17.1	Vytváření faktorů	97
17.2	Pořadí úrovní	98
17.3	Transformace úrovní	101
18	Práce se stringy	104
18.1	Detekce stringů	104
18.2	Separace stringů	105
18.3	Transformace stringů	106

IV Vizualizace dat	108
19 Struktura grafů	109
19.1 Grammar of graphics	109
19.2 Struktura ggplot2 grafů	109
20 Vizualizace kategorických proměnných	113
20.1 Vizualizace jedné proměnné	113
20.2 Vizualizace více proměnných	116
21 Vizualizace numerických proměnných	121
21.1 Vizualizace jedné proměnné	121
21.2 Vizualizace více proměnných	125
22 Kombinované grafy	128
22.1 Boxploty	128
22.2 Bodové grafy	129
22.3 Histogramy a grafy hustoty	131
23 Facety	134
23.1 Jednorozměrné facety	134
23.2 Vícerozměrné facety	135
24 Vzhled grafů	138
24.1 Barvy	139
24.2 Tvar	146
24.3 Velikost a průhlednost	146
24.4 Formátování os	147
24.5 Nadpisy, názvy a poznámky	150
24.6 Celková tematika grafu (<i>themes</i>)	150
25 Pokročilé grafy	154
25.1 Polární koordináty	154
25.2 Skládání geomů	156
25.3 Více zdrojů dat	158
26 Export grafů	161
26.1 Export pomocí ggsave()	161
26.2 Rasterová versus vektorová grafika	162

V Pokročilé R	164
27 Vlastní funkce	165
27.1 Počet chybějících hodnot v proměnné	165
27.2 Graf pro likertovské položky	168
28 For loops (cykly)	173
28.1 Kdo je členem gangu?	173
28.2 Průměr každé numerické proměnné	175
28.3 Histogram pro každou numerickou proměnnou	177
28.4 Boostraping	180
29 Co dál?	183
29.1 Statistika	183
29.2 Vizualizace dat	183
29.3 Oragnizace práce	183
29.4 Na vše ostatní je tu Big Book of R	184

Předmluva

Učit se R je běh na dlouhou trať. Je to cesta, která znamená mnohem větší časovou investici než zvládnutí softwaru s GUI, jako je například SPSS. Odměnou je mnohem větší flexibilita a v ruce univerzální nástroj pro zpracování dat, analýzu, vizualizaci, ale i programování a automatizaci. R je každý den využíváno nespočtým množstvím studentů, výzkumníků a dalších odborníků pro analýzu dat, ale i vytváření webových stránek a aplikací nebo psaní knih (včetně této!). Jak bylo kdysi proneseno na dnes již zapomenutém kousku internetu:

Evelyn Hall: I would like to know how (if) I can extract some of the information from the summary of my nlme.

Simon Blomberg: This is R. There is no if. Only how.

—Evelyn Hall and Simon ‘Yoda’ Blomberg, R-help (April 2005)

Tento text je průvodcem pro návštěvníky poprvé vstupující do světa R, který je provede základy datové analýzy od instalace všeho nezbytného softwaru, až po manipulaci a vizualizaci dat. Slouží zároveň jako podklad pro výuku kurzu [Úvod do analýzy dat v R](#) na katedře sociologie FF Univerzity Karlovy. Vznik tohoto textu byl podpořen [NMS Market Research](#).

Tato kniha je licencovaná pod [Creative Commons Attribution-NonCommercial 2.0 Generic](#) a je možné ji volně šířit pro nekomerční účely.

R, Rstudio, Tidyverse

Pokud čtete tuto knihu, jste pravděpodobně odhodlaní vrhnout se do světa analýzy dat. Nováčkům ve světě R ale hrozí při první návštěvě jisté zmatení. Zničehonic se na ně ze všech stran začne valit řada nových pojmů, ve kterých se může ne jeden začátečník ztratit. To je zcela pochopitelné, R existuje již více než 20 let a za tu dobu se kolem něj rozrostl bohatý ekosystém rozšíření, organizací a akcí. Předtím, než se pustíme do detailů, si proto představíme tři pojmy, které by všichni uživatelé R měli znát: R, Rstudio a Tidyverse.

R

R je volně šiřitelný, otevřený programovací jazyk zaměřený specificky na vizualizaci a statistickou analýzu dat. Jeho počátky sahají do poloviny 90. let minulého století, kdy začal být vytvářen dvěma pracovníky Aucklandské university, Rossem Ihakem a Robertem Gentlemanem. Od té doby se stal jedním z nejpobulárnějších jazyků pro analýzu dat a drží se mezi nejpobulárnějšími jazyky vůbec. R je široce využíváné jak pro akademický výzkum, tak v komerční i veřejné sféře.

Zatímco funkčnost jiných statistické programů, jako například SPSS nebo Excel, je omezená na autory před-připravené nástroje a postupy, možnosti R jsou téměř neomezené. Kromě základních i pokročilých statistických analýz je možné R využít k psaní knih, článků, internetových stránek (včetně této) nebo webových aplikací. R je také využíváno řadou předních statistiků, jejichž práce se týká i sociálněvědního výzkumu.

i Česká stopa v R

Přestože jeho počátky sahají na Nový Zéland, do historie R se významně zapsalo i několik rodáků z Česka. Jedním ze současných členů hlavního vývojářského týmu je [Šimon Urbánek](#), který se mimo jiné zasloužil o vytvoření R verze pro MacOS. Verzi pro Windows, a nejen ji, spravuje druhý český člen týmu, [Tomáš Kalibera](#). Neobyčejně velkou měrou přispěl k rozvoji R i [Jan Vítek](#).

Cenou za široké možnosti R jsou vyšší nároky na jeho osvojení, jelikož na rozdíl od programů s grafickým rozhraním vyžaduje práce s R alespoň základní znalosti programování. Naštěstí již dnes existuje řada zdrojů a komunit, které mohou s tímto problémem pomoci. Pro inspiraci můžeme zmínit blog [R-bloggers](#), skupinu [Rladies](#) nebo komunitní projekt [Tidytuesday](#).

Rstudio

Dávno už jsou pryč časy, kdy práce s programovacími jazyky znamenala psaní kódu v jednoduché příkazové řádce (nebo nedej bože prorážení dírkovacích štítků!). Dnešní uživatelé mohou využívat sofistikovaných programů, jejich cílem je usnadnit každodenní práci. Těmto programům se říká integrované vývojářské prostředí (*Integrated Development Environments*) a jasně nejpopulárnějším *IDE* pro R je v současné době Rstudio.

Rstudio, vyvíjené stejnojmennou společností, bude kontrolovat váš kód, napovídat vám jména funkcí, exportovat grafy a mnoho dalšího. Nejedná se samozřejmě o jediné vývojářským prostředí pro R (konkurenty jsou například [VScode](#), [Vim](#) nebo [ESS Emacs](#)), představuje však výbornou rovnováhu mezi výkonem a uživatelskou přívětivostí a skvělou volbou pro všechny nováčky.

Tidyverse

Přestože jsou možnosti R nesmírně široké, zdaleka ne všechny nástroje vám budou k dispozici hned po jeho instalaci. Většina rozšíření pro R je distribuovaná formou balíčků (*packages*), které jsou volně dostupné ke stažení.

Balíčků, které do R přináší nové funkce, dnes existují desítky tisíc. Jednou z nejpopulárnějších rodin takových balíčků je [Tidyverse](#), která rozšiřuje možnosti R zejména v oblastech manipulace a vizualizace dat. Mnoho úkonů, které se v základním R provádí velmi zdlouhavě nebo krkolomně, jsou v Tidyverse záležitostí na jeden dva řádky. Jednotlivé balíčky jsou také designovány tak, aby si spolu navzájem rozuměli a využívali identickou syntax. Stinnou stránkou živelné popularity R je, že řada jeho vývojářů má značně rozdílné představy o psaní počítačového kódu. To vede k mnoha různým konvencím, které jsou pro běžného uživatele matoucí (např. mají funkce začínat velkým, nebo malým písmenem? mají se slova oddělovat potřítkem, nebo tečkou?). Všechny balíčky Tidyverse se drží jednotného stylu a je proto velice jednoduché jejich funkce kombinovat bez zbytečných zmatků. Jedná se tak o další způsob, jak ulehčit ponoření do R.

Je nutné zmínit, že využívání Tidyverse není striktně nutné. Tidyverse vzniklo dlouho po vzniku samotného R a do dneška existuje mnoho uživatelů, pro které představuje základní R (nebo jiné balíčky pro analýzu dat) ideální pracovní prostředí. Nicméně, stejně jako u Rstudia, Tidyverse představuje skvělou rovnováhu mezi uživatelskou přívětivostí a flexibilitou práce.

Proč ne grafická rozhraní?

Nakonec si dovolíme krátce vyjádřit k tématu, pravidelně objevuje vždy, když dojde na výuku analýzy dat.

Na začátku jsme zmínili, že R je programovací jazyk a pro práci s ním je nutné znát základy programování. To striktně řečeno není úplně pravda. Protože je v R možné udělat téměř cokoli, je možné v něm vytvořit i grafické rozhraní, a tím práci s ním přiblížit “klikacímu” softwaru jako je SPSS. Těchto rozhraní již dnes existuje celá řada, mezi nejpůvodnější se řadí například **Jamovi** a **JASP**. S využitím těchto rozhraní je možné analýzy “naklikávat” místo psaní programovacího kódu, čímž se výrazně snižuje vstupní bariéra. Grafická rozhraní ale podle našeho názoru mají tři velké slabiny, které dříve nebo později převáží nad jakýmkoliv potencionálními výhodami:

- **Grafická rozhraní nikdy nepokryjí vše, co R nabízí a co potřebujeme:** Přestože většina grafických rozhraní nabízí široké možnosti pro jednoduchou manipulaci s daty a základní statistické postupy, žádné z nich nepokrývá všechny potřeby průměrného výzkumníka, což platí zvláště pro pokročilejší analýzy. Jinak řečeno, ten kdo se rozhodne vážněji zabývat kvantitativní analýzou, se dříve nebo později alespoň lehkému programování nevyhne. A čas do té doby strávený v grafických rozhraních mu v tu chvíli nebude příliš užitečný.
- **Grafické rozhraní jsou časově neefektivní:** Nedokonalá nabídka není jediným problémem grafických rozhraní. I kdyby v nich byly obsaženy všechny nezbytné funkce, práce s grafickými rozhraními bude v dlouhodobém horizontu vždy pomalejší, než psaní kódů. Jedním z největších výhod, které počítače přinášejí, je možnost automatizace. Proč ručně vytvářet tučt kontingenčních tabulek nebo kontrolovat desítky proměnných, pokud to může počítač udělat za nás? Práce se skriptem nám umožní zadat počítači příkaz a nechat ho, ať ho sám provede na jakkoliv velkém počtu případů. Takovéto efektivitu grafická rozhraní zkrátka nikdy nedosáhnou.
- **Klikání svádí k nereprodukovatelnosti:** Počítačový skript není jen způsob, jak počítači říkat, co má dělat. Jedná se zároveň o detailní záznam celé naší práce. Kdokoli, ať už mi sami nebo lidé se zájmem o naši práci, se mohou v budoucnu podívat, jak jsme v analýze postupovali a případně se naší prací inspirovat nebo ji vylepšit. To je nejen skvělý nástroj pro ušetření času, ale i cesta ke zkvalitnění vědeckého výzkumu jako takového. Práce v grafickém rozhraní bohužel řadu lidí svádí k rychlému naklikávání, po kterém často nezůstane nic kromě řady pochybných výsledků, jejichž původem si není nikdo jistý.

Struktura knihy

Tato kniha je rozdělena do pěti sekcí.

První sekce názvem **Trocha teorie** vám pomůže nainstalovat všechny nezbytný software, předá vám tipy pro organizaci práce a seznámí vás se základním fungováním R jako programovacího jazyka.

V druhé sekci **Manipulace s dataframy** se seznámíme s importem a exportem dat a se základní manipulací s daty, jako filtrování sloupců a řádků datasetů a se převodem dat mezi širokým a dlouhým formátem.

Třetí v pořadí je sekce **Manipulace s proměnnými** si ukážeme jak transformovat a sumarizovat proměnné. Krátce se také dotkneme práce s strukturovaným a nestrukturovaným textem.

Čtvrtou sekcí je **Vizualizace dat**, jejímž obsahem je vytváření základních i pokročilých grafů, upravování jejich vzhledu a nakonec jejich export z R pro další použití.

Poslední sekce nese název **Pokročilé R** a krátce si v ní představíme komplexnější, ale o to užitečnější funkce R, jmenovitě vytváření vlastních funkcí a *for loop* cykly.

Data

Tato kniha využívá několik datasetů.

Countries

Prvním a hlavním je dataset `countries`. Dataset je možné stáhnout [zde](#) (klikněte pravým tlačítkem na odkaz a zvolte *Uložit jako...*). Popis proměnných naleznete v následující tabulce.

Jméno proměnné	Popis	Zdroj
<code>country</code>	Jméno země	
<code>code</code>	Dvoumístný ISO kód země	
<code>gdp</code>	HDP země v milionech euro (2018)	Eurostat
<code>population</code>	Populace země k 1. lednu 2018	Eurostat
<code>area</code>	Celková rozloha země	CIA factbook
<code>eu_member</code>	Je země členem Evropské unie? (2019)	Evropská unie
<code>postsoviet</code>	Byla země součástí Východního bloku?	Wikipedie
<code>life_exp</code>	Naděje na dožití při narození (2017)	OSN
<code>uni_prc</code>	Podíl lidí s vysokoškolským vzděláním ve věku 15 až 64 let (2018)	Eurostat
<code>poverty_risk</code>	Podíl lidí ohrožených chudobou (2017)	Eurostat
<code>material_dep</code>	Podíl lidí s materiální deprivací, 3 nebo méně položek (2017)	Eurostat
<code>hdi</code>	Index lidského rozvoje (2018)	OSN
<code>foundation_date</code>	Datum vzniku/zformování země	Wikipedie
<code>maj_belief</code>	Největší náboženská skupina v zemi (2018)	Pew Researcher Center

Dogs

Dataset věnovaný psím plemenům, původně z projektu [TidyTuesday](#). Jedná se o dva datasety, [první](#) obsahuje vlastnosti psích plemen, [druhý](#) popularitu plemen v čase (pro stažení klikněte pravým tlačítkem na odkaz a zvolte *Uložit jako...*).

Ukraine

Poslední data se týkají postojů občanů České republiky k válce na Ukrajině z března 2022. Data pochází z dotazníkového šetření Centra pro výzkum veřejného mínění Akademie věd. [První](#) dataset obsahuje odpovědi respondentů, [druhý](#) popis jednotlivých proměnných.

Část I

Trocha teorie

1 Instalace softwaru

Tato kapitola je věnována instalaci R, Rstudia a balíčků Tidyverse na všech platformách a pár základním tipům na usnadnění práce. Instalace softwaru se značně liší podle operačního systému počítače, takže si dejte pozor, která část instrukcí je pro vás relevantní!

Celý proces sestává ze čtyř kroků:

1. Instalace R
2. Instalace Rstudia
3. Instalace Tidyverse
4. Základní nastavení Rstudia

1.1 Instalace R

Úplně prvním krokem pro práci s R je, snad nepřekvapivě, instalace jazyka samotného.

1.1.1 Windows

R je možné stáhnout z oficiálních stránek projektu <https://www.r-project.org>. Čeští uživatelé budou pravděpodobně chtít stahovat z českého serveru, který je dostupný na adrese <https://mirrors.nic.cz/R/index.html>. Zde klikněte na *Download R for Windows* a poté na *base*. Po stažení je možné R nainstalovat jako jakýkoliv jiný program.

Varování

Možná víte, že existují dva typy procesorů, 32bitové a 64bitové. Setkat se dnes s 32bitovým procesorem je dnes poměrně vzácné, pokud ale takový počítač máte je nutné si dát pozor, jakou verzi R instalujete. Poslední verze R, která podporuje 32bitové procesory je 4.1., což *není* ta nejnovější. Pokud naopak máte 64bitový procesor, můžete si s klidem nainstalovat aktuální verzi R. Pokud si nejste jistí, můžete si procesor svého počítače [zkontrolovat v nastavení](#). Obdobné omezení se týká i Rstudia, které podporuje 32bitové procesory pouze do verze 1.2.

Ti z vás, kteří preferují instalaci softwaru přes manager, mohou využít [Chocolatey](#), Po [instalaci manageru](#) samotného stačí do terminálu zadat

```
choco install r
```

Upozorňujeme ale, že z naší zkušenost se správčům Chocolatey balíčku ne vždy daří držet krok s aktuální oficiální verzí .

1.1.2 MacOS

R je možné stáhnout stejně jako u Windows verze z <https://mirrors.nic.cz/R/>, kde zvolte *Download R for MacOS*. Pokud máte Macbook s M1 procesorem (tedy Macbook z roku 2020 nebo mladší), zvolte verzi *arm64*. U starších verzí zvolte základní verzi R. Po stažení je možné nainstalovat jako jakýkoliv jiný software.

Pokud preferujete instalaci pomocí software manageru, je možné využít [Homebrew](#). Pro instalaci Homebrew otevřete terminál a použijte příkaz

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

a řiďte se instrukcemi. Po úspěšné instalaci Homebrew je možné nainstalovat R pomocí příkazu:

```
brew install r
```

1.1.3 Linux

Konkrétní podoba instalace R pro Linuxu závisí na tom, jakou distribuci používáte. R oficiálně podporuje tři distribuce a to Debian, Ubuntu a Fedora/Redhead. Instrukce pro instalaci jsou dostupné na <https://mirrors.nic.cz/R/>. Velkou pozornost doporučujeme věnovat repozitářům pro instalaci balíčků, jelikož jejich napojení se liší distribuci od distribuce a jejich správnou přípravou si do budoucna ušetříte mnoho času!

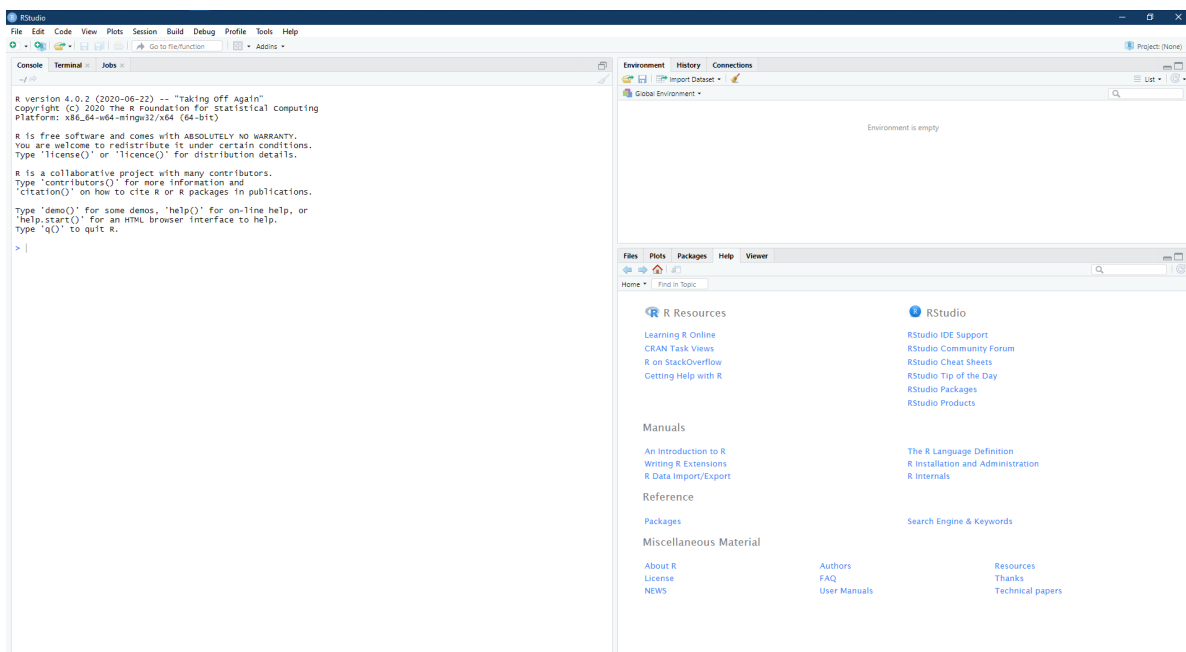
Pokud používáte jinou než jednu z oficiálně podporovaných distribucí, konzultuje svůj package manager.

1.2 Instalace Rstudio

Po instalaci R jste teoreticky připraveni pro další práci! Rychle ale zjistíte, že obsluhovat R pouze z příkazové řádky není ani zdaleka ideální. Analytici a vývojáři proto využívají řadu programů, zvané *Integrated Developer Environments (IDE)*, které psaní kódu usnadňují. Jedním z nejlepších (podle nás dokonce nejlepší!) IDE pro práci v R je **Rstudio**. Rstudio pro všechny operační systémy je dostupné ke stažení na <https://www.rstudio.com/products/rstudio/download/#download>. Instalace probíhá klasickým způsobem.

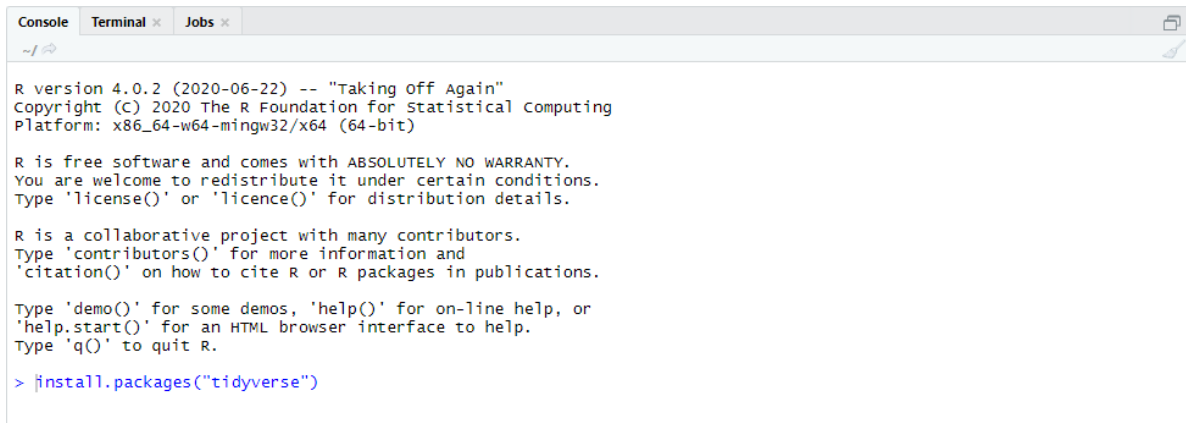
Ti z vás, kteří používají software manager, mohou využít Chocolatey pro Windows (`choco install r.studio`), Homebrew pro MacOS (`brew install rstudio`), případně konzultovat dokumentaci k vaší Linux distribuci.

Rstudio při prvním spuštění vypadá zhruba takto:



1.3 Instalace Tidyverse

Posledním krokem je instalace sady R balíčků Tidyverse, které budeme využívat v rámci této knihy. Tyto balíčky lze nainstalovat uvnitř Rstudia pomocí příkazu `install.packages("tidyverse")` zadaného do konzole na levé straně (nezapomeňte na uvozovky!):



```
Console Terminal x Jobs x
~/
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

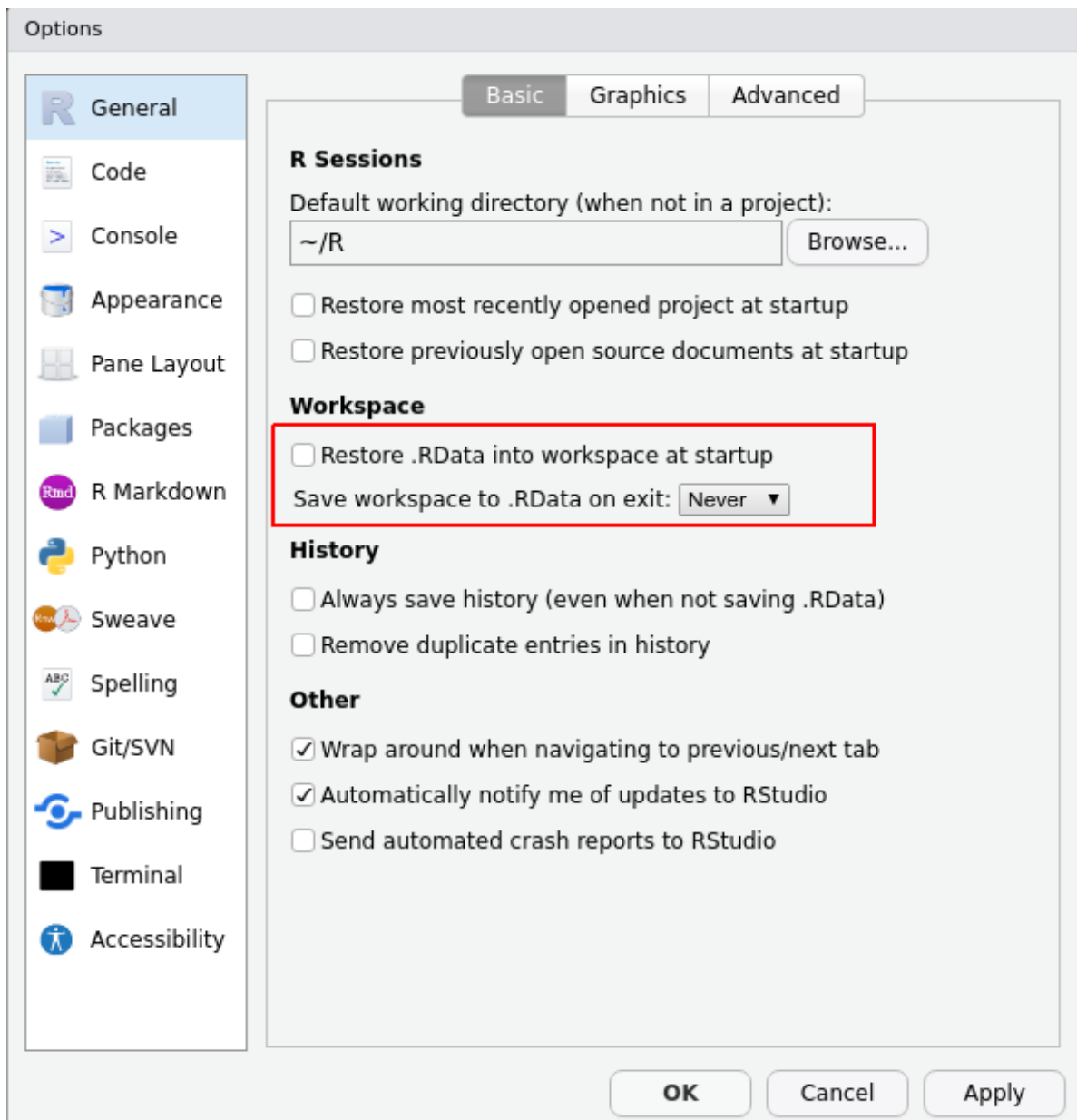
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |install.packages("tidyverse")
```

Instalace všech potřebných balíčků trvá zpravidla pár minut (s výjimkou některých uživatelů Linuxu, kteří musí balíčky kompilovat. Vy si počkáte zhruba tři čtvrtě hodiny). Úspěšná instalace bude zakončená větou `package tidyverse successfully unpacked and MD5 sums checked`.

1.4 Základní nastavení Rstudio

Jako úplně poslední věc se vyplatí změnit dvě výchozí nastavení, kterými si dlouhodobě ušetříte práci i nervy. Rstudio ve výchozím nastavení při ukončení ukládá všechny nahraná data a další vámi vytvořené objekty a znovunahraje je pokaždé, když R znovu zapnete. To je ovšem v praxi spíše na škodu, protože to znamená, že zanedlouho budete mít vaše prostředí zaneřádné daty z předchozích analýz a projektů. Tomu se dá jednoduše zabránit tím, že si v Rstudiu otevřete záložku *Tools*, a v ní volbu *Global options*. V tomto nastavení odškrtněte možnost *Restore .Rdata into workspace at startup* a zároveň nastavte možnost *Save workspace to .Rdata on exit* na *never* tak, jak je to na obrázku níže.

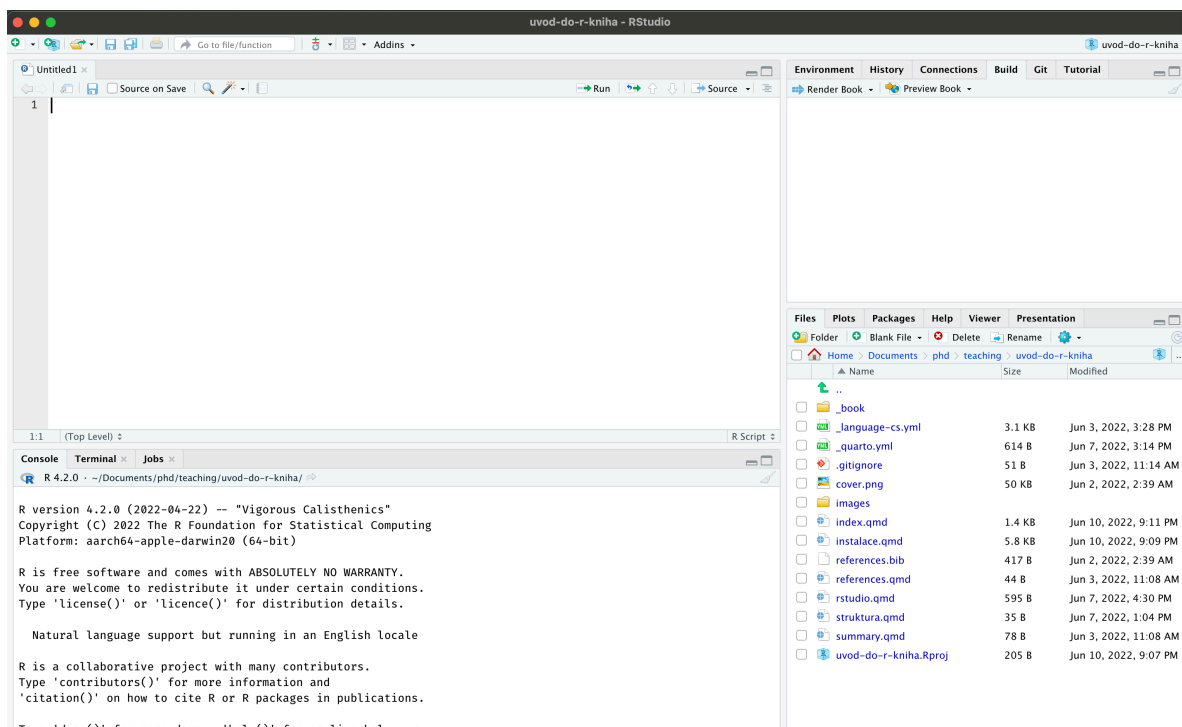


2 První pohled na Rstudio

Rstudio je *Integrated Developer Environment (IDE)*, nástroj pro ulehčení práce s programovacími jazyky. Jeho hlavní prací je pomoci vám s psaním kódu, hledáním chyb, instalací balíčků a mnohým dalším. Tato kapitola slouží jako základní představení Rstudia a některých užitečných funkcí.

2.1 Orientace v Rstudiu

Prvním krokem, který učiníte téměř vždy při zapnutí Rstudia, je otevření nového skriptu. To je možné udělat pomocí klávesové zkratky **Ctrl + Shift + N** na Windows a Linuxu, případně **Cmd + Shift + N** na MacOS. A nebo můžete kliknout na *File -> New File -> R Script* v pravém horním rohu. Vaše Rstudio bude vypadat zhruba takto:



Rstudio je rozděleno do čtyřech oblastí:

Skript se nachází v levém horním rohu a je jedním z nejdůležitějších součástí Rstudia. Zde budete psát kód říkající R co má dělat, od importu dat až vytváření statistických modelů. Většinu času strávíte právě v tomto okně.

Komentování kódu

Jednou z nejužitečnějších vlastností skriptu je možnost komentovat si vlastní kód. Komentář vytvoříte pomocí **#**. R bude vše za tímto znakem ignorovat, čímž se vám otevírá prostor pro vlastní poznámky. Například:

```
# Dnes se seznámíme s R
```

Konzoli najdeme v levém dolním rohu. V konzoli bude R zobrazovat výsledky vašeho kódu, upozornění a chybové hlášky, takže se ji vyplatí pozorně sledovat! I zde můžete zadávat instrukce pro R, ale pouze řádek po řádku, takže budeme preferovat psání kódu ve skriptu.

V pravé vrchní části se nachází okno sloužící hned několika účelům. Tím nejdůležitějším pro začínající analytiku je **Environment**, ve kterém uvidíte importovaná data a další objekty, které během své práce vytvoříte.


Poslední část Rstudia je v pravé spodní části a najdeme v ní hned několik užitečných věcí. Tou první je záložka **Files**, ve které uvidíme obsah vašeho současného *pracovního adresáře* (o tom později). Druhou důležitou záložkou je **Plots**, kde se budou zobrazovat vámi vytvořené grafy. Poslední záložkou je **Help**, obsahující dokumentaci k R a jeho funkcím. Do ní zavítáte pokaždé, pokud si nebudete jistí jak některá z funkcí funguje.

2.2 Nastavení

Přestože Rstudio je možné bez větších problémů používat v jeho základním nastavení, v průběhu času si ho pravděpodobně budete chtít pro větší pohodlí. Většina nastavení Rstudia je dostupná v záložce *Tools -> Global Options* na vrchní liště. Pokud jste se řídili našimi instrukcemi pro instalaci (viz Kapitola 1), tak už jste do nastavení Rstudia na chvíli zavítali, abyste vypnuli automatické ukládání pracoviště. Pokud jste tak ještě neučinili, silně doporučujeme to udělat teď.

Možnosti nastavení, které Rstudio nabízí, jsou široké a doporučujeme si je v klidu všechny projít. Pro začátek doporučujeme věnovat pozornost třem záložkám, a to **General**, **Appearance** a **Pane Layout**. V záložce **General**, kromě již zmíněného nastavení pro ukládání pracoviště, stojí za pozornost zejména *Default working directory (when not in a project)*. Toto je adresář, do kterého bude R ukládat všechny výstupy, pokud neřeknete jinak. Pokud si rádi udržujete na svém počítači pořádek, můžete R přesměrovat do vlastní složky.

V záložce Appearance je možné nastavit velikost a font písma a celkový vzhled Rstudia. Pokud je na vás výchozí písmo příliš malé nebo se vám nelíbí výchozí barevné schéma, zde je možné to napravit.

 Více schémat, víc!

Rstudio nabízí malý výběr barevných schémat (*themes*). Pokud vám není žádné z nich po chuti, je možné si vytvořit vlastní nebo si stáhnout schéma vytvořené jinými uživateli. K tomu je ideální stránka <https://tmtheme-editor.herokuapp.com>. Zde si můžete prohlédnout galerii schémat (gruvbox je naše oblíbené!) nebo si vytvořit vlastní. Jakmile najdete schéma, se kterým jste spokojení, stáhněte si ho a nainportujte ho v záložce Appearance.

Poslední záložkou, do které mnoho uživatelů zavítá jako do jedné z prvních, je Pane Layout. Zde si můžete upravit rozložení Rstudia. Chcete mít konzoli napravo od skriptu? Nepotřebujete některou ze záložek? Zde si můžete nastavit vše k vaší spokojenosti.

V menu nastavení samozřejmě najdete mnoho dalšího. Uživatelé Pythonu budou jistě potěšeni záložkou stejného jména. Programátoři mohou nastavit své version control nástroje v záložce Git/SVN. Pokud plánujete využívat Rstudio pro psaní reportů, záložky Rmarkdown a Spelling jsou pro vás. Nemusíte se ale stresovat, pokud pro většinu z těchto možností nevidíte v tuto chvíli využití. Postupem času se možná dostanete do situace, kdy se vám vyplatí s těmito možnostmi pohrát. Do té doby bude dobře sloužit výchozí nastavení.

3 Jak si organizovat práci

Pokud byste se zeptali, co dělá datového analytika dobrým analytikem, většina lidí by vám asi odpověděla, že je to dobrá znalost statistických technik. Další by možná řekli, že je to schopnost psát počítačový kód nebo vytvářet poutavé grafy. A všechny tyto věci opravdu jsou extrémně důležité. Neméně důležitou, zato však často opomíjenou schopností, je ale také schopnost efektivně organizovat svoji vlastní práci. Ta se týká všeho, od organizace souborů v počítači, až po pojmenovávání proměnných. Předtím než se vrhneme do R samotného, si tedy řekneme několik organizačních tipů.

3.1 Rstudio projekty

První tip se týká organizace souborů, se kterými pracujeme. Ty mohou být cokoli od datasetů, dotazníků a codebooků až po vytvořené grafy a reporty. Naše první rada je zde jednoduchá: **Každý projekt, na kterém pracujete, by měl mít svou vlastní složku.** To se může zdát triviální, znepokojivě velké množství lidí si ovšem osvojilo zvyk ukládat všechny své soubory v jedné obří složce. Ve výsledku to vede pouze ke zmatenému hledání, který z souborů pojmenovaný *data-kopie3.csv* obsahuje data, která hledáme.

Pokud používáte Rstudio, můžeme organizaci souborů usnadnit ještě o něco více. **Každý projekt by měl mít svůj vlastní Rstudio projekt.** Rstudio projekt je v podstatě jen složka, obsahující soubor s koncovkou *.Rproj*. To se může zdát triviální, tyto projekty jsou ale extrémně užitečné, protože umožňují Rstudiu automaticky nastavit pracovní adresář (s ním se setkáme při importu dat), ukládat historii vaší práce pro každý pracovní projekt zvlášť a mnoho dalšího. Jedná se tedy o skvělý nástroj, jak si udržet pořádek, zvláště pokud pracujete na několika projektech najednou.

Nový Rstudio projekt vytvoříte kliknutím na *File -> New Project...* v levém horním rohu. V otevřeném menu zvolte *New Directory -> New Project* a vyberte si název a adresu, kde má být projekt vytvořen. Pokud už máte složku, ze které byste chtěli vytvořit Rstudio projekt, stačí zvolit *Existing Directory*. Po vytvoření se nový Rstudio projekt automaticky otevře. Pokud byste Rstudio projekt zavřeli, můžete ho znovu otevřít v pravém horním rohu kliknutím na malou modrou ikonu R, za kterou následuje buď *Project: (None)*, případně název současně otevřeného projektu. Na tomto místě můžete také přepínat mezi projekty nebo je zavřít.

3.2 Organizace projektů

Teď, když je vaše práce organizovaná do (Rstudio) projektů, můžeme se zaměřit na to, jak organizovat jednotlivé soubory. **Udržujte přehlednou a konzistentní strukturu napříč všemi projekty.**

Všechny vaše soubory by v rámci jednoho projektu měli být roztrženy do srozumitelně pojmenovaných podsložek. Konkrétní struktura projektu závisí na osobních preferencích a povaze práce, nám se obecně osvědčilo následující schéma:

```
project/  
|-data-raw/  
|-data-cleaned/  
|-scripts/  
|-documentation/  
|-outputs/  
|-project.Rproj
```

Složka projektu by měla obsahovat alespoň následující podsložky a soubory. V podsložce *data-raw* jsou uchována data se kterými pracujeme, a to v takové podobě, v jaké se k nám dostala. Tato syrová data nikdy nepřepisujeme! Slouží jako poslední záchrana, pokud by bylo nutné provést všechny analýzy znovu od začátku. Naproti tomu, složka *data-cleaned* je pro již zpracovaná data. Nachází se zde vyčištěná data ze složky *data-raw*, připravená k další analýze. Data v té složce můžeme přepisovat, protože v případě potřeby je můžeme vždy znovuvytvořit pomocí našich skriptů. Skripty samotné bydlí ...(dramatická pauza)... ve složce *scripts*. Zde asi není nutné mnoho vysvětlovat. Ve složce *documentation* je uchována dokumentace k projektu. Zpravidla se jedná o PDF verze dotazníků, popis sběru dat, codebooky a podobně. Poslední složkou je *output*, do které ukládáme všechny naše výstupy, ať už se jedná o dílčí grafy nebo celé reporty. V kořenovém adresáři se nachází pouze soubor *.Rproj* (který Rstudiu říká, že tato složka je Rstudio projekt).

Na konci kurzu Úvodu do R by složka vašeho Rstudio projektu mohla vypadat zhruba nějak takto:

```
uvod-do-r/  
|-data-raw/  
  |-countries.csv  
  |-cvvm-cerven-2019.csv  
|-data-cleaned/  
  |-countries-clean.csv  
|-scripts/  
  |-01-import-export.R
```



```
| -02-data-manipulation.R
| -03-data-visualization.R
| -04-final-homework.R
|-documentation/
| -cvvm-codebook.pdf
|-outputs/
| -vomacka-intro-r-homework.docx
|-uvod-do-r.Rproj
```

Jakmile si najdete svou preferovanou strukturu svých souborů, dodržujte ji napříč všemi projekty. To vám umožní se rychle zorientovat i v projektech, na kterých jste řadu týdnů nebo dokonce měsíců nepracovali.

Samozřejmě, ne všechny projekty mohou mít úplně identickou strukturu a občas je nutné strukturu složek přizpůsobit konkrétnímu projektu. I projekt, který obsahuje kód k této knize, je organizovaný výrazně jinak! Výše popsané schéma ale z naší zkušenosti představuje solidní základ pro projekty, na kterých sociální vědci zpravidla pracují.

3.3 Pojmenovávání souborů

Pořádek se vyplatí udržovat nejen při organizaci souborů, ale i při jejich pojmenovávání. Nadevše ostatní **by vaše soubory měli mít srozumitelná, krátká jména**. Datové soubory by měli být pojmenované tak, aby bylo ze jména jasné, jaká data obsahují. Například, pro data sesbírána Centrem pro výzkum veřejného mínění v červnu 2019 preferujeme názvy jako `cvvm-cerven-2019.csv`, spíše než `V0619.csv`. Pro oddělení více slov doporučujeme používat buď `-` nebo `_`, naopak se vyhýbejte mezerám. Přestože v dnešní době si většina programů dokáže s mezerami ve jménech souborů poradit, čas od času je možné narazit na situace, kde jsou mezery problematické. Skripty by měli být očíslované v pořadí odrážejícím postup analýzy. Názvy by zpravidla měli obsahovat pouze malá písmena.

💡 “Version control” aneb konec `report-v1-finalni3.docx`

Problémem, který je nám všem jistě dobře známý, je jak udržovat pořádek v souborech, které jsou průběžně aktualizovány. Ať už se jedná o textové dokumenty, které prochází korekturami a zpětnou vazbou, nebo skripty které jsou pravidelně aktualizovány, většina lidí se dříve nebo později dostane do situace, kdy zírá do obrazovky a říká si *“Počkat, která verze mého reportu je ta aktuální?”* A nedejbože pokud bychom potřebovali zjistit, čím přesně se od sebe dvě různé verze stejného dokumentu liší.

Jednou možností, jak tomu předejít, je skálopevně dodržet některou pojmenovávací konvenci a poctivě číslovat každou novou verzi všech souborů. V dnešní době už ale existují i lepší řešení. Většina úložišť poskytuje službu zvanou *version control*,

tedy automatické sledování provedených změn. Místo toho, abyste po každé změně vytvářeli novou kopii souboru, pracujete pouze s jednou kopií a necháte na počítači, aby zaznamenával celou historii úprav. U každé větší změny můžete do historie zanechat krátkou poznámku, v čem se tato verze liší od té předchozí. Version control je dostupná pro většinu úložišť včetně [Google Drive](#) a [OneDrive](#). Pokud píšete velké množství kódu, doporučujeme využít některé ze specializovaných úložišť jako je [Github](#).

3.4 Kódovací styly

Kódovací styly (*coding styles*) představují seznam pravidel pro psaní dobře čitelného kódu. Silně **doporučujeme dodržovat jeden kódovací styl**. Dodržování vámi vybraného stylu pomůže váš kód udržovat dobře čitelný a přehledný, a to nejen pro vaše budoucí já, ale i pro vaše spolupracovníky.

Jednou z typických věcí, kterou kódovací styly upravují, je pojmenovávání proměnných. Způsobů pojmenovávání proměnných existuje více, mezi ty nejpobulárnější patří následující tři:

snake_case je styl, který používá malá písmena a slova odděluje podtržítkem. Například proměnná obsahující měsíční příjem respondenta by ve *snake_case* stylu vypadalo jako `monthly_income`.

camelCase styl pro oddělení slov využívá velkých písmen, zbylá písmena jsou malá. Průměrný měsíční příjem by v tomto stylu byl `monthlyIncome`.

kebab-case je styl podobný *snake_case*, místo podtržítkek ale využívá pomlček. Naše proměnná příjmu by vypadalo jako `monthly-income`.

Kód v této knize se řídí [Tidyverse coding style guide](#). To rozhodně není jediný používaný styl (např. [Google má svůj vlastní](#)) a rozhodně se nedá říct, že by byl lepší než všechny ostatní. V budoucnu si možná vybudujete svůj vlastní styl, ušitý na míru vašim potřebám. Ze začátku ovšem doporučujeme vybrat si jeden z populárních stylů a dát si záležet na jeho dodržování. Vaše budoucí já i vaši kolegové vám za to poděkují.

4 Typy objektů

Jednou z velkých životních pravd je, že vše v R je objekt. Některé objekty jsou velmi jednoduché, jiné mají komplexní strukturu. Každý objekt s má své využití a všechna data, se kterými budeme pracovat, budou uložena v některém z nich. Vyplatí se proto mít alespoň základní přehled o tom, s jakými typy objekty se v R můžeme setkat.

4.1 Atomové vektory

Nezákladnějšími objekty je takzvané atomické vektory (*atomic vectors*). Atomické vektory jsou základním stavebním kamenem R a všechny ostatní objekty, se kterými se setkáme, z nich vychází. Každý vektor je složen z určitého počtu elementů, tedy dílčích částí. Atomový vektor s jedním elementem představuje základní jednotku informace. Příkladem takového vektoru je:

```
"Fred"
```

```
[1] "Fred"
```

Výše zmíněný je takzvaný *character* vektor, obsahující jeden element, **Fred**. Vektory ale mohou obsahovat i více elementů:

```
c("Fred", "Daphne", "Velma", "Shaggy", "Scooby")
```

```
[1] "Fred" "Daphne" "Velma" "Shaggy" "Scooby"
```

Na rozdíl od předchozího příkladu, tento vektor obsahuje pět elementů. Všimněte si, že elementy jsou spojeny do jednoho vektoru pomocí funkce `c()` (zkratka pro *combine*). R zná čtyři typy atomových vektorů:

Tabulka 4.1: Typy atomových vektorů

Typ	Popis	Příklady
character	Písmena a další znaky. Poznáte je podle toho, že elementy jsou “obaleny” úvozovkami (" nebo ').	"Fred", "?", "1"
integer	Celá čísla. Spolu s typem <i>double</i> tvoří skupinu numerických (<i>numeric</i>) vektorů.	-1, 316, 17
double	Desetinná čísla. Zkratka pro <i>double precision floating point format</i> . Spolu s typem <i>integer</i> tvoří skupinu numerických (<i>numeric</i>) vektorů.	1.32, 0.1, -9.0
logical	Binární vektor, který může nabývat pouze dvou hodnot: pravda (<i>TRUE</i>) a nepravda (<i>FALSE</i>).	TRUE, FALSE

(Technicky existují ještě dva další typy atomových vektorů, *raw* a *complex*, s těmi se ovšem běžný uživatel nikdy nesetká, takže je přeskočíme.)

1 není “1”

Při práci s R je třeba si dát pozor na to, že ne vše, co vypadá jako číslo, nutně číslo je. R vám s radostí spočítá průměr vektoru `c(1,2,3)`, pokud byste se pokusili spočítat průměr vektoru `c("1", "2", "3")`, narazíte na problém a chybovou hlášku. Proč? Protože zatímco první vektor je typu *numeric* (konkrétně *integer*), druhý vektor je typu *character*. R tedy druhý vektor vidí v podstatě jako písmena a pro písmena se průměr spočítat přeci nedá!

Pro atomové vektory platí, že jejich elementy musí být stejného typu. Není tedy možné vytvořit atomový vektor, který by byl kombinací znaků a čísel (`c(18, "Fred")`). Pokud se to pokusíte, R vás buď zastaví nebo převede všechny elementy do stejného typu (`c("18", "Fred")`). Na toto automatické převádění elementů typů si dávejte pozor, jedná se o častý zdroj chyb a problémů.

Kromě běžných hodnot, kterých mohou elementy nabývat, existují tři speciální hodnoty, se kterými se budeme setkávat. Těmi jsou NA a NULL a NaN. NA představuje chybějící hodnotu ve statistickém smyslu slova. Setkáme se s ní pokud respondenti odmítnou odpovědět na některou z otázek v dotazníkovém šetření nebo pokud R nemá dostatek informací pro výpočet nějaké veličiny. Jedná se tedy o hodnotu existující, ale nám neznámou. Naproti tomu NULL reprezentuje absenci platné hodnoty. Setkáme se s ní v situaci, kdy žádné pozorování v datasetu neodpovídá námi specifikovaným filtrům. Hodnota NaN je zkratkou pro “*Not a Number*”. Pokud se s ní setkáte, znamená to zpravidla, že jste se dopočítali někam, kam jste nechtěli.

4.2 Faktory

Komplikovanějším typem vektoru jsou takzvané faktory. Jedná se v podstatě *integer* vektory, jejichž hodnotám bylo přiřazeno slovní označení (*label*):

```
[1] Agree    Neutral  Disagree
Levels: Agree Disagree Neutral
```

Přestože se faktory na první pohled tváří jako běžné *character* vektory, každé kategorii byla přiřazena číselná hodnota. V našem případě "Agree" = 1, "Disagree" = 2, "Neutral" = 3" (všimněte si, že číselné hodnoty byly přiřazeny podle abecedního pořadí slovních označení). Toho mnoho R funkcí využívá při statistických výpočtech, v rámci kterých je třeba zakódovat kategorické proměnné do číselných hodnot. S faktory se proto v datové analýze setkáme velmi často, ať už se bude jednat o odpovědi na likertovské položky v dotazníkových šetřeních, název bydliště nebo třeba název prodaného produktu.

Další důležitou vlastností faktoru je, že mohou nabývat pouze hodnot, které byly definovány při jejich vytvoření:

```
factor(x = c("Agree", "Neutral", "Disagree", "Don't know"),
       levels = c("Agree", "Neutral", "Disagree"))
```

```
[1] Agree    Neutral  Disagree <NA>
Levels: Agree Neutral Disagree
```

Přestože se v "datech" se vyskytují čtyři různé hodnoty, hodnota "Don't know" byla po vytvoření faktoru převedena na NA. Důvodem je, že jsme při vytváření našeho faktoru uvedli pouze tři platné hodnoty: "Agree", "Neutral", "Disagree" a žádná další nebude naším faktorem akceptována. Tato vlastnost se hodí zejména v situacích, kdy víme, jakých hodnot může proměnná nabývat, například u likertovských položek, a všechny ostatní naměřené hodnoty jsou nutně chybné.

Nakonec ještě zmiňme, že kromě klasických faktorů existují také takzvané *ordered factors*. Historicky se jednalo o faktory, které nemají pevně dané pouze to, jakých hodnot mohou elementy nabývat, ale i v jakém pořadí mají být seřazeny. Většina současných funkcí mezi klasickými a *ordered* faktory nerozlišuje, takže se jimi zpravidla nemusíte trápit.

Klasický faktor lze vytvořit funkcí `factor()`, *ordered* faktor poté pomocí funkce `ordered()`.

4.3 Matice a tabulky

Do této chvíle jsme viděli pouze jednorozměrné vektory, jako je například řada jmen v *character* vektoru. V datové analýze se ale budeme běžně setkávat i s vícerozměrnými objekty, z nichž tím nejzákladnějším je matice (*matrix*). Jedná se o starou známou matici, kterou si můžete pamatovat z hodin matematiky:

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

V příkladu výše vidíme dvourozměrnou matici s devíti elementy. Matice mohou mít i více dimenzí a v takovém případě jim říkáme *arrays*.

Matice jsou samozřejmě extrémně důležité pro jakoukoliv manipulaci s daty. Pokud si již ovšem z hodin lineární algebry mnoho nepamätujete, nemusíte panikařit. Mnoho funkcí sice matice interně využívá, jejich uživatelé to ovšem často ani nepostřehnou. Matematické operace jsou v R značně abstrahované a většina běžných datových analytiků proto s maticemi interaguje minimálně. Pokud byste je někdy potřebovali vytvořit, poslouží vám k tomu funkce `matrix()` a `array()`.

Typem objektu, se kterým se setkáte o něco častěji, je tabulka (*table*). Tabulky jsou také maticemi, jejichž dimenzím byla přiřazena slovní označení. Nejčastěji se s tabulkami setkáme při agregaci vektorů. Příkladem tabulky je například

```
Dislike  Like
       7    13
```

Tato tabulka je výsledkem funkce `table()`, aplikované na atomový *character* vektor obsahující sedmkrát hodnotu `Dislike` a třináctkrát hodnotu `Like`. Studenti sociálních věd také jistě budou znát kontingenční tabulky, tedy frekvenční tabulky pro dvě nebo více proměnných:

```
          Preference
Transport Dislike Like
    Bike          4    6
    Bus           5    5
    Car           4    6
```

Jak matice, tak tabulky, mohou nepřekvapivě obsahovat pouze numerické elementy.

4.4 Listy

Všechny předchozí typy objektů mohli uchovávat pouze elementy stejného typu. Realita je ovšem komplikovanější a je na nás, abychom se jí přizpůsobili.

Základním typem objektu pro uchovávání elementů různého typu je list:

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

```
[[2]]  
[1] "Fred"
```

```
[[3]]  
[1] TRUE
```

Listy mohou uchovávat objekty různých typů, v podstatě bez jakýchkoliv omezení. List může dokonce obsahovat jiný list! Poradí si dokonce i s objekty různé délky:

```
[[1]]  
[1] "Fred" "Daphne" "Velma" "Shaggy" "Scooby"
```

```
[[2]]  
[1] 42.0 1.3 666.0
```

```
[[3]]  
[1] TRUE FALSE
```

```
[[4]]  
[[4]] [[1]]  
[1] "Car"
```

```
[[4]] [[2]]  
[1] "Bus"
```

```
[[4]] [[3]]  
[1] "Bike"
```

Díky své flexibilitě se listy využívají primárně pokud chceme uchovávat velmi různorodá data na jednom místě. Většina výstupů statistických analýz bude právě ve formě listu. List je možné vytvořit funkcí `list()`.

4.5 Dataframy

To nejlepší jsme si nechali nakonec. Zdaleka nejužitečnějším typem objektu pro datového analytika je *dataframe*:

```
      V1 V2   V3
1   Fred 16 FALSE
2 Daphne 16 FALSE
3   Velma 15 FALSE
4 Shaggy 17 FALSE
5 Scooby  3  TRUE
```

Dataframy slouží stejnému účelu jako spreadsheetsy v jiných softwarech. Je v nich uchovávána drtivá většina všech dat a pracuje s nimi většina funkcí, se kterými se v této knize setkáme. U běžného dataframu platí, že každý sloupec představuje jednu proměnnou a každý řádek jedno pozorování (např. respondenta).

Zvláštní vlastností dataframů je, že všechny jeho elementy musí být stejně dlouhé. Jinak řečeno, pro každou proměnnou musíme mít stejný počet pozorování. Co když tomu tak není? V takovém případě vstupuje do hry hodnota *NA*, zmíněná dříve, která kóduje chybějící hodnoty. Například v následujícím dataframu máme ve druhém sloupci jen čtyři hodnoty, plus jednu *NA*:

```
      V1 V2   V3
1   Fred 16 FALSE
2 Daphne 16 FALSE
3   Velma 15 FALSE
4 Shaggy NA  FALSE
5 Scooby  3  TRUE
```

Jak jsme již zmínili, naprostá většina naší práce bude probíhat v dataframech. Konkrétně budeme využívat speciální typ dataframu, zvaný *tibble*. *Tibble* je druh dataframu pocházející z Tidyverse, a většinou se chová identicky jako jeho základní varianta. Hlavním rozdílem je o něco hezčí vzhled.

Klasický dataframe je možný vytvořit funkcí `data.frame()`, *tibble* poté pomocí funkce `tibble()`.

5 Jména objektů

5.1 Přiřazování jmen

V předchozí kapitole jsme si představili nejdůležitější typy objektů. Řekli jsme si také, jak můžeme objekty vytvářet. *Character* vektor bychom například vytvořili takto:

```
c("Fred", "Daphne", "Velma", "Shaggy", "Scooby")
```

```
[1] "Fred" "Daphne" "Velma" "Shaggy" "Scooby"
```

S takto vytvořeným vektorem se nám ale nebude dobře pracovat. To proto, že R ho vytvoří, vytiskne do konzole a promptně zapomene. Pokud si chceme data odložit na později, je nutné objektu, ve kterém jsou uložena, přiřadit jméno. Přiřazování jmen je velmi jednoduché:

```
name <- c("Fred", "Daphne", "Velma", "Shaggy", "Scooby")
```

V tomto případě jsme našemu vektoru přiřadili jméno **name**. Obsah vektoru nebyl vytisknut v konzoli. Místo toho ho R uložilo do paměti a pokud používáte Rstudio, můžete ho vidět v pravém horním rohu v záložce Environment. K přiřazování jmen se využívá operátor `<-`, který nejnadhěji vytvoříte pomocí klávesové `Alt + -` (resp. `option + -` na MacOS). Alternativně můžete použít `=`, výsledek bude stejný:

```
name = c("Fred", "Daphne", "Velma", "Shaggy", "Scooby")
```

Jakmile má objekt přiřazené jméno, můžeme na něj v budoucnu odkazovat. Pokud bychom chtěli zjistit kolik elementů náš vektor má, můžeme použít funkci `length()`:

```
length(name)
```

```
[1] 5
```

Jak je vidět, objekt **name** obsahuje pět elementů.

i “<-” nebo “=” ?

Někteří čtenáři si teď možná nejsou jistí, který z operátorů by měli používat pro pojmenovávání objektů, <- nebo =? Krátce řečeno, na vaší volbě nezáleží.

Pro delší odpověď je třeba znát trochu historie. R původně vzniklo na základě jazyka jménem S, který pro pojmenovávání objektů používal právě <-. Tento operátor převzalo i R, primárně pro zpětnou kompatibilitu. Od doby, kdy lidé ještě používali S, již dnes uběhlo skoro 20 let a zpětná kompatibilita s tímto jazykem není moc důležitá. Naopak přibýlo uživatelů, kteří kromě R používají i jazyky jako Python a Javascript, využívající = operátor. Tito uživatelé přirozeně tíhnou k využívání = ve všech situacích.

Technicky vzato existuje velmi malé množství případů, kdy na rozdíl mezi <- a = záleží. Operátor <- má vyšší prioritu než =, což znamená, že pokud se R dostane do situace, kdy neví, který z nich vyhodnotit dřív, vybere si <-. V praxi k takovým situacím ale dochází naprosto minimálně. Zájemci o více detailů viz <https://stackoverflow.com/a/51564252>.

Jména můžeme stejným přiřazovat i elementům uvnitř složitějších objektů než jsou vektory, jako jsou listy a dataframy. Pokud bychom chtěli vytvořit dataframe a pojmenovat jednotlivé sloupce (proměnné), udělali bychom to následovně:

```
gang <- data.frame(name = c("Fred", "Daphne", "Velma", "Shaggy", "Scooby"),
                  age = c(16, 16, 15, 17, 3),
                  is_dog = c(FALSE, FALSE, FALSE, FALSE, TRUE))
```

Náš nový dataframe můžeme zobrazit v konzoli pomocí funkce `print()`:

```
print(gang)
```

```
  name age is_dog
1  Fred  16  FALSE
2 Daphne 16  FALSE
3  Velma 15  FALSE
4 Shaggy 17  FALSE
5 Scooby  3   TRUE
```

Nejenže můžeme pracovat s naším dataframem pomocí jeho jména (`name`), ale každý sloupec dataframu má jméno, které jsme mu přiřadili. O tom, jak pracovat s jednotlivými sloupci, si povíme v příští kapitole.

💡 Jména elementů

Méně používaná, ale občas užitečná, je možnost pojmenovávat jednotlivé elementy vektoru. Například:

```
c(Fred = 16, Daphne = 16, Velma = 15, Shaggy = 17, Scooby = 3)
```

```
Fred Daphne Velma Shaggy Scooby  
16 16 15 17 3
```

To se hodí zejména v například případech, kdy chceme mít informaci o to, co jednotlivé hodnoty znamenají, ale nechceme pro ně vytvářet novou proměnnou v dataframu.

5.2 Pravidla pojmenování

Přestože R nabízí značnou volnost v tom, jak své objekty pojmenujete, je nutné dodržovat alespoň některá pravidla. Jména musí začínat buď písmenem nebo tečkou. Pokud začínají tečkou, druhý znak nesmí být číslice (například `.2scale` tedy není použitelné jméno). Jména také mohou obsahovat pouze písmena, číslice, tečky nebo podtržítka (`_`). Žádné `$`, `~` a podobně.

Následující slova také nemůžou být jmény objektů: `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, `NA_character_` a Těmto výrazům se říká rezervovaná slova a jsou využívána pro vnitřní fungování R. Už jsme se například setkali s výrazem `NA`, který je rezervován pro vyjádření chybějících hodnot.

Kromě těchto pravidel, které musíte dodržovat ať se vám to líbí nebo ne, také silně doporučujeme, aby jména vašich objektů byla krátká, ale srozumitelná. Věk respondentů dotazníkového šetření by měl být ideálně uložen v proměnné `age`, spíše než `I2` nebo `age_of_respondents`. Řiďte se také jednou z jmenovacích konvencí popsaných v kapitole [Jak si organizovat práci](#). My osobně preferujeme `snake_case`, tedy všechna písmena malá a slova oddělená podtržítkem.

6 Funkce

Jedním z centrálních typů objektů, o kterém jsme zatím nemluvili, jsou funkce. Funkce jsou objekty které nám umožňují manipulovat jinými objekty. Poznat funkci je jednoduché, protože jméno každé z nich je následované závorky (). V předchozích kapitolách jsme již funkce dokonce používali, ať už se jednalo o `c()`, `data.frame()` nebo `install.packages()`. Teď se na ně konečně podíváme pořádně.

6.1 Používání funkcí

Používat funkce je jednoduché. Každá funkce obsahuje argumenty, pomocí kterých funkci upřesňujete, co přesně má vykonat. Tyto argumenty si píšete právě do závorek za jménem funkce. Funkce, kterou jsme viděli opakovaně, je například naše staré dobré `c()`:


```
name <- c("Fred", "Daphne", "Velma", "Shaggy", "Scooby")
```

V tomto případě jsme použili funkci `c()` pro vytvoření nového vektoru. Této funkci jsme zadali pět argumentů, definujících z jakých elementů se má nový vektor skládat. Obdobně bychom na náš nový objekt `name` mohli použít funkci `print()`:

```
print(name)
```

```
[1] "Fred" "Daphne" "Velma" "Shaggy" "Scooby"
```

V tomto případě jsme funkci `print()` specifikovali pouze jeden argument, a to který objekt má vytisknout do konzole.

 Ukládejte si své výsledky!

Častým zdrojem zmatení u nových uživatelů bývá, že R zdánlivě dělá co mu říkají, ale výsledky nikdy neukládá! Toto nedorozumění je nejsnažnější vysvětlit na praktickém příkladu.

Co kdybychom chtěli zaokrouhlit čísla v následujícím vektoru?

```
age <- c(16.45, 16.52, 15.9, 17.1, 3.234)
```

Zaokrouhlení čísel je jednoduchá záležitost, pro kterou můžeme využít funkci `round()`:

```
round(age)
```

```
[1] 16 17 16 17 3
```

A je zaokrouhleno. Nebo ne? Pokud se podíváme na vektor `age`, zjistíme že pořad obsahuje původní čísla:

```
age
```

```
[1] 16.450 16.520 15.900 17.100 3.234
```

Proč si R odmítá zapamatovat, že jsme čísla zaokrouhlili? Protože jsme mu neřekli, že má výsledek funkce `round()` uložit do paměti. Vektor se zaokrouhlenými čísly je objekt jako každý jiný a pokud ho chceme využívat v budoucnu, musíme mu přiřadit jméno. Pokud nám nevadí přijít o původní nezaokrouhlená jména, můžeme klidně použít jméno původního vektoru:

```
age <- round(age)
age
```

```
[1] 16 17 16 17 3
```

6.2 Argumenty funkcí

Jak jsme zmínili, každá funkce má argumenty, které ovlivňují její fungování. V předchozích příkladech sloužili argumenty primárně pro určení toho, se kterými daty má funkce pracovat. Většina argumentů ale upravuje primárně to, *co* má funkce s daty dělat. Pro ukázkou si vytvořme nový vektor:

```
age <- c(16, 16, 17, 15, NA)
```

Jedná se o numerický vektor, jehož poslední hodnota je neznámá (`NA`). Co kdybychom chtěli spočítat průměr těchto hodnot? K tomu poslouží funkce `mean()`, pokud bychom ji ale aplikovali na náš vektor, narazili bychom na problém:

```
mean(age)
```

```
[1] NA
```

R nám říká, že průměr těchto čísel je NA, tedy neznámý. Proč? Narážíme tu na jistou pedantnost typickou pro R. R nám svým způsobem říká *“Alespoň jedno z čísel v tomto vektoru je neznámé a může teoreticky nabývat jakékoliv hodnoty. Proto i výsledný průměr může nabývat jakékoliv hodnoty, a je tedy sám neznámý”*. V tom má R jistě pravdu. Co kdybychom se ale spokojili s tím, že budeme neznámé hodnoty ignorovat a spočítat průměr jen pro známá čísla? Přesně k tomu má funkce `mean()` argument `na.rm` (*remove NAs*). Tento argument může nabývat dvou hodnot `TRUE` a `FALSE`. Ve výchozím nastavení je tento argument nastaven na `FALSE`, což mi ale můžeme jednoduše změnit:

```
mean(age, na.rm = TRUE)
```

```
[1] 16
```

A je to! Pomocí argumentu `na.rm` jsme změнили fungování funkce `mean()` tak, aby ignorovalo neznámé hodnoty.

6.3 Funkce a vnořené objekty

V předchozí kapitole jsme si řekli, že většina dat je uchovávaných v `dataframech`. Jeden takový `dataframe` můžeme vytvořit pomocí:

```
gang <- data.frame(name = c("Fred", "Daphne", "Velma", "Shaggy", "Scooby"),
                  age   = c(16, 16, 15, 17, 3),
                  is_dog = c(FALSE, FALSE, FALSE, FALSE, TRUE))
```

Co kdybychom chtěli spočítat počet členů členů Scoobyho gangu? Jako první se nabízí možnost:

```
length(name) # Error: object 'name' not found
```

To ovšem nebude fungovat, protože R nemůže najít žádný objekt jménem `name`. Tento objekt je totiž vnořený (*nested*) uvnitř jiného objektu, `gang` a R nebude prohledávat všechny existující objekty pokaždé, když mu řekneme, aby aplikovalo některou funkci. Je tedy na nás, abychom R navedli, kde má proměnnou `name` hledat.

Toho lze docílit několika způsoby. Tím prvním je pomocí operátoru `$`. Tímto operátorem můžeme navigovat vnořenými objekty, například jím můžeme vybrat proměnnou `name` v dataframu `gang`:

```
length(gang$name)
```

```
[1] 5
```

R teď ví, že objekt `name` by mělo hledat unvitř objektu `gang`.

Specifikování vnořených objektu pomocí `$` je asi nejpoužívanější způsob pokud pracujeme s dataframy, není ale jediný. Alternativní způsob představuje indexování pomocí hranatých závorek `[]`. Ty lze aplikovat několika způsoby. Prvním z nich je skrze jméno vnořeného objektu:

```
gang["name"]
```

```
   name
1  Fred
2 Daphne
3  Velma
4 Shaggy
5 Scooby
```

Alternativně můžeme využít pořadí řádků a sloupců v objektu. `name` je první proměnnou v dataframu `gang` a můžeme ji tedy vybrat následovně:

```
gang[, 1]
```

```
[1] "Fred" "Daphne" "Velma" "Shaggy" "Scooby"
```

Všimněte si, že závorky v tomto případě obsahují čárku (`[, 1]`). To proto, že pomocí hranatých závorek můžeme vybírat jak sloupce, tak řádky. Pořadí řádku se z konvence píše na první pozici, sloupce na druhé. Kdybychom se chtěli dozvědět více o Fredovi, mohli bychom použít:

```
gang[1, ]
```

```
   name age is_dog
1 Fred  16  FALSE
```

💡 Vylučovací metoda

Indexování je možné využít i pro výběr všech sloupců/řádků kromě zmíněných. Pro vybrání všech sloupců kromě třetího použijeme `gang[, -3]`.

Oboje možnosti je samozřejmě možné kombinovat. Hodnotu prvního řádku a prvního sloupce bychom získali následovně:

```
gang[1,1]
```

```
[1] "Fred"
```

Pokud tedy pracujeme s vnořenými objekty, a to budeme téměř neustále, nesmíme R nikdy zapomenout říct, kde má hledat.

i `[[]]` je více než `[]`

Čas od času se můžete setkat s kódem využívajícím dvojité závorky (`[[]]`), místo jednoduchých (`[]`). Každá z těchto variant má své využití.

Všimněme si, jaký typ objektu vrátí následující kód:

```
gang["name"]
```

```
  name
1  Fred
2 Daphne
3  Velma
4 Shaggy
5 Scooby
```

Jedná se o dataframe, stejně jako byl původní objekt, pouze byly odstraněny všechny sloupce kromě toho se jménem `name`. Co naproti tomu dělá následující kód?

```
gang[[ "name" ]]
```

```
[1] "Fred"  "Daphne" "Velma"  "Shaggy" "Scooby"
```

Tento kód vrátil stejné hodnoty, ale v jiném formátu. Už se nejedná o (filtrovaný) dataframe, ale o atomický vektor. Rozdíl mezi těmito dvěma způsoby vybírání vnořených objektů je důležitý, protože argumenty funkcí často očekávají data v určitém formátu.

6.4 Řetězení funkcí

Všechny příklady, které jsme zatím viděli, aplikovali vždy pouze jednu funkci. Asi ovšem tušíte, že v reálné analýze budeme muset na naše data aplikovat mnohem více funkcí, než se dostaneme ke kýženým výsledkům. To s sebou přináší praktický problém. Jak na sebe efektivně řetězit větší počet funkcí tak, aby byl náš kód stále čitelný? V principu existují tři varianty.

První možnost je aplikovat funkci jednu po druhé a ukládat mezivýsledky do nových objektů:

```
me_awake <- wake_up(me)
me_clean  <- wash(me_awake)
me_fed    <- eat_breakfest(me_clean)
me_working <- go_to_work(me_fed)
```

Tento postup je analogický tomu, co jsme dělali dosud. Aplikujeme funkci a její výsledek uložíme do nového objektu. Jedná se o vcelku přehledný postup, nevýhodou ovšem je, že vytváříme velké množství objektu, které zabírají místo a zneřehledňují naše prostředí.

Alternativně je možné na sebe funkce “nabalovat”:

```
me_working <- go_to_work(eat_breakfest(wash(wake_up(me))))
```

Tímto se vyhneme vytváření nových objektů, výsledný kód ovšem není příliš čitelný. Hlavním problémem je, že pokud chceme vědět, co tento kód dělá, je nutné ho číst od středu. Jako první je aplikovaná funkce v “jádro”, tedy `wake_up()`, a poté všechny ostatní směrem k okraji. Funkce `go_to_work()` je aplikovaná jako poslední a to i přesto, že je na řádce jako první.

Poslední, námi preferovanou, metodu je využívání takzvaných *pipes*. Používat budeme *pipes* z balíčku `magrittr`, který je součástí `Tidyverse`. Ty vypadají takto: `%>%` a aplikuje se následovně:

```
me_working <- me %>%
  wake_up() %>%
  wash() %>%
  eat_breakfest() %>%
  go_to_work()
```

Pipes (`%>%`) vezmou objekt nalevo od nich a vloží ho do funkce napravo. První *pipe* tedy vezme objekt `me` a vloží ho do funkce `wake_up()`. Druhá *pipe* vezme výsledek funkce `wake_up()` a vloží ho do funkce `wash()`. Takto celý řetězec pokračuje dále až po funkci `go_to_work()`. Výsledek celého řetězce je uložen do objektu `me_working` tak, jak jsme zvyklí. Protože psát jednotlivé *pipes* ručně by bylo otravné, existuje pro ně v Rstudiu klávesová zkratka **Shift + Ctrl + M** (respektive **Shift + Command + M** na MacOS).

Pipes jsou preferovaný způsob řetězení funkcí v Tidyverse a budou využívány ve zbytku této knihy. Jejich hlavní výhodou je, že výsledný kód je dobře čitelný, protože je možné ho číst zleva doprava, tak jak jsme zvyklí u normálního textu. Na druhou stranu, někteří lidé argumentují že takto psaný kód zabírá příliš mnoho místa.

i Tidyverse vs základní *pipes*

Pipes byly v R dlouhou dobu čistě Tidyverse záležitostí. Od verze 4.1. jsou ale podporovány i základní instalací R a je tedy možné využívat tento způsob řetězení funkcí bez nutnosti instalovat další balíčky. *Pipes* v základním R vypadají a chovají se trochu odlišně od svých Tidyverse příbuzných. Základní verze *pipes* vypadá takto: `|>`. Příklad s řetězením funkcí by tedy vypadal následovně:

```
me_working <- me |>
  wake_up() |>
  wash() |>
  eat_breakfest() |>
  go_to_work()
```

Kromě odlišného vzhledu se také obě verze chovají trochu jinak. Hlavním rozdílem je, že používají jiný “*placeholder*” pro specifikaci argumentů. Obě verze ve výchozím nastavení vloží objekt na jejich levé straně do prvního argumentu funkce napravo. Pokud bychom chtěli vložit objekt do jiného než prvního argumentu, je nutné využít právě *placeholder*. Tidyverse *pipe* používá jako *placeholder* tečku. Například, pokud bychom chtěli vložit objekt `iris` do argumentu `data`, který je na druhém místě funkce `lm()`:

```
iris %>% lm(Sepal.Width ~ Species, data = .)
```

Naproti tomu základní *pipe* využívá jako *placeholder* podtržítka:

```
iris |> lm(Sepal.Width ~ Species, data = _)
```

Kromě *placeholderů* se obě verze *pipes* liší i interním fungováním. Ve zkratce řečeno, tidyverse *pipe* je flexibilnější a umí více věcí, základní *pipe* je zhruba dvakrát až třikrát rychlejší.

Pokud byste si chtěli základní verzi *pipe* vyzkoušet, můžete upravit klávesovou zkratku `Shift + Ctrl + M` tak, že půjdete do *Tools -> Global Options -> Code -> Use native pipe operator*.

6.5 Dokumentace funkcí

Po všem tom povídání si teď možná říkáte, jak si má člověk zapamatovat, co která funkce dělá, nemluvě o tom, jaké má argumenty. Naštěstí pro nás si toho moc nazpaměť pamatovat nemusíme, protože každá funkce má svou vlastní dokumentaci. Ta obsahuje popis funkce, výčet všech jejích argumentů, detaily o jejím fungování a příklady použití. Dokumentaci pro vybranou funkci můžeme zobrazit pomocí funkce `help()`, případně ?:

```
help(mean)
?mean
```

Obě výše zmíněné funkce mají stejný výsledek, a to otevření dokumentace pro funkci `mean()`. Dokumentace všech funkcí má stejnou strukturu, složenou z následujících součástí.

První sekcí je *Description*, která obsahuje krátký popis funkce, v tomto případě vysvětlení, že funkce `mean()` počítá aritmetický průměr.

V sekci *Usage* je k vidění výchozí nastavení funkce, vidíme například, že argument `trim` má výchozí hodnotu 0 a argument `na.rm` je nastavený na `FALSE`.

Sekce *Arguments* nepřekvapivě popisuje jednotlivé argumenty, k čemu slouží a jakých hodnot mohou nabývat.

Následuje sekce *Value*, která popisuje výsledek dané funkce, tedy co dostaneme, pokud funkci aplikujeme.

Občas přítomná je také sekce *Details*, která poskytuje další detaily o fungování funkce. Tato sekce se objevuje hlavně u funkcí pro výpočet statistických modelů a podobně komplikovanějších funkcí.

References je klasickým seznamem literatury. Najdeme zde všechny texty citované v dokumentaci a odkazy na další užitečné práce.

See Also je seznamem příbuzných funkcí, které by uživatele mohli zajímat. Vidíme například, že je nám doporučena funkce `weighted.mean()` pro výpočet váženého průměru.

Examples je poslední sekcí dokumentace, která obsahuje ukázky použití funkce v praxi.

6.6 Vytváření vlastních funkcí

Jedním z největších předností R je, že se při naší práci nemusíme spoléhat pouze na funkce, které pro nás připravili jiné lidé, ale můžeme si vytvořit funkce na míru našim potřebám. Vytvoření nové funkce je velmi jednoduché, pomocí funkce `function`.

Funkci, kterou základní instalace R překvapivě postrádá, je výpočet počtu chybějících hodnot v proměnné. Ne, že by se jednalo o obtížný úkol. Lze k tomu využít kombinaci dvou funkcí, `is.na()` a `sum()`.

Funkce `is.na()` zkontroluje, jestli každý element vektoru chybějící hodnota a vrátí nám nový logický vektor, který bude mít hodnotu `TRUE` v případě chybějících hodnot a hodnotu `FALSE` v případě těch platných. Například:

```
age <- c(NA, 16, 17, NA, 3)
is.na(age)
```

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

Jak vidíme, na první a čtvrtý element jsou chybějící hodnoty. Nyní můžeme použít funkci `sum()`, který při aplikaci na logický vektor vrátí počet `TRUE` hodnot:

```
sum(is.na(age))
```

```
[1] 2
```

A opravdu, dozvěděli jsme se, že v našem vektoru jsou dvě chybějící hodnoty. Nabízí se ale otázka, jestli by se kombinace funkcí `sum(is.na())` nedala nějak zjednodušit. Přeci jen, počítat chybějící hodnoty budeme relativně často, a čím méně závorek v našem kódu, tím menší šance, že některou z nich zapomeneme uzavřít.

Vytvoříme si proto vlastní funkci, která bude počítat chybějící hodnoty za nás. Taková funkce by mohla vypadat třeba takto:

```
count_missings <- function(var) {
  sum(is.na(var))
}
```

Jako první musíme naší funkci vymyslet jméno. V tomto případě použijeme popisné `count_missings`. Novou funkci vytvoříme pomocí funkce `function()`. Do kulatých závorek vypíšeme, jaké argumenty by naše nová funkce měla mít. V našem případě bude stačit pouze jediný argument, a to `var`. Následují spojené závorky a uvnitř to hlavní, tedy popis toho, co má naše nová funkce dělat. V tomto případě spočítá počet chybějících hodnot. Všimněte si, že se zde znovu objevu argument `var`, který jsme definovali v předchozím kroku.

A to je vše. Teď už můžeme používat naší novou funkci a ušetřit si trochu psaní:

```
count_missings(age)
```

[1] 2

7 R balíčky

Přestože základní instalace R teoreticky obsahuje vše, co potřebujeme pro datovou analýzu, v praxi se nám vyplatí stáhnout si rozšiřující balíčky, které pro nás připravili ostatní členové a členky R komunity. Koneckonců, proč se trápit psáním našich vlastních funkcí, což často vyžaduje netriviální programátorské a statistické znalosti, když můžeme využít léty ověřený balíček od některého z uznávaných autorů. V této kapitole si ukážeme základy instalace a udržování nových balíčků.

7.1 Instalace balíčků

Většina balíčků pro R je dostupná v repozitáři zvaném [Comprehensive R Archive Network](#) (CRAN). CRAN je spravován centrálním vývojářským týmem R a všechny balíčky v něm podstupují přísnou technickou kontrolu. Ta zajišťuje, že všechny balíčky fungují na všech mainstreamových operačních systémech, mají kompletní dokumentaci a neobsahují žádný škodlivý kód (ovšem pozor na to, že CRAN neručí za věcnou správnost funkcí! To, jestli vám statistické a další funkce dají správný výsledek, je zodpovědností jednotlivých autorů). Instalace balíčků z CRAN je jednoduché, stačí použít funkci `install.packages()`:

```
install.packages("tidyverse")
```

Všimněte si, že název balíčku, který chceme nainstalovat, v tomto případě `tidyverse`, musí být v úvozovkách. Tato funkce také automaticky stáhne všechny prerekvizity potřebné pro zvolený balíček.

Funkci `install.packages()` je možné použít také k aktualizaci balíčků. Pokaždé, když ji použijete, bude stažena nejnovější dostupná verze zvoleného balíčku. Seznam nainstalovaných balíčků, pro které jsou dostupné aktuálnější verze, můžeme získat pomocí funkce `old.packages()` (bez jakýchkoliv argumentů). Pokud máte zastaralých balíčků více, můžete je aktualizovat všechny najednou pomocí funkce `update.packages()`.

7.2 Nahrání balíčku

Poté co je balíček nainstalován, je před jeho použitím třeba ještě nahrát (“zapnout”). R nenahrává všechny nainstalované balíčky, aby zbytečně neplýtvalo paměti. Nahrát balíček

je nutné pokaždé, když restartujete R. Nahrání samotné je jednoduchý proces pomocí funkce `library()`:

```
library(tidyverse)
```

Všimněte si, že u funkce `library()` už název balíčku nemusí být v uvozovkách (ale může, pokud preferujete konzistenci).

i *install.packages* versus *library*

Začínající uživatelé si občas nejsou jistí rozdílem mezi funkcemi `install.packages()` a `library()`, potažmo mezi instalací a nahráním balíčku. Instalace balíčku je proces zahrnující stažení balíčku z internetu a jeho následné nahrání. To je nutné udělat pouze jednou a balíček od té chvíle bude na vašem počítači. Nahráním balíček aktivujete, což vám umožní přístup k jeho funkcím. Nahrávat balíčky je nutné pokaždé, když spustíte R.

Pokud se během analýzy rozhodnete, že již balíček nepotřebujete, můžete ho vypnout pomocí funkce `detach()`. Tím přijmete o funkce v něm obsažené, až do chvíle, kdy znovu použijete funkci `library()`. Pokud chcete balíček odinstalovat úplně, použijte funkci `remove.packages()`. Ovšem pozor! Tato funkce smaže daný balíček z vašeho počítače. Pokud si své rozhodnutí v budoucnu rozmyslíte, budete si muset balíček znovu stáhnout a nainstalovat.

7.3 Konflikty mezi balíčky

Protože jsou balíčky pro R vytvářeny nezávisle na sobě velkým množstvím lidí, dostanou se čas od času do vzájemného konfliktu. Nejčastějším konfliktem je, že dva různí autoři použijí pro své funkce stejný název. Příkladem může být balíček `dplyr`, součást Tidyverse, který obsahuje funkci `filter()`. Funkce `filter()` je už ale obsažená v balíčku `stats`, který je součástí základní instalace. Pokud k situaci je tato dojde, R bude preferovat funkci pocházející z balíčku, který byl nahrán později (v tom případě tedy `dplyr`). Pokud bychom chtěli využít funkci `filter()` z balíčku `stats`, je nutné specifikovat v jakého balíčku (v odborném žargonu *namespace*) ji má R hledat, čehož docílíme pomocí `::` v tomto formátu:

```
stats::filter()
```

Analogicky, pokud bychom chtěli použít verzi z `dplyr`, použili bychom `dplyr::filter()`.

7.4 Kde hledat balíčky

Už jsme zmínili, že většinu balíčků, které budete potřebovat, je možné získat z CRAN. Kromě něj ale existují i další repozitáře, na kterých vývojáři sdílí svou práci.

Tím nejpoblárnějším je dnes [Github](#). Tato stránka je populární nejen u uživatelů R, ale i všech ostatních jazyků. dalšími populárními možnostmi jsou [R-forge](#) a, zvláště u kolegů z biologie a chemie, [Bioconductor](#).

Tyto repozitáře se od CRAN liší ve dvou ohledech. Tím prvním je, že balíčky nejsou zdaleka tak přísně kontrolovány, co se týče kvality. To na jednu stranu urychluje proces publikace, na druhou stranu musí být koncoví uživatelé opatrnější ohledně toho, co instalují na svůj počítač. Druhým aspektem je, že instalace z těchto repozitářů zpravidla vyžaduje více kroků, než jen využití `install.packages()`. Doporučujeme konzultovat dokumentaci k balíčkům v těchto repozitářích. Proces instalace ovšem není o tolik náročnější.

Část II

Manipulace s dataframy

8 Import a export dat

Předtím, než můžeme vytvářet dechberoucí grafy a komplexní modely, je nutné nejdříve naše data dostat do R. Tato kapitola bude věnovaná importu a exportu dat v nejběžnějších formátech, k čemuž využijeme balíčky `readr` a `haven`, oba součástí Tidyverse.

8.1 Pracovní adresář

Předtím, než se pustíme do importu dat samotného, se musíme seznámit s konceptem pracovního adresáře (*working directory*). Pracovní adresář je výchozí složka na vašem počítači, ze kterého bude R importovat a exportovat všechny soubory, pokud mu neřeknete jinak. Cestu k vašemu současnému pracovnímu adresáři zjistíte pomocí funkce `getwd` (*get working directory*):

```
getwd()
```

```
[1] "/Users/ales/Documents/cuni/teaching/uvod-do-r-kniha"
```

V našem případě je pracovním adresářem složka `/Users/ales/Documents/phd/teaching/uvod-do-r-kniha`. Pokud bychom po R chtěli naimportovat nějaký datový soubor, R ho bude hledat v této složce. Stejně tak, pokud bych exportoval vytvořený graf, bude uložen do této složky. Na koncept pracovního adresáře je dobré si zvyknout rychle, protože mnoho problémů, které začínající uživatelé mají během importu dat, je způsobeno buď odkazováním na špatnou složku nebo neznalostí jejich pracovního adresáře.

💡 Ať se o to postará Rstudio

Jednou z velkých předností Rstudio projektů (viz Sekce 3.1) je automatické nastavení pracovního adresáře do kmenové složky vašeho projektu při startu. To zaručuje, že pracovní adresář bude vždy poblíž vašich dat, což výrazně ulehčuje jejich import.

Výchozí pracovní adresář (mimo Rstudio projekt) je možné nastavit v *Tools -> Global Options -> General*. Pracovní adresář je také možné nastavit ručně pomocí funkce `setwd()`, tuto možnost ale silně nedoporučujeme. Problém spočívá v tom, že jakákoliv adresa na vašem počítači je platná jen pro váš počítač. Pokud byste složku s vaším projektem přesunuli na jiné

místo nebo poslali kolegovi, bylo by nutné měnit všechny pracovní adresáře ručně. Mnohem lepší je spoléhat na automatické nastavení pomocí Rstudio projektů.

8.2 Import dat

8.2.1 Comma seperated values

Zdaleka nejčastějším typem souborů, se kterými se pravděpodobně setkáte, jsou takzvané *comma separated values (CSV)* soubory. Ty se dají poznat jednoduše podle koncovky `.csv`. Data tohoto typu můžeme naimportovat do R pomocí funkce `read_csv()` z balíčku `readr`, která je také součástí balíčku `tidyverse`.

Možnosti, jak říct R, kde má soubor hledat, jsou dvě. Tou preferovanou je využít relativní cesty (*relative path*). Relativní cesta začíná ve vašem pracovní adresáři a můžeme jít specifikovat následovně:

```
library(tidyverse) # nezapomeňte na nahrání balíčku!  
  
countries <- read_csv("data-raw/countries.csv")
```

Tento příkaz říká R, aby se v pracovním adresáři podívalo do složky `data-raw` a v ní hledalo soubor `countries.csv`. Nalezený soubor potom naimportuje jako dataframe a pojmenuje `countries`.

Alternativně je možné specifikovat úplnou cestu k souboru (*full path*):

```
countries <- read_csv("/Users/ales/Documents/phd/teaching/uvod-do-r-kniha/data-raw/countries.csv")
```

Oproti předchozímu příkladu je plná cesta mnohem delší. Silně doporučujeme plné cesty *nevyužívat*, a to ze stejného důvodu, ze kterého byste neměli ručně nastavovat pracovní adresáře. Výše uvedená cesta bude fungovat pouze na jednom konkrétním počítači a pouze dokud zůstane složka s projektem na stejném místě. Používáním plných cest si zaděláváte na problém ve chvíli, kdy budete přesouvat svoji práci z jednoho počítače na druhý.

i read_csv versus read.csv

Pro import dat do R není nezbytně nutné využívat balíček `readr`, potažmo `tidyverse`. Základní instalace R obsahuje funkci `read.csv()`, pomocí které byste mohli data importovat stejným způsobem. My ale preferujeme `read_csv()`, protože je rychlejší a dává nám větší kontrolu nad tím, jak jsou data importována.

Data lze stejným způsobem stahovat i z internetu:

```
countries <- read_csv("https://raw.githubusercontent.com/Sociology-FA-CU/Uvod_do_analyzy_d
```

Bohužel, ne všechna data uložená ve formátu `.csv` jsou opravdu hodnoty oddělené čárkami. Přestože tyto atypické formáty mohou vzniknout více způsoby, primárním zdrojem problémů je většinou Microsoft Excel. Ten je distribuován v řadě regionálních verzí, z nichž každá se chová trochu jinak. Konkrétně verze pro střední Evropu využívá pro oddělování hodnot středníky (;), protože střeoevropské země historicky využívají čárku pro oddělení desetinných míst. To vede k řadě otravných problémů při importu a exportu dat, zvláště v mezinárodních týmech.

V případě, že se setkáte s datovým souborem, který nepoužívá klasické oddělovače, máte dvě možnosti. Tou první je pomocí argumentů funkce `read_csv()` ručně upravit, které hodnoty mají být viděny jako oddělovače sloupců a které jako oddělovače desetinných míst. Například:

```
countries <- read_csv("data-raw/countries.csv",  
                      locale = locale(grouping_mark = ";",  
                                      decimal_mark = ","))
```

Protože problémy s importem dat produkovaných ve střední Evropě jsou extrémně časté, balíček `readr` obsahuje funkci `read_csv2()`, která plní stejný účel jako kód výše. Druhou možností je tedy ulehčit si psaní a využít ji:

```
countries <- read_csv2("data-raw/countries.csv")
```

Peklo jménem locale encoding

Kromě problému s oddělovači se při práci s neanglickým textem setkáte pravděpodobně ještě s jedním problémem: nesprávným zobrazením českých znaků (resp. znaků, které nejsou obsaženy v anglosaské abecedě). Uchovávání textu ve výpočetní technice je komplexním problémem, pro který existuje velké množství standardů. Autoři většiny operačních systémů, včetně Linuxu a MacOS, se dnes již shodli na využívání univerzálního standardu zvaného UTF-8. Ne tak ovšem Microsoft a Windows využívá několik desítek standardů v závislosti na regionální verzi operačního systému. To vede k problémům při analýze dat, jelikož data vytvořená na jedné regionální verzi Windows se nemusí zobrazit správně na jiné regionální verzi (nebo jiném operačním systému). Pokud k tomu dojde, je nutné specifikovat standard kódování textu (*locale encoding*) manuálně. V českém prostředí se nejčastěji setkáme s encodingem `Windows-1252`, import dat by tedy vypadal následovně

```
countries <- read_csv("data-raw/countries.csv",  
                     locale = locale(encoding = "Windows-1252"))
```

Dalšími populárními verzemi je `Windows-1250`, případně již zmiňovaný UTF-8. Pokud

žádná z těchto možností nepovede ke správnému importu dat, buď vám pomáhej. Seznam existujících standardů je dostupný na [Wikipedii](#).

8.2.2 RDS

Formát `.rds` je specifický pro R. Na rozdíl od `.csv` souborů, `.rds` formát uchovává také metadata, jako je například pořadí kategorií ve faktoru nebo atributy proměnných v dataframě. Import těchto souborů je velmi podobný importu tomu, co jsme již viděli:

```
countries <- read_rds("data-raw/countries.rds")
```

`.rds` soubory nemusí obsahovat pouze jednoduchá tabulková data, ale i složitější objekty, jako jsou listy. Hodí se proto například pro ukládání vytvořených statistických modelů.

8.2.3 SPSS a Stata

Poměrně velké množství sociálněvědních dat je uloženo ve formátech vlastních SPSS a Stata, jelikož tyto programy dlouhou dobu dominovali v akademickém prostředí. Základní instalace R neobsahuje funkce, pomocí kterých bychom mohli data v tomto formátu nainportovat, naštěstí ale pro tento účel existuje několik šikovných balíčků. Jedním z nich je balíček `haven`. Pro import dat vytvořených v SPSS:

```
countries <- read_spss("data-raw/countries.sav") # funkce z balíčku haven
```

Analogicky, pro import dat ze programu Stata:

```
countries <- read_stata("data-raw/countries.dta")
```

8.3 Export dat

Export dat probíhá velmi podobně, jako jejich import. Zatímco importovací funkce začínají slovesem `read_`, exportovací naopak `write_`. Například pro export ve formátu `.csv` do složky `data-clean` v našem pracovním adresáři:

```
write_csv(x = countries, file = "data-clean/countries.csv")
```

Všimněte si, že při exportu dat nepřirazuje výsledku funkce žádné jméno, protože nevytváříme nový objekt uvnitř R. Místo toho pomocí argumentu `x` specifikujeme, které objekt chceme exportovat, a pomocí argumentu `file` poté kam a pod jakým jménem.

Analogicky bychom mohli využít funkce `write_rds()`, `write_sav` (pro export do SPSS formátu) a `write_dta()` (pro export od Stata).

Pozor na lomítka

Pokud používáte Windows, dejte si pozor na lomítka v cestách k souborům. R očekává, že budete používat *forward-slash* (/), tedy například `project/data-raw/countries.csv`. Naopak Windows používá v cestách *back-slash* (\) a cesta by tedy vypadal následovně `project\data-raw\countries.csv`. Při práci v R je nutné používat / pokud cestu kopírujete odjinud, je nutné lomítka ručně opravit. Uživatelé ostatních operačních systémů se patáliemi s lomítky nemusí trápit.

9 První pohled na dataframe

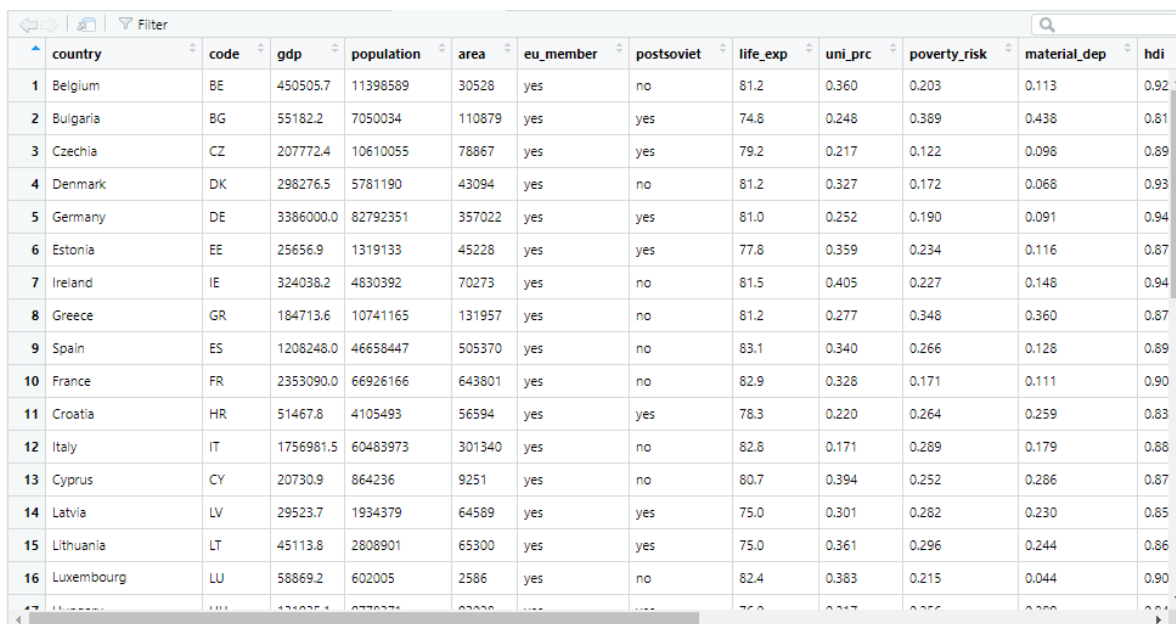
Dataframe je zdaleka nejběžnější objekt pro uchovávání dat v R a tato kapitola je proto věnována právě jim. Pracovat budeme s dataframe `countries`:

```
library(tidyverse) # tidyverse bude od teď standardem naší práce.
countries <- read_csv("data-raw/countries.csv")
```

9.1 Pohled na dataframe

Jako první je dobré se ujistit, že data byla importována správně. Dataframe je možné si prohlédnout pomocí `View()` (pozor, s velkým V!):

```
View(countries)
```



The screenshot shows the RStudio View window displaying a dataframe with 17 rows and 12 columns. The columns are: country, code, gdp, population, area, eu_member, postsoviet, life_exp, uni_prc, poverty_risk, material_dep, and hdi. The rows are numbered 1 through 17, corresponding to countries from Belgium to Luxembourg. The interface includes a search bar at the top right and a filter icon at the top left.

	country	code	gdp	population	area	eu_member	postsoviet	life_exp	uni_prc	poverty_risk	material_dep	hdi
1	Belgium	BE	450505.7	11398589	30528	yes	no	81.2	0.360	0.203	0.113	0.92
2	Bulgaria	BG	55182.2	7050034	110879	yes	yes	74.8	0.248	0.389	0.438	0.81
3	Czechia	CZ	207772.4	10610055	78867	yes	yes	79.2	0.217	0.122	0.098	0.89
4	Denmark	DK	298276.5	5781190	43094	yes	no	81.2	0.327	0.172	0.068	0.93
5	Germany	DE	3386000.0	82792351	357022	yes	yes	81.0	0.252	0.190	0.091	0.94
6	Estonia	EE	25656.9	1319133	45228	yes	yes	77.8	0.359	0.234	0.116	0.87
7	Ireland	IE	324038.2	4830392	70273	yes	no	81.5	0.405	0.227	0.148	0.94
8	Greece	GR	184713.6	10741165	131957	yes	no	81.2	0.277	0.348	0.360	0.87
9	Spain	ES	1208248.0	46658447	505370	yes	no	83.1	0.340	0.266	0.128	0.89
10	France	FR	2353090.0	66926166	643801	yes	no	82.9	0.328	0.171	0.111	0.90
11	Croatia	HR	51467.8	4105493	56594	yes	yes	78.3	0.220	0.264	0.259	0.83
12	Italy	IT	1756981.5	60483973	301340	yes	no	82.8	0.171	0.289	0.179	0.88
13	Cyprus	CY	20730.9	864236	9251	yes	no	80.7	0.394	0.252	0.286	0.87
14	Latvia	LV	29523.7	1934379	64589	yes	yes	75.0	0.301	0.282	0.230	0.85
15	Lithuania	LT	45113.8	2808901	65300	yes	yes	75.0	0.361	0.296	0.244	0.86
16	Luxembourg	LU	58869.2	602005	2586	yes	no	82.4	0.383	0.215	0.044	0.90
17

Obrázek 9.1: Pohled na náš dataframe

`View()` zobrazí dataframe v novém interaktivním okně, pomocí kterého lze zkontrolovat, jestli byla data nahrána správně, jestli jsou proměnné správně pojmenované a všechny text se zobrazuje bez problémů. U větších dat může být ovšem funkce být poněkud pomalá. Lepší je proto podívat se pouze na výsek dat. Funkce `head()` umožňuje zobrazit několik prvních řádků dataframu (a analogicky funkce `tail()` zobrazí poslední řádky):

```
head(countries, n = 3)
```

```
# A tibble: 3 x 17
  country code      gdp popula~1  area eu_me~2 posts~3 life_~4 uni_prc pover~5
  <chr>   <chr>   <dbl> <dbl> <dbl> <chr>   <chr>   <dbl> <dbl> <dbl>
1 Belgium BE      450506. 11398589 30528 yes    no      81.2  0.36  0.203
2 Bulgaria BG       55182.  7050034 110879 yes    yes     74.8  0.248  0.389
3 Czechia CZ      207772. 10610055  78867 yes    yes     79.2  0.217  0.122
# ... with 7 more variables: material_dep <dbl>, hdi <dbl>,
#   foundation_date <date>, maj_belief <chr>, dem_index <dbl>, di_cat <chr>,
#   hd_title_name <chr>, and abbreviated variable names 1: population,
#   2: eu_member, 3: postsoviet, 4: life_exp, 5: poverty_risk
```

Poslední možností je vytisknout dataframe přímo do konzole, což však s výjimkou velmi malých dat není příliš přehledné.

9.2 Sumarizace dataframu

Balíček `dplyr` z Tidyverse nabízí o něco kompaktnější funkci pro prohlédnutí našich dat, `glimpse()`:

```
glimpse(countries)
```

```
Rows: 38
Columns: 17
$ country      <chr> "Belgium", "Bulgaria", "Czechia", "Denmark", "Germany"~
$ code         <chr> "BE", "BG", "CZ", "DK", "DE", "EE", "IE", "GR", "ES", ~
$ gdp          <dbl> 450505.7, 55182.2, 207772.4, 298276.5, 3386000.0, 2565~
$ population   <dbl> 11398589, 7050034, 10610055, 5781190, 82792351, 131913~
$ area         <dbl> 30528, 110879, 78867, 43094, 357022, 45228, 70273, 131~
$ eu_member    <chr> "yes", "yes", "yes", "yes", "yes", "yes", "yes", "yes"~
$ postsoviet   <chr> "no", "yes", "yes", "no", "yes", "yes", "no", "no", "n~
$ life_exp     <dbl> 81.2, 74.8, 79.2, 81.2, 81.0, 77.8, 81.5, 81.2, 83.1, ~
```



```

$ uni_prc      <dbl> 0.360, 0.248, 0.217, 0.327, 0.252, 0.359, 0.405, 0.277~
$ poverty_risk <dbl> 0.203, 0.389, 0.122, 0.172, 0.190, 0.234, 0.227, 0.348~
$ material_dep <dbl> 0.113, 0.438, 0.098, 0.068, 0.091, 0.116, 0.148, 0.360~
$ hdi          <dbl> 0.92, 0.81, 0.89, 0.93, 0.94, 0.87, 0.94, 0.87, 0.89, ~
$ foundation_date <date> 1831-07-21, 1989-11-10, 1993-01-01, 2053-05-19, 1949--
$ maj_belief    <chr> "catholic", "orthodox", "nonbelief", "protestantism", ~
$ dem_index     <dbl> 7.78, 7.03, 7.69, 9.22, 8.68, 7.97, 9.15, 7.29, 8.08, ~
$ di_cat        <chr> "Flawed democracy", "Flawed democracy", "Flawed democr~
$ hd_title_name <chr> "King - Philippe", "President - Rumen Radev", "Preside~

```

Alternativní možností je generická funkce `summary()`:

```
summary(countries)
```

```

      country      code      gdp      population
Length:38      Length:38      Min.   : 10735      Min.   : 38114
Class :character Class :character 1st Qu.: 43947      1st Qu.: 2075301
Mode  :character Mode  :character Median : 201612      Median : 7001444
                        Mean  : 484601      Mean  :16754743
                        3rd Qu.: 458715      3rd Qu.:11398589
                        Max.   :3386000      Max.   :82792351
                        NA's   :3           NA's   :1

      area      eu_member      postsoviet      life_exp
Min.   : 160      Length:38      Length:38      Min.   :74.80
1st Qu.: 41344      Class :character Class :character 1st Qu.:76.80
Median : 73874      Mode  :character Mode  :character Median :81.00
Mean   :156019
3rd Qu.:242305
Max.   :783562

      uni_prc      poverty_risk      material_dep      hdi
Min.   :0.1550      Min.   :0.1220      Min.   :0.0420      Min.   :0.7600
1st Qu.:0.2200      1st Qu.:0.1770      1st Qu.:0.0820      1st Qu.:0.8425
Median :0.3010      Median :0.2200      Median :0.1280      Median :0.8800
Mean   :0.2915      Mean   :0.2403      Mean   :0.1799      Mean   :0.8739
3rd Qu.:0.3630      3rd Qu.:0.2820      3rd Qu.:0.2590      3rd Qu.:0.9200
Max.   :0.4050      Max.   :0.4160      Max.   :0.4810      Max.   :0.9500
NA's   :3           NA's   :5           NA's   :5

      foundation_date      maj_belief      dem_index      di_cat
Min.   :1291-08-01      Length:38      Min.   :4.370      Length:38
1st Qu.:1919-11-11      Class :character 1st Qu.:6.670      Class :character
Median :1975-06-01      Mode  :character Median :7.710      Mode  :character

```

```
Mean      :1930-10-08      Mean      :7.639
3rd Qu.   :1991-08-28      3rd Qu.  :8.680
Max.      :2053-05-19      Max.     :9.870
NA's      :1

hd_title_name
Length:38
Class :character
Mode  :character
```

V některých případech nepotřebujeme prohlížet celý dataframe. Pokud si nejsme jistí, jak se jmenují proměnné v našem dataframu, pomůžeme nám funkce `names()`:

```
names(countries)
```

```
[1] "country"      "code"          "gdp"           "population"
[5] "area"         "eu_member"     "postsoviet"    "life_exp"
[9] "uni_prc"      "poverty_risk"  "material_dep"  "hdi"
[13] "foundation_date" "maj_belief"    "dem_index"     "di_cat"
[17] "hd_title_name"
```

Celkový počet proměnných lze zjistit pomocí funkce `ncol()`, případně `length()`, pro kontrolu počtu řádků potom příbuzná `nrow()`:

```
ncol(countries)
```

```
[1] 17
```

```
nrow(countries)
```

```
[1] 38
```

10 Práce se sloupci

První dimenzí dataframu jsou sloupce, reprezentující zpravidla naše proměnné. Pro práci se sloupci nabízí Tidyverse šikovnou funkci `select()` a pár jejích příbuzných. V této kapitole si ukážeme, jak efektivně vybírat sloupce v dataframu, přejmenovávat je a řadit podle našich přání.

10.1 Výběr sloupců

Předmětem analýzy v mnoha případech není celý dataframe, ale pouze jeho výsek. Základní aplikace již zmíněné funkce `select()` je přímočará. Prvním argumentem je dataframe, který chceme filtrovat, zbylými sloupce, které chceme zachovat:

```
select(countries, country, life_exp, postsoviet)
```

```
# A tibble: 38 x 3
  country  life_exp postsoviet
  <chr>    <dbl> <chr>
1 Belgium    81.2 no
2 Bulgaria   74.8 yes
3 Czechia    79.2 yes
4 Denmark    81.2 no
5 Germany    81    yes
6 Estonia    77.8 yes
7 Ireland    81.5 no
8 Greece     81.2 no
9 Spain      83.1 no
10 France    82.9 no
# ... with 28 more rows
```

Pokud by naším cílem bylo se některého sloupce zbavit, využijeme mínusu (-), podobně jako u indexování (viz. Sekce 6.3). Pro vyřazení více sloupců využijeme již dobře známou funkci `c()`:

```
select(countries, -c(country, life_exp, postsoviet))
```

```
# A tibble: 38 x 14
  code      gdp popul~1  area eu_me~2 uni_prc pover~3 mater~4  hdi foundati~5
  <chr>   <dbl>   <dbl> <dbl> <chr>   <dbl>   <dbl>   <dbl> <dbl> <date>
1 BE     4.51e5  1.14e7  30528 yes     0.36    0.203   0.113  0.92 1831-07-21
2 BG     5.52e4  7.05e6  110879 yes     0.248   0.389   0.438  0.81 1989-11-10
3 CZ     2.08e5  1.06e7  78867 yes     0.217   0.122   0.098  0.89 1993-01-01
4 DK     2.98e5  5.78e6  43094 yes     0.327   0.172   0.068  0.93 2053-05-19
5 DE     3.39e6  8.28e7  357022 yes     0.252   0.19    0.091  0.94 1949-05-23
6 EE     2.57e4  1.32e6  45228 yes     0.359   0.234   0.116  0.87 1918-02-24
7 IE     3.24e5  4.83e6  70273 yes     0.405   0.227   0.148  0.94 1937-12-29
8 GR     1.85e5  1.07e7  131957 yes     0.277   0.348   0.36   0.87 1975-11-19
9 ES     1.21e6  4.67e7  505370 yes     0.34    0.266   0.128  0.89 1978-12-06
10 FR    2.35e6  6.69e7  643801 yes     0.328   0.171   0.111  0.9  1958-10-05
# ... with 28 more rows, 4 more variables: maj_belief <chr>, dem_index <dbl>,
#   di_cat <chr>, hd_title_name <chr>, and abbreviated variable names
#   1: population, 2: eu_member, 3: poverty_risk, 4: material_dep,
#   5: foundation_date
```

10.2 Pomocné funkce

Ručně vypisovat všechny proměnné, které chceme vybrat, je u větších dataframů zdlouhavá činnost. Naštěstí pro nás obsahuje balíček `dplyr` řadu pomocných funkcí (*selection helpers*).

Nezákladnější pomocnou funkcí je `:`, která vybere všechny sloupce v rozpětí. Například, pro vybrání `country`, `area` a všech proměnných mezi nimi:

```
select(countries, country:area)
```

Pokud chceme vybrat všechny proměnné v dataframu, nemusíme využívat `:`, stačí využít funkci `everything()`. Možnost vybrat úplně všechny proměnné se nemusí zdát na první pohled užitečná, nachází ale často využití při převodu dat mezi širokým a dlouhým formátem (viz. níže).

Sadou užitečných pomocných funkcí jsou `starts_with()`, `ends_with()` a `contains()`. Funkce `starts_with()` vybere všechny sloupce začínající stejnými znaky, `ends_with()` naopak všechny sloupce končící stejně. `contains()` identifikuje sloupce, jejichž název obsahuje specifikovaný řetězec znaků. Například pro vybrání všech proměnných, jejichž název obsahuje podtržítka:

```
select(countries, contains("_"))
```

Poslední pomocnou funkcí, kterou si zde ukážeme, je `where()`. Pomocí ní lze vybrat všechny sloupce splňující danou logickou podmínku. Pro vybrání všech numerických proměnných:

```
select(countries, where(is.numeric))
```

Analogicky by bylo možné aplikovat funkce `is.character`, `is.factor` nebo `is.logical`. Tyto funkce jsou uvnitř `where()` použity bez závorek.

Dokumentace k pomocným funkcím je dostupná pomocí `help("tidyr_tidy_select")`.

10.3 Přejmenování proměnných

Ne vždy budeme spokojeni s tím, jak jsou naše proměnné pojmenované. Způsobů, jak proměnnou přejmenovat je řada, preferovanou metodou v rámci Tidyverse je využití funkce `rename()`. Její aplikace je jednoduchá, nové jméno je vždy specifikované ve formátu `nove_jmeno = stare_jmeno`. Pokud by se nám například nelíbilo jméno proměnné `uni_prc`, můžeme ho změnit na výstižnější `university_educated`. Tady poprvé narážíme na řetězení funkcí pomocí *pipes*, představených v kapitole věnované funkcím (Sekce 6.4):

```
countries %>%  
  rename(university_educated = uni_prc) %>%  
  select(country, university_educated)
```

```
# A tibble: 38 x 2  
  country university_educated  
  <chr>          <dbl>  
1 Belgium          0.36  
2 Bulgaria         0.248  
3 Czechia          0.217  
4 Denmark          0.327  
5 Germany          0.252  
6 Estonia          0.359  
7 Ireland          0.405  
8 Greece           0.277  
9 Spain            0.34  
10 France          0.328  
# ... with 28 more rows
```

Dataframe `countries` již není obsažen ve funkci `filter()`, ale je do ní poslán skrze *pipe* (`%>%`). Vzpomeňme si, že *pipe* vezme objekt na její levé straně a vloží ho do prvního argumentu funkce napravo.

O něco komplexnější funkcí je `rename_with()`. Ta umožňuje přejmenovávat proměnné funkce programátorsky. Co kdybychom například chtěli převést názvy proměnných ze *snake_case* na *kebab-case*? Jediné, co pro to musíme udělat je změnit podtržítka v názvech proměnných na pomlčky. Jednou možností by bylo ručně přepsat názvy všech proměnných. Efektivnější variantou je využít funkce `rename_with()` v kombinaci s funkcí `str_replace()`:

```
countries %>%
  rename_with(str_replace, pattern = "_", replacement = "-") %>%
  select(country, contains("-"))
```

```
# A tibble: 38 x 11
  country eu-memb~1 life--2 uni-p~3 pover~4 mater~5 foundati~6 maj-b~7 dem-i~8
  <chr>    <chr>          <dbl>  <dbl>  <dbl>  <dbl> <date>      <chr>    <dbl>
1 Belgium yes           81.2   0.36   0.203  0.113 1831-07-21 cathol~  7.78
2 Bulgaria yes           74.8   0.248  0.389  0.438 1989-11-10 orthod~  7.03
3 Czechia yes           79.2   0.217  0.122  0.098 1993-01-01 nonbel~  7.69
4 Denmark yes           81.2   0.327  0.172  0.068 2053-05-19 protes~  9.22
5 Germany yes           81     0.252  0.19   0.091 1949-05-23 cathol~  8.68
6 Estonia yes           77.8   0.359  0.234  0.116 1918-02-24 nonbel~  7.97
7 Ireland yes           81.5   0.405  0.227  0.148 1937-12-29 cathol~  9.15
8 Greece  yes           81.2   0.277  0.348  0.36   1975-11-19 orthod~  7.29
9 Spain   yes           83.1   0.34   0.266  0.128 1978-12-06 cathol~  8.08
10 France yes           82.9   0.328  0.171  0.111 1958-10-05 cathol~  7.8
# ... with 28 more rows, 2 more variables: `di-cat` <chr>,
# `hd-title_name` <chr>, and abbreviated variable names 1: `eu-member`,
# 2: `life-exp`, 3: `uni-prc`, 4: `poverty-risk`, 5: `material-dep`,
# 6: `foundation-date`, 7: `maj-belief`, 8: `dem-index`
```

10.4 Pořadí proměnných

Pořadí proměnných v dataframu je možné upravovat pomocí funkce `relocate()`. Tu je možné využít pro jednotlivé proměnné i v kombinaci s pomocnými funkcemi. Pomocí argumentů `.before` a `.after` je možné určit novou pozici nových sloupců. Pokud bychom chtěli oddělit numerické proměnné od kategoriálních, využijeme následující kombinací funkcí:

```
relocate(countries, where(is.numeric), .after = last_col())
```

```

# A tibble: 38 x 17
  country code eu_member postsoviet foundatio~1 maj_b~2 di_cat hd_ti~3    gdp
  <chr>   <chr> <chr>   <chr>   <date>     <chr>   <chr> <chr>   <dbl>
1 Belgium BE    yes     no       1831-07-21 cathol~ Flawe~ King -- 4.51e5
2 Bulgaria BG    yes     yes      1989-11-10 orthod~ Flawe~ Presid~ 5.52e4
3 Czechia CZ    yes     yes      1993-01-01 nonbel~ Flawe~ Presid~ 2.08e5
4 Denmark DK    yes     no       2053-05-19 protes~ Full ~ Queen ~ 2.98e5
5 Germany DE    yes     yes      1949-05-23 cathol~ Full ~ Presid~ 3.39e6
6 Estonia EE    yes     yes      1918-02-24 nonbel~ Flawe~ Presid~ 2.57e4
7 Ireland IE    yes     no       1937-12-29 cathol~ Full ~ Presid~ 3.24e5
8 Greece  GR    yes     no       1975-11-19 orthod~ Flawe~ Presid~ 1.85e5
9 Spain   ES    yes     no       1978-12-06 cathol~ Full ~ King -- 1.21e6
10 France FR    yes     no       1958-10-05 cathol~ Flawe~ Presid~ 2.35e6
# ... with 28 more rows, 8 more variables: population <dbl>, area <dbl>,
#   life_exp <dbl>, uni_prc <dbl>, poverty_risk <dbl>, material_dep <dbl>,
#   hdi <dbl>, dem_index <dbl>, and abbreviated variable names
#   1: foundation_date, 2: maj_belief, 3: hd_title_name

```

11 Práce s řádky

Obdobně jako je funkce `select()` neocenitelným pomocníkem pro práci se sloupci dataframů, její příbuzná `filter()` nám dobře poslouží pro filtrování řádků. Tato kapitola je věnována právě jí, ale také rodině funkcí `slice()` a funkci `arrange()`.

11.1 Filtrování řádků

Pro filtrování řádků je třeba trocha výrokové logiky. Základními logickými operátory v R jsou `==` (EQUAL), `|` (OR) a `&` (AND). Nepřekvapivě, pro HIGHER THAN používám `>=`, naopak pro LOWER THAN slouží `<=`. Negace se provádí pomocí vykřičníku, tedy například NOT EQUAL je `!=`.

Vybavení těmito znalostmi, filtrování řádků není obtížný úkol. Hlavní funkcí je zde `filter()`:

```
filter(countries, postsoviet == "yes" & gdp > 100000)
```

```
# A tibble: 5 x 17
  country code      gdp popula-1  area eu_me~2 posts~3 life_~4 uni_prc pover~5
  <chr>  <chr>    <dbl>    <dbl> <dbl> <chr>    <chr>    <dbl>  <dbl>  <dbl>
1 Czechia CZ      207772. 10610055  78867 yes     yes      79.2    0.217  0.122
2 Germany DE     3386000  82792351 357022 yes     yes      81      0.252  0.19
3 Hungary HU     131935.  9778371  93028 yes     yes      76      0.217  0.256
4 Poland  PL     496462. 37976687 312685 yes     yes      77.8    0.272  0.195
5 Romania RO     202884. 19530631 238391 yes     yes      75.2    0.155  0.357
# ... with 7 more variables: material_dep <dbl>, hdi <dbl>,
#   foundation_date <date>, maj_belief <chr>, dem_index <dbl>, di_cat <chr>,
#   hd_title_name <chr>, and abbreviated variable names 1: population,
#   2: eu_member, 3: postsoviet, 4: life_exp, 5: poverty_risk
```

Podmínky filtrování lze kombinovat, například vyfiltrovat pouze postsovětské země s hrubým domácím produktem větším než 100 000 milionů euro.

💡 %in% místo |

Občas je naším cílem vyfiltrovat řádky obsahující některou z vybraných hodnot kategoriální proměnné, například všechny náboženské skupiny spadající pod křesťanství. Jednou možností je:

```
filter(countries, maj_belief == "catholic" | maj_belief == "orthodox" | maj_belief == "p
```

Tento přístup funguje, je ale zbytečně květnatý. Místo něj je možné aplikovat operátor %in%, pomocí kterého můžeme vyfiltrovat všechny hodnoty objevující se ve zvoleném vektoru:

```
filter(countries, maj_belief %in% c("catholic", "orthodox", "protestantism"))
```

Oba tyto příkazy vedou ke stejnému výsledku, ten druhý je ale výrazně kompaktnější.

Jak si jistě dokážete představit, funkce `select()` a `filter()` jsou často využívané dohromady:

```
countries %>%  
  filter(postsoviet == "yes") %>%  
  select(country, postsoviet, life_exp)
```

A tibble: 16 x 3

	country <chr>	postsoviet <chr>	life_exp <dbl>
1	Bulgaria	yes	74.8
2	Czechia	yes	79.2
3	Germany	yes	81
4	Estonia	yes	77.8
5	Croatia	yes	78.3
6	Latvia	yes	75
7	Lithuania	yes	75
8	Hungary	yes	76
9	Poland	yes	77.8
10	Romania	yes	75.2
11	Slovenia	yes	80.9
12	Slovakia	yes	77.4
13	North Macedonia	yes	75.9
14	Albania	yes	76.4
15	Serbia	yes	76.3
16	Bosnia and Herzegovina	yes	77.3

11.2 Řezání dataframů

V některých případech budeme chtít filtrovat na základě pořadí řádků dataframů. K tomu nám poslouží rodina funkcí `slice` z balíčku `dplyr`.

Prvním členem této rodiny je funkce `slice()`. Její aplikace je velmi podobná klasickému indexování pomocí hranatých závorek. Například výběr prvního řádku v dataframů:

```
slice(countries, 1)
```

Je ekvivalentní `countries[1,]` a vrátí první řádek dataframů. Obdobně podobné jsou i funkce `slice(countries, -1)` a `countries[-1,]`, které vrátí všechny řádky kromě prvního. Funkce `slice()` je však pouze základem pro řadu dalších užitečných funkcí.

Další dvě funkce, které nám již svým fungováním budou povědomé jsou `slice_head()` a `slice_tail()`. Ty, obdobně jako funkce `head()` a `tail()`, vrátí prvních n řádků v dataframů. Na rozdíl od svých příbuzných ze základní instalace R, ovšem `slice` funkce umožňují vybrat nejen absolutní, ale i relativní počet řádků. Například pro vybrání prvních deseti procent dataframů:

```
slice_head(countries, prop = 0.1)
```

Pro vybrání absolutního počtu řádku slouží argument `n`.

O něco zajímavější jsou funkce `slice_max()` a `slice_min()`. Ty umožňují vybrat n řádků s nejvyšší, respektive nejnižší, hodnotou dané proměnné. Pomocí těchto funkcí můžeme například jednoduše zjistit, které tři země v našem dataframů mají nejvyšší naději na dožití:

```
slice_max(countries, order_by = life_exp, n = 3) %>%  
  select(country, life_exp)
```

```
# A tibble: 3 x 2  
  country    life_exp  
  <chr>      <dbl>  
1 Switzerland 83.3  
2 Spain       83.1  
3 France      82.9
```

a které naopak nejnižší:

```
slice_min(countries, order_by = life_exp, n = 3) %>%  
  select(country, life_exp)
```

```
# A tibble: 3 x 2
  country  life_exp
  <chr>    <dbl>
1 Bulgaria 74.8
2 Latvia   75
3 Lithuania 75
```

Funkce `slice_max()` a `slice_min()` jsem zde zkombinovaly s funkcí `select()`, abychom vybrali jen relevantní proměnné.

Posledním členem rodiny je funkce `slice_sample()`, která vybere náhodné řádky dataframu. Tato funkce najde uplatnění zejména v simulačních studiích a technikách.

```
slice_sample(countries, n = 3)
```

11.3 Group_by()

V tuto chvíli si možná někteří čtenáři říkají, jaké je využití `slice` funkcí oproti jejich klasickým variantám, jako je `head()` nebo `tail()`. Jednou z jejich velkých předností je možnost kombinovat je s funkcí `group_by()`.

Funkce `group_by()` umožňuje rozdělit dataframu na podskupiny a aplikovat funkce z balíčku `dplyr` na každou z podskupin zvlášť. Podskupiny jsou definované kategoričnou proměnnou v dataframu. Tímto způsobem můžeme zjistit nejen které země se těší nejvyšší naději na dožití obecně, ale i to, jak jsou na tom západní a postsovětské země zvlášť:

```
countries %>%
  group_by(postsoviet) %>%
  slice_max(order_by = life_exp, n = 3) %>%
  select(country, postsoviet, life_exp)
```

```
# A tibble: 6 x 3
# Groups:   postsoviet [2]
  country  postsoviet life_exp
  <chr>    <chr>      <dbl>
1 Switzerland no          83.3
2 Spain    no          83.1
3 France   no          82.9
4 Germany  yes         81
5 Slovenia yes         80.9
6 Czechia  yes         79.2
```

Zatímco mezi západními zeměmi vedou Švýcarsko, Španělsko a Francie, v postsovětské skupině je to Německo, Slovinsko a Česká republika. Třídít je možné i pomocí většího počtu proměnných, například pro třídění podle postsovětské historie a členství v Evropské unii bychom použili `group_by(postsoviet, eu_member)`. Jak jistě tušíte, funkce `group_by()` má mnoho využití a budeme se s ní setkávat opakovaně i následujících kapitolách.

11.4 Pořadí řádků

Posledním typem operace, kterou si v této kapitole představíme, je řazení řádků pomocí funkce `arrange()`. Pořadí zemí v dataframu `countries` podle naděje na dožití získáme jednoduše:

```
countries %>%
  arrange(life_exp) %>%
  select(country, life_exp)
```

```
# A tibble: 38 x 2
  country      life_exp
  <chr>        <dbl>
1 Bulgaria      74.8
2 Latvia        75
3 Lithuania     75
4 Romania       75.2
5 North Macedonia 75.9
6 Hungary       76
7 Serbia        76.3
8 Albania       76.4
9 Turkey        76.4
10 Montenegro   76.8
# ... with 28 more rows
```

Při bližším pohledu zjistíme, že země jsou seřazený vzestupně. Nejhůře se vede Bulharsko a Litva s Lotyšskem. Co kdyby nás ale zajímaly země s nejvyšší nadějí na dožití? Pro sestupné řazení zkombinujeme funkci `arrange()` s funkcí `desc()`:

```
countries %>%
  arrange(desc(life_exp)) %>%
  select(country, life_exp)
```

```
# A tibble: 38 x 2
```

```
country    life_exp
<chr>      <dbl>
1 Switzerland 83.3
2 Spain      83.1
3 France     82.9
4 Italy      82.8
5 Norway     82.5
6 Luxembourg 82.4
7 Sweden     82.4
8 Iceland    82.4
9 Austria    81.9
10 Netherlands 81.6
# ... with 28 more rows
```

Nejvyšší naději na dožití se těší Švýcarsko, se Španělskem v těsném závěsu.

12 Široký a dlouhý formát

Důležitými pojmy v analýze jsou široký (*wide*) a dlouhý (*long*) formát uchovávání dat. Naštěstí pro nás se nejedná o nic složitého, jde pouze o způsob orientace dataframů. V širokém formátu jsou uchovány horizontálně, zatímco v dlouhém formátu jsou data orientovány vertikálně. Obsah dataframu se nemění, jediným rozdílem je forma:

Široký formát				Dlouhý formát		
country	gdp	life_exp	poverty_risk	country	name	value
Czechia	207772.4	79.2	0.122	Czechia	gdp	207772.400
Germany	3386000.0	81.0	0.190	Czechia	life_exp	79.200
Norway	368388.9	82.5	0.160	Czechia	poverty_risk	0.122
				Germany	gdp	3386000.000
				Germany	life_exp	81.000
				Germany	poverty_risk	0.190
				Norway	gdp	368388.900
				Norway	life_exp	82.500
				Norway	poverty_risk	0.160

Široký vs Dlouhý formát

Zpravidla je široký formát intuitivnější pro lidi, zatímco ten dlouhý se lépe čte počítačům. V praxi proto budeme převádět data mezi formáty často a poslouží nám k tomu dvojice funkcí z balíčku `tidyr`: `pivot_wider()` a `pivot_longer()`.

12.1 Z širokého do dlouhého formátu

Vraťme se k datasetu `countries`. V rámci analýzy nás může zajímat, jaká je nejvyšší pozorovaná hodnota každé z numerických proměnné. Jako bonus bychom také rádi věděli, které zemi tato hodnota patří.

Možností, jak se dostat ke kýženému výsledku, je více. Jedna z těch elegantnějších zahrnuje převedení dataframu do dlouhého formátu. Začneme tím, že z dataframu vybereme proměnnou `country` a všechny numerické proměnné, k čemuž poslouží funkce `select()`. Poté bude následovat funkce `pivot_longer()`. Tato funkce má jeden povinný argument, a

to `cols`, pomocí kterého specifikujeme, které proměnné chceme převést do širokého formátu. Není nutné vybrat všechny existující proměnné, naopak pro naše potřeby je lepší nechat proměnnou `countries` v původní podobě. Kromě `cols`, jsou dalšími dvěma užitečnými argumenty `names_to` a `values_to`, pomocí kterých lze specifikovat názvy nově vytvořených sloupců. První z argumentů určí název sloupce obsahující názvy původních proměnných, druhý poté název sloupce, ve kterém budou uchovány naměřené hodnoty.

```
countries %>%
  select(country, where(is.numeric)) %>%
  pivot_longer(cols = -country,
               names_to = "variable",
               values_to = "max_value")
```

```
# A tibble: 342 x 3
  country variable      max_value
  <chr>    <chr>          <dbl>
1 Belgium gdp             450506.
2 Belgium population 11398589
3 Belgium area         30528
4 Belgium life_exp      81.2
5 Belgium uni_prc        0.36
6 Belgium poverty_risk  0.203
7 Belgium material_dep  0.113
8 Belgium hdi            0.92
9 Belgium dem_index     7.78
10 Bulgaria gdp         55182.
# ... with 332 more rows
```

Tento kód provede výše popsané. Pomocí kombinace funkcí `select()` a pomocné funkce `where()` (viz. Sekce 10.2) vybereme proměnné. Funkce `pivot_longer()` se postará o zbytek. Argumentem `cols` určíme, že převedeny do dlouhého formátu mají být všechny proměnné kromě `country` a nově vytvořené proměnné se mají jmenovat `variable` a `max_value`. Výsledkem je dataframe s menším počtem sloupců, zato výrazně větším počtem řádků. Každý řádek reprezentuje hodnotu jedné proměnné v jedné zemi.

Teď už zbývá jen vybrat pro každou z původních proměnných nejvyšší naměřenou hodnotu. Toho docílíme pomocí nám již známých funkcí `group_by()` a `slice_max()`. A to je vše! Z výsledného dataframu lze vyčíst že nejvyššímu pozorovanému Indexu lidského rozvoje (gdp) se těší Norsko, nebo že nejvyšší podíl lidí ohrožených chudobou je 41,6 % a trpí jím Severní Makedonie.

```

countries %>%
  select(country, where(is.numeric)) %>%
  pivot_longer(cols = -country,
               names_to = "variable",
               values_to = "max_value") %>%
  group_by(variable) %>%
  slice_max(max_value)

```

```

# A tibble: 9 x 3
# Groups:   variable [9]
  country      variable      max_value
  <chr>        <chr>          <dbl>
1 Turkey      area             783562
2 Norway      dem_index        9.87
3 Germany     gdp              3386000
4 Norway      hdi               0.95
5 Switzerland life_exp         83.3
6 North Macedonia material_dep    0.481
7 Germany     population      82792351
8 North Macedonia poverty_risk    0.416
9 Ireland     uni_prc          0.405

```

12.2 Z dlouhého do širokého formátu

Jak již asi tušíte, opakem `pivot_longer()` je funkce `pivot_wider()`, pomocí které je možné řádky “roztáhnout” do sloupců. Tato funkce má dva povinné argumenty, a to `names_from` a `values_from`. První z těchto argumentů převede hodnoty ve vybraném sloupci na názvy nových sloupců. Druhý argument poté nové sloupce zaplní hodnotami ze zvolené proměnné.

Poněkud umělým, ale názorným příkladem může být, pokud by naším cílem bylo vytvořit dataframe obsahující minimální hodnotu ohrožení chudobou podle převažujícího náboženského vyznání. Tento dataframe by měl být dobře srozumitelný pro naše čtenáře, a měl by proto mít podobu kontingenční tabulky.

Začneme podobně, jako v předchozím cvičení. Nejdříve vybereme všechny relevantní proměnné, tedy převažující náboženské vyznání (`maj_belief`), členství v Evropské unii (`eu_member`) a HDP (`gdp`). Poté kombinací funkcí `group_by()` a `slice_min()` získáme nejnižší pozorované hodnoty ohrožení chudobou pro každou kombinaci převažujícího vyznání a členství v EU. Nakonec už zbývá pouze využít funkce `pivot_wider()` pro převedení proměnné `maj_belief()` do sloupců a zaplnění nově vzniklých sloupců pomocí hodnot z `poverty_risk`. A máme hotovo


```

countries %>%
  select(maj_belief, eu_member, poverty_risk) %>%
  group_by(maj_belief, eu_member) %>%
  slice_min(poverty_risk) %>%
  pivot_wider(names_from = maj_belief, values_from = poverty_risk)

```

```

# A tibble: 2 x 6
# Groups:   eu_member [2]
  eu_member catholic  islam nonbelief orthodox protestantism
  <chr>          <dbl> <dbl>    <dbl>    <dbl>         <dbl>
1 no            0.181 0.413    NA        0.367          0.16
2 yes           0.163 NA      0.122    0.252          0.157

```

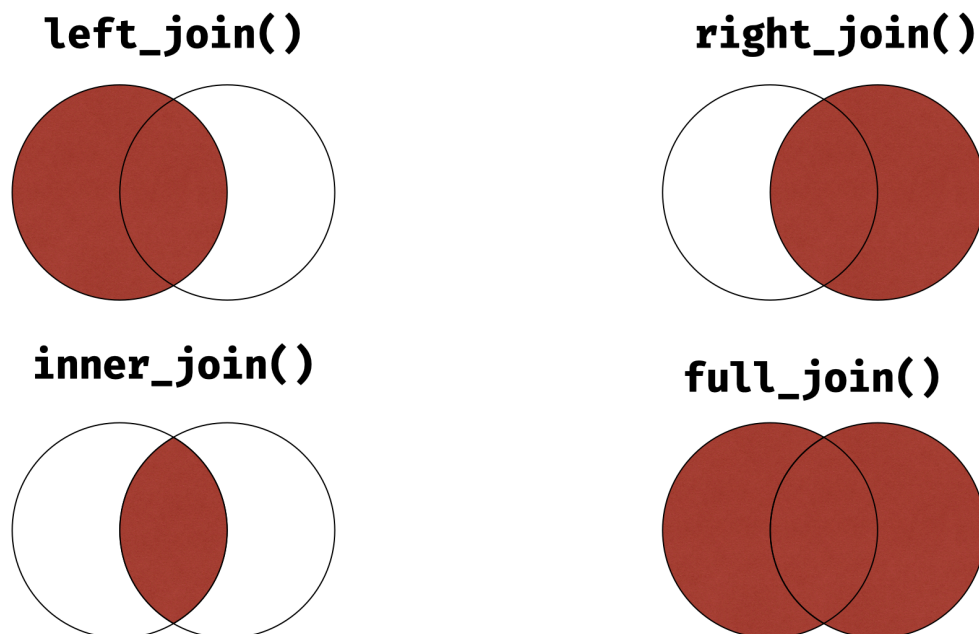
Všimněme si, že některé buňky jsou prázdné, protože naše data neobsahují všechny kombinace náboženského vyznání a členství v Evropské unii.

13 Spojování dataframů

V ideálním světě by všechna data potřebná pro naši analýzu byla připravena v jednom úhledném datasetu. Praxe je ovšem mnohem krutější a nutí nás si data čistit svépomocí. Jednou z častých nutností je spojovat větší počet dílčích datasetů dohromady, k čemuž nám poslouží sada funkcí `*_join()`.

13.1 Spojovací funkce

Spojovat dva datasety lze více způsoby, všechny ale vyžadují klíč, tedy proměnou, která je přítomná v obou dataframech a pomocí které budou propojeny jednotlivé řádky.



Obrázek 13.1: `*_join()` funkce

Funkcí pro spojení dvou dataframů nabízí Tidyverse více. První z nich, `left_join()`, připojí k prvnímu specifikovanému dataframu všechny řádky druhého dataframu se shodnou hodnotou

klíče. Řádky, které se nachází pouze v druhém dataframu, ale ne v prvním, jsou ztraceny. Analogickou funkcí je poté `right_join()`, která zachová pouze řádky s hodnotami klíče, nacházejícím se v druhém dataframu. `inner_join()` je nejpřísnější z funkcí a při spojení dataframů zachová pouze řádky s hodnotami nacházejícími se v obou datasetech. Naopak funkce `full_join()` je nejliberálnější a zachová při spojení všechna data.

Pro ukázkou spojovacích funkcí opustíme dataset `countries` a místo toho se podíváme do zvířecí říše, konkrétně na data z American Kennel Club. Jedná se o dva dasety, které jsou dostupné v rámci [TidyTuesday](#). Importovat je lze přímo ze repozitáře projektu:

```
breed_traits <- read_csv('https://raw.githubusercontent.com/rfordatascience/tidyuesday/master/data/akc/breed_traits.csv')
breed_ranks <- read_csv('https://raw.githubusercontent.com/rfordatascience/tidyuesday/master/data/akc/breed_ranks.csv')
```

První z těchto datasetů obsahuje hodnocení o vlastnostech psích plemen jak byly hodnoceny členy klubu. Vlastností tu najdeme celou řadu, od délky kožichu po přátelskost nebo slintavost. Druhý dataset obsahuje informace o popularitě plemen za posledních zhruba 10 let, plus pár popisných odkazů:

```
names(breed_traits)
```

```
[1] "Breed" "Affectionate With Family"
[3] "Good With Young Children" "Good With Other Dogs"
[5] "Shedding Level" "Coat Grooming Frequency"
[7] "Drooling Level" "Coat Type"
[9] "Coat Length" "Openness To Strangers"
[11] "Playfulness Level" "Watchdog/Protective Nature"
[13] "Adaptability Level" "Trainability Level"
[15] "Energy Level" "Barking Level"
[17] "Mental Stimulation Needs"
```

```
names(breed_ranks)
```

```
[1] "Breed" "2013 Rank" "2014 Rank" "2015 Rank" "2016 Rank" "2017 Rank"
[7] "2018 Rank" "2019 Rank" "2020 Rank" "links" "Image"
```

Všimněme si, že oba datasety obsahují proměnnou `Breed`, tedy plemeno psa. To bude naším klíčem, tedy proměnou, pomocí které spojíme oba dataframy dohromady. Důvodem, proč nejsou oba datasety spojené už od začátku, je že ne všechna plemena obsažená v `breed_traits` se umístila v ročním hodnocení, a chybí tedy v `breed_ranks`. Při spojování dat je tedy na nás, jak se touto komplikací vypořádáme.

První možností je vzít dataframe `breed_traits` a přilepit k němu `breed_ranks`, pomocí funkce `left_join()`. Výsledkem bude dataframe, který obsahuje všechny informace z `breed_traits` a pokud se některé plemeno neumístilo v žebříčku z `breed_ranks`, bude mít v proměnných hodnocení chybějící hodnotu:

```
left_join(breed_traits, breed_ranks, by = "Breed")
```

```
# A tibble: 10 x 27
  Breed Affec~1 Good ~2 Good ~3 Shedd~4 Coat ~5 Drool~6 Coat ~7 Coat ~8 Openn~9
  <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> <chr>   <chr>   <dbl>
1 Retr~     5     5     5     4     2     2 Double Short     5
2 Fren~     5     5     4     3     1     3 Smooth Short     5
3 Germ~     5     5     3     4     2     2 Double Medium  3
4 Retr~     5     5     5     4     2     2 Double Medium  5
5 Bull~     4     3     3     3     3     3 Smooth Short     4
6 Pood~     5     5     3     1     4     1 Curly Long     5
7 Beag~     3     5     5     3     2     1 Smooth Short     3
8 Rott~     5     3     3     3     1     3 Smooth Short     3
9 Poin~     5     5     4     3     2     2 Smooth Short     4
10 Dach~     5     3     4     2     2     2 Smooth Short     4
# ... with 17 more variables: `Playfulness Level` <dbl>,
# `Watchdog/Protective Nature` <dbl>, `Adaptability Level` <dbl>,
# `Trainability Level` <dbl>, `Energy Level` <dbl>, `Barking Level` <dbl>,
# `Mental Stimulation Needs` <dbl>, `2013 Rank` <dbl>, `2014 Rank` <dbl>,
# `2015 Rank` <dbl>, `2016 Rank` <dbl>, `2017 Rank` <dbl>, `2018 Rank` <dbl>,
# `2019 Rank` <dbl>, `2020 Rank` <dbl>, links <chr>, Image <chr>, and
# abbreviated variable names 1: `Affectionate With Family`, ...
```

Všimněme si, že například němečtí ovčáci nebyli hodnoceni a u proměnných 2013 Rank až 2019 Rank tedy mají chybějící hodnotu. Naopak pro buldoky jsou k dispozici všechna data.

Alternativou k `left_join()` je funkce `right_join()`. Ta provede velmi podobnou operaci, jako jsme viděli výše, výchozím dataframem zde ale bude `breed_ranks`.

```
right_join(breed_traits, breed_ranks, by = "Breed")
```

```
# A tibble: 10 x 27
  Breed Affec~1 Good ~2 Good ~3 Shedd~4 Coat ~5 Drool~6 Coat ~7 Coat ~8 Openn~9
  <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> <chr>   <chr>   <dbl>
1 Bull~     4     3     3     3     3     3 Smooth Short     4
```

```

2 Pood~      5      5      3      1      4      1 Curly  Long      5
3 Beag~      3      5      5      3      2      1 Smooth Short     3
4 Rott~      5      3      3      3      1      3 Smooth Short     3
5 Dach~      5      3      4      2      2      2 Smooth Short     4
6 Boxe~      4      5      3      2      2      3 Smooth Short     4
7 Pome~      5      3      3      2      3      1 Double Long      3
8 Hava~      5      5      5      2      3      1 Double Long      5
9 Brit~      3      4      4      3      3      1 Double Short     3
10 Pugs      5      5      4      4      2      1 Smooth Short     5
# ... with 17 more variables: `Playfulness Level` <dbl>,
#   `Watchdog/Protective Nature` <dbl>, `Adaptability Level` <dbl>,
#   `Trainability Level` <dbl>, `Energy Level` <dbl>, `Barking Level` <dbl>,
#   `Mental Stimulation Needs` <dbl>, `2013 Rank` <dbl>, `2014 Rank` <dbl>,
#   `2015 Rank` <dbl>, `2016 Rank` <dbl>, `2017 Rank` <dbl>, `2018 Rank` <dbl>,
#   `2019 Rank` <dbl>, `2020 Rank` <dbl>, links <chr>, Image <chr>, and
#   abbreviated variable names 1: `Affectionate With Family`, ...

```

V tomto případě již plemena jako zlatý retrívr nebo německý ovčák ve výsledném dataframu nenajdeme vůbec, protože nejsou obsažena v dataframu `breed_ranks`.

 Jak napravo, tak nalevo

Výsledek funkce `left_join(breed_traits, breed_ranks)` je ekvivalentní funkci `right_join(breed_ranks, breed_traits)`.

Pro zachování pouze plemen, která jsou obsažena v obou dataframech, lze aplikovat funkci `inner_join()`. Výsledný dataframe bude mít mnohem méně řádků, než ty předchozí, pouze 49, protože většina plemen není v dataframu `breed_ranks`:

```
inner_join(breed_traits, breed_ranks, by = "Breed")
```

Poslední verzí je permissivní `outer_join()`, která spojí oba dataframy a zachová přitom všechny řádky:

```
full_join(breed_traits, breed_ranks, by = "Breed")
```

13.2 Kterou spojovací funkci použít?

Každá z výše zmíněných funkcí se hodí pro jinou situaci. Která je ta pravá? Pokud je hlavním cílem naší práce analýza charakteristik jednotlivých plemen, bude pro nás nejužitečnější `left_join(breed_traits, breed_ranks)`. Na druhou stranu, pokud by pro naši analýzu bylo

stěžejní roční hodnocení, uplatili bychom spíše `right_join(breed_traits, breed_ranks)`, protože plemena, který nebyla hodnocena, pro nás nejsou zajímavá. Pro analýzu vztahů mezi charakteristikami a hodnocením pro nás budou užitečná pouze plemena, pro která máme k dispozici všechny informace, a ty bychom získali pomocí `inner_join(breed_traits, breed_ranks)`. Nakonec, pokud by naším cílem bylo jen datasety spojit, aniž bychom přišli o jakýkoliv data, například pro jejich uskladnění, využili bychom `full_join(breed_traits, breed_ranks)`.

Část III

Manipulace s proměnnými

14 Transformace proměnných

Transformace proměnných patří mezi nejběžnější operace při práci s daty. Od standardizace proměnných, přes jejich čištění, až po vytváření proměnných nových, obsah následující kapitoly budou denní chléb každého analytika.

Hlavním tahounem je zde funkce `mutate()` z balíčku `dplyr`. Přestože transformace proměnných lze provádět i bez ní, má tato funkce několik předností.

14.1 Jednoduché transformace

Funkce `mutate()` přijímá jako svůj první argument dataframe, dalšími argumenty jsou poté jednotlivé transformace:

```
countries %>%
  mutate(gdp_milliards = gdp / 1000,
         poverty_risk = poverty_risk * 100) %>%
  select(country, gdp, gdp_milliards, poverty_risk)
```

```
# A tibble: 38 x 4
  country      gdp gdp_milliards poverty_risk
  <chr>      <dbl>      <dbl>      <dbl>
1 Belgium  450506.      451.       20.3
2 Bulgaria  55182.       55.2       38.9
3 Czechia  207772.      208.       12.2
4 Denmark  298276.      298.       17.2
5 Germany  3386000      3386       19
6 Estonia   25657.       25.7       23.4
7 Ireland  324038.      324.       22.7
8 Greece   184714.      185.       34.8
9 Spain    1208248      1208.      26.6
10 France  2353090      2353.      17.1
# ... with 28 more rows
```


Zde je vidět nejen aplikace funkce `mutate()`, ale i její hlavní primární výhoda. Protože jejím výsledkem je dataframe s provedenými transformacemi, je možné na ní navázat dalšími funkcemi, jako je `select()` nebo `filter()`. Novým proměnným také můžeme jednoduše přiřadit jméno (viz Kapitola 5). Pokud uložíme výsledek transformace pod novým jménem, bude vytvořena nová proměnná (v našem případě `gdp_milliards`). Pokud použijeme jméno již existující proměnné, bude přepsána hodnotami (`poverty_risk`).

Jednou z analýz, která se v datasetu nabízí, je srovnání ekonomické produktivity zemí, a jednou z nejpobulárnějších metrik ekonomické produktivity je HDP. Čtenáři se znalostmi ekonomie ale již jistě tuší problém. HDP se silně odvíjí od počtu obyvatel a není tedy úplně smysluplné porovnávat obří Německo s maličkým Českem. Pro serióznější analýzu by proto bylo lepší využít standardizovanější míru, jakou je například HDP na hlavu. Tuto proměnnou náš dataset neobsahuje, nemusíme ale smutnit. Máme k dispozici jak HDP, tak počet obyvatel a od kýženého výsledku nás dělí jedna matematická operace.

Pro rychlé srovnání zemí by také bylo vhodné převést data do standardizovaných jednotek. Takovou jednotkou jsou mimo jiné *z skóry*, získatelné odečtením průměru proměnné od každé naměřené hodnoty a vydělením rozdílu směrodatnou odchylkou, tedy $z_i = \frac{x_i - \bar{x}}{sd(x)}$. Protože se jedná o populární formu standardizace, R pro ni nabízí funkci `scale()`:

```
countries %>%
  mutate(gdp_pc = gdp / population,
         gdp_pc_scaled = scale(gdp_pc)) %>%
  select(country, gdp_pc, gdp_pc_scaled)
```

```
# A tibble: 38 x 3
  country   gdp_pc gdp_pc_scaled[,1]
  <chr>     <dbl>         <dbl>
1 Belgium  0.0395         0.357
2 Bulgaria 0.00783       -1.06
3 Czechia  0.0196        -0.534
4 Denmark  0.0516         0.897
5 Germany  0.0409         0.419
6 Estonia  0.0194        -0.540
7 Ireland  0.0671         1.59
8 Greece   0.0172        -0.640
9 Spain    0.0259        -0.252
10 France  0.0352         0.162
# ... with 28 more rows
```

Z transformované proměnné `gdp_pc_scaled` je vidět, že z skór České republiky je -0.53, naše HDP na hlavu se tedy nachází zhruba půl směrodatné odchylky pod průměrem. Naopak Irsko se těší HDP na hlavu o 1.5 směrodatné odchylky vyšší, než je průměr všech zemí v datasetu.

14.2 Transformace po skupinách

Ve výše zmíněných příkladech byly transformace aplikovány na vybrané proměnné jako celek. Co když ale není naším cílem transformovat všechny hodnoty stejným způsobem?

Pro detailnější analýzu ekonomické produktivity zemí může být zajímavé zohlednit jejich politickou historii. Jak si například Česká republika vede ve srovnání s ostatními *postsovětskými* zeměmi? Pro zodpovězení této otázky je nutné aplikovat funkci `scale()` na každou skupinu proměnné `postsoviet` zvlášť. Naštěstí pro nás, tato operace nemůže být jednodušší, a to díky funkci `group_by()`, se kterou jsme se již setkali při řezání dataframů (Sekce 11.3):

```
countries %>%
  group_by(postsoviet) %>%
  mutate(gdp_pc = gdp / population,
         gdp_pc_scaled = scale(gdp_pc)) %>%
  ungroup() %>%
  select(country, postsoviet, gdp_pc, gdp_pc_scaled)
```

```
# A tibble: 38 x 4
  country postsoviet  gdp_pc gdp_pc_scaled[,1]
  <chr>    <chr>         <dbl> <dbl>
1 Belgium no           0.0395 -0.245
2 Bulgaria yes          0.00783 -0.781
3 Czechia yes          0.0196  0.522
4 Denmark no           0.0516  0.329
5 Germany yes          0.0409  2.89
6 Estonia yes          0.0194  0.507
7 Ireland no           0.0671  1.07
8 Greece  no           0.0172 -1.31
9 Spain   no           0.0259 -0.894
10 France no           0.0352 -0.453
# ... with 28 more rows
```

Přestože tento dataframe na první pohled vypadá velmi podobně jako ten předchozí, hodnoty proměnné `gdp_pc_scaled` jsou odlišné. Česká republika má nyní hodnotu 0.52. Nachází se tedy zhruba půl směrodatné odchylky nad průměrem ostatních *postsovětských* zemí. Naopak z skóru Irska se snížil na 1.1, protože ve srovnání s ostatními *západními* zeměmi je jeho HDP na hlavu pouze jednu směrodatnou odchylku nad průměrem.

Tohoto srovnání jsme dosáhli právě tím, že jsme před aplikací funkce `mutate()` rozdělili dataframe pomocí `group_by()` a všechny následující operace tedy budou prováděny pro západní a postsovětské funkce zvlášť.

! Po použití vypněte

Jakmile jednou aplikujete funkci `group_by()`, bude aktivní ve všech následujících krocích. To může vést ke zmatkům, zpravidla proto, že na ni zapomenete a aplikujete funkce na každou skupinu zvlášť, aniž byste si to uvědomovali. Proto pokaždé, když skončíte s transformací dat nezapomeňte seskupování ukončit pomocí `ungroup()`.

14.3 Řádkové operace

Přesuňme se teď od ekonomické produktivity k palčivějším tématům. Jedním z ekonomicko-sociálních problémů, se kterými se musí každá země vypořádat, jsou obyvatelé ohrožení chudobou (proměnná `poverty_risk`) a obyvatelé v materiální deprivaci (`material_dep`). Naneštěstí pro nás nemáme k dispozici podíl obyvatel ohrožených alespoň jedním z těchto rizik, můžeme ale získat alespoň konzervativní odhad. Maximální možný podíl lidí ohrožených chudobou *nebo* v materiální deprivaci je možné získat jednoduše součtem obou hodnot pro každou zemi.

Tento krapet koprbatý problém nám poslouží pro demonstraci řádkových (*rowwise*) transformací. R ve svém výchozím nastavení aplikuje funkce po sloupcích (*columnwise*). To s sebou přináší poněkud zákeřnou komplikaci při snaze sečíst dvě hodnoty na stejném řádku dataframu. Pokud chceme aplikovat funkci po řádcích, nikoliv po sloupcích, je nutné využít funkce `rowwise()`. Ta funguje velmi obdobně jako `group_by()`, a to včetně jejího “vypnutí” pomocí `ungroup()`:

```
countries %>%  
  rowwise() %>%  
  mutate(poverty_or_dep = sum(poverty_risk, material_dep, na.rm = TRUE)) %>%  
  ungroup() %>%  
  select(country, poverty_or_dep)
```

```
# A tibble: 38 x 2  
  country poverty_or_dep  
  <chr>      <dbl>  
1 Belgium      0.316  
2 Bulgaria      0.827  
3 Czechia       0.22  
4 Denmark       0.24  
5 Germany       0.281  
6 Estonia       0.35  
7 Ireland       0.375
```

```

8 Greece          0.708
9 Spain           0.394
10 France         0.282
# ... with 28 more rows

```

Pomocí funkce `rowwise()` jsme získali součet podílu lidí ohrožených chudobou a lidí v materiální deprivaci pro každou ze zemí. Jak je vidět, alespoň do jedné z těchto kategorií v České republice spadá maximální 22 % obyvatel.

14.4 Podmíněné transformace

Jednou z myšlenkových operací, ve které počítače vynikají, je rigidní *“pokud je splněna podmínka, udělej X”*. Pojdme toho využít.

V předchozí sekci jsme porovnávali země na základě standardizovaného HDP na hlavu. Co kdybychom tuto analýzu chtěli vzít o krok dále a vytvořit novou kategoriální proměnnou, jejíž hodnota bude *Above average* pro země s nadprůměrným HDP na hlavu, a *Below average* pro země podprůměrné.

K tomu nám dobře poslouží funkce `if_else()`, která má tři povinné argumenty. Tím prvním je podmínka, jejímž výsledkem musí být buď hodnota “pravda” (`TRUE`) nebo “nepravda” (`FALSE`). Druhým argumentem je operace, která bude provedena, pokud je zmíněná podmínka splněna, třetím argumentem poté nepřekvapivě operace provedené v případě nesplnění podmínky. Aplikace pro náš konkrétní případ by vypadala následovně:

```

countries %>%
  mutate(gdp_pc_scaled = scale(gdp / population),
         gdp_pc_cat = if_else(gdp_pc_scaled > 0,
                              true = "Above average",
                              false = "Below average")) %>%
  select(country, gdp_pc_scaled, gdp_pc_cat)

```

```

# A tibble: 38 x 3
  country gdp_pc_scaled[,1] gdp_pc_cat
  <chr>          <dbl> <chr>
1 Belgium      0.357 Above average
2 Bulgaria    -1.06  Below average
3 Czechia     -0.534 Below average
4 Denmark      0.897 Above average
5 Germany      0.419 Above average
6 Estonia     -0.540 Below average

```

```

7 Ireland          1.59 Above average
8 Greece          -0.640 Below average
9 Spain           -0.252 Below average
10 France          0.162 Above average
# ... with 28 more rows

```

Co kdybychom ale chtěli, aby výsledkem operace byly více než dvě hodnoty? Možná nám přijde, že klasifikovat země pouze jako nadprůměrné a podprůměrné je příliš redukcionistické (populární to výčitka mezi sociology). Země bychom místo toho raději rozdělili do čtyř kategorií:

- `below average` pro země se z skóre nižším než -1
- `slightly below average` pro země v intervalu -1 až 0
- `slightly above average` analogicky pro země mezi 0 a 1
- `above average` pro ty se z skórem vyšším, než 1.

Jednou z možností je využít řadu na sebe navazujících `if_else` funkcí. Tento postup by *technicky* fungoval, povede ale k mnoha slzám a frustracím přímo úměrným množství funkcí, které je třeba správně zřetězit. Elegantnějším řešením je využít funkci `case_when()`, která byla vytvořena právě pro tento případ:

```

countries %>%
  mutate(gdp_pc = scale(gdp / population),
         gdp_pc_cat = case_when(gdp_pc < -1 ~ "Below average",
                                gdp_pc <= 0 ~ "Slightly below average",
                                gdp_pc <= 1 ~ "Slightly above average",
                                gdp_pc > 1 ~ "Above average",
                                TRUE ~ "Unknown")) %>%
  select(country, gdp_pc, gdp_pc_cat)

```

```

# A tibble: 38 x 3
  country  gdp_pc[,1] gdp_pc_cat
  <chr>      <dbl> <chr>
1 Belgium    0.357 Slightly above average
2 Bulgaria  -1.06  Below average
3 Czechia   -0.534 Slightly below average
4 Denmark    0.897 Slightly above average
5 Germany    0.419 Slightly above average
6 Estonia   -0.540 Slightly below average
7 Ireland    1.59  Above average
8 Greece    -0.640 Slightly below average

```

```

9 Spain          -0.252 Slightly below average
10 France         0.162 Slightly above average
# ... with 28 more rows

```

Funkce `case_when()` má oproti dosavadním funkcím atypickou syntax. Každá z logických podmínek je kondezovaná do formule `podmínka ~ vysledek`. První řádek v této funkci, `gdp_pc < -1 ~ "Below average"`, tedy říká “pokud je hodnota proměnné `gdp_pc` menší než `-1`, vrať hodnotu *Below average*”. Pokud tato podmínka splněná není, funkce zkontroluje podmínku následující. Podmínky jsou ověřovány jedna po druhé, přičemž podmínky na vyšších místech jsou ověřeny dříve. Speciální podmínkou je `TRUE ~ vysledek`, která je je vždy splněna. To se hodí pokud jsou v datech přítomny hodnoty, které nespĺňují žádnou z předchozích podmínek. Kdy se může stát, že hodnota nespĺňuje žádnou z našich podmínek? Například, pokud se jedná o hodnotu chybějící:

```

countries %>%
  mutate(gdp_pc = scale(gdp / population),
         gdp_pc_cat = case_when(gdp_pc < -1 ~ "Below average",
                                gdp_pc <= 0 ~ "Slightly below average",
                                gdp_pc <= 1 ~ "Slightly above average",
                                gdp_pc > 1 ~ "Above average",
                                TRUE ~ "Unknown")) %>%
  select(country, gdp_pc, gdp_pc_cat) %>%
  filter(is.na(gdp_pc))

```

```

# A tibble: 4 x 3
  country          gdp_pc[,1] gdp_pc_cat
  <chr>            <dbl> <chr>
1 Liechtenstein    NA Unknown
2 Montenegro       NA Unknown
3 Turkey           NA Unknown
4 Bosnia and Herzegovina NA Unknown

```

15 Sumarizace proměnných

Sumarizace proměnných je prováděně velmi podobně jako jejich transformace, slouží k ní ale funkce `summarise()`. Pokud jste si již osvojili funkci `mutate()` z předchozí kapitoly, budete i zde jako doma.

15.1 Jednoduchá sumarizace

Jak již bylo zmíněno, základní aplikace `summarise()` je velmi podobná transformaci proměnných. Výpočet průměru a směrodatné odchylky průměrné naděje na dožití je tedy jednoduchý. V rámci `summarise()` je možné aplikovat nejen klasické funkce jako jsou `mean()` nebo `sd()`, ale i základní matematické operace. Stejně tak je možné i používat i funkce vnořené. Toho využijme pro výpočet průměrné absolutní odchylky (*mean absolute deviation*), alternativy ke směrodatné odchylce:

```
countries %>%
  summarise(mean = mean(life_exp, na.rm = TRUE),
            sd   = sd(life_exp, na.rm = TRUE),
            mae  = mean(abs(life_exp - mean(life_exp, na.rm = TRUE)), na.rm = TRUE))
```

```
# A tibble: 1 x 3
  mean   sd   mae
<dbl> <dbl> <dbl>
1  79.6  2.82  2.58
```

Průměrná naděje na dožití v našem datasetu je 79.6 let, se směrodatnou odchylkou 2.8 roku a průměrnou absolutní odchylkou 2.6 roku. Protože naděje na dožití některých zemí neznáme, je nutné využít `na.rm = TRUE` pro odstranění chybějících hodnot (viz Sekce 6.2).

15.2 Sumarizace po skupinách

Funkci `summarise()` lze jako mnoho již představených kombinovat s funkcí `group_by()` pro skupinovou analýzu. Pro získání průměru, směrodatné odchylky a průměrné absolutní odchylky naděje na dožití postsovětských a západních zemí:

```

countries %>%
  group_by(postsoviet) %>%
  summarise(mean = mean(life_exp, na.rm = TRUE),
            sd   = sd(life_exp, na.rm = TRUE),
            mae  = mean(abs(life_exp - mean(life_exp, na.rm = TRUE)), na.rm = TRUE))

# A tibble: 2 x 4
  postsoviet mean    sd  mae
  <chr>      <dbl> <dbl> <dbl>
1 no         81.4  1.76  1.06
2 yes        77.1  1.97  1.57

```

Postsovětské země mají v průměru nižší naději na dožití, než ty západní, jsou ale také mezi nimi větší rozdíly, což je možné vidět jak na základě směrodatné, tak absolutní odchylky.

16 Transformace a sumarizace více proměnných

V předchozích dvou kapitolách jsme si představili, jak transformovat a sumarizovat proměnné. Vždy jsme však pracovali maximálně s jednou nebo dvěma proměnnými najednou. V praxi ovšem nejsou neobvyklé situace, ve kterých je nutné aplikovat určitou funkci na desítky, ne-li stovky proměnných najednou. Naštěstí pro nás, Tidyverse pro tyto příležitosti nabízí funkci `across()`.

16.1 Transformace většího množství proměnných

Dataset `countries` obsahuje několik kategoriálních proměnných, mezi nimi `postsoviet`, `eu_member`, `maj_belief` a `di_cat`. Všechny tyto proměnné jsou typu *character*, pro analýzu by ovšem bylo lepší je převést na typ *factor* (pro typy objektů viz Kapitola 4).

Jednou možností je aplikovat funkci `as.factor()` na každou proměnnou zvlášť:

```
countries %>%
  mutate(postsoviet = as.factor(postsoviet),
         eu_member   = as.factor(eu_member),
         maj_belief  = as.factor(maj_belief),
         di_cat      = as.factor(di_cat)) %>%
  select(postsoviet, eu_member, maj_belief, di_cat) %>%
  head(5)
```

```
# A tibble: 5 x 4
  postsoviet eu_member maj_belief di_cat
  <fct>      <fct>      <fct>      <fct>
1 no        yes        catholic   Flawed democracy
2 yes       yes        orthodox   Flawed democracy
3 yes       yes        nonbelief   Flawed democracy
4 no        yes        protestantism Full democracy
5 yes       yes        catholic   Full democracy
```

Tímto kódem dosáhneme našeho cíle, nejedná se však o nejelegantnější řešení, jelikož opakovaně aplikujeme stejnou funkci na každou z proměnných zvlášť. To nejen náš kód prodlužuje, ale zároveň zvyšuje šanci, že na některém řádku uděláme chybu. Alternativou je funkce `across()`:

```
countries %>%
  mutate(across(.cols = c(postsoviet, eu_member, maj_belief, di_cat),
                .fns = as.factor)) %>%
  select(postsoviet, eu_member, maj_belief, di_cat) %>%
  head(5)
```

```
# A tibble: 5 x 4
  postsoviet eu_member maj_belief    di_cat
  <fct>      <fct>      <fct>      <fct>
1 no        yes        catholic    Flawed democracy
2 yes       yes        orthodox    Flawed democracy
3 yes       yes        nonbelief   Flawed democracy
4 no        yes        protestantism Full democracy
5 yes       yes        catholic    Full democracy
```

Funkce `across()` má dva nezbytné argumenty. Prvním z nich je `.col`, pomocí kterého specifikujeme proměnné, na které chceme naši funkci aplikovat. Argument `.fns` poté specifikuje funkci samotnou. Výsledek je stejný jako v předchozím případě, náš kód je ale kompaktnější.

Tímto ovšem výhody funkce `across()` nekončí. Proměnné je v ní možné specifikovat nejen tím, že je vyjmenuje jednu po druhé, ale i pomocí pomocných funkcí, se kterými jsme se již setkali v kapitole věnované výběrům sloupců (Sekce 10.2).

Pokud bychom například chtěli zaokrouhlit všechny numerické proměnné v datasetu na dvě desetinná místa, není nutné jejich názvy vypisovat ručně. Stačí využít kombinace funkcí `where()` a `is.numeric()`:

```
countries %>%
  mutate(across(.cols = where(is.numeric),
                .fns = round, 2)) %>%
  select(where(is.numeric)) %>%
  head(5)
```

```
# A tibble: 5 x 9
  gdp population area life_exp uni_prc poverty_risk mater~1 hdi dem_i~2
  <dbl>      <dbl> <dbl>   <dbl>  <dbl>      <dbl>  <dbl> <dbl>  <dbl>
```

```

1 450506. 11398589 30528 81.2 0.36 0.2 0.11 0.92 7.78
2 55182. 7050034 110879 74.8 0.25 0.39 0.44 0.81 7.03
3 207772. 10610055 78867 79.2 0.22 0.12 0.1 0.89 7.69
4 298276. 5781190 43094 81.2 0.33 0.17 0.07 0.93 9.22
5 3386000 82792351 357022 81 0.25 0.19 0.09 0.94 8.68
# ... with abbreviated variable names 1: material_dep, 2: dem_index

```

V rámci `across()` je také možné specifikovat argumenty pro aplikovanou funkci. Výše jsem určili, že numerické proměnné mají být zaokrouhlené na dvě desetinná místa pomocí `.fns = round, 2`, kde 2 je argument funkce `round()`. Alternativně bychom mohli využít takzvanou tilda notaci (*tilda notation*):

```

countries %>%
  mutate(across(.cols = where(is.numeric),
                .fns = ~round(., 2))) %>%
  select(where(is.numeric)) %>%
  head(5)

```

```

# A tibble: 5 x 9
   gdp population area life_exp uni_prc poverty_risk mater~1 hdi dem_i~2
   <dbl>      <dbl> <dbl>   <dbl> <dbl>      <dbl> <dbl> <dbl> <dbl>
1 450506. 11398589 30528 81.2 0.36 0.2 0.11 0.92 7.78
2 55182. 7050034 110879 74.8 0.25 0.39 0.44 0.81 7.03
3 207772. 10610055 78867 79.2 0.22 0.12 0.1 0.89 7.69
4 298276. 5781190 43094 81.2 0.33 0.17 0.07 0.93 9.22
5 3386000 82792351 357022 81 0.25 0.19 0.09 0.94 8.68
# ... with abbreviated variable names 1: material_dep, 2: dem_index

```

Na rozdíl od předchozího příkladu, funkci `round()` zde předchází tilda (`~`) a prvním argumentem je `..`. Tato tečka (`.`) slouží jako *placeholder* pro proměnné dosazované do funkce `round()`. Jinak řečeno, funkce `across()` postupně dosadí každou proměnnou specifikovanou pomocí argumentu `.cols` na místo placeholderu `..`. Tilda notace je o něco komplexnější, než předchozí způsob, je ale o mnoho flexibilnější, protože nám umožňuje kontrolovat, do kterého argumentu budou námi proměnné dosazeny.

16.2 Sumarizace většího množství proměnných

Funkci `across()` je možné aplikovat v rámci `summarise()` identicky jako v případě `mutate()`. Toho využijeme primárně pro výpočet deskriptivních statistik. Stejně jako v předchozích kapitál, i zde můžeme funkce navazovat na sebe:

```
countries %>%
  summarise(across(.cols = where(is.numeric),
                    .fns = mean, na.rm = TRUE)) %>%
  mutate(across(.cols = everything(),
                .fns = round, 2))
```

```
# A tibble: 1 x 9
```

```
  gdp population   area life_exp uni_prc poverty_risk mater~1   hdi dem_i~2
  <dbl>   <dbl> <dbl>   <dbl> <dbl>   <dbl>   <dbl> <dbl> <dbl>
1 484601. 16754743 156019.   79.6   0.29     0.24    0.18 0.87   7.64
# ... with abbreviated variable names 1: material_dep, 2: dem_index
```

Všimněme si, že při výpočtu průměru numerických proměnných bylo nutné odstranit chybějící proměnné pomocí `na.rm = TRUE` (s tímto argumentem jsme se již setkali, viz Sekce 6.2). Všechny získané průměry jsme poté zaokrouhlili pomocí `mutate()`.

Výsledkem jsou data v širokém formátu (Kapitola 12). Pro čitelnost bude lepší je převést do formátu dlouhého:

```
countries %>%
  summarise(across(.cols = where(is.numeric),
                    .fns = mean, na.rm = TRUE)) %>%
  mutate(across(.cols = everything(),
                .fns = round, 2)) %>%
  pivot_longer(cols = everything(),
               names_to = "variable",
               values_to = "mean")
```

```
# A tibble: 9 x 2
```

```
  variable      mean
  <chr>         <dbl>
1 gdp          484601.
2 population   16754743
3 area         156019.
4 life_exp     79.6
5 uni_prc      0.29
6 poverty_risk 0.24
7 material_dep 0.18
8 hdi          0.87
9 dem_index    7.64
```

16.3 Analýza po skupinách

Stejně v předchozích kapitolách, i zde můžeme aplikovat funkci `group_by()` pro skupinovou sumarizaci (a transformaci) dat:

```
countries %>%
  group_by(eu_member) %>%
  summarise(across(.cols = where(is.numeric),
                    .fns = mean, na.rm = TRUE)) %>%
  mutate(across(.cols = -eu_member,
                .fns = round, 2))

# A tibble: 2 x 10
  eu_member    gdp popul~1  area life_~2 uni_prc pover~3 mater~4  hdi dem_i~5
  <chr>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 no        152949.  1.19e7 1.45e5  78.6  0.27  0.31  0.26  0.84  6.87
2 yes       567514.  1.83e7 1.60e5  79.9  0.3  0.23  0.17  0.88  7.89
# ... with abbreviated variable names 1: population, 2: life_exp,
# 3: poverty_risk, 4: material_dep, 5: dem_index
```

Výsledkem je dataframe, který sumarizuje numerické proměnné pro západní a postsovětské země zvlášť. Všimněme si, že při zaokrouhlování je nutné funkci `round()` aplikovat na všechny proměnné s výjimkou proměnné `eu_member`.

Stejně jako v předchozí sekci, i zde je pro čitelnost lepší převést data do dlouhého formátu. Výsledkem bude dataset vhodný pro vizualizaci nebo statistické modelování:

```
countries %>%
  group_by(eu_member) %>%
  summarise(across(.cols = where(is.numeric),
                    .fns = mean, na.rm = TRUE)) %>%
  mutate(across(.cols = -eu_member,
                .fns = round, 2)) %>%
  pivot_longer(cols = -eu_member,
               names_to = "variable",
               values_to = "mean")

# A tibble: 18 x 3
  eu_member variable          mean
  <chr>      <chr>          <dbl>
1 no        gdp            152949.
```

2 no	population	11949585.
3 no	area	144874.
4 no	life_exp	78.6
5 no	uni_prc	0.27
6 no	poverty_risk	0.31
7 no	material_dep	0.26
8 no	hdi	0.84
9 no	dem_index	6.87
10 yes	gdp	567514.
11 yes	population	18299258.
12 yes	area	159999.
13 yes	life_exp	79.9
14 yes	uni_prc	0.3
15 yes	poverty_risk	0.23
16 yes	material_dep	0.17
17 yes	hdi	0.88
18 yes	dem_index	7.89

Na rozdíl od počítače, lidským čtenářům tento formát zpravdila nepřijde příliš přirozený. Ideálně proto data převedeme zpět do širšího formátu, abychom mohli jednoduše porovnat, která skupina zemí si vede lépe:

```
countries %>%
  group_by(eu_member) %>%
  summarise(across(.cols = where(is.numeric),
                    .fns = mean, na.rm = TRUE)) %>%
  mutate(across(.cols = -eu_member,
                 .fns = round, 2)) %>%
  pivot_longer(cols = -eu_member,
               names_to = "variable",
               values_to = "mean") %>%
  pivot_wider(names_from = eu_member,
              values_from = mean) %>%
  mutate(difference = no - yes)
```

```
# A tibble: 9 x 4
  variable      no      yes difference
  <chr>        <dbl>   <dbl>     <dbl>
1 gdp          152949.  567514.  -414565.
2 population  11949585. 18299258. -6349673.
3 area        144874.  159999.  -15124.
4 life_exp     78.6    79.9     -1.31
```

5	uni_prc	0.27	0.3	-0.0300
6	poverty_risk	0.31	0.23	0.08
7	material_dep	0.26	0.17	0.09
8	hdi	0.84	0.88	-0.0400
9	dem_index	6.87	7.89	-1.02

16.4 Sumarizace více proměnných bez použití across()

Přestože je kombinace funkcí `summarise()` a `across()` velmi užitečná, výsledný dataset není vždy ve formátu, se kterým je jednoduché dále pracovat, zvláště pokud aplikujeme více než jednu funkci najednou. Existuje ovšem trik, využívající převodu mezi širokým a dlouhým formátem, kterým si můžeme práci ulehčit.

Naším cílem může být spočítat průměr, směrodatnou odchylku, maximum a minimum všech numerických proměnných. Jednou variantou je aplikovat funkci `across()` a specifikovat více funkcí pomocí `lst()`. Tato funkce umožňuje aplikovat několik funkcí v rámci jednoho `across()` Výsledek ovšem není příliš vzhledný:

```
countries %>%
  summarise(across(.cols = where(is.numeric),
                  .fns = lst(mean, sd, min, max), na.rm = TRUE))

# A tibble: 1 x 36
  gdp_m~1 gdp_sd gdp_min gdp_max popul~2 popul~3 popul~4 popul~5 area_~6 area_sd
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 484601. 7.94e5 10735. 3386000 1.68e7 2.41e7 38114 8.28e7 156019. 189008.
# ... with 26 more variables: area_min <dbl>, area_max <dbl>,
#   life_exp_mean <dbl>, life_exp_sd <dbl>, life_exp_min <dbl>,
#   life_exp_max <dbl>, uni_prc_mean <dbl>, uni_prc_sd <dbl>,
#   uni_prc_min <dbl>, uni_prc_max <dbl>, poverty_risk_mean <dbl>,
#   poverty_risk_sd <dbl>, poverty_risk_min <dbl>, poverty_risk_max <dbl>,
#   material_dep_mean <dbl>, material_dep_sd <dbl>, material_dep_min <dbl>,
#   material_dep_max <dbl>, hdi_mean <dbl>, hdi_sd <dbl>, hdi_min <dbl>, ...
```

Výsledek není nepoužitelný, abychom se ovšem dostali k čitelné tabulce, museli bychom několikrát využít převodu mezi širokým a dlouhým formátem.

Alternativním způsobem je vybrat proměnné, se kterými chceme pracovat, převést data do dlouhého formátu a poté již aplikovat klasickou funkci `summarise()`. Nakonec jen výsledky zaokrouhlíme:

```

countries %>%
  select(where(is.numeric)) %>%
  pivot_longer(everything()) %>%
  group_by(name) %>%
  summarise(mean = mean(value, na.rm = TRUE),
            sd   = sd(value, na.rm = TRUE),
            min  = min(value, na.rm = TRUE),
            max  = max(value, na.rm = TRUE)) %>%
  mutate(across(.cols = where(is.numeric),
                .fns = round, 2))

```

```
# A tibble: 9 x 5
```

	name	mean	sd	min	max
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	area	156019.	189008.	160	783562
2	dem_index	7.64	1.3	4.37	9.87
3	gdp	484601.	793693.	10735.	3386000
4	hdi	0.87	0.05	0.76	0.95
5	life_exp	79.6	2.82	74.8	83.3
6	material_dep	0.18	0.13	0.04	0.48
7	population	16754743	24110721.	38114	82792351
8	poverty_risk	0.24	0.08	0.12	0.42
9	uni_prc	0.29	0.08	0.16	0.41

Hlavní výhodou této metody je, kromě podle našeho názoru větší přehlednosti, že umožňuje specifikovat argumenty pro každou statistickou funkci zvlášť.

17 Práce s faktory

S faktory jsme se již setkali na začátku naší cesty (Sekce 4.2) a přišel čas navštívit znovu. Faktory představují hlavní typ objektů pro práci s kategorickými proměnnými a jsou široce využívané od vizualizaci dat po statistické modelování. Vyplatí se proto na ně podívat blíže. K jejich manipulaci nám poslouží balíček `forcats`, který je součástí Tidyverse.

17.1 Vytváření faktorů

Náš dataframe `countries` obsahuje řadu kategorických proměnných, mezi nimi také `maj_belief`, tedy převažující náboženská skupina v dané zemi. Tato proměnná je uložena jako objekt typu *character*:

```
class(countries$maj_belief)
```

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

Pro převedení této proměnná stačí pouze využít funkce `as.factor()`. Výsledný faktor bude obsahovat úrovně (*levels*) odpovídající pozorovaným hodnotám původní proměnné. Vzpomeňte si, že faktory nemohou nabývat jiných hodnot, než těch specifikovaných při jejich vzniku:

```
countries$maj_belief <- as.factor(countries$maj_belief)
```

```
class(countries$maj_belief)
```

```
[1] "factor"
```

```
levels(countries$maj_belief)
```

```
[1] "catholic"      "islam"          "nonbelief"      "orthodox"  
[5] "protestantism"
```

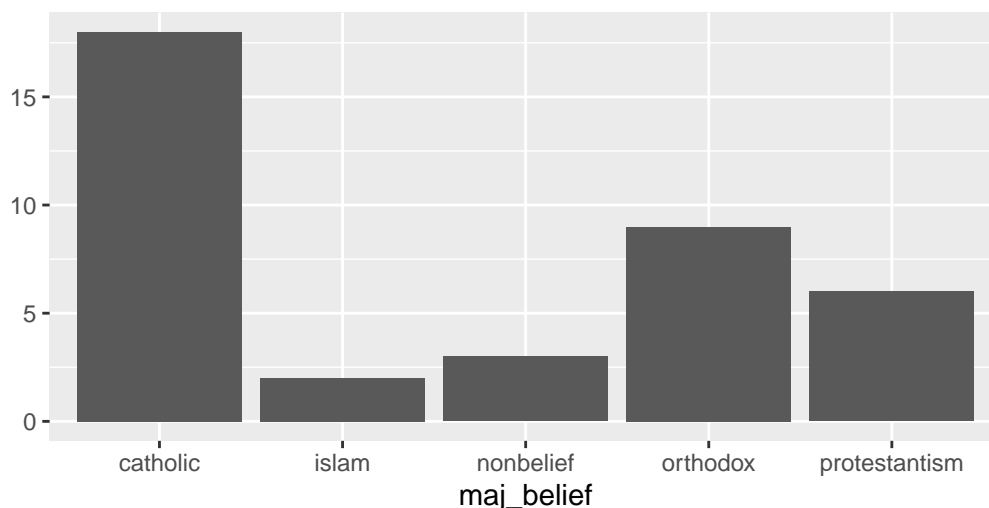
Všimněme si, že pořadí jednotlivých úrovní odpovídá abecednímu pořadí. Kromě funkce `as.factor()` je možné pro vytvoření faktoru využít také Tidyverse funkci `as_factor()`. Primárním rozdílem mezi nimi je, že funkce `as.factors()` uspořádá úrovně v abecedním pořadí, zatímco funkce `as_factor()` v pořadí, v jakém se jednotlivé kategorie objeví v datech. Primární výhodou druhé z funkcí je, že dojde ke stejnému výsledku bez ohledu na jazyk operačního systému.

17.2 Pořadí úrovní

Jednou ze situací, ve kterých je nutné převést kategorické proměnné na faktory, je vizualizace dat. Pokud by nás zajímalo náboženské složení zemí v našem datasetu, můžeme data vizualizovat pomocí funkce `qplot()` (o které se dozvíme více v příštích kapitolách):

```
qplot(x = maj_belief, data = countries)
```

Warning: `qplot()` was deprecated in ggplot2 3.4.0.



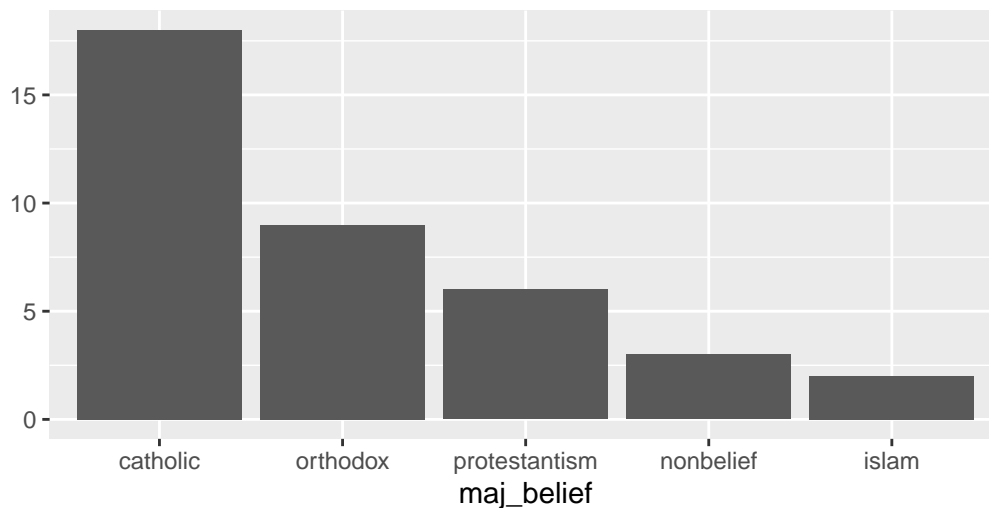
Výsledný graf je funkční, nicméně nepříliš estetický. Kategorie v grafu jsou seřazeny podle pořadí úrovní našeho faktoru, vhodnější by ale bylo, aby byly seřazeny sestupně podle frekvence výskytu.

⚠ Varování

Funkce pro vizualizaci dat převádí kategorické proměnné na faktory automaticky.

Pokud chceme změnit pořadí kategorií v grafu, nestačí pouze seřadit řádky datasetu, je třeba změnit pořadí úrovní faktoru. Způsobů, jak řadit úrovně je více. Tím základním je specifikovat pořadí úrovní explicitně, pomocí funkce `fct_relevel()`. Ta přijímá jako první argument faktor, který chceme transformovat a dále jednotlivé úrovně v pořadí, v jakém je chceme uložit.

```
countries$maj_belief <- fct_relevel(countries$maj_belief,  
                                  "catholic",  
                                  "orthodox",  
                                  "protestantism",  
                                  "nonbelief",  
                                  "islam")  
  
qplot(x = maj_belief, data = countries)
```



A je to! Sloupce jsou seřazeny. Ovšem manuálně vypisovat všechny kategorie je zdlouhavá záležitost. Lepší variantou je nechat R, aby pořadí úrovní určilo za nás. K tomu nejdříve budeme muset spočítat frekvenci výskytu jednotlivých kategorií. K tomu je možné využít již nám dobře známou kombinaci funkcí `group_by()` a `summarise()`. Uvnitř `summarise()` použijeme funkci `n()`, která vrátí počet pozorování v rámci dané skupiny.

```
countries %>%  
  group_by(maj_belief) %>%  
  summarise(n = n())
```

```
# A tibble: 5 x 2
```

```

maj_belief      n
<fct>          <int>
1 catholic      18
2 orthodox       9
3 protestantism  6
4 nonbelief      3
5 islam          2

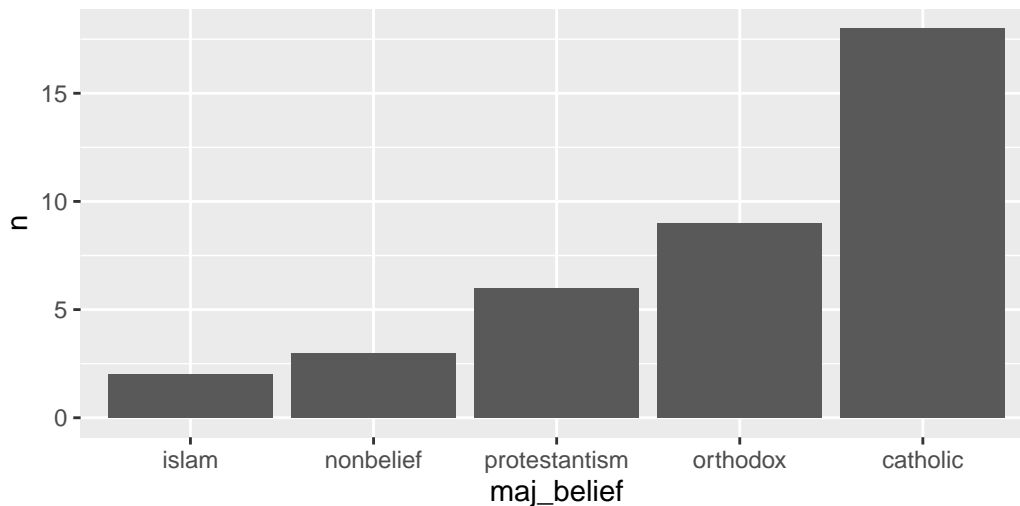
```

Dalším krokem je zrušit seskupení a aplikovat funkci `fct_reorder()`, kde prvním argumentem je faktor a druhým proměnná, podle které úrovně faktoru seřadíme. Nakonec už jen zbývá aplikovat `qplot()`, tentokrát včetně argumentu `y`:

```

countries %>%
  group_by(maj_belief) %>%
  summarise(n = n()) %>%
  ungroup() %>%
  mutate(maj_belief = fct_reorder(maj_belief, n)) %>%
  qplot(x = maj_belief, y = n, data = ., geom = "col")

```



Tento kód si zaslouží několik vysvětlivek. Zaprvé, protože se jedná o agregovaná data, je nutné specifikovat proměnnou pro osu Y(argument `y`). Dále je nutné funkci `qplot()` říct, že výsledkem má být sloupcový graf (`geom = col`). Nakonec je nutné specifikovat, že náš dataframe má být dosazen do argumentu `data` a to pomocí *placeholderu* `..`. Vzpomeňme si, že s *placeholdery* jsme si již setkali, když jsme transformovali větší množství proměnných pomocí funkce `across()` (viz. Sekce 16.1).

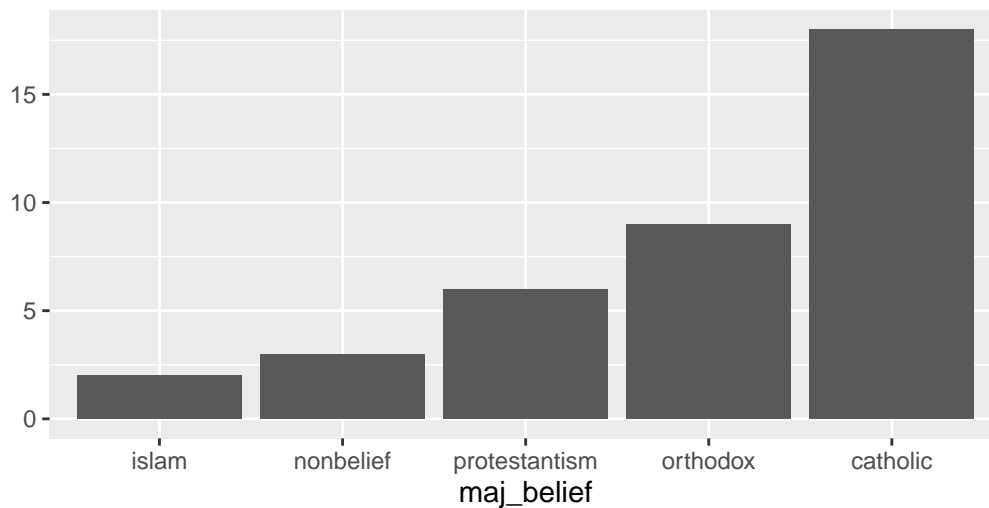
💡 count()

Pokud vám série funkci `group_by()` `%>% summarise(n = n())` `%>% ungroup()` přijde zbytečně zdlouhavá, nejste sami. Autoři Tidyverse mají stejný názor a připravili pro nás proto funkci `count()`. Ta je v podstatě je zkratkou pro výše zmíněnou sérii funkci. Následující skript tedy bude mít stejný výsledek, jako ten výše:

```
countries %>%  
  count(maj_belief) %>%  
  mutate(maj_belief = fct_reorder(maj_belief, n)) %>%  
  qplot(x = maj_belief, y = n, data = ., geom = "col")
```

V našem posledním grafu jsou kategorie seřazené vzestupně. V našem původním datasetu jsou ovšem pořadí sestupně. Pokud bychom chtěli pořadí kategorií obrátit, můžeme k tomu využít funkci `fct_rev()`

```
countries$maj_belief <- fct_rev(countries$maj_belief)  
qplot(x = maj_belief, data = countries)
```



17.3 Transformace úrovní

Kromě řazení úrovní faktoru je také často budeme chtít transformovat. Nejčastěji tak, že budeme chtít změnit název úrovně. Jednou z úrovní naší proměnné `maj_belief` je `protestantism`, jejíž název morfologicky neodpovídá ostatním. Rádi bychom proto

změnili název kategorie z `protestantism` na `protestant`. K tomu nám poslouží funkce `fct_recode()`:

```
countries$maj_belief <- fct_recode(countries$maj_belief,
                                  "protestant" = "protestantism")

levels(countries$maj_belief)
```

```
[1] "islam"      "nonbelief"  "protestant" "orthodox"   "catholic"
```

Jak je vidět, aplikace této funkce je snadná, stačí specifikovat faktor, a v následujících argumentech poté názvy úrovní ve formátu *nový název = starý název*. Změnit je možné i více názvu najednou.

Kromě ručního přepisování názvů je možné měnit úrovně také programátorsky, pomocí funkce `fct_relabel()`. Pro hezčí vzhled našich grafů bychom například chtěli, aby název každé úrovně začínal velkým písmenem. Převedení prvních písmen na kapitálky je možné dosáhnout pomocí funkce `str_to_title()`. Nelze ji ale aplikovat přímo, jelikož tím bychom změnili formát proměnné z *factor* na *character*. Místo toho ji použijeme v kombinaci s `fct_relabel()`. Prvním argumentem je faktor samotný, druhým poté funkce, kterou chceme na názvy úrovní aplikovat:

```
countries$maj_belief <- fct_relabel(countries$maj_belief, str_to_title)

levels(countries$maj_belief)
```

```
[1] "Islam"      "Nonbelief"  "Protestant" "Orthodox"   "Catholic"
```

Poslední transformací, kterou si zde ukážeme, je kolapsování kategorií. I to lze provádět jak ručně, tak programátorsky. První variantou je `fct_collapse()`, pomocí které můžeme například sloučit existující kategorie `Protestant`, `Catholic` a `Orthodox` do nové kategorie `Christian`:

```
countries$maj_belief_collapsed <- fct_collapse(countries$maj_belief,
                                               Christianity = c("Protestant",
                                                                "Catholic",
                                                                "Orthodox"))

levels(countries$maj_belief_collapsed)
```

```
[1] "Islam"          "Nonbelief"      "Christianity"
```

Alternativou je slučování na základě četností pomocí funkce `fct_lump()`. Pro sloučení všech kategorií, kromě tří nejpočetnějších lze využít argument `n`:

```
countries$maj_belief_n <- fct_lump(countries$maj_belief,  
                                  n = 3,  
                                  other_level = "Other")  
  
levels(countries$maj_belief_n)
```

```
[1] "Protestant" "Orthodox"    "Catholic"    "Other"
```

Obdobně, pro sloučení všech kategorií, které tvoří alespoň 20 % všech pozorovaných hodnot, lze využít argument `prop`:

```
countries$maj_belief_prop <- fct_lump(countries$maj_belief,  
                                      prop = 0.2,  
                                      other_level = "Other")  
  
levels(countries$maj_belief_prop)
```

```
[1] "Orthodox" "Catholic" "Other"
```

18 Práce se stringy

Předmětem této kapitoly jsou *stringy*, tedy nestrukturovaný text. O analýze nestrukturovaného toho lze napsat mnoho, mnohem více, než kolik dokáže pojmut tato kniha. Představíme si proto pouze úplné základy a to s pomocí balíčku `stringr`, který je součástí Tidyverse.

18.1 Detekce stringů

Náš data frame `countries` obsahuje proměnnou `hd_title_name`. Jedná se o titula a jméno hlavy dané země (k roku 2018). Na rozdíl od ostatních kategoriálních proměnných jsou jak titul, tak jméno osoby v jednom sloupci. Práce s nimi tedy bude vyžadovat o něco jiný přístup, než na jaký jsme zvyklí.

Jedním z nejběžnějších úkonů je vyhledávání vzorců (*patterns*) v textu. Pro vybraní zemí, jejichž hlavou je král, je možné zkombinovat již známou funkci `filter()` s funkcí `str_detect()`. Tato funkce vrátí hodnotu `TRUE` pro všechny řádky, ve kterých se nachází zvolený vzorec znaků, v našem případě *“King”*:

```
countries %>%  
  filter(str_detect(hd_title_name, pattern = "King")) %>%  
  select(country, hd_title_name)
```

```
# A tibble: 5 x 2  
  country      hd_title_name  
  <chr>        <chr>  
1 Belgium    King - Philippe  
2 Spain      King - Felipe VI  
3 Netherlands King - Willem-Alexander  
4 Sweden     King - Carl XVI Gustaf  
5 Norway     King - Harald V
```

Pomocí stejné funkce je možné hledat i více vzorců na jednou. Pro vyhledání všech království našem datasetu vyhledáme všechny hlavy států s titulem *“King”* nebo *“Queen”*. Oba hledané vzorce oddělíme znakem `|`, značící logický operátor *OR*:


```
countries %>%
  filter(str_detect(hd_title_name, pattern = "King|Queen")) %>%
  select(country, hd_title_name)
```

```
# A tibble: 7 x 2
  country      hd_title_name
  <chr>        <chr>
1 Belgium     King - Philippe
2 Denmark     Queen - Margrethe II
3 Spain       King - Felipe VI
4 Netherlands King - Willem-Alexander
5 Sweden      King - Carl XVI Gustaf
6 United Kingdom Queen - Elizabeth II
7 Norway      King - Harald V
```

V některých případech nám bude stačit vědět, kolikrát se určitý vzorec vyskytuje v datech. K tomu využijeme funkci `str_count()`. Protože pracujeme s vektorem stringů, zkombinujeme ji s funkcí `sum()`, abychom získali celkový počet monarchů napří všemi zeměmi:

```
sum(str_count(countries$hd_title_name, pattern = "King|Queen"))
```

```
[1] 7
```

18.2 Separace stringů

Pro usnadnění budoucí práce by bylo lepší proměnnou `hd_title_name` rozdělit do dvou nových proměnných. První z nových proměnných bude titul hlavy státu (`title`), druhou poté samotné jméno státníka (`name`). Toho nejjednodušeji docílíme pomocí funkce `separate()` z balíčku `tidyr`. Prvním argumentem této funkce, je string, který chceme rozdělit. Druhým argumentem, `into`, je vektor obsahující jména nových proměnných. Třetím argumentem je separátor (`sep`), který rozděluje obsah první a druhé z nových proměnných:

```
countries %>%
  select(hd_title_name) %>%
  separate(hd_title_name,
           into = c("title", "name"),
           sep = "-") %>%
  head(5)
```

Warning: Expected 2 pieces. Additional pieces discarded in 4 rows [5, 11, 19, 30].

```
# A tibble: 5 x 2
  title      name
  <chr>     <chr>
1 King      " Philippe"
2 President " Rumen Radev"
3 President " Miloš Zeman"
4 Queen     " Margrethe II"
5 President " Frank"
```

Tento kód téměř funguje, jak má, s jedním drobným problémem. Jak nás upozorňuje varování `Warning: Expected 2 pieces. Additional pieces discarded in 4 rows [5, 11, 19, 30]`. v několika jménech se objevil náš separátor - více než jednou. Protože jsme ale specifikovali pouze dvě nové proměnné, `title` a `name`, zahodili jsme omylem část jmen na řádcích 5, 11, 19 a 30. Napravit to můžeme pomocí argumentu `extra = "merge"`, pomocí kterého zachováme všechny jména celé:

```
countries %>%
  select(hd_title_name) %>%
  separate(hd_title_name,
           into = c("title", "name"),
           sep = "-",
           extra = "merge") %>%
  head(5)
```

```
# A tibble: 5 x 2
  title      name
  <chr>     <chr>
1 King      " Philippe"
2 President " Rumen Radev"
3 President " Miloš Zeman"
4 Queen     " Margrethe II"
5 President " Frank-Walter Steinmeier"
```

18.3 Transformace stringů

V některých případech je nutné stringy transformovat, buď do podoby vhodné pro analýzy nebo naopak do podoby vhodné pro prezentaci výstupů. Balíček `stringr` pro transformaci

stringů nabízí hned několik funkcí. Funkce `str_to_lower()` převede všechna písmena na malá, `str_to_upper()` naopak na velká. `str_to_sentence()` převede první písmeno na velké a zbytek na malá, a nakonec `str_to_title()` převede první písmeno každého slova na velké a zbytek na malá:

```
str_to_lower(countries$hd_title_name) %>% head(5)
```

```
[1] "king - philippe"           "president - rumen radev"  
[3] "president - miloš zeman"   "queen - margrethe ii"  
[5] "president - frank-walter steinmeier"
```

```
str_to_upper(countries$hd_title_name) %>% head(5)
```

```
[1] "KING - PHILIPPE"           "PRESIDENT - RUMEN RADEV"  
[3] "PRESIDENT - MILOŠ ZEMAN"   "QUEEN - MARGRETHE II"  
[5] "PRESIDENT - FRANK-WALTER STEINMEIER"
```

```
str_to_sentence(countries$hd_title_name) %>% head(5)
```

```
[1] "King - philippe"           "President - rumen radev"  
[3] "President - miloš zeman"    "Queen - margrethe ii"  
[5] "President - frank-walter steinmeier"
```

```
str_to_title(countries$hd_title_name) %>% head(5)
```

```
[1] "King - Philippe"           "President - Rumen Radev"  
[3] "President - Miloš Zeman"    "Queen - Margrethe Ii"  
[5] "President - Frank-Walter Steinmeier"
```

Část IV

Vizualizace dat

19 Struktura grafů

V této kapitole se seznámíme se základy vizualizace dat pomocí balíčky `ggplot2`, který je (nepřekvapivě) součástí *Tidyverse*. S tímto balíčkem jsme se již krátce setkali v kapitole věnované faktorům (Kapitola 6), kde jsme využívali funkci `qplot()`. Balíček `ggplot2` ovšem nabízí mnohem více.

19.1 Grammar of graphics

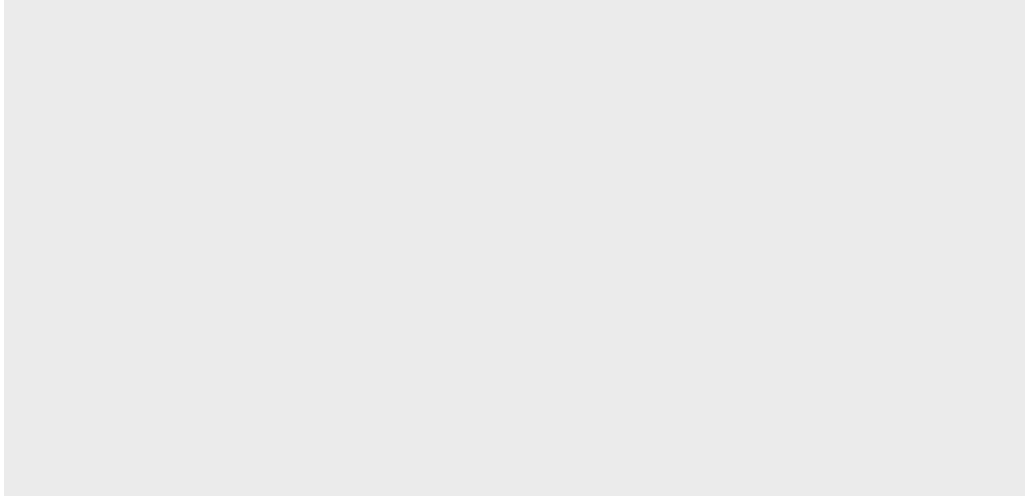
Ačkoliv to nemusí být na první pohled zřejmé, i vizualizace dat je předmětem vědeckého výzkumu a teorie. Teoretické základy balíčku `ggplot2` leží v takzvané “*grammar of graphics*” (Wilkinson, 2005). Základními pilíři tohoto paradigmatu, tak jak je implementované zde, jsou takzvané *scales*, *geoms* a *themes*:

- *Scales* jsou dimenze grafu, v kterých se nachází data. Jedná se osy grafu, ale také například o barvu nebo velikost.
- *Geoms* jsou objekty, které fyzicky reprezentují data v grafu. Jde například o sloupce ve sloupcovém grafu nebo body v bodovém grafu.
- *Themes* kontrolují estetickou stránku grafu, jako velikost nebo font písma, barvu pozadí nebo zda jsou v grafu přítomné vodící přímký.

19.2 Struktura `ggplot2` grafů

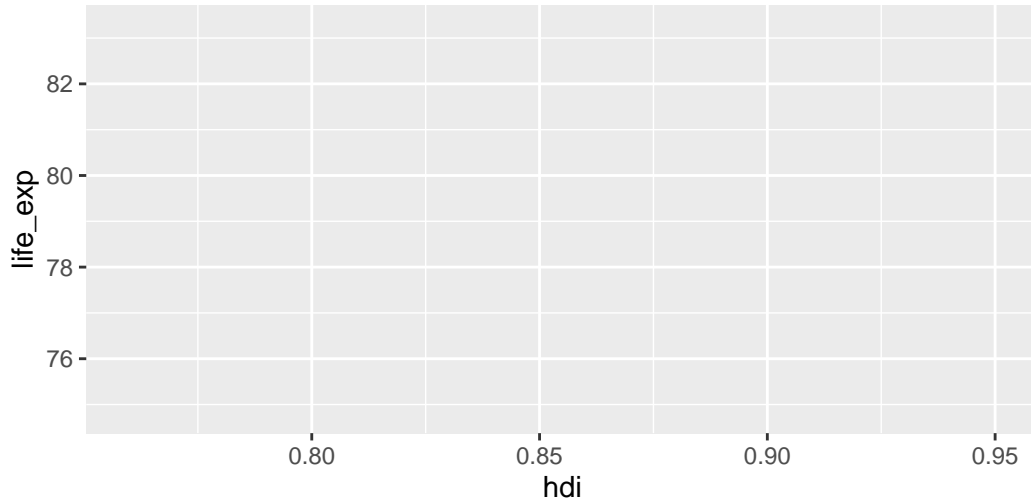
Jednotlivé složky se na sebe nanášejí ve vrstvách (*layers*). Tvorba každého grafu bude začínat funkcí `ggplot()`:

```
ggplot(data = countries)
```



Funkce `ggplot()` vyžaduje minimálně argument `data`, pomocí kterého specifikujeme náš dataframe. Výsledkem je prázdné plátno. Druhým krokem je specifikovat dimenze (*scales*) našeho grafu. Toho docíleme pomocí funkce `aes()` (zkratka pro *aesthetics*) a argumentu `mapping`:

```
ggplot(data = countries,  
       mapping = aes(x = hdi, y = life_exp))
```



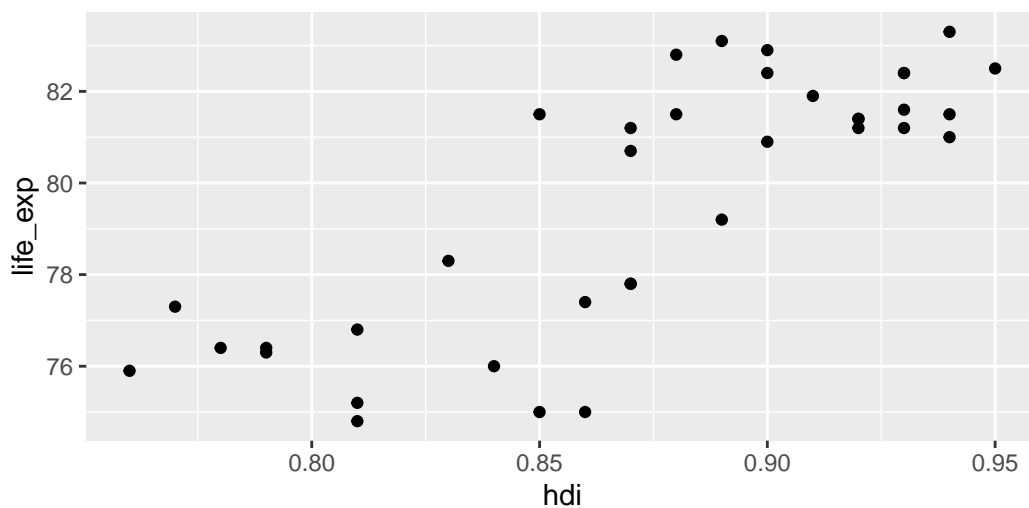
Pomocí funkce `aes()` jsme definovali dvě dimenze (*scales*). Ose X jsme přiřadili proměnnou `hdi`, a na osu Y jsme “namapovali” proměnnou `life_exp`. Výsledkem je graf s popsánými osami, nicméně pořád bez dat.

💡 Mapování

Slovo “mapovat” je zde používáno v matematickém významu, tedy ve smyslu přiřazování elementů jednoho setu k elementům druhého setu. V našem případě přiřazujeme proměnné v datech k dimenzím v grafu. Od toho je odvozeno také jméno argumentu `mapping`.

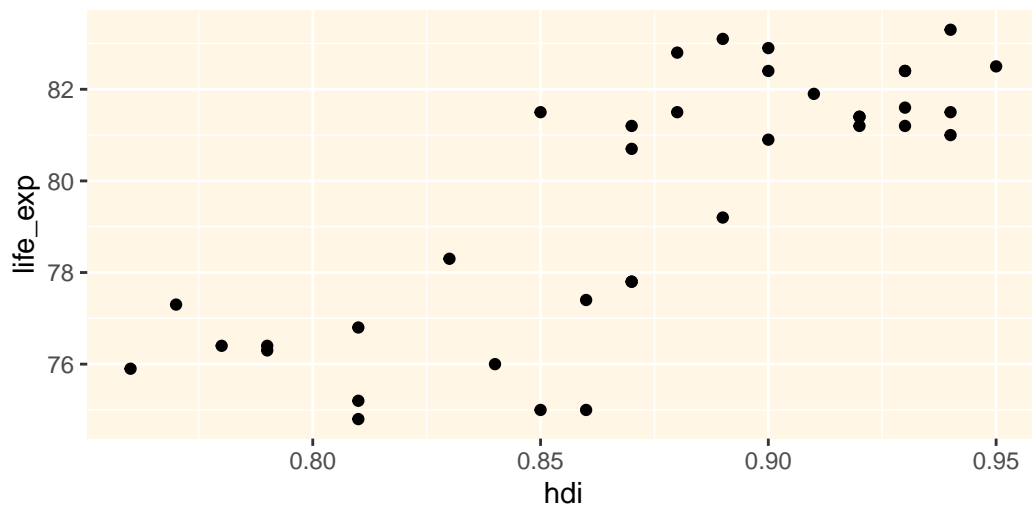
Dalším krokem je přidání geomů, tedy objektů, které budou reprezentovat jednotlivé pozorování. V našem případě se nabízí zobrazit jednotlivé země jako body, využijeme tedy funkce `geom_point()`:

```
ggplot(data = countries,  
       mapping = aes(x = hdi, y = life_exp)) +  
  geom_point()
```



Výsledkem je starý známý bodový graf (*scatter plot*), ve kterém je každý řádek dataframu reprezentovaný bodem. Posledním krokem je úprava vizuální stránky grafu, jako například barvy pozadí. Toho docílíme pomocí funkce `themes()`, která má řadu argumentů, mezi nimi i `panel.background`. Na to, jak přesně tato funkce funguje, se zaměříme v budoucích kapitolách:

```
ggplot(data = countries,  
       mapping = aes(x = hdi, y = life_exp)) +  
  geom_point() +  
  theme(panel.background = element_rect(fill = "#fff6e5"))
```



A to je v podstatě celá logika balíčku `ggplot2`! V následujících kapitolách si představíme nejpoužívanější dimenze/*scales* a *geoms*, a ponoříme se také do fungování funkce `theme()`.

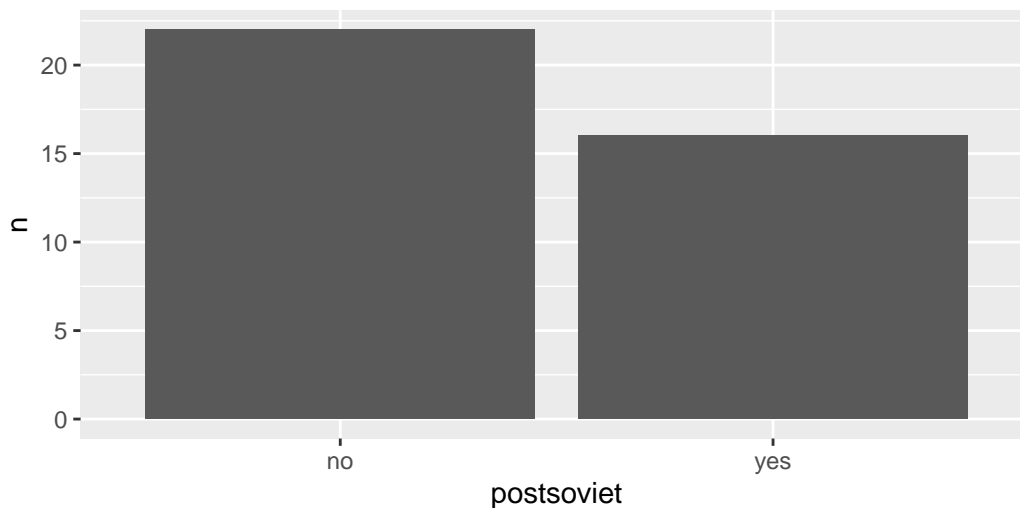
20 Vizualizace kategorických proměnných

V této kapitole si představíme nejčastější typy grafů pro vizualizaci kategorických proměnných.

20.1 Vizualizace jedné proměnné

Začněme vizualizací jedné kategoriální proměnné. Sloupcové grafy jsou pravděpodobně nejpopulárnějším typem vizualizace, se kterým se setkáme. Pro vytvoření sloupcového grafu je dobré si vybavit, které proměnné se nachází na jednotlivých osách. Na ose X se nachází název kategorie, na ose Y poté frekvence výskytu. Vstupními daty pro funkci `ggplot()` bude tedy dataframe s těmito dvěma proměnnými. Data budou reprezentovaná pomocí sloupců, které přidáme funkcí `geom_col()`:

```
countries %>%  
  count(postsoviet) %>%  
  ggplot(aes(x = postsoviet, y = n)) +  
  geom_col()
```



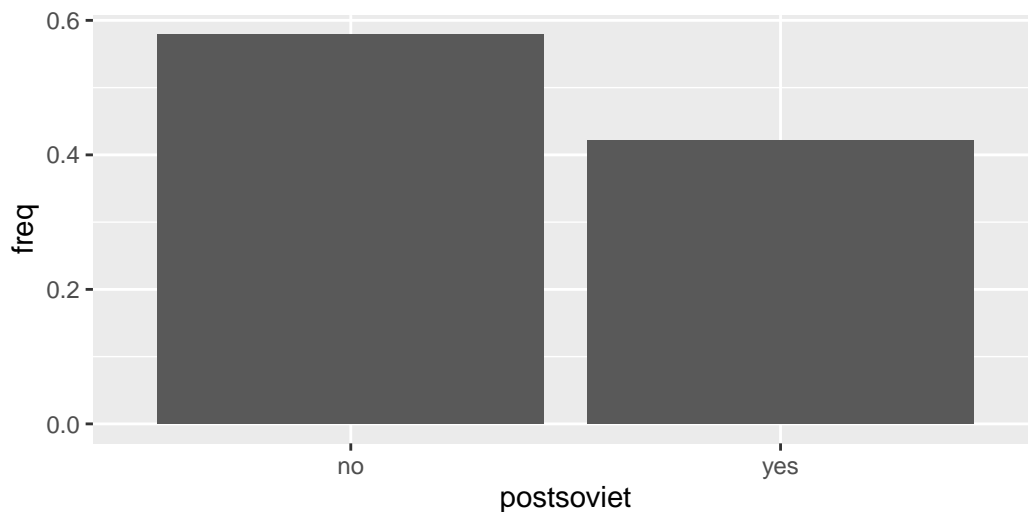
💡 geom_bar()

Pokud je cílem našeho sloupcového grafu zobrazit (absolutní) četnost jednotlivých kategorií, můžeme nahradit `geom_col()` funkcí `geom_bar()`, která automaticky frekvenci výskytu všech skupin. Celý kód by vypadal následovně:

```
ggplot(countries,  
       aes(x = postsoviet)) +  
  geom_bar()
```

Pokud bychom v grafu chtěli zobrazit relativní frekvenci výskytu kategorií, spočítáme procentuální zastoupení před vytvořením grafu.

```
countries %>%  
  count(postsoviet) %>%  
  mutate(freq = n / sum(n)) %>%  
  ggplot(aes(x = postsoviet, y = freq)) +  
  geom_col()
```



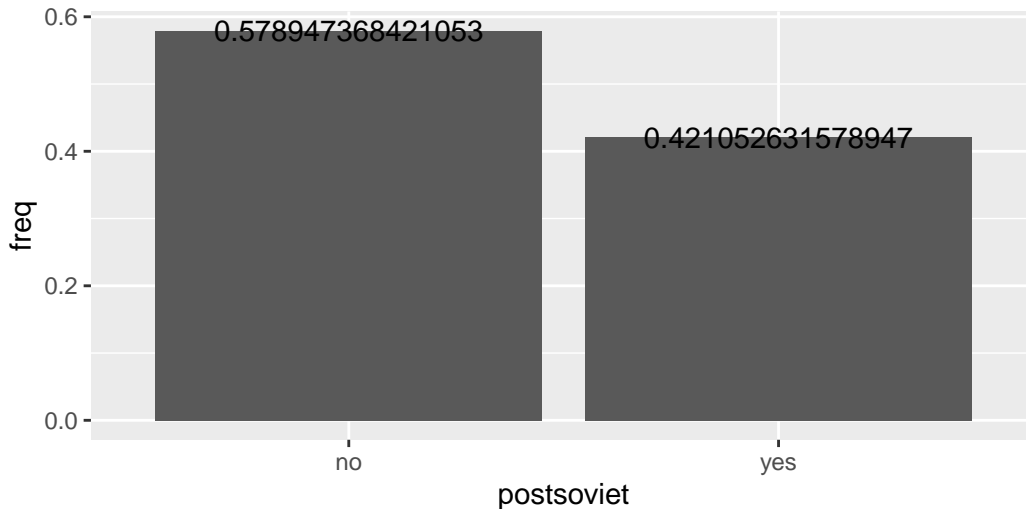
💡 Orientace grafu

Názvy kategorií nemusí být nutně na ose X. Pro otočení grafu o 90 stupňů stačí použít `ggplot(aes(x = freq, y = postsoviet))`.

Do grafu také můžeme přidat popisky jednotlivých sloupců. Nejdříve je nutné napojit proměnou obsahující frekvenci jednotlivých kategorií na dimenzi `label`. Popisky přidáme do

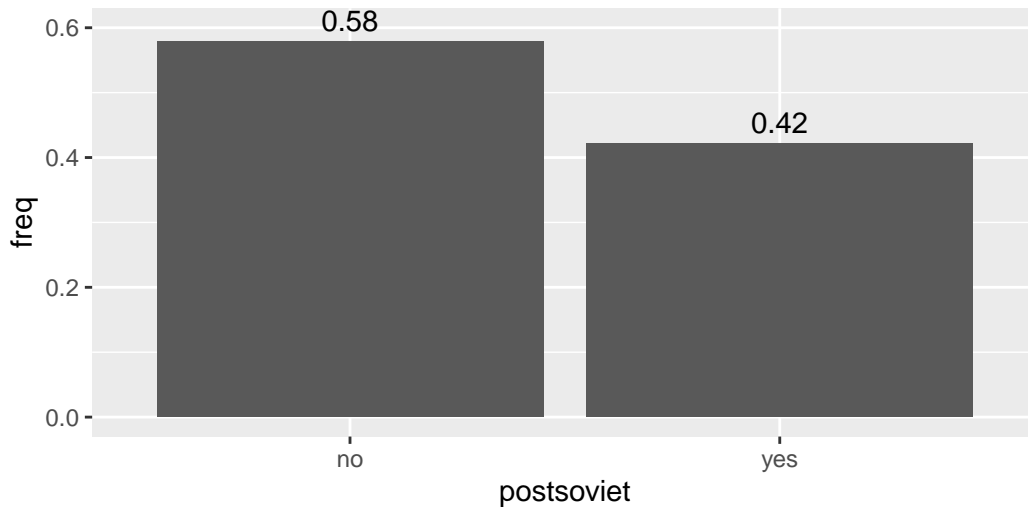
grafu pomocí funkcce `geom_text()`:

```
countries %>%  
  count(postsoviet) %>%  
  mutate(freq = n / sum(n)) %>%  
  ggplot(aes(x = postsoviet, y = freq, label = freq)) +  
  geom_col() +  
  geom_text()
```



Výsledkem je funkční, ale nepřilíš vzhledný graf. Aby náš graf byl použitelný, je nutné čísla v popiscích zaokrouhlit a popisky samotné posunout tak, aby nepřekrývali sloupce. Zaokrouhlení dosáhneme pomocí funkce `round()`, kterou můžeme aplikovat přímo uvnitř funkce `ggplot()`. Pro posunutí popisků na vertikální ose je možné využít argument `vjust` uvnitř funkce `geom_text()`, k posouvání na horizontální ose poté slouží `hjust`. Hodnoty argumentů `vjust` a `hjust` jsou ve stejných jednotkách, jako proměnná na dané ose.

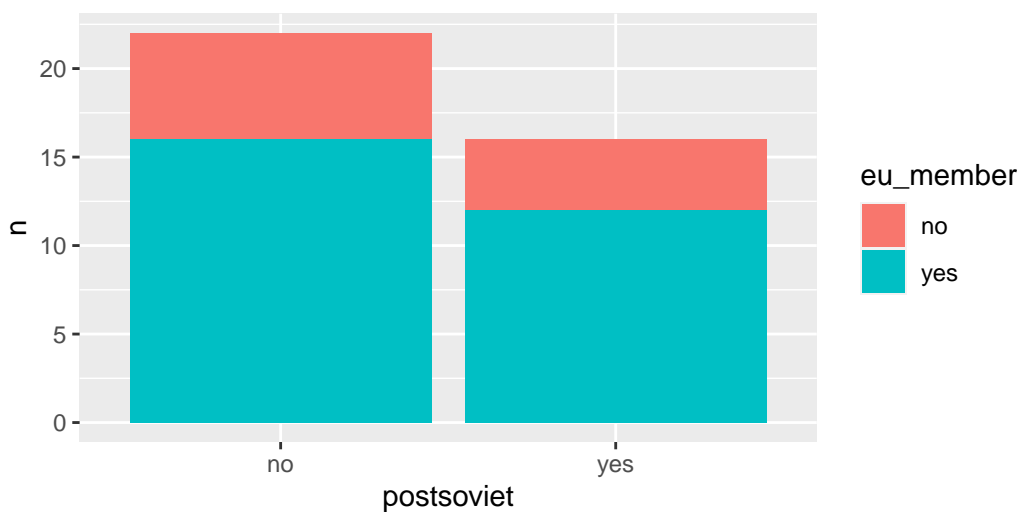
```
countries %>%  
  count(postsoviet) %>%  
  mutate(freq = n / sum(n)) %>%  
  ggplot(aes(x = postsoviet, y = freq, label = round(freq, 2)) ) +  
  geom_col() +  
  geom_text(vjust = -0.5)
```



20.2 Vizualizace více proměnných

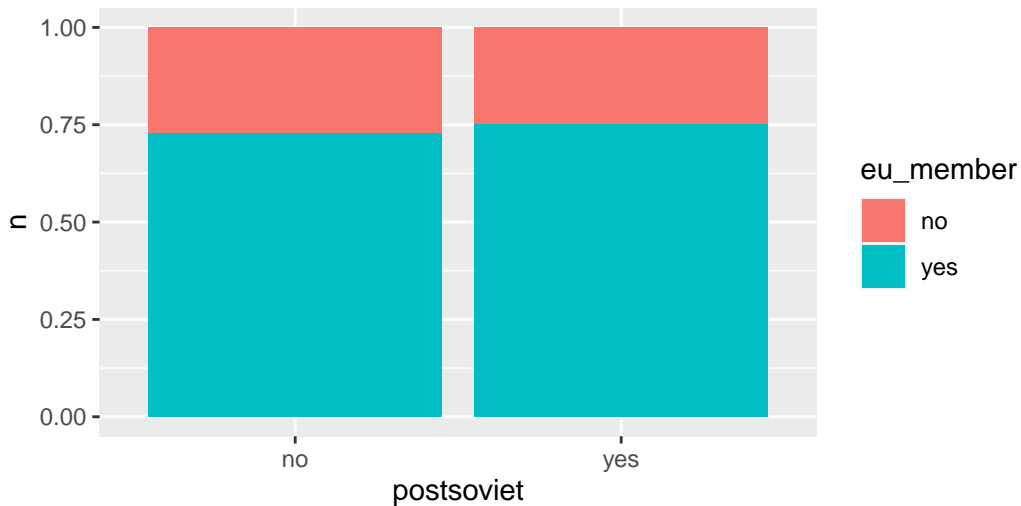
Pro vizualizaci většího počtu proměnných pomocí sloupcových grafů zpravidla využíváme barev, pro rozlišení jednotlivých kategorií. Jedna kategorická proměnná bude tedy namapovaná na osu X, druhá poté na barvu sloupce. Frekvence dané kategorie bude na ose Y:

```
countries %>%
  count(postsoviet, eu_member) %>%
  ggplot(aes(x = postsoviet, fill = eu_member, y = n)) +
  geom_col()
```



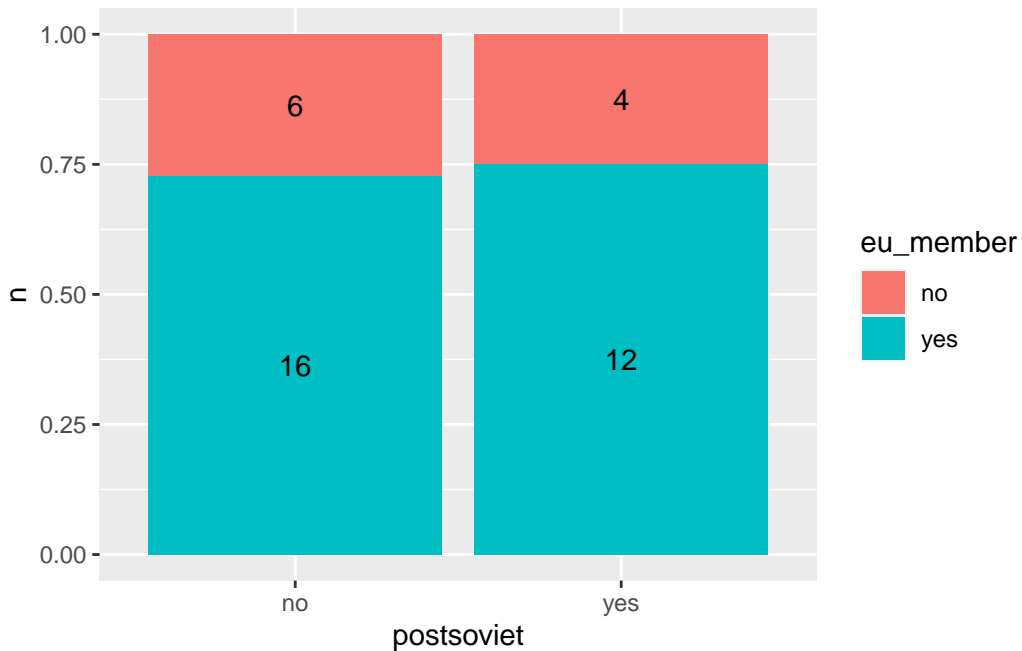
Všimněme si, že v našem grafu jsou nyní jednotlivé kategorie naskládány na sebe. Jejich pozici je možné upravovat pomocí argumentu `position`, pro který je výchozí hodnota `position = "stack"`. První alternativou je `argument = "fill"`, který je obdobný `stack`, ale velikost sloupců je standardizována. Graf tedy zobrazuje relativní frekvenci jednotlivých kategorií:

```
countries %>%  
  count(postsoviet, eu_member) %>%  
  ggplot(aes(x = postsoviet, fill = eu_member, y = n)) +  
  geom_col(position = "fill")
```



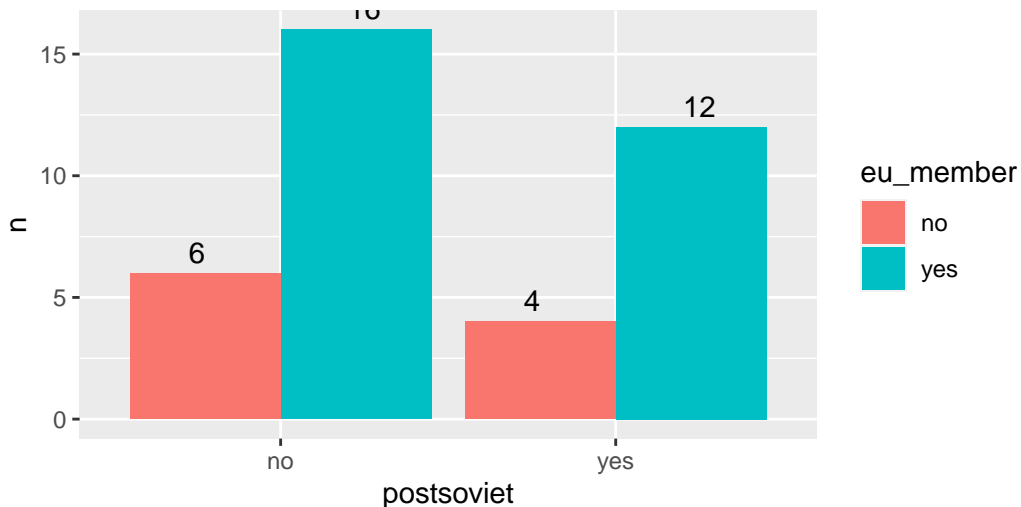
Stejně jako u jednoduchého sloupcového grafu, i do skládaných grafů je možné přidat popisky. Je ale nutné sladit jejich pozici s pozicí sloupců. Pokud jsme jako sloupců zvolili `position = "fill"`, je nutné stejnou pozici zvolit i pro popisky. Také je nutné popisky zarovnat doprostřed jednotlivých dlaždic. Obojího docíleme pomocí `position = position_fill(vjust = 0.5)`:

```
countries %>%  
  count(postsoviet, eu_member) %>%  
  ggplot(aes(x = postsoviet, fill = eu_member, y = n, label = n)) +  
  geom_col(position = "fill") +  
  geom_text(position = position_fill(vjust = 0.5))
```



Druhou alternativou je `position = "dodge"`, pomocí které je možné vyskládat jednotlivé sloupce vedle sebe. Obdobně jako u předchozí varianty přidáme popisky, tentokrát ale pomocí `position_dodge()`. jednotlivé sloupce jsou od sebe zpravidla vzdálené jednu "jednotku". Stejně jako u jednoduchého grafu také popisky posuneme lehce nahoru:

```
countries %>%
  count(postsoviet, eu_member) %>%
  ggplot(aes(x = postsoviet, fill = eu_member, y = n, label = n)) +
  geom_col(position = "dodge") +
  geom_text(position = position_dodge(width = 1), vjust = -0.5)
```

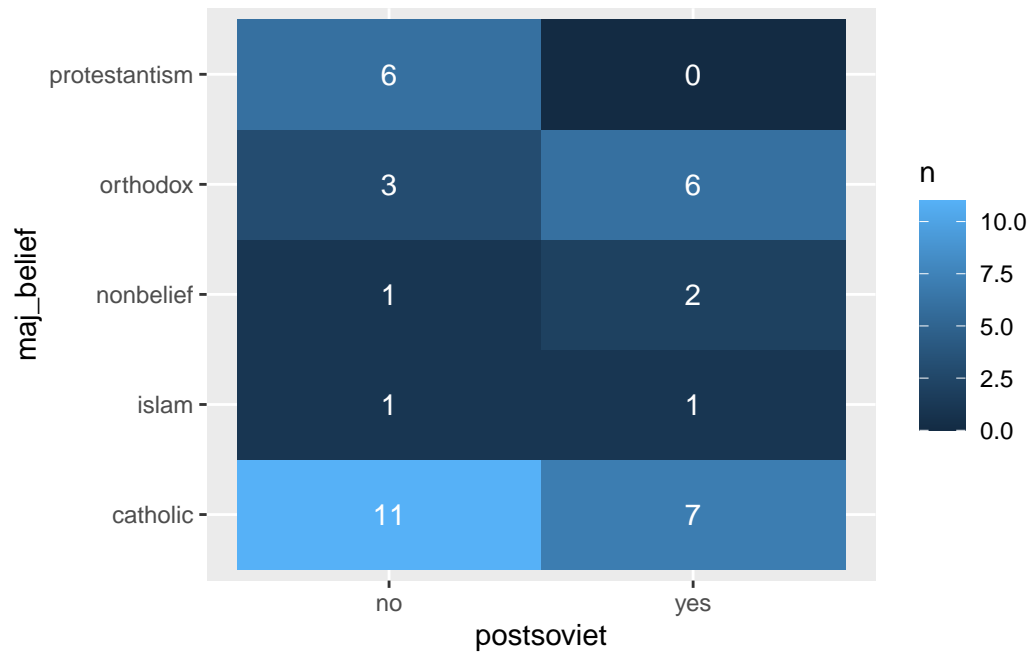


Alternativou klasických sloupcových grafů je *heat map*. Pro vytvoření *heat* mapy nejdříve získáme frekvenci výskytu kombinací jednotlivých kategorií, obdobně jako když jsme vytvářeli sloupcový graf. Poté jen napojíme jednu z kategorických proměnných na osu X, druhou na osu Y a frekvenci výskytu na barvu jednotlivých “dlaždic”. Graf dokončíme pomocí funkce `geom_tile()`. Stejně jako v předchozích grafech můžeme přidat popisky pro jednotlivé dlaždice pomocí `geom_text()`.

🔥 Chybějící kombinace kategorií

V případě, že se v naší datech nevyskytují některé kombinace kategorií, je nutné proměnné převést na faktory a do funkce `count()` přidáme argument `.drop = FALSE`. Pokud bychom to neudělali, objevily by se v naší *heat* mapě mezery.

```
countries %>%
  mutate(across(c(maj_belief, postsoviet),
                 as.factor)) %>%
  count(postsoviet, maj_belief, .drop = F) %>%
  ggplot(aes(x = postsoviet, y = maj_belief,
            label = n, fill = n)) +
  geom_tile() +
  geom_text(color = "white")
```



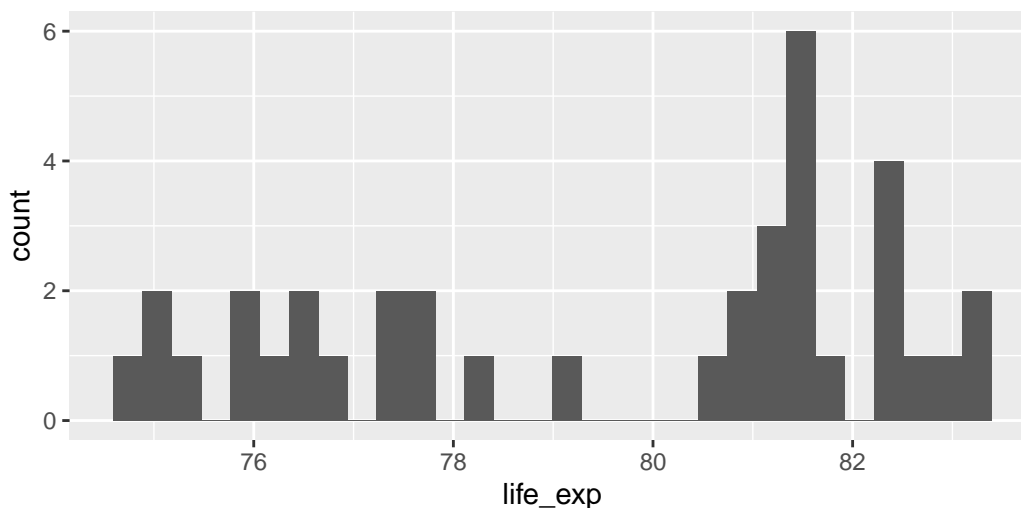
21 Vizualizace numerických proměnných

V předchozí kapitole jsme si ukázali nejčastější způsoby vizualizace kategorických proměnných, v této se pustíme do proměnných numerických.

21.1 Vizualizace jedné proměnné

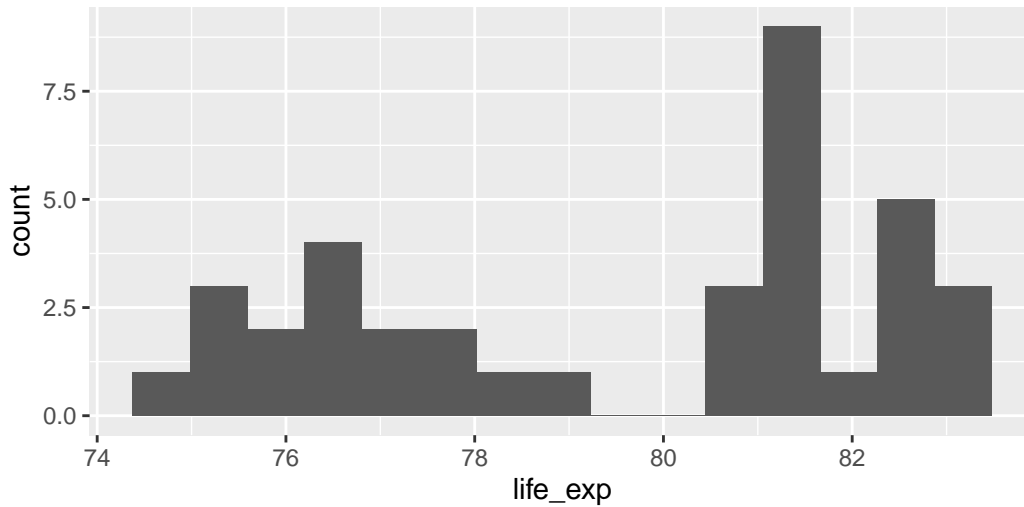
Numerické proměnné jsou zpravidla vizualizovány pomocí histogramu, tedy sloupcové grafu, který zobrazuje frekvenci jednotlivých hodnot shluknutých do menšího počtu kategorií (v angličtině zvaných *bins*). Vytvoření histogramu je přímočaré:

```
ggplot(countries,  
       aes(x = life_exp)) +  
  geom_histogram()
```



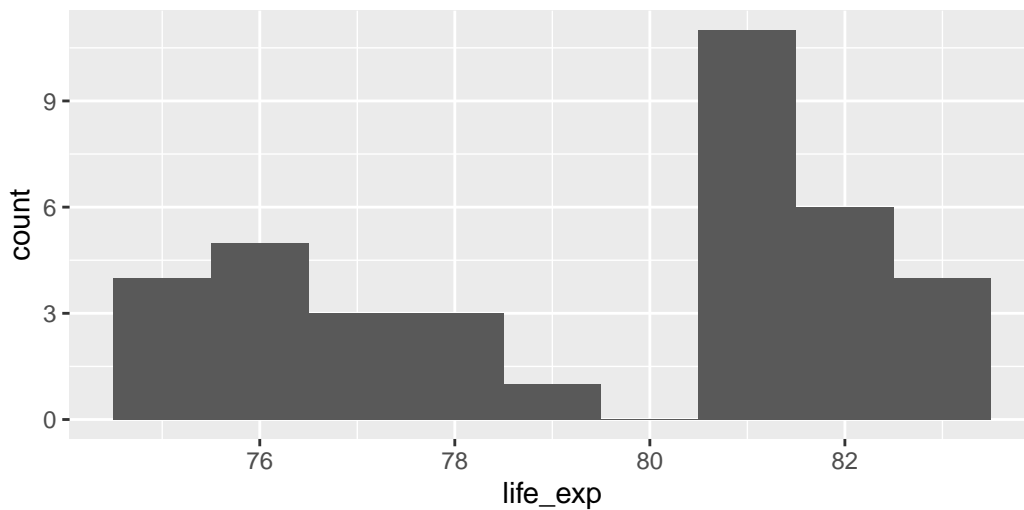
Počet kategorií je možné kontrolovat pomocí jednoho ze dvou argumentů. Prvním z nich je `bins`, pomocí kterého je možné kontrolovat celkový počet kategorií. Například, pro 15 kategorií zvolíme následující:

```
ggplot(countries,  
       aes(x = life_exp)) +  
  geom_histogram(bins = 15)
```



Druhým argumentem je `binwidth`, pomocí kterého je možné specifikovat šířku jednotlivých kategorií. Pokud chceme, aby kategorie měly šířku jednoho (v našem případě) roku, použijeme následující kód:

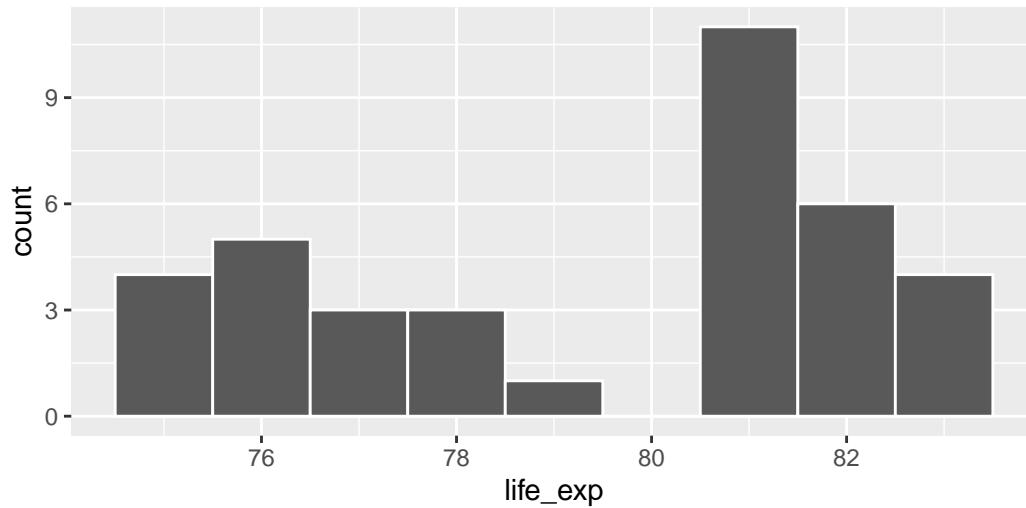
```
ggplot(countries,  
       aes(x = life_exp)) +  
  geom_histogram(binwidth = 1)
```



💡 Tip

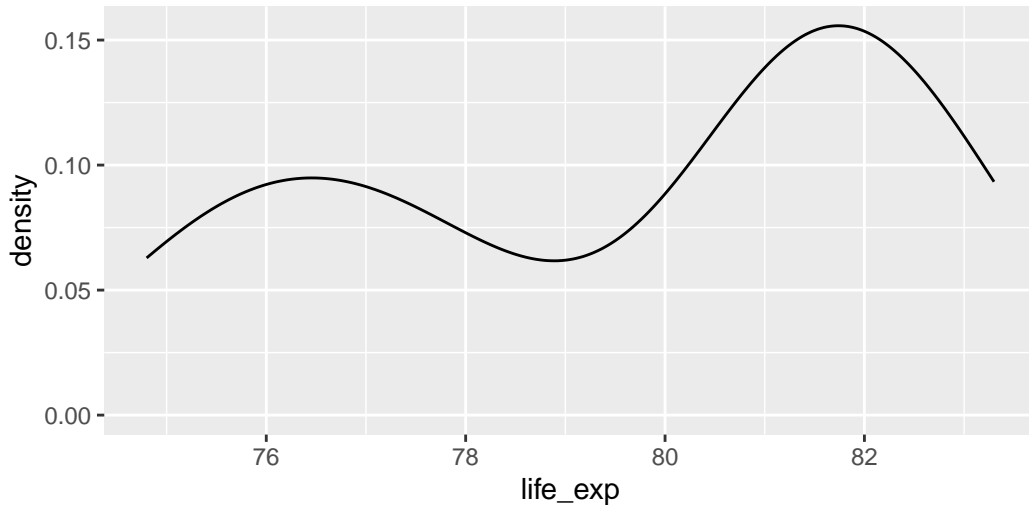
Někteří lidé preferují, když jsou jednotlivé kategorie (*bins*) vizuálně oddělené. Toho můžeme docílit tak, že nastavíme barvu manuálně:

```
ggplot(countries,  
       aes(x = life_exp)) +  
  geom_histogram(binwidth = 1, color = "white")
```



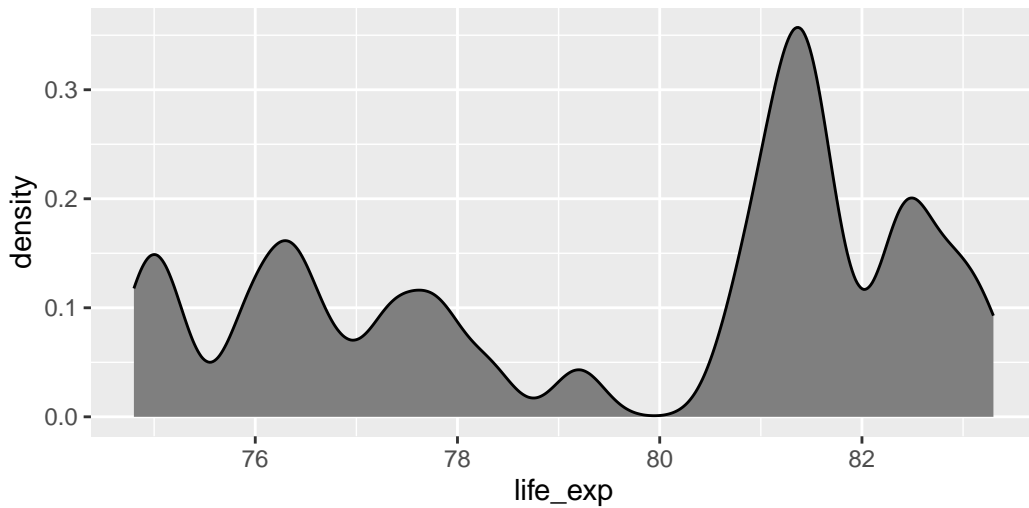
Alternativou k histogramu je graf hustoty (*density plot*). Ty na rozdíl od histogramů nekategorizují vizualizovanou proměnnou, místo toho odhadují podobu spojitého rozdělení, kterou proměnná nabývá:

```
ggplot(countries,  
       aes(x = life_exp)) +  
  geom_density()
```



Míru “vyhlazení” (*smoothing*) grafu hustoty je možné kontrolovat pomocí argument `bw` (*bandwidth* zkráceně). Nižší hodnoty povedou k menšímu vyhlazení. Někteří také mohou preferovat, pokud je plocha pod křivkou hustoty vybarvená, čehož lze docílit pomocí argumentu `fill`:

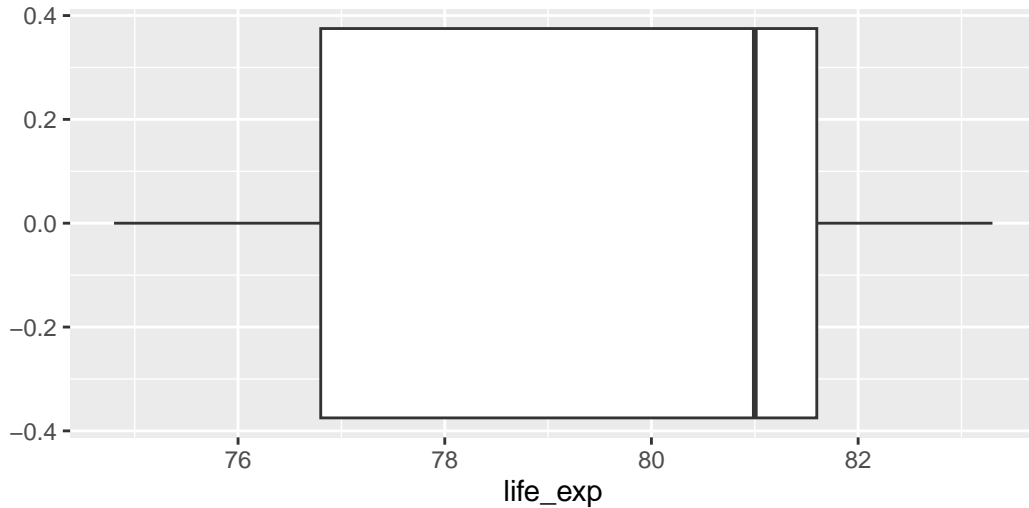
```
ggplot(countries,
       aes(x = life_exp)) +
  geom_density(bw = 0.25, fill = "grey50")
```



Poslední možností je boxplot, který zobrazuje vybrané kvartily proměnné. Hranice krabice “krabice” zobrazují první a třetí kvartil, úsečka uvnitř krabice reprezentuje medián a “fousky”

grafu reprezentují mezikvartilové rozpětí vynásobené konstantou (zpravidla 1,5):

```
ggplot(countries,  
       aes(x = life_exp)) +  
  geom_boxplot()
```

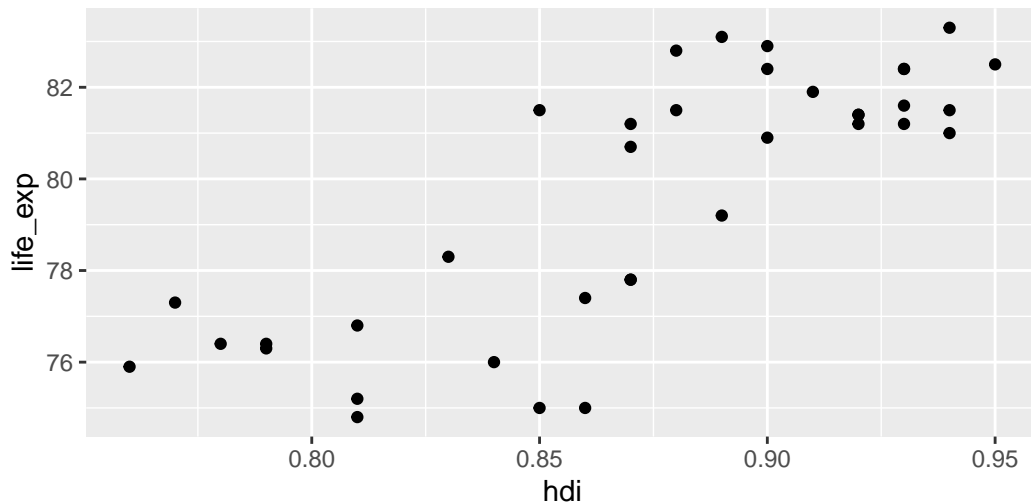


Z grafu výše je možné vyčíst, že medián naděje na dožití našich zemí je 81 let. První kvartil je zhruba 76,8 let a třetí kvartil je zhruba 81,5 let. Fousky grafy, které zpravidla reprezentují hranice pro odlehlá pozorování mají hodnoty 74,8 a 83,2 let.

21.2 Vizualizace více proměnných

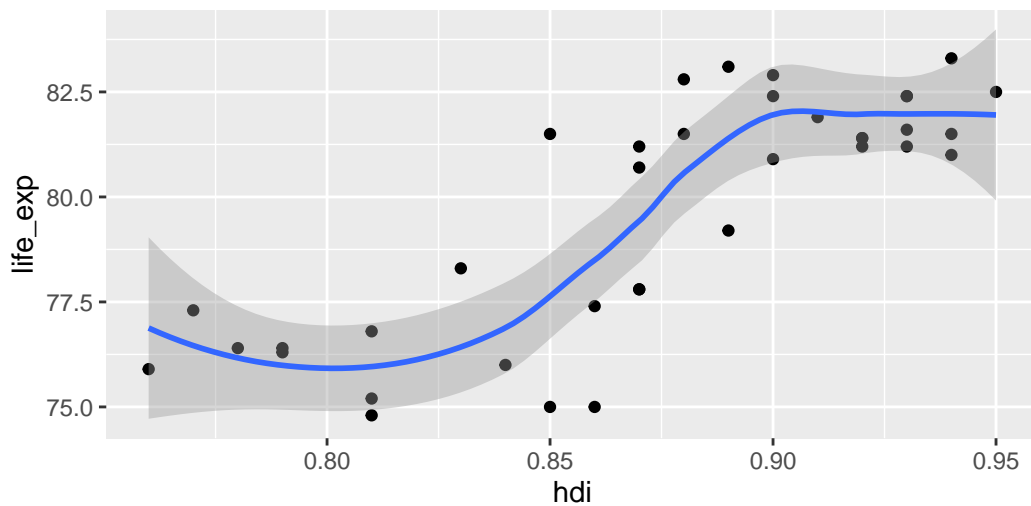
Vztah dvou numerických proměnných je typicky zobrazen pomocí bodového grafu, známého také jako *scatterplot*. Pro vytvoření bodového grafu stačí přiřadit jednu proměnnou na osu X a druhou na osu Y. Poté jen zobrazíme data pomocí funkce `geom_point()`:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp)) +  
  geom_point()
```



Pro lepší přehled může být užitečné přidat křivku vyjadřující vztah mezi proměnnými. Balíček `ggplot2` na to poskytuje užitečnou funkci zvanou `geom_smooth()`. Ve výchozím nastavení tato funkce zobrazí křivku reprezentující takzvanou *locally estimated scatterplot smoothing (loess)* regresi, neparametrickou techniku pro popis vztahů mezi numerickými proměnnými, včetně 95 % intervalů spolehlivosti:

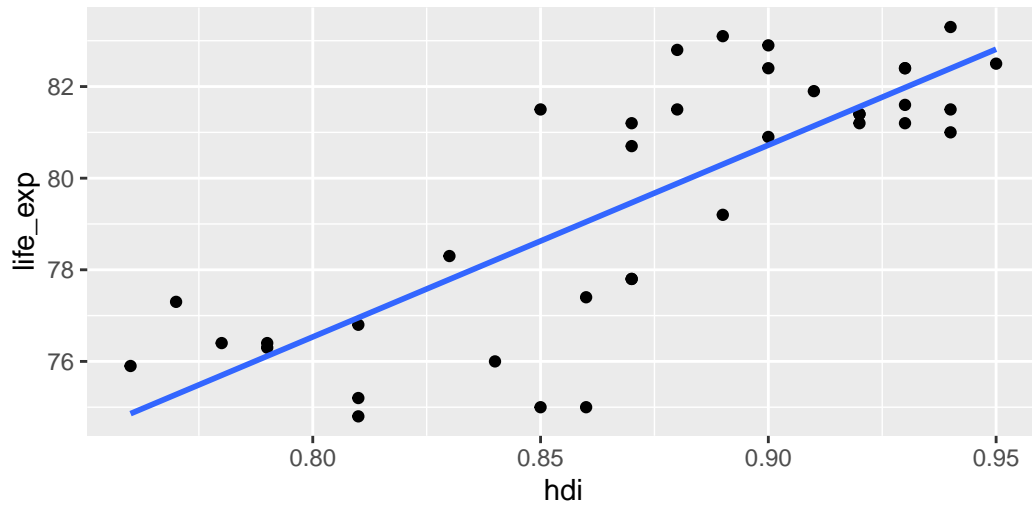
```
ggplot(countries,
       aes(x = hdi, y = life_exp)) +
  geom_point() +
  geom_smooth()
```



Kromě *loess* regrese můžeme aplikovat také klasickou lineární regresi, pomocí argumentu

method = "lm" . Také se můžeme zbavit intervalů spolehlivosti pomocí se = FALSE:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp)) +  
  geom_point() +  
  geom_smooth(method = "lm", se = FALSE)
```



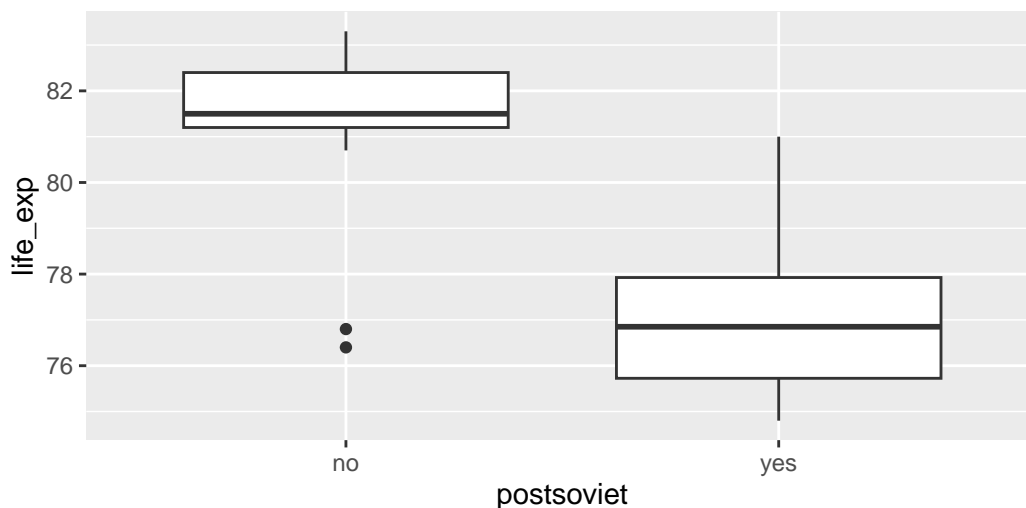
22 Kombinované grafy

Nejdříve jsme si ukázali jak na vizualizaci kategoriální proměnných. Poté jak na vizualizaci numerických proměnných. Teď už nás čeká jen jejich kombinace. Princip vytváření kombinovaných grafu je stejný, jako u grafů jednodušších, je ale nutné upozornit na pár chytáků.

22.1 Boxploty

Pro vytvoření boxplotu pro větší počet skupin stačí přidat kategoričnou proměnnou na jednu z os:

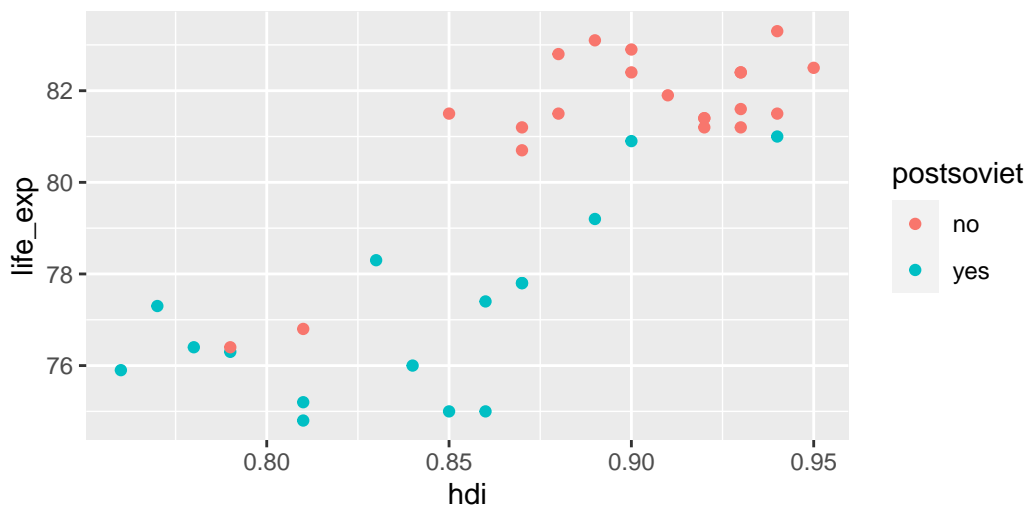
```
ggplot(countries,  
       aes(x = postsoviet, y = life_exp)) +  
  geom_boxplot()
```



22.2 Bodové grafy

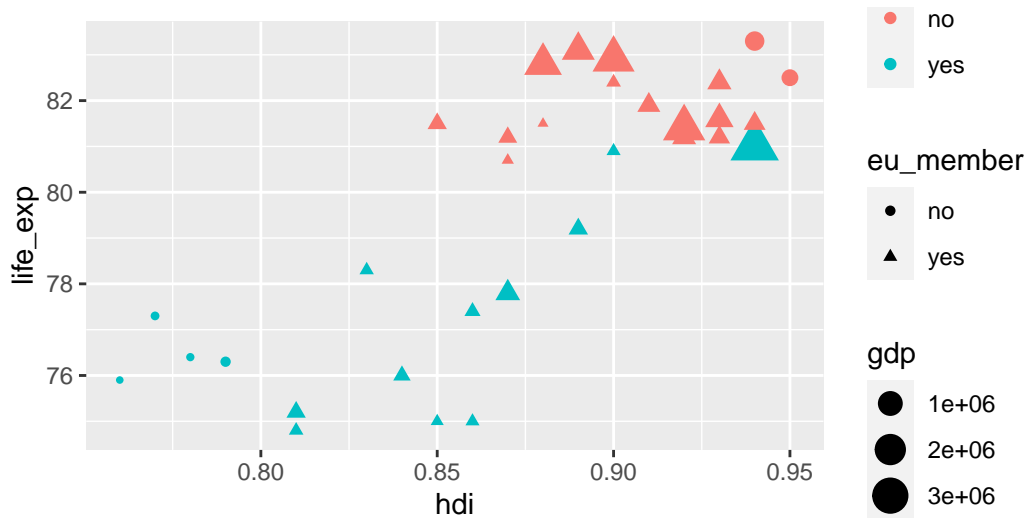
V případě, že jsou obě osy grafu obsazeny numerickými proměnnými, jako je to například v případě bodových grafů, musí být kategorické proměnné namapované na jiné dimenze. Nejčastějším kandidátem je barva:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp, color = postsoviet)) +  
  geom_point()
```



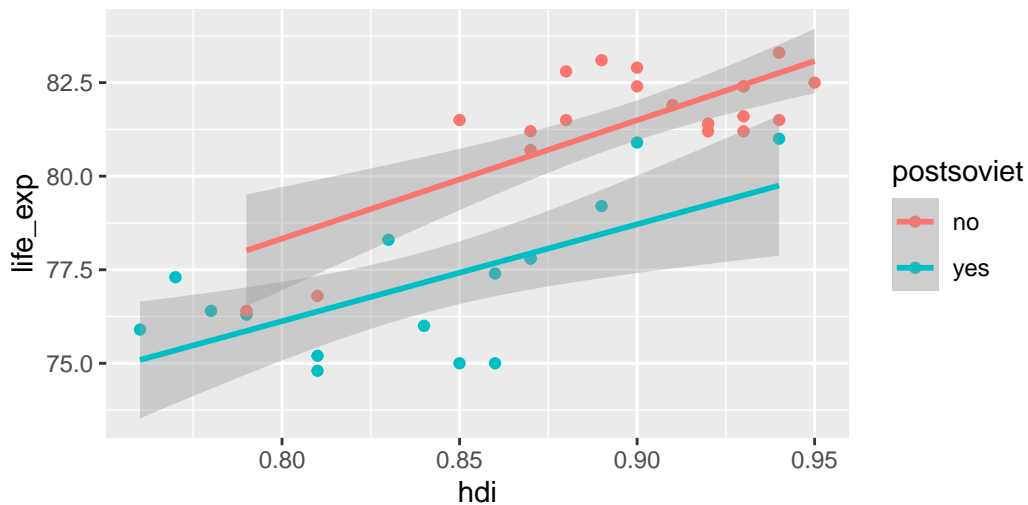
Co kdybychom chtěli ale do grafu zapojit více proměnných? V takovém případě můžeme využít dimenzí tvaru (`shape`) a velikosti (`size`). Získáme tak (poněkud překombinovaný) graf zobrazující až pět proměnných:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp, color = postsoviet,  
           shape = eu_member, size = gdp)) +  
  geom_point()
```



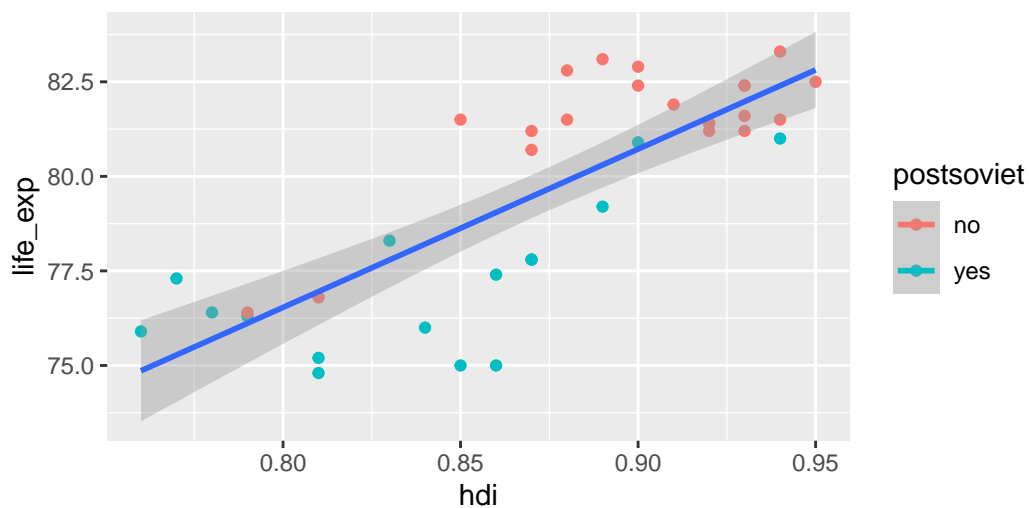
Poznámku si zaslouží bodové grafy obsahující regresní přímky (nebo křivky) vytvořené pomocí `geom_smooth()`. Ve výchozím nastavení bude do grafu přidána přímka pro každou kategorii:

```
ggplot(countries,
       aes(x = hdi, y = life_exp, color = postsoviet)) +
  geom_point() +
  geom_smooth(method = "lm")
```



Pokud bychom chtěli jednu přímku pro celý graf, je nutné přidat, argument `group = 1`, pomocí kterého řekneme grafu, že pro potřeby výpočtu vlastností přímky patří všechna pozorování do jedné skupiny:

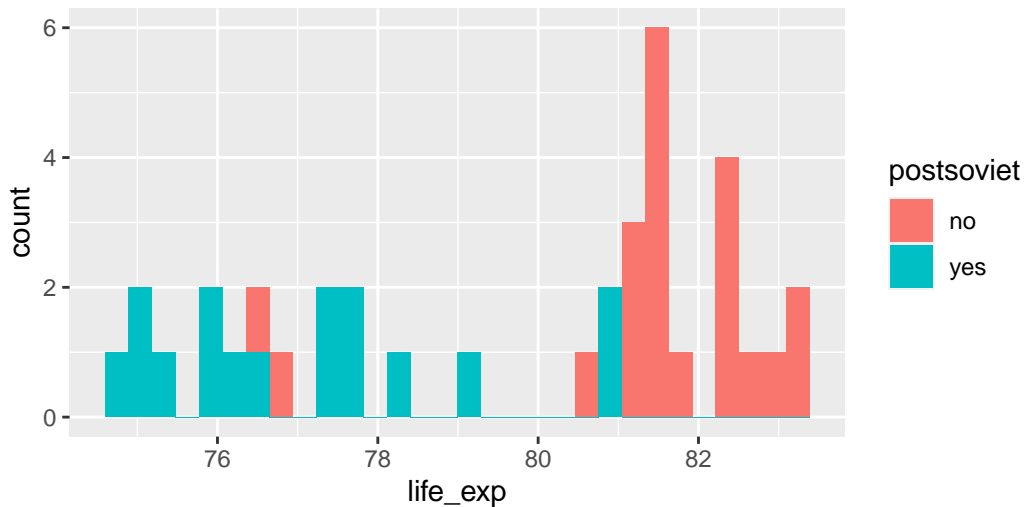
```
ggplot(countries,
       aes(x = hdi, y = life_exp, color = postsoviet, group = 1)) +
  geom_point() +
  geom_smooth(method = "lm")
```



22.3 Histogramy a grafy hustoty

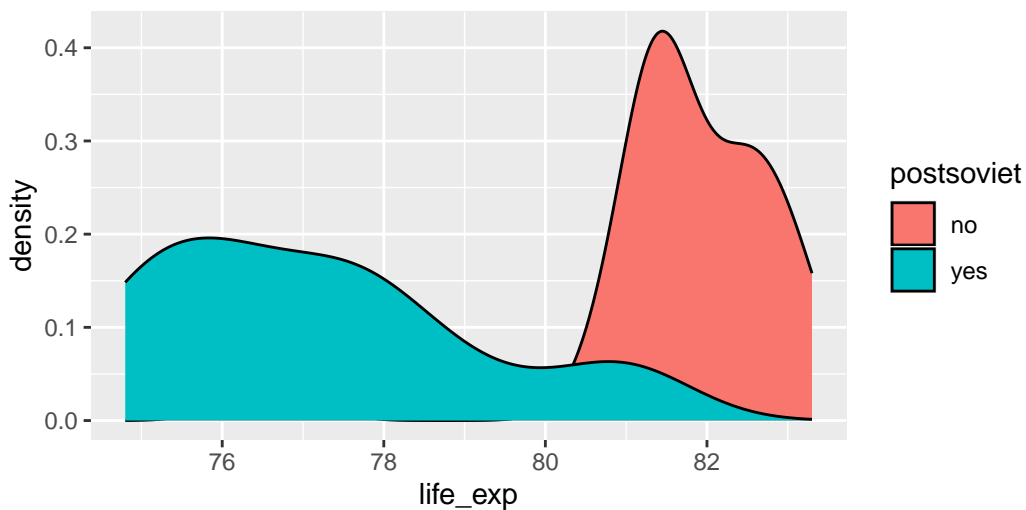
Zapojení kategoričkých proměnných do histogramů probíhá obdobně, jako u bodových grafu, a využijeme k tomu dimenzi barvy.

```
ggplot(countries,
       aes(x = life_exp, fill = postsoviet)) +
  geom_histogram()
```



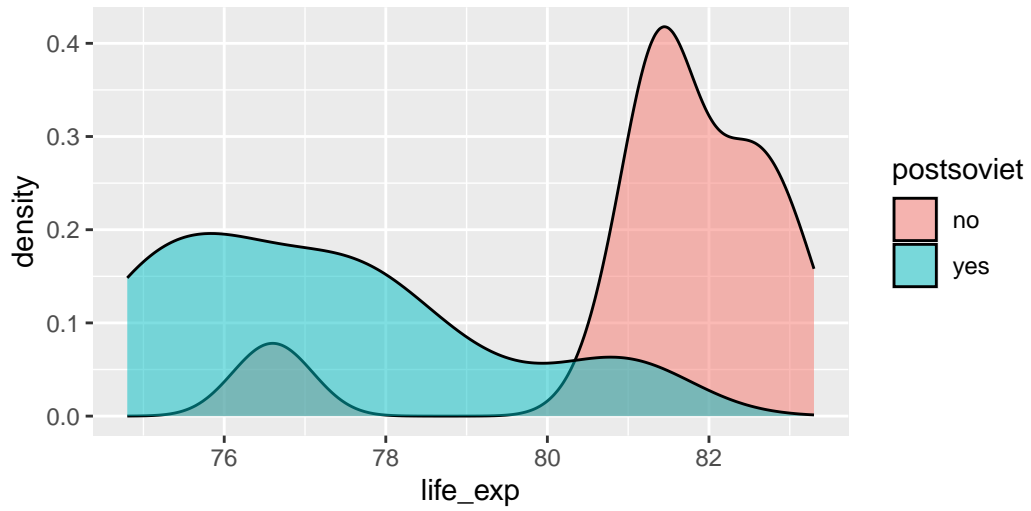
U grafů hustoty je situace o něco komplikovanější, protože rozdělení se mohou překrývat. V grafu níže tak nevidíme dvě západní země s nízkou nadějí na dožití:

```
ggplot(countries,
       aes(x = life_exp, fill = postsoviet)) +
  geom_density()
```



Řešením je zvýšit průhlednost rozdělení, čehož docílíme pomocí argumentu `alpha`. Ten může nabývat hodnot od 0 do 1, kde 0 je naprosto průhledná a 1 je naprosto neprůhledná:

```
ggplot(countries,  
       aes(x = life_exp, fill = postsoviet)) +  
  geom_density(alpha = 0.5)
```



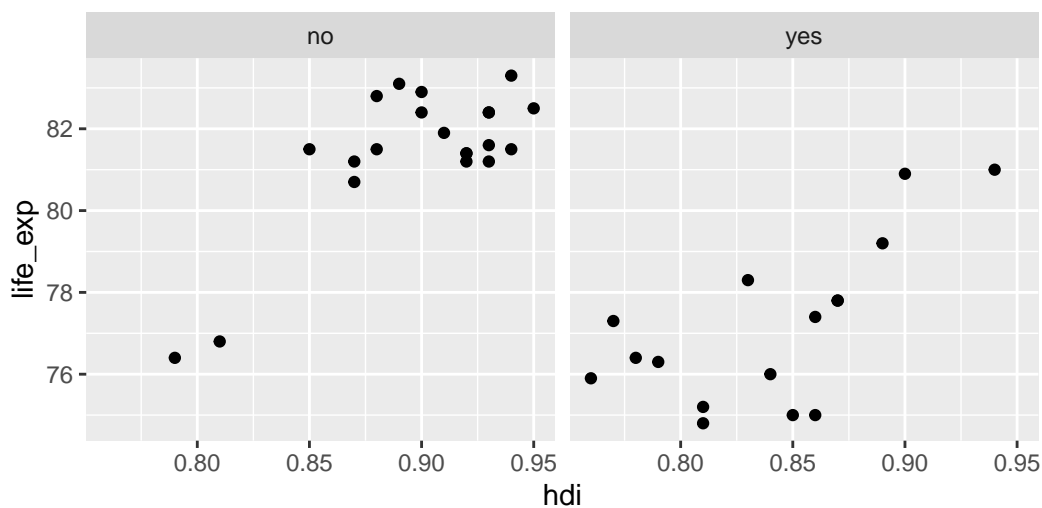
23 Facety

Užitečným nástrojem pro vizualizaci více skupin (nebo více proměnných) jsou facety (*facets*), zvané také *small multiples*. Ty umožňují rozdělit jeden graf do sady menších facet.

23.1 Jednorozměrné facety

Rozdělení grafu na facety je přímočaré, stačí k normálnímu grafu připojit funkci `facet_wrap()`. Uvnitř ní je poté nutné specifikovat kategorickou proměnnou, podle které se budou facety dělit. Tato proměnná je zadána v, na první pohled zvláštním formátu, jelikož ji vždy musí předcházet tilda (~). Proč tomu tak je bude jasnější, až začneme vytvářet facety na základě více proměnných:

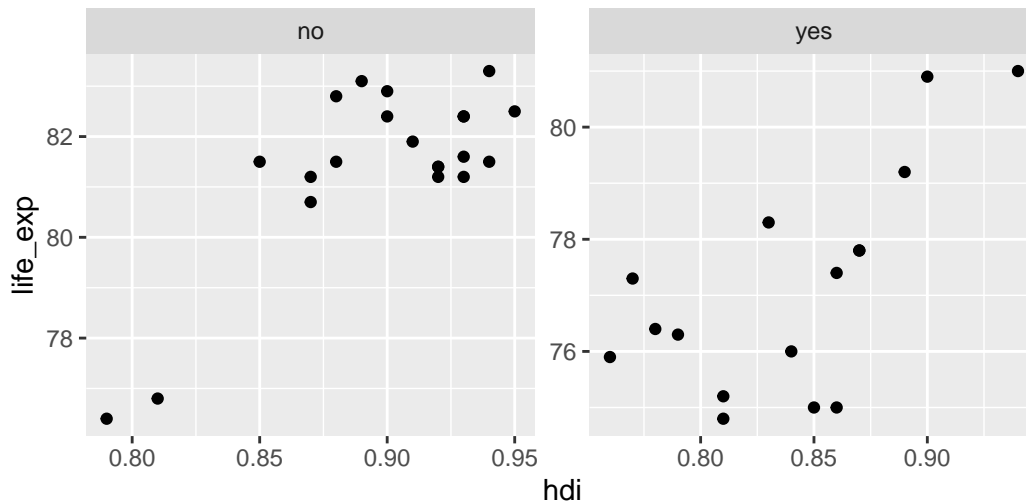
```
ggplot(countries,  
       aes(x = hdi, y = life_exp)) +  
  geom_point() +  
  facet_wrap(~postsoviet)
```



Výsledkem jsou dva menší grafy, jeden pro západní země (nadepsaný `no`) a druhý pro postsovětské (`yes`). Ve výchozím nastavení sdílí všechny dílčí grafy stejné rozpětí os. Změnit

to můžeme pomocí argumentu `scale`. Pokud bychom chtěli, aby každý z facet měla svou vlastní horizontální osu, použijeme `scale = "free_x"`. Analogicky, pro vlastní vertikální osu je možné aplikovat `scale = "free_y"`. Pokud mají všechny dílčí grafy mít své vlastní osy, využijeme `scale = "free"`:

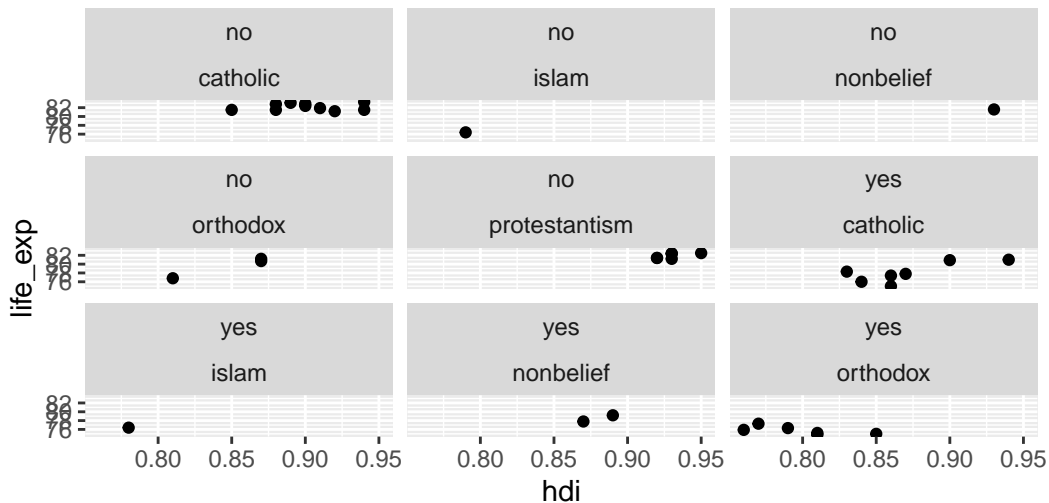
```
ggplot(countries,  
       aes(x = hdi, y = life_exp)) +  
  geom_point() +  
  facet_wrap(~postsoviet,  
            scales = "free")
```



23.2 Vícerozměrné facety

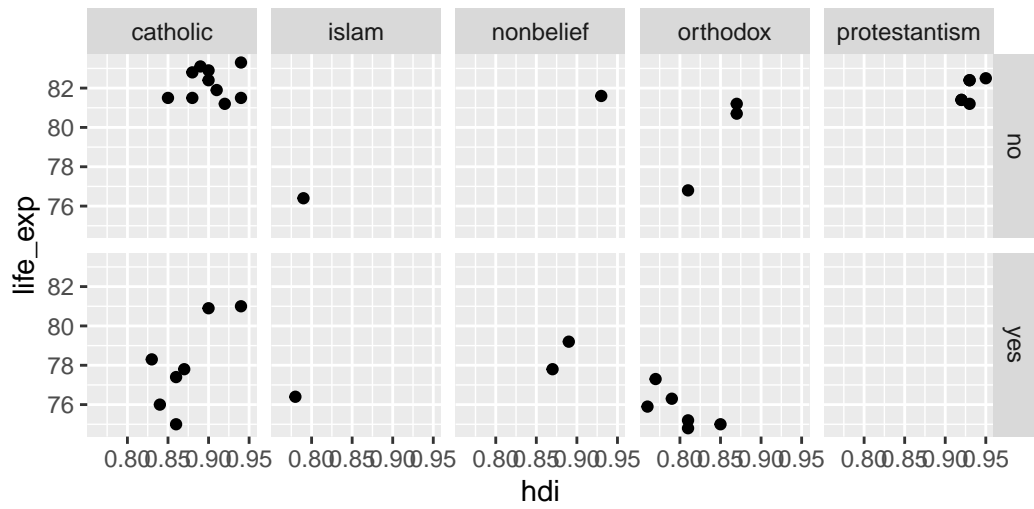
Facety je možné vytvářet na základě více než jedné proměnné, a to hned dvěma způsoby. Tím prvním je využít již známou funkci `facet_wrap()`:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp)) +  
  geom_point() +  
  facet_wrap(~postsoviet + maj_belief)
```



Počet řádků v “tabulce” grafů je možné kontrolovat pomocí argumentu `nrow`, pro počet sloupců poté analogicky `ncol`. Tímto způsobem můžeme vytvořit facetu pro každou kombinaci kategorií obou proměnných. V takto nestrukturovaných facetách může ovšem být obtížné se zorientovat. Lepší variantou proto může být funkce `facet_wrap()`. I ta vytváří facety pro každou kombinaci kategorií, organizuje je ale do tabulky. U této funkce je také nejvíce zřejmé, proč se při vytváření facet využívá tilda (`~`). Jedná se totiž o formuli, pomocí které definujeme vztah mezi proměnnými. V našem případě je výsledná tabulka facet založená na vztahu proměnných `postsoviet` a `maj_belief`:

```
ggplot(countries,
       aes(x = hdi, y = life_exp)) +
  geom_point() +
  facet_grid(postsoviet~maj_belief)
```

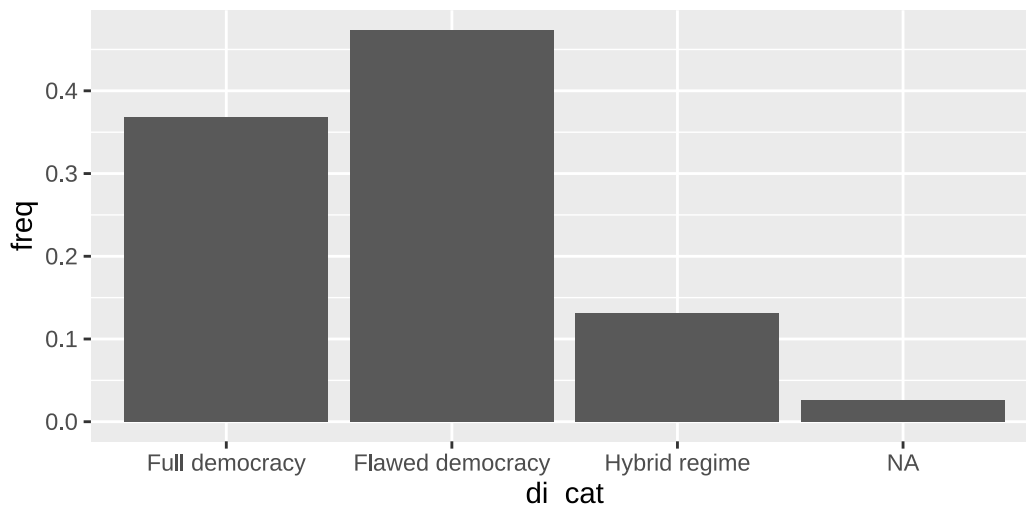
24 Vzhled grafů

Zatímco předchozí kapitoly byly věnované struktuře grafů, v této kapitole se zaměříme na jejich vzhled. Podíváme se detailněji na upravování barev, textu i podoby všech částí grafu. Pomocníky nám kromě `ggplot2` budou také balíček `scales` a `RColorBrewer`. Tyto tři balíčky jsou instalovány společně, `scales` a `RColorBrewer` ovšem nejsou aktivovány pomocí `library(tidyverse)`.

V rámci této kapitoly se budeme opakovaně vracet ke sloupcovému grafu zobrazujícímu četnost kategorií proměnné `di_cat` a pro ulehčení práce si proto připravíme nový dataframe `dem_countries`, obsahující relativní frekvence všech kategorií.

```
dem_countries <- countries %>%
  count(di_cat) %>%
  mutate(freq = n / sum(n),
         di_cat = fct_relevel(di_cat,
                              "Full democracy",
                              "Flawed democracy",
                              "Hybrid regime"))

ggplot(dem_countries,
       aes(x = di_cat, y = freq)) +
  geom_col()
```

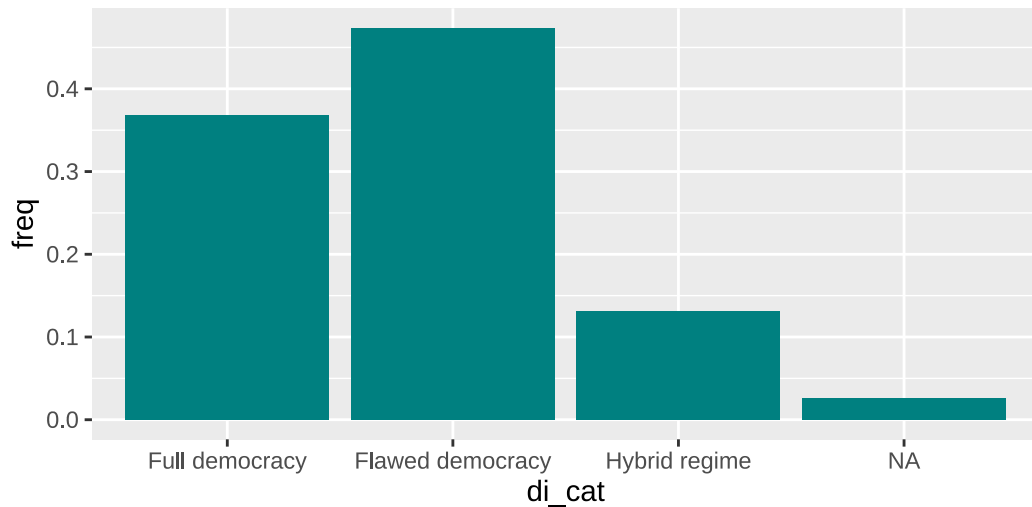


24.1 Barvy

Barvy jsou v R specifikované pomocí hex kódů, tedy kombinace znaku # a šesti dalších číslic a písmen. Například černé barvě přísluší kód #000000, zatímco bílá #ffffff. Kódy barev jsou dostupné na mnoha místech, jakým je třeba stránka <https://www.color-hex.com>. Uživatelé Rstudia mohou také využít balíček `colourpicker`, přidávající šikovný *addin* (rozšíření) Rstudio, pomocí kterého je výběr barev nadmíru snadný.

Nejjednodušší je změnit barev všech sloupců najednou. Předtím, než se do toho pustíme, je ale třeba si ujasnit rozdíl mezi argumenty `color` a `fill`. Většina objektů (*geomů*) pomocí kterých `ggplot2` je složena ze dvou částí: obrysu a výplně. Barvu obrysu kontrolujeme pomocí argumentu `color`, barvu výplně pomocí `fill`. Protože u sloupcových grafů je dominantní výplň sloupců, použijeme pro změnu vzhledu právě argument `fill`, a to přímo uvnitř funkce `geom_col()`:

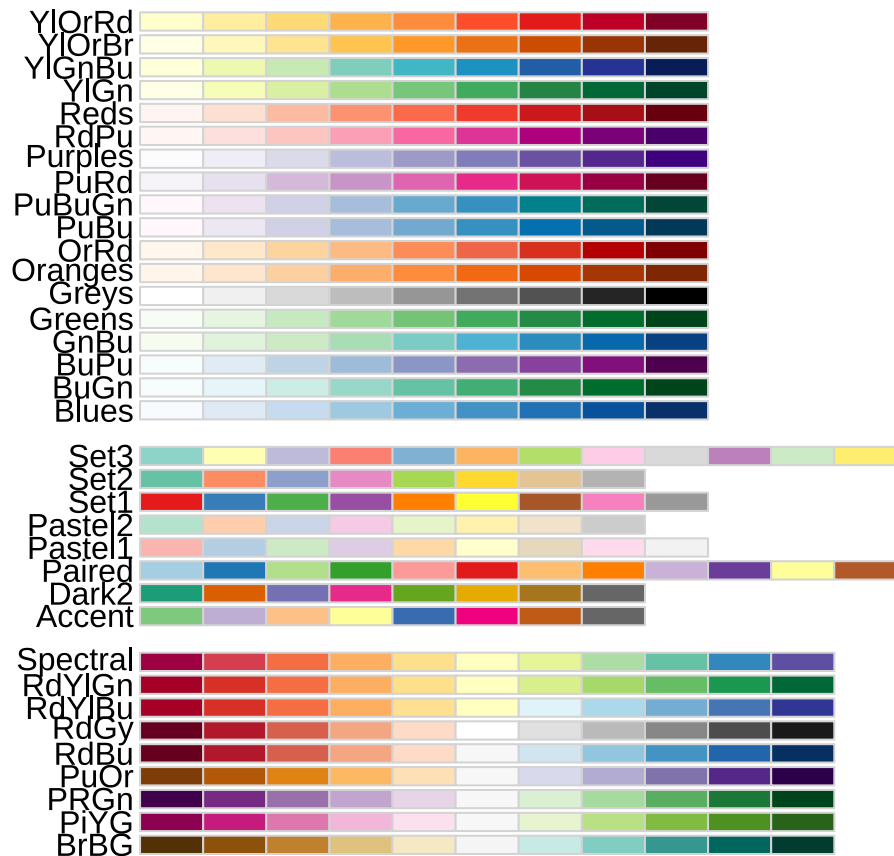
```
ggplot(dem_countries,
       aes(x = di_cat, y = freq)) +
  geom_col(fill = "#008080")
```



Komplexnějším úkonem je aplikace palety barev. Základní nabídku palet, kterou přináší balíček `RColorBrewer`, je možné zobrazit pomocí funkce `display.brewer.all()` (nesmíme ale zapomenout nejdříve balíček aktivovat!):

```
library(RColorBrewer)
```

```
display.brewer.all()
```

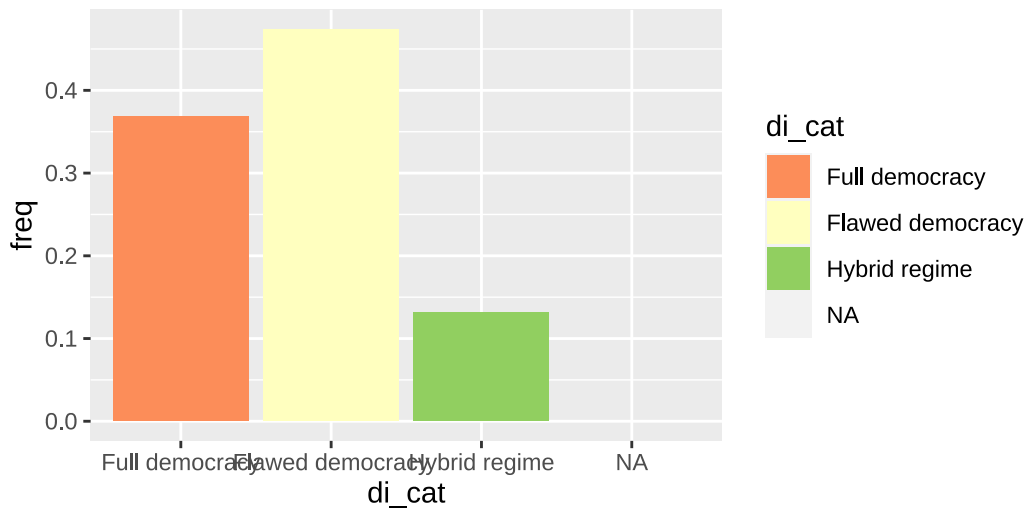


Palety jsou rozděleny do tří skupin. První skupinou jsou takzvané *sequential* palety, tedy palety vhodné pro vizualizaci stupňující se intenzity. Hodí se zejména pro jednapolární proměnné, kde nula reprezentuje absenci, jako například podíl nezaměstnaných. Druhou skupinou jsou *qualitative* palety, vhodné pro nominální proměnné, jako je převažující náboženská skupina v zemi. Poslední skupinou jsou *diverging* palety, určené pro bipolární proměnné. Tato skupina palet je vhodná pokud nízké hodnoty reprezentují opak vysokých hodnot. Příkladem bipolární proměnné je například škála demokracie-autoritářství.

Pro aplikaci palety musíme nejdřív jednotlivé kategorie na ose X namapovat na barvu výplně (`fill`). Poté ke zbytku kódu přidáme funkci `scale_fill_brewer()`. Ta je součástí širší rodiny funkcí, začínajících slovem `scale_`, které kontrolují vzhled jednotlivých dimenzí. Jelikož v tuto chvíli pracujeme s dimenzí `fill`, používáme skupinu funkcí `scale_fill`. A protože je naším cílem využít paletu z `RColorBrewer`, funkce kterou hledáme je právě `scale_fill_brewer()`:

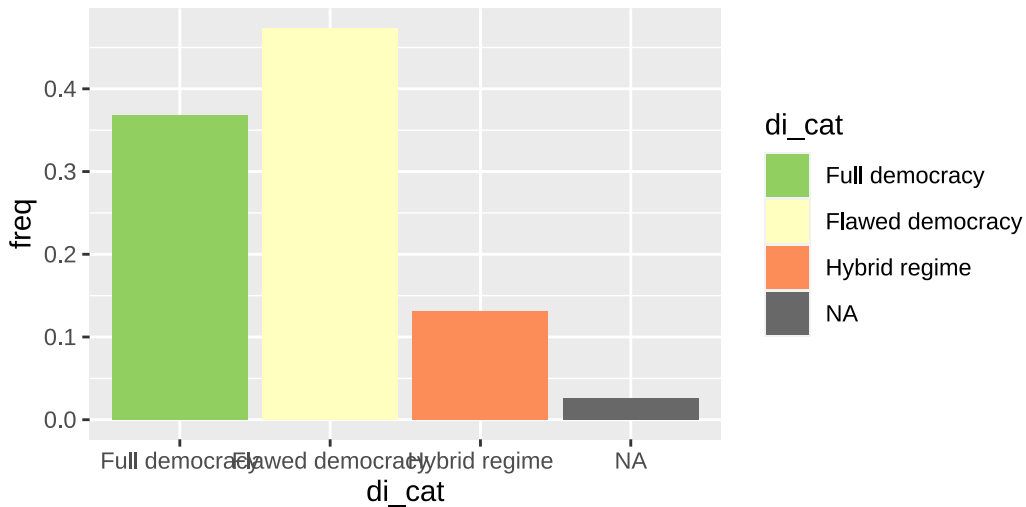
```
ggplot(dem_countries,
  aes(x = di_cat, y = freq, fill = di_cat)) +
  geom_col() +
```

```
scale_fill_brewer(palette = "RdYlGn")
```



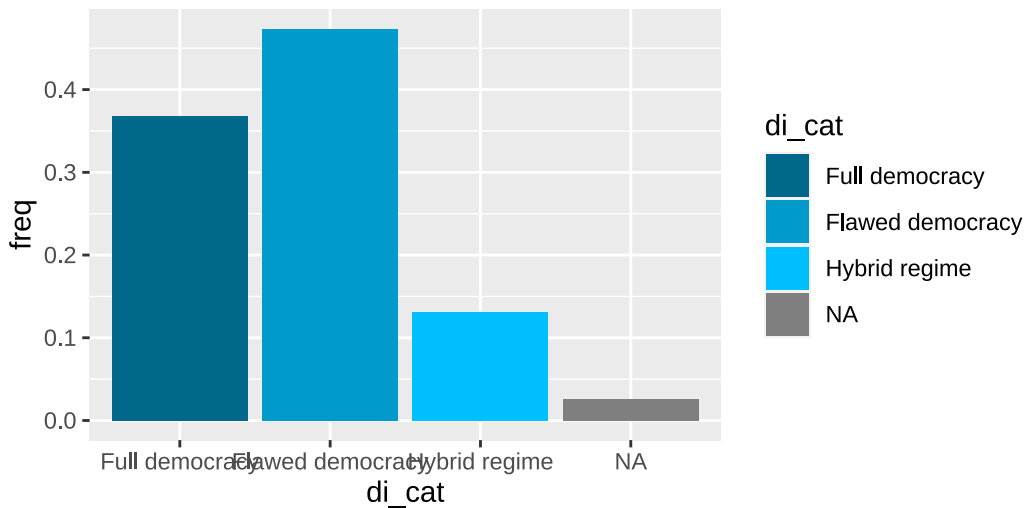
Funkce `scale_color_brewer()` má několik užitečných argumentů. Prvním z nich je `direction`, pomocí které je možné kontrolovat orientaci barev. V našem případě by bylo pravděpodobně vhodnější, aby země s rozvinutější mírou demokracie byly označeny zeleně. Toho docílíme pomocí `direction = -1`. Druhým z užitečných argumentů je `na.value`, pomocí které je možné kontrolovat barvu sloupce reprezentující chybějící hodnoty (NA). V tuto chvíli je barva NA sloupce stejná jako barva pozadí grafu, což není úplně ideální. Použijeme proto tmavší odstín šedé, s hex kódem `#696868`:

```
ggplot(dem_countries,  
       aes(x = di_cat, y = freq, fill = di_cat)) +  
  geom_col() +  
  scale_fill_brewer(palette = "RdYlGn",  
                   direction = -1,  
                   na.value = "#696868")
```



Pokud nám nevyhovuje žádná z předpřipravených palet, je možné barvy jednotlivých kategorií zadat i ručně, k čemuž využijeme funkce `scale_fill_manual()`:

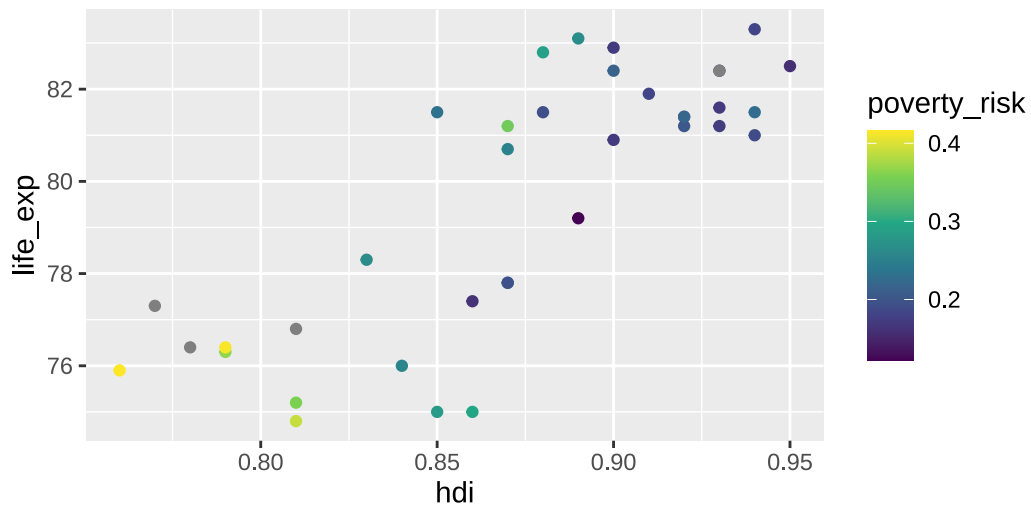
```
ggplot(dem_countries,
       aes(x = di_cat, y = freq, fill = di_cat)) +
  geom_col() +
  scale_fill_manual(values = c("#00688B", "#009ACD", "#00BFFF", "#7D7D7D"))
```



Nakonec je dobré zmínit ještě speciální typ barevných palet, takzvané *continuous* palety. Ty slouží k barevné vizualizaci spojitých proměnných. `ggplot2` nabízí dvě *continuous* palety, `gradient` pro unipolární proměnné a `viridis` pro bipolární. Obě je možné aplikovat pomocí

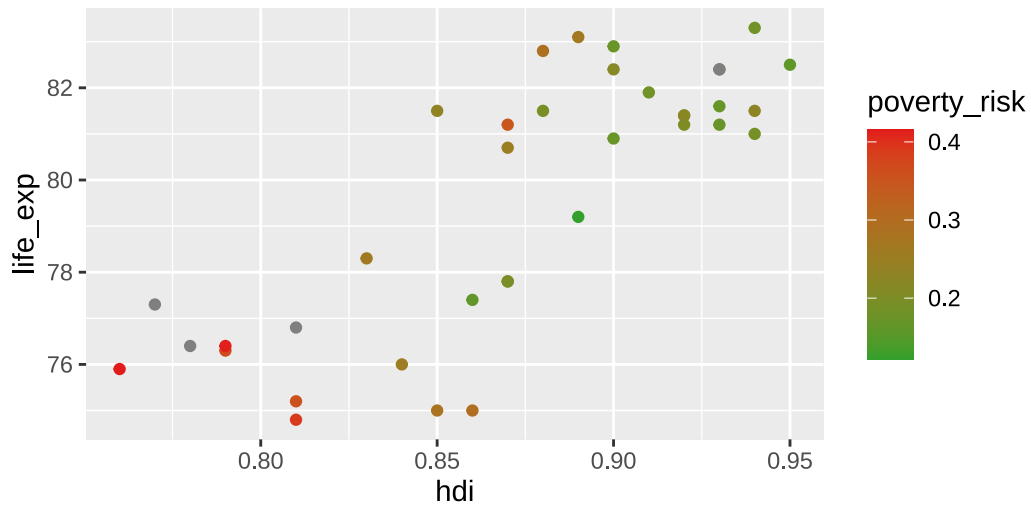
funkcí `scale_color_continuous()` nebo `scale_fill_continuous()` podle toho, zda jde o barvu obrysu nebo barvu výplně:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp, color = poverty_risk)) +  
  geom_point() +  
  scale_color_continuous(type = "viridis")
```



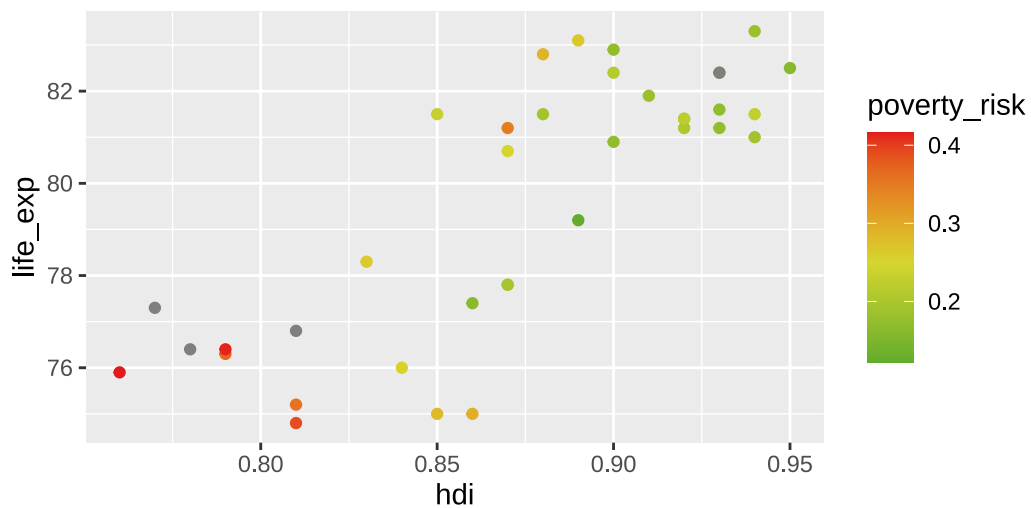
Pokud nám nevyhovuje žádná z palet, je možné zvolit barvy vlastní, a to hned dvěma způsoby. Prvním možností je funkce `scale_color_gradient()` (případně `scale_fill_gradient()`), pomocí které můžeme barvu minima a maxima. Funkce interpoluje barvu zbylých hodnot:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp, color = poverty_risk)) +  
  geom_point() +  
  scale_color_gradient(low = "#33A02C", high = "#E31A1C")
```

Druhou možností je funkce `scale_color_gradient2()` (a analogicky `scale_fill_gradient2()`), pomocí které je možné specifikovat tři barvy: minima, maxima a středu. Poté jen stačí specifikovat střední hodnotu barevné škály:

```
ggplot(countries,
       aes(x = hdi, y = life_exp, color = poverty_risk)) +
  geom_point() +
  scale_color_gradient2(low = "#33A02C", mid = "#D6D62D", high = "#E31A1C",
                       midpoint = 0.25)
```



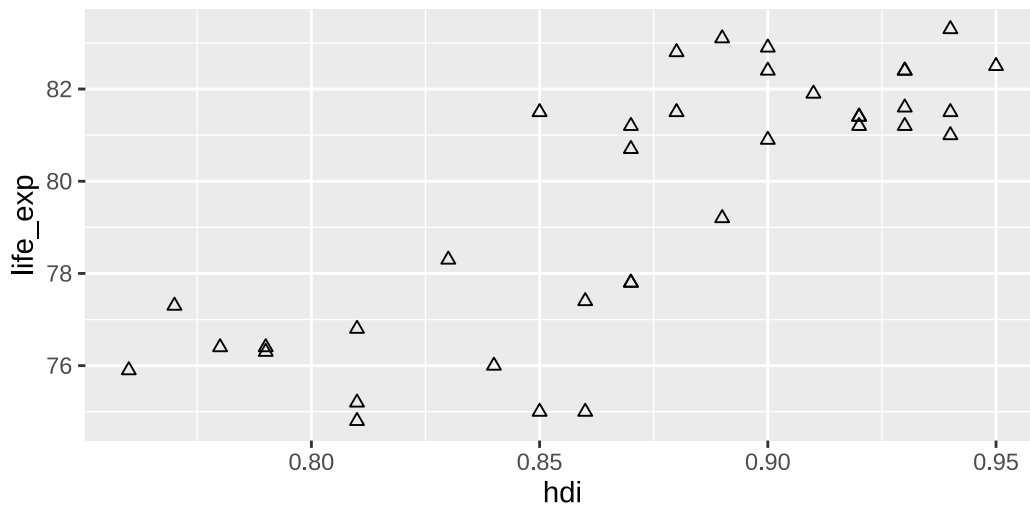
24.2 Tvar

U některých geomů, například `geom_point()`, je možné určit určit jejich tvar a to pomocí argumentu `shape`. R obsahuje 26 základních tvarů, které je aplikovat pomocí jejich číselných kód. Vychozím tvarem je ten s hodnotou 1:

13	14	15	16	17	18	19	20	21	22	23	24	25
⊗	⊠	■	●	▲	◆	●	●	○	□	◇	△	▽
0	1	2	3	4	5	6	7	8	9	10	11	12
□	○	△	+	×	◇	▽	⊠	✱	⊞	⊕	⊗	⊞

Tvar objektů je možné specifikovat pomocí stejných pravidel, jako jejich barvu. Plošně je možné zvolit tvar pomocí argumentu `shape`, v případě škál bychom využili funkce `scale_shape_manual()`:

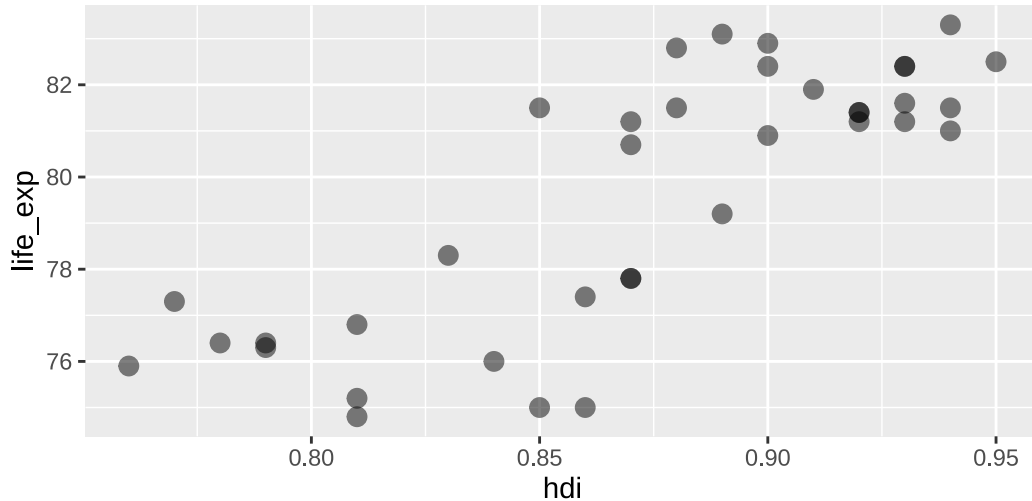
```
ggplot(countries,  
       aes(x = hdi, y = life_exp)) +  
  geom_point(shape = 24)
```



24.3 Velikost a průhlednost

Průhlednost objektu je možné upravovat argumentem `alpha`, se kterým jsme již letmo setkali v předchozí kapitole (Sekce 22.3). `alpha` nabývá hodnot od 0 (zcela průhledná) do 1 (zcela neprůhledná). Argument `size` poté slouží ke kontrole velikosti geomů a může nabývat jakékoli pozitivní hodnoty:

```
ggplot(countries,
       aes(x = hdi, y = life_exp)) +
  geom_point(alpha = 0.5, size = 3)
```

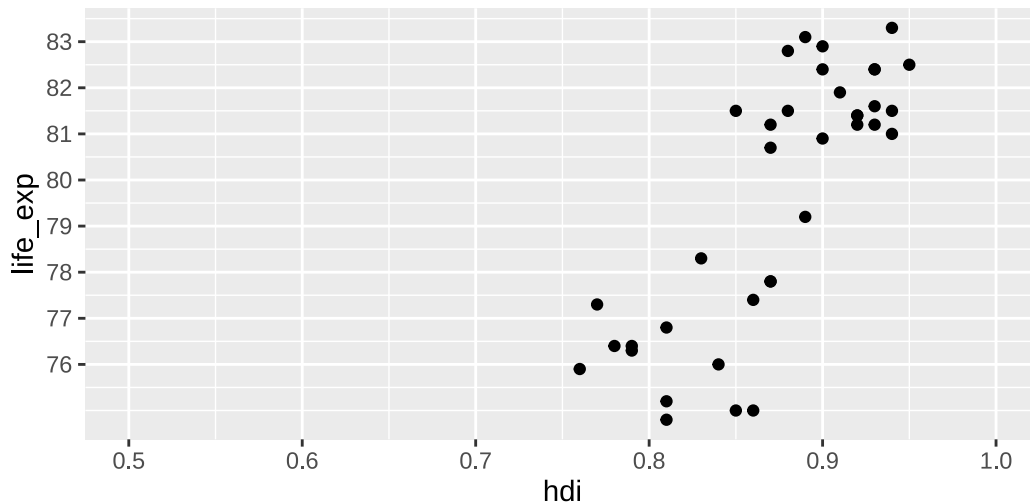


24.4 Formátování os

O formátování vertikální a horizontální osy stará rodina funkcí `scale_x` a `scale_y`. Pokud je na dané ose numerická proměnná, použijeme funkci `scale_x_continuous()` (resp. `scale_y_continuous()`). Pokud jde o proměnnou kategorickou, využijeme funkci `scale_x_discrete()` a `scale_y_discrete()`.

U numerických proměnných jsou dvěma nejužívanějšími argumenty `limits` a `breaks`. Prvním z nich lze určit rozpětí osy, a to vektorem obsahujícím spodní a horní limit. Pokud bychom chtěli omezit rozpětí horizontální osy mezi hodnotami 0.5 a 1, použijeme `limits = c(0.5, 1)`. Pro určení pouze jednoho z limitů nahradíme druhou hodnotu `NA`, např. `c(NA, 1)`. Druhým argumentem, `breaks`, poté upravíme hodnoty, které se na ose ukazují:

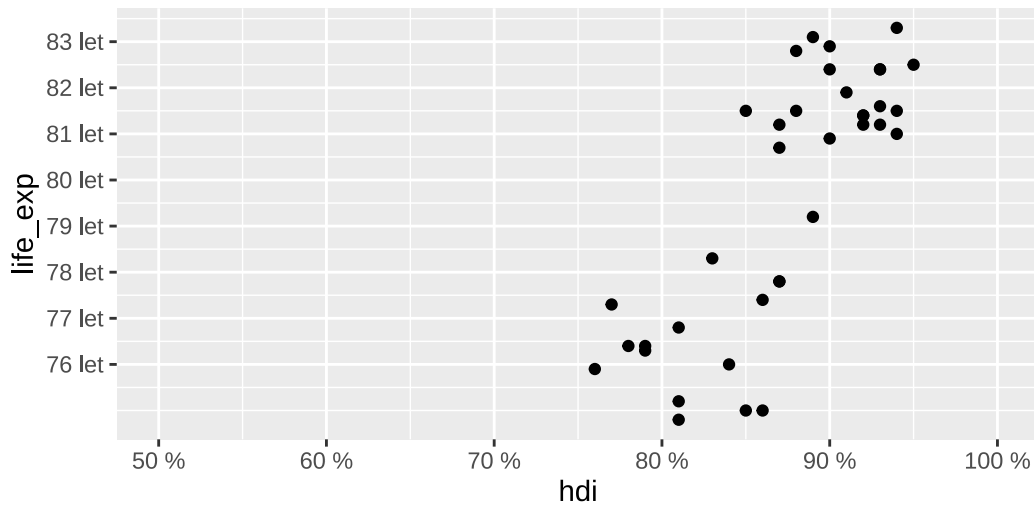
```
ggplot(countries,
       aes(x = hdi, y = life_exp)) +
  geom_point() +
  scale_x_continuous(limits = c(0.5, 1)) +
  scale_y_continuous(breaks = 76:83)
```



Kromě toho, jaké hodnoty se na osách zobrazí, je možné upravovat i jejich formát. K tomu nám pomůže balíček `scales`, v kombinaci s argumentem `labels`. Tento balíček obsahuje sadu funkcí, jako například `number_format()`, `percent_format()` nebo `date_format()`. Funkcí `number_format()` můžeme přidat prefix (argument `prefix()`), sufix (`suffix`), převést proměnnou na jiné jednotky (`scale`) nebo upravit počet desetinných míst `accuracy` a jejich oddělovač (`decimal.mark`). Funkce `percent_format()` funguje obdobně, automaticky ale také převádí desetinná čísla na procenta:

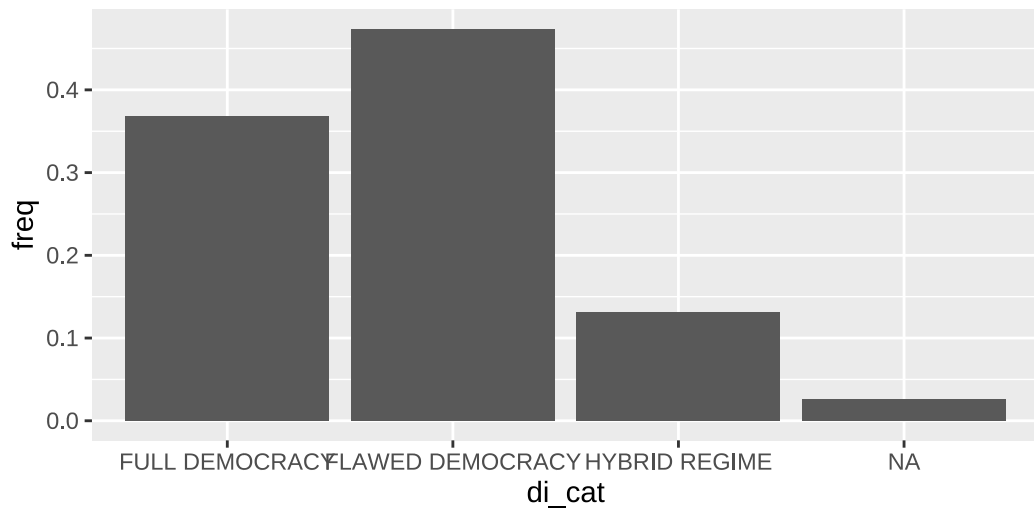
```
library(scales)

ggplot(countries,
       aes(x = hdi, y = life_exp)) +
  geom_point() +
  scale_x_continuous(limits = c(0.5, 1),
                    labels = percent_format(accuracy = 1, suffix = "%")) +
  scale_y_continuous(breaks = 76:83,
                    labels = number_format(suffix = " let"))
```



V argumentu `labels` je možné použít i další funkce. Pro formátování textu je možné využít například funkcí, se kterými jsme se setkali v kapitole věnované stringům (Sekce 18.3):

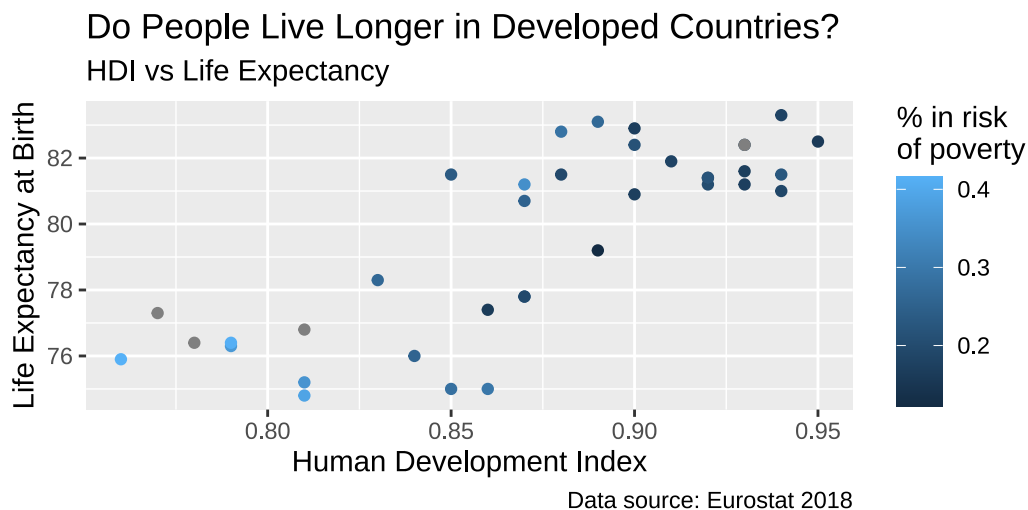
```
ggplot(dem_countries,
  aes(x = di_cat, y = freq)) +
  geom_col() +
  scale_x_discrete(labels = str_to_upper)
```



24.5 Nadpisy, názvy a poznámky

Všechny textové popisy grafů je možné ovládat pomocí funkce `labs()`. Pomocí ní můžeme určit nadpis grafu (`title`), podnadpis (`subtitle`), poznámky (`caption`) a názvy všech použitých dimenzí:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp, color = poverty_risk)) +  
geom_point() +  
labs(title = "Do People Live Longer in Developed Countries?",  
     subtitle = "HDI vs Life Expectancy",  
     caption = "Data source: Eurostat 2018",  
     x = "Human Development Index",  
     y = "Life Expectancy at Birth",  
     color = "% in risk\nof poverty")
```



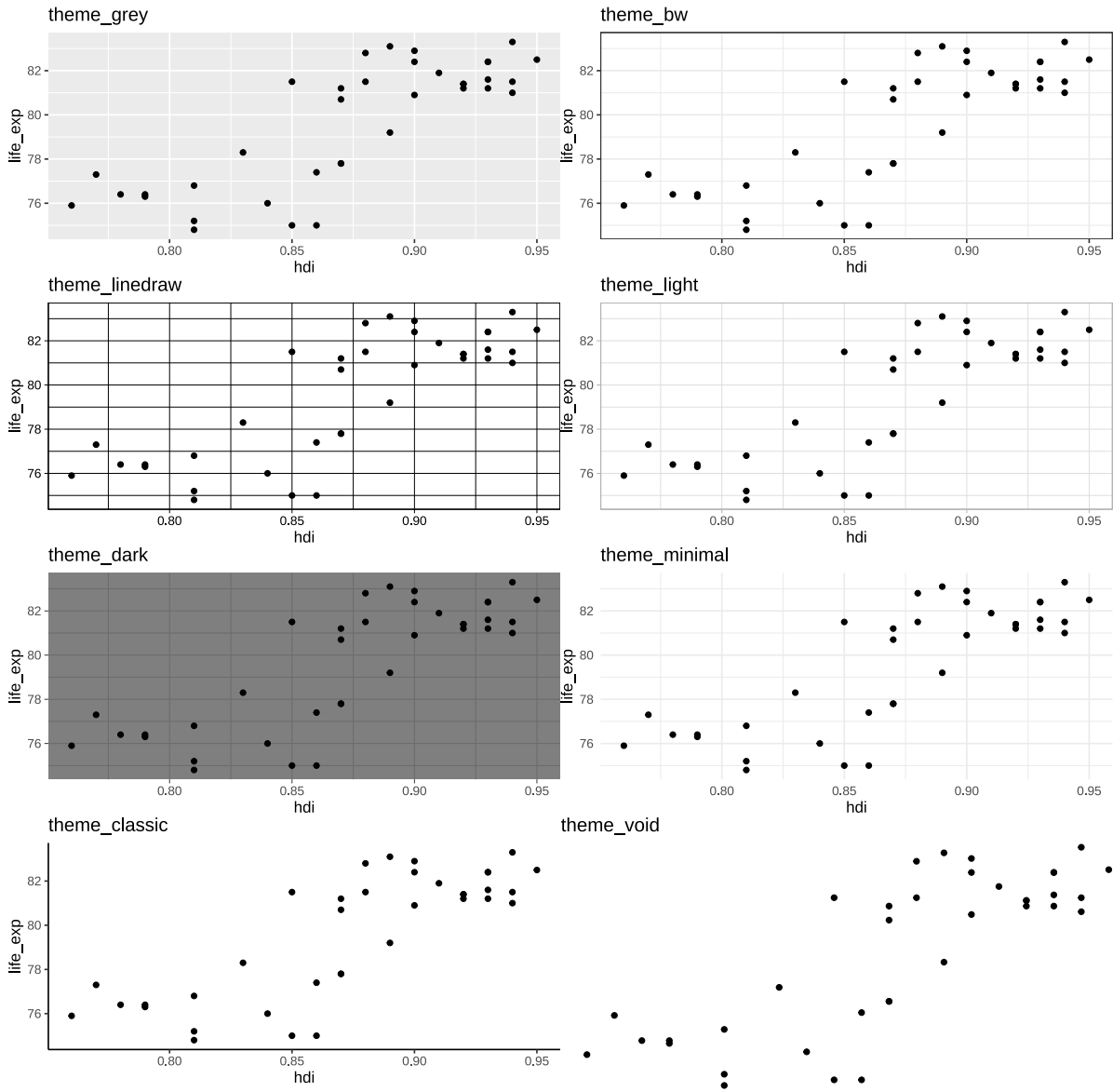
💡 Text na více řádcích

Pokud chceme aby text v grafu byl zalomený na více řádků, použijeme zvláštní znak `\n`, například `"% in risk\nof poverty"`.

24.6 Celková tématika grafu (*themes*)

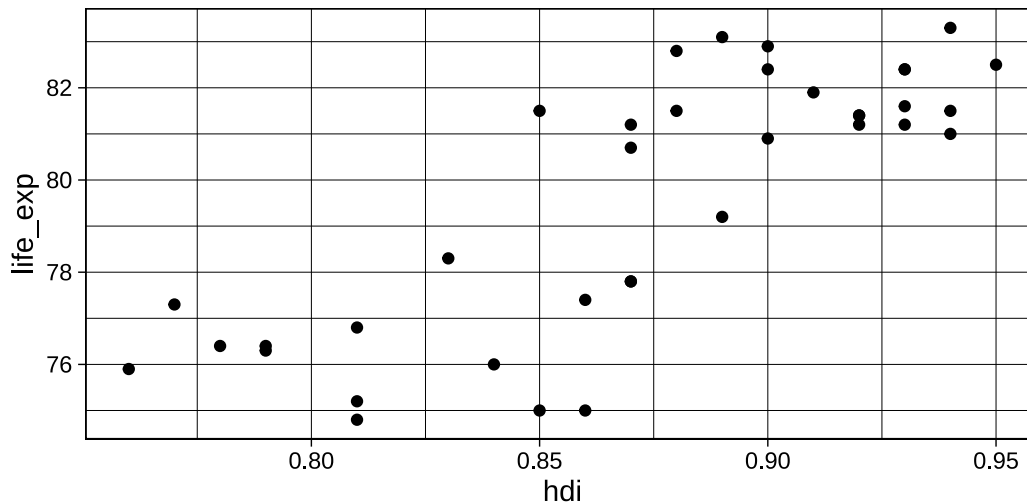
Poslední sekce této kapitoly je věnovaná celkové tématice grafu (anglicky *theme*). Pomocí funkce `theme` je možné ovládat všechny aspekty grafu, které nebyli popsány výše. `ggplot2`

obsahuje sadu předpřipravených tématik, které můžeme aplikovat na každý graf:



Pro aplikaci vybrané tématiky stačí připojit její funkci ke grafu:

```
ggplot(countries,  
       aes(x = hdi, y = life_exp)) +  
  geom_point() +  
  theme_linedraw()
```



Kromě předpřipravených tématik je možné také upravovat vzhled grafu manuálně, pomocí funkce `theme()`. Tato funkce má několik desítek argumentů, které nám umožní kontrolovat i ty nejmenší detaily. My si ukážeme pouze ty nejpoužívanější.

Prvním aspektem, který budeme chtít často kontrolovat, je pozice legendy. Toho docílíme pomocí argumentu `legend.position`. Ten může nabývat buď jedné ze čtyř předpřipravených pozic (`top`, `bottom`, `left` a `right`). Alternativně je možné použít dvojici koordinátů, oba koordináty mohou nabývat hodnoty mezi hodnotami 0 a 1. Dvojice `c(1,1)` umístí legendu pravého horního rohu, `c(0,0)` do levého horního rohu a `c(0.5, 0.5)` přímo na střed.

Dále je možné upravovat font textu, a to pomocí argumentu `text`. Ten přijímá funkci `element_text()`, pomocí které je možné specifikovat font (`family`), velikost (`size`) nebo zda má být text kurzívou/tučně (`face`). Pokud je naším cílem upravit pouze některý text, je možné využít cílené argumenty jako `title` nebo `axis.text.x`

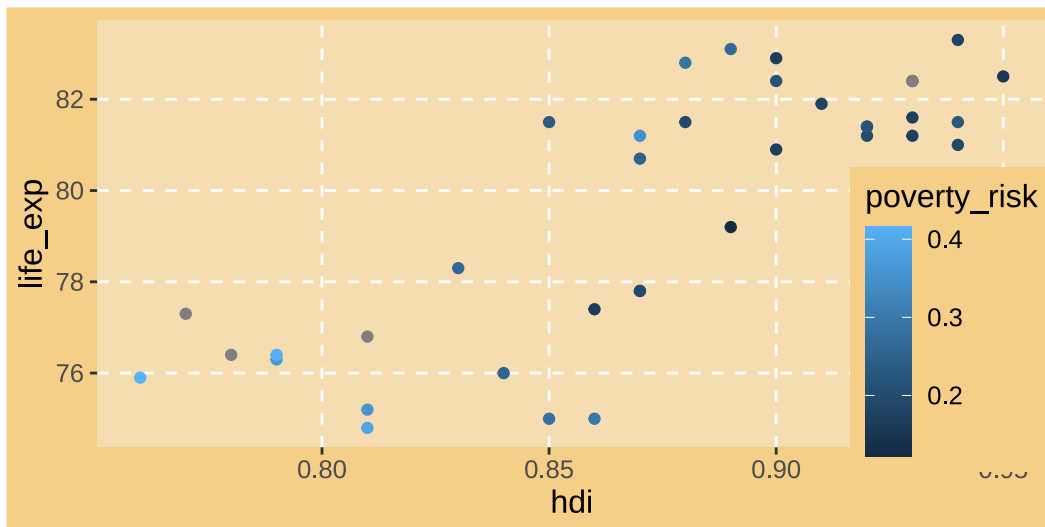
Barvu pozadí grafu je možné ovládat pomocí grafu. Každý graf je rozdělený do dvou částí. *Panel* je vnitřní oblast grafu, ve které se nachází geomy, zatímco *plot* je vnější oblast obsahující popisky a legendu. Vlastnosti obou se dají upravovat nezávisle na sobě pomocí argumentů `panel.background` a `plot.background`. Pokud náš graf obsahuje legendu, můžeme její vzhled upravit obdobně pomocí argumentu `legend.background`.

Návodné čáry grafu kontroluje skupina argumentů `panel.grid`. Čáry se rozlišují na primární (`panel.grid.major`) a sekundární (`panel.grid.minor`). Upravovat také můžeme pouze návodné čáry pro specifickou osu pomocí `panel.grid.major.x` a `panel.grid.major.y` (resp. `panel.grid.minor.x` a `panel.grid.minor.y`). Vzhled čar je možné upravit pomocí funkce `element_line()`. Pro odebrání čáry, nebo jakéhokoliv jiného elementu grafu, je možné využít funkce `element_blank()`.


```

ggplot(countries,
       aes(x = hdi, y = life_exp, color = poverty_risk)) +
  geom_point() +
  theme(legend.position = c(0.9, 0.3),
       text = element_text(family = "Calibri", size = 12),
       panel.background = element_rect(fill = "#F5DDB1"),
       plot.background = element_rect(fill = "#F5CE87"),
       legend.background = element_rect(fill = "#F5CE87"),
       panel.grid.major = element_line(linetype = "dashed"),
       panel.grid.minor = element_blank())

```



💡 Upravování výchozích tématik

Pokud chceme upravit jednu z výchozích tématik, například `theme_linedraw()`, použijeme obě funkce za sebou:

```

ggplot(countries,
       aes(x = hdi, y = life_exp, color = poverty_risk)) +
  geom_point() +
  theme_linedraw() +
  theme(legend.position = "bottom")

```

25 Pokročilé grafy

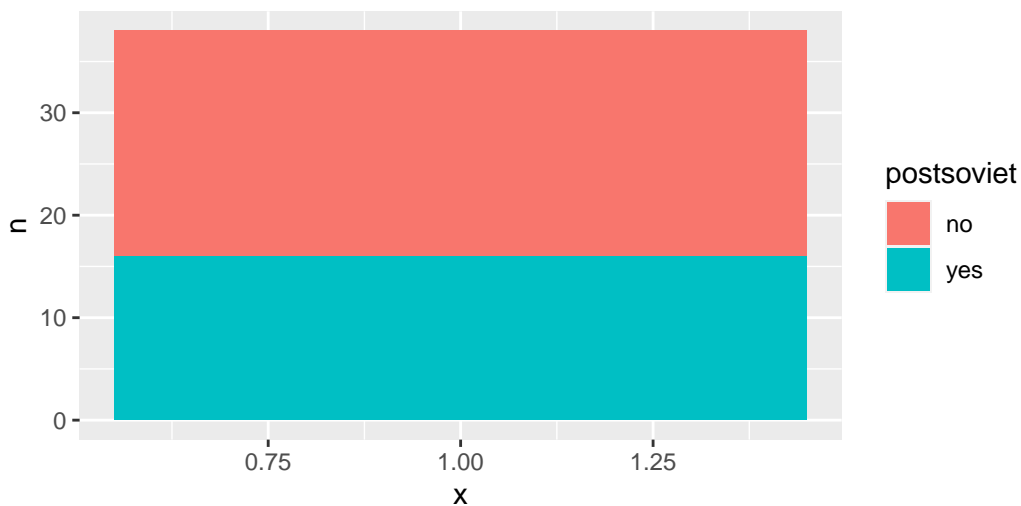
Předchozí kapitoly byly věnovány základům práce s balíčkem `ggplot2`. V této kapitole si ukážeme příklady pokročilejších technik, které při vizualizaci dat můžeme využít.

25.1 Polární koordináty

Většina grafů využívá karteziánské koordináty - objekty v grafu jsou mapované na horizontální a vertikální osu. Čas od času se ovšem vyplatí využít jiný systém. Jedním z nejpoužívanějších jsou polární koordináty.

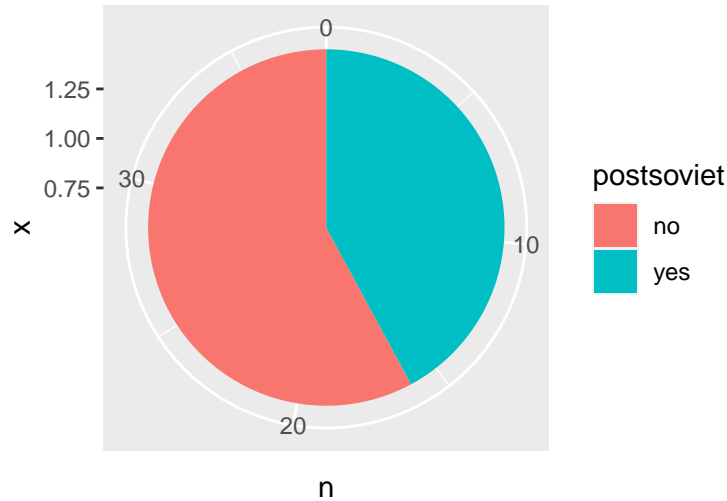
Přestože si to možná neuvědomujeme, každý z nás se již s polárními koordinátami setkal. Slouží k vytváření koláčových grafů, které nejsou ničím jiným, než stočenými skládanými sloupcovými grafy. Začneme vytvořením skládaného sloupcového grafu:

```
countries %>%  
  count(postsoviet) %>%  
  ggplot(aes(x = 1, y = n, fill = postsoviet)) +  
  geom_col(position = "stack")
```



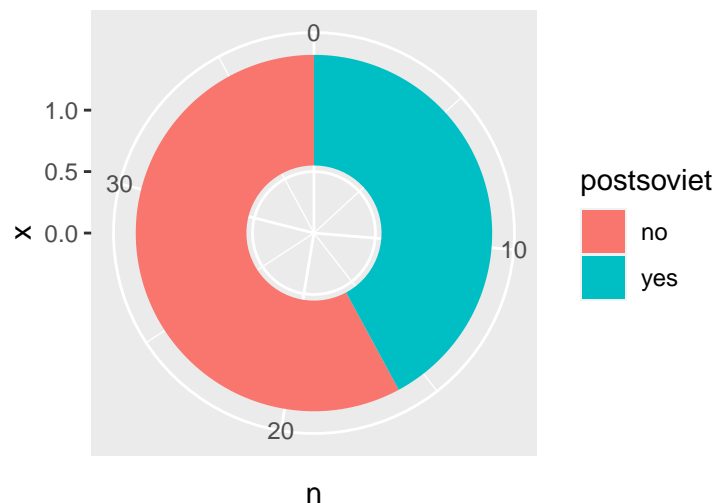
Nyní jen stačí použít funkci `coord_polar()` pro aplikace polárních koordinátů. Argumentem `theta` určíme, kterou z os “obtočíme” kolem středu grafu:

```
countries %>%  
  count(postsoviet) %>%  
  ggplot(aes(x = 1, y = n, fill = postsoviet)) +  
  geom_col(position = "stack") +  
  coord_polar(theta = "y")
```



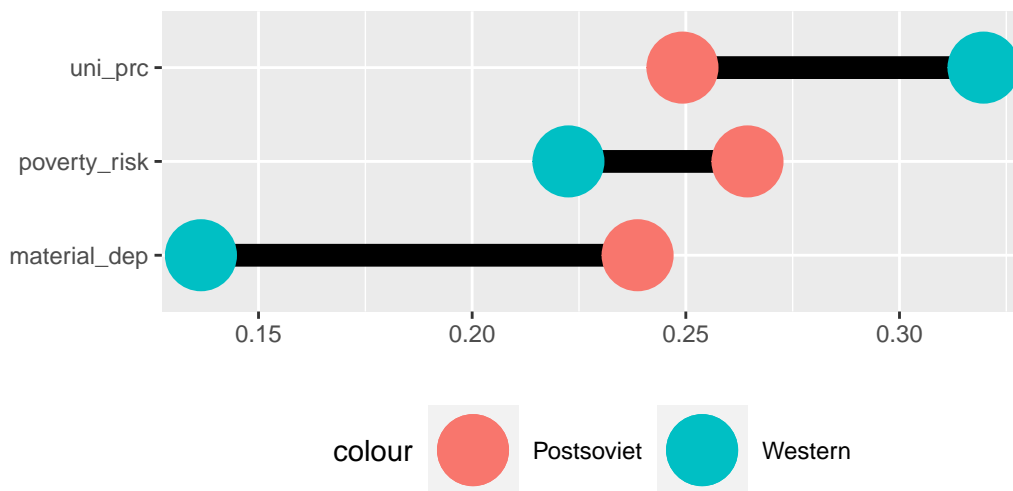
A je to, koláčový graf je hotový! Obdobným způsobem je možné vytvářet i další varianty. Například takzvaný donut chart, tedy koláčový graf s dírou ve středu, vytvoříme jednoduše tak, že necháme prostor mezi začátkem horizontální osy a sloupcem:

```
countries %>%  
  count(postsoviet) %>%  
  ggplot(aes(x = 1, y = n, fill = postsoviet)) +  
  geom_col() +  
  scale_x_continuous(limits = c(0, NA)) +  
  coord_polar(theta = "y")
```



25.2 Skládání geomů

Mnoho komplexních grafů je možné vytvořit kombinací několika vrstev geomů. K tomu nám pomůže fakt, že každá vrstva `ggplot2` grafů může mít svůj vlastní zdroj dat a své vlastní mapování. Následující graf se nazývá barbell chart a využívá se pro srovnání zpravidla dvou skupin napříč několika proměnnými. Přestože tento graf může na první pohled vypadat komplikovaně, jedná se jen o dvě sady bodů spojené úsečkou.



Začněme přípravou dat. Pro každou ze skupin proměnné `postsoviet` spočítáme průměr proměnných `material_dep`, `poverty_risk` a `uni_prc`. Data převedeme do dlouhého formátu a poté zpět do širšího. Spočítáme rozdíl mezi oběma skupinami pro každou z proměnných a nakonec dáme sloupcům lepší názvy:

```

countries %>%
  group_by(postsoviet) %>%
  summarise(across(.cols = c(material_dep, poverty_risk, uni_prc),
                  .fns = mean, na.rm = TRUE)) %>%
  pivot_longer(cols = -postsoviet) %>%
  pivot_wider(names_from = postsoviet, values_from = value) %>%
  mutate(diff = yes-no) %>%
  rename(western = no,
         postsoviet = yes)

```

```
# A tibble: 3 x 4
```

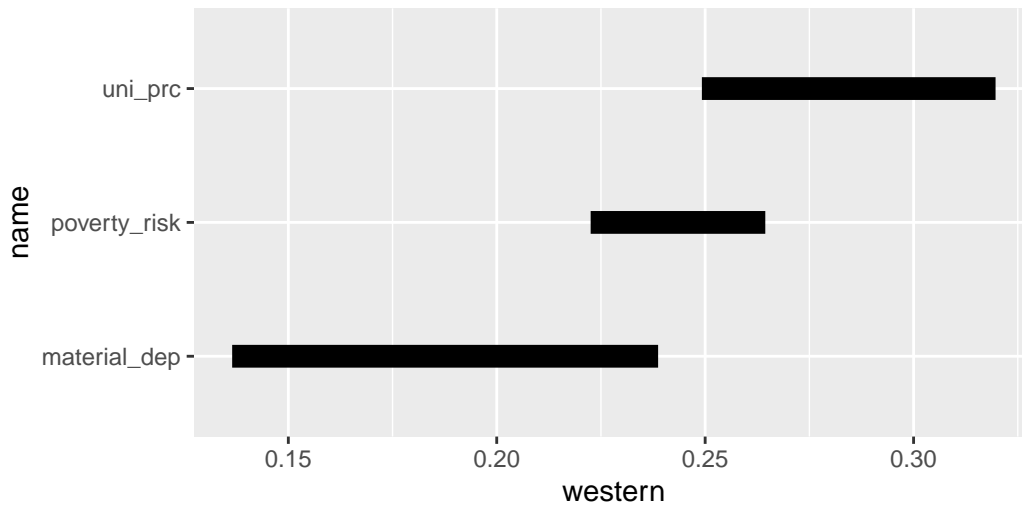
	name	western	postsoviet	diff
	<chr>	<dbl>	<dbl>	<dbl>
1	material_dep	0.137	0.239	0.102
2	poverty_risk	0.223	0.264	0.0419
3	uni_prc	0.320	0.249	-0.0705

Druhým krokem je vytvořením grafu obsahujícím úsečku spojujícím obě skupiny. Pro vytvoření úsečky využijeme funkce `geom_segment()`, které vyžaduje čtyři argumenty: `x`, `xend`, `y` a `yend`.

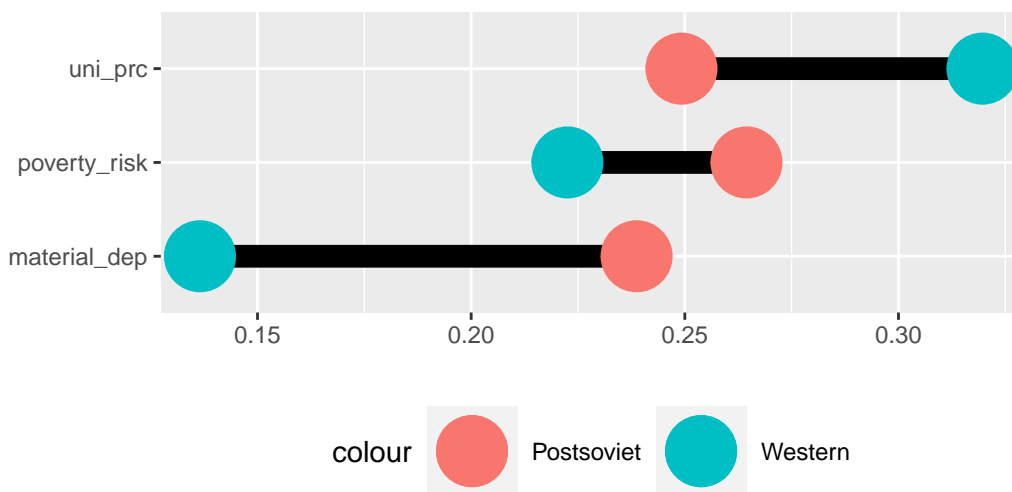
```

countries %>%
  group_by(postsoviet) %>%
  summarise(across(.cols = c(material_dep, poverty_risk, uni_prc),
                  .fns = mean, na.rm = TRUE)) %>%
  pivot_longer(cols = -postsoviet) %>%
  pivot_wider(names_from = postsoviet, values_from = value) %>%
  mutate(diff = yes-no) %>%
  rename(western = no,
         postsoviet = yes) %>%
  ggplot(aes(y = name)) +
  geom_segment(aes(x = western, xend= postsoviet, yend = name), size = 4)

```

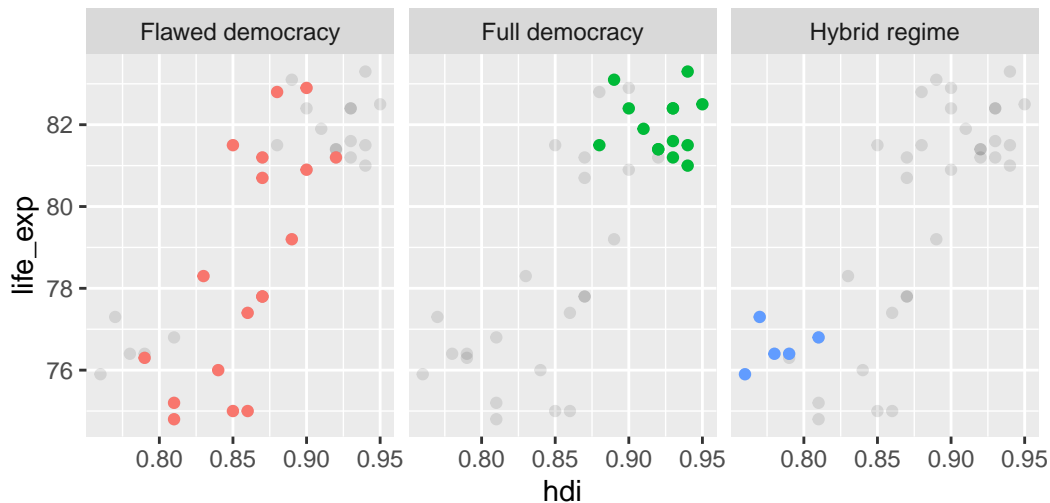


V posledním kroce přidáme body na každou stranu úsečky. Protože průměr každé skupiny je samostatné proměnná, budeme muset použít dvě vrstvy geomů. Dále také budeme muset ručně definovat dimenzi barvy, pro správné vytvoření legendy. Nakonec jen upravíme popisky:



25.3 Více zdrojů dat

Jeden graf může být vytvořen z několika dataframů. To se může hodit například v situacích, kdy chceme vytvořit graf obsahující facety zvýrazňující určitou skupinu dat:

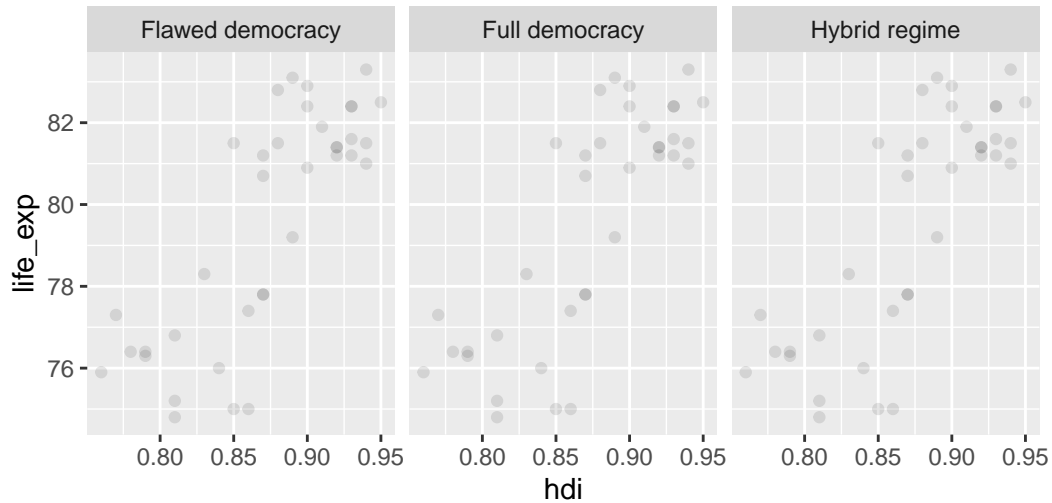


Graf výše využívá faktu, že facety recyklují všechna pozorování, které nepatří do jedné konkrétní facety. Začneme tím, že vytvoříme nový dataframe `countries2`, který je téměř stejný jako `countries`, ale neobsahuje proměnnou `di_cat` (a rovnou se zbavíme chybějících hodnot):

```
countries2 <- countries %>%
  filter(!is.na(di_cat)) %>%
  select(-di_cat)
```

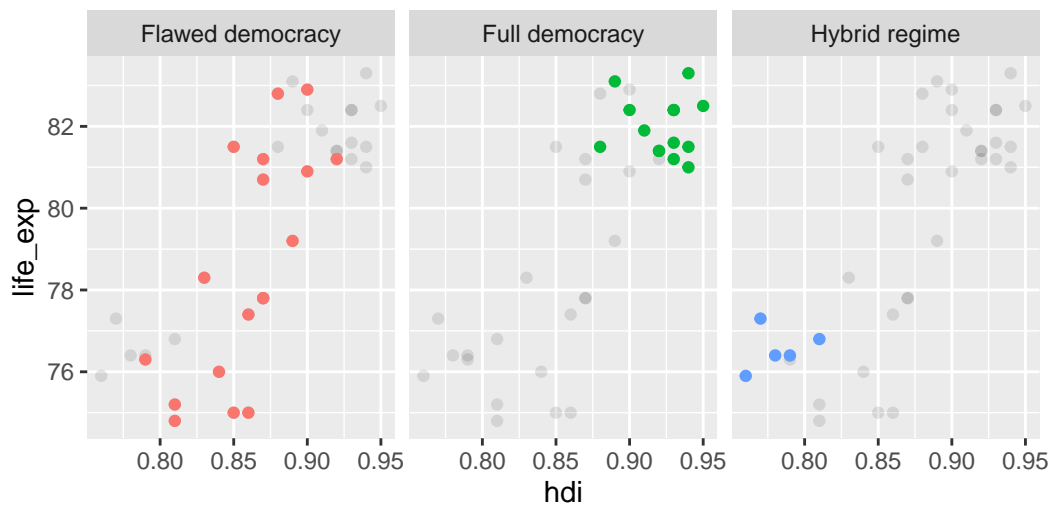
Poté vytvoříme bodový graf pro proměnné `hdi` a `life_exp` rozdělený do facet podle proměnné `di_cat`. Zdroj dat ale nespecifikujeme uvnitř funkce `ggplot()`, ale až ve funkci `geom_point()`. Jako zdroj dat použijeme `countries2`. Protože tento dataframe neobsahuje facetovou proměnnou, budou všechny body zobrazeny ve všech facetech:

```
countries %>%
  filter(!is.na(di_cat)) %>%
  ggplot(aes(x = hdi, y = life_exp)) +
  geom_point(data = countries2, alpha = 0.1) +
  facet_wrap(~di_cat)
```



Nyní přidáme druhou vrstvu bodů, tentokrát založenou na dataframě `countries`. Tento dataframe již facetovou proměnnou obsahuje, takže body budou zobrazeny jen pro relevantní facetu. Také rovnou skryjeme nepotřebnou legendu:

```
countries %>%
  filter(!is.na(di_cat)) %>%
  ggplot(aes(x = hdi, y = life_exp)) +
  geom_point(data = countries2, alpha = 0.1) +
  facet_wrap(~di_cat) +
  geom_point(aes(color = di_cat), show.legend = FALSE)
```

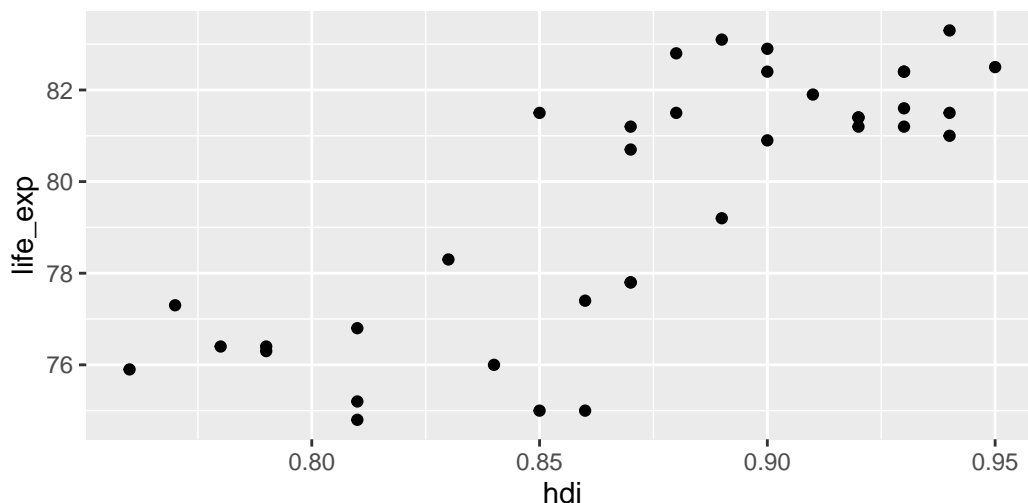


26 Export grafů

Poté, co jsme si vyiplali náš graf do nejmenšího detailu ho zbývá pouze vyexportovat z R pro další užití. V této kapitole si ukážeme, jak naše grafy exportovat ve správných rozměrech a formátu. Jako příklad nám poslouží následující graf, zobrazující vztah mezi indexem lidského rozvoje a nadějí na dožití:

```
hdi_life_plot <- ggplot(countries,  
                        aes(x = hdi, y = life_exp)) +  
                        geom_point()
```

```
hdi_life_plot
```



26.1 Export pomocí ggsave()

Grafy vytvořené pomocí balíčku `ggplot2` je možné uložit pomocí funkce `ggsave()`. Hlavními argumenty této funkce jsou `plot` a `filename`. Pomocí prvního z argumentů určíme, jaký graf chceme exportovat, pomocí druhého název souboru (včetně koncovky). Pokud bychom chtěli uložit graf do jiné složky, než je náš pracovní adresář, využijeme argument `path`. Dále je možné určit rozměry grafu, pomocí argumentů `width` a `height`. Jednotky, ve kterých jsou

rozměry grafu definovány, upravujeme pomocí argumentu `unit`. Na výběr máme centimetry ("cm"), milimetry ("mm"), palce ("in") a pixely ("px"). Formát exportovaného obrázku je možné nastavit pomocí `device`, na výběr máme jak formáty rasterové grafiky (např. `png` a `jpeg`), tak vektorové (`svg` nebo `wmf`). V případě, že exportujeme graf jako rasterový obrázek, je nutné ještě zvolit vhodné rozlišení, čehož docílíme pomocí argumentu `dpi`. "Dots per Inch" je (dnes možná lehce archaická) míra, udávající počet kapek inkoust na jeden palec při tisku obrázku. Pro tisk na papír rozměru A4 se zpravidla využívá `dpi = 300`, pro obrazovky s vysokým rozlišením (retina) poté `dpi = 600`.

Následující příkaz uloží graf `hdi_life_plot` jako soubor `hdi_life_plot.png` do složky `plots`. Graf bude exportován v `png` formátu, s rozměry 14x10 centimetrů. Protože plánujeme graf vložit do textového dokumentu, nastavíme rozlišení na 300 DPI.

```
ggsave(plot = hdi_life_plot,  
        filename = "hdi_life_plot.png",  
        path = "plots",  
        device = "png",  
        units = "cm",  
        width = 14,  
        height = 10,  
        dpi = 300)
```

26.2 Rasterová versus vektorová grafika

Jedním z důležitých rozhodnutí při exportu grafů je volba formátu, ve kterém budou uloženy. Obecně máme dvě volby: rasterové a vektorové obrázky.

Rasterové obrázky jsou složeny z velkého množství malých čtverečků, které dohromady skládají celkový obraz. Jedná se o typ obrázku, který používá například fotoaparát vašeho mobilního telefonu. Výhodou rasterové grafiky je schopnost uchovávat komplexní obrázky v malých souborech. Naopak nevýhodou je, že rozlišení i rozměry rasterových obrázku jsou pevně dané. Pokud bychom rasterový obrázek příliš přiblížili, dojde ke ztrátě detailu. Změna rozměrů zase obrázek může zdeformovat.

Vektorové obrázky jsou souborem matematických funkcí, které vykreslují celkový obraz. S vektorou grafikou se setkáte nejčastěji v profesionálním grafickém softwaru, ať už v kontextu grafického designu nebo tisku. Výhodou vektorové grafiky je, že možné dynamicky upravovat rozměry grafu. Obrázky je také možné upravovat i po jejich exportu a budou také vždy perfektně ostré, protože jejich rozlišení je vypočítáváno dynamicky. Nevýhodou je relativně velké množství úložného prostoru, které vyžadují.

V praxi se nejčastěji setkáte s rasterovými obrázky, pokud budete vytvářet grafy vlastní potřeby (např. do školní práce nebo malého reportu). Naopak pokud spolupracujete s grafikem,

jehož prací je vyvíjet vaše výstupy k dokonalosti, exportem grafů do vektorové grafiky dotyčným usnadníte mnoho práce.

Část V
Pokročilé R

27 Vlastní funkce

R nabízí širokou nabídku funkcí pro analýzu dat. Tuto nabídku je možné dále rozšířit balíčky jako `tidyverse`. Čas od času se ale dostaneme do situace, kdy nám žádná z předpřipravených funkcí nestačí. Naštěstí pro nás nám R umožňuje jednoduše vytvářet funkce vlastní. Obecná poučka říká, že každý kus kódu, který se v našem skriptu opakuje víc než dvakrát, by měl být nahrazen funkcí. Tím si nejen ušetříme čas při analýze, ale také snížíme šanci, že se někde v kódu upíšeme a uděláme chybu.

Funkce v R jsou objekty a je tedy možné je vytvářet stejným způsobem, jakým bychom vytvořili například `dataframe`. Vlastní funkci vytvoříme pomocí funkce `function()`, následované složenými závorkami. Uvnitř jednoduchých závorek můžeme definovat jednotlivé argumenty, uvnitř složených závorek poté definujeme funkci samotnou. Obecně vypadá definice nové funkce takto:

```
nazev_funkce <- function(argument1, argument2, ...) {  
  # Definice funkce  
}
```

Vytváření funkcí si ukážeme na několika případech.

27.1 Počet chybějících hodnot v proměnné

Jednou z častých operací, kterou v rámci analýzy budeme provádět, je počítat množství chybějících hodnot v proměnné. Poněkud překvapivě, R neobsahuje funkci, která by pro nás chybějící hodnoty spočítala. Musíme proto využít kombinace funkcí `is.na()` a `sum()`. Například pro spočítání chybějících hodnot u proměnné `gdp`:

```
sum(is.na(countries$gdp))
```

```
[1] 3
```

Pro pohodlnost si vytvoříme vlastní funkci jménem `count_na()`. Pro začátek bude mít tato funkce jeden argument, a to proměnnou, pro které chceme počet chybějících hodnot spočítat:

```

count_na <- function(x) {
  na_count <- sum(is.na(x))

  return(na_count)
}

```

Útroby této funkce vypadají podobně jako předchozí kód, jen název proměnné je nahrazen generickým argumentem `x`. Výsledek je uložený do objektu `na_count` (existujícím pouze uvnitř této funkce). Spočítanou hodnotu poté exportujeme z naší funkce pomocí `return()`. Takto vytvořenou funkci můžeme využívat tak, jak jsme zvyklí:

```

countries %>%
  summarise(across(.cols = everything(),
                  .fns = count_na)) %>%
  pivot_longer(cols = everything(),
              values_to = "missings_count")

```

```

# A tibble: 17 x 2
  name          missings_count
  <chr>          <int>
1 country         0
2 code            0
3 gdp             3
4 population      1
5 area            0
6 eu_member       0
7 postsoviet      0
8 life_exp        1
9 uni_prc         3
10 poverty_risk   5
11 material_dep   5
12 hdi            0
13 foundation_date 0
14 maj_belief     0
15 dem_index      1
16 di_cat         1
17 hd_title_name  0

```

Naše funkce spočítá absolutní frekvenci chybějících hodnot v proměnné. Co kdybychom ale chtěli relativní frekvenci, tedy podíl chybějících hodnot z celkového množství pozorování? Toho docílíme tak, že naši funkci rozšíříme o další argument, `relative`. Pokud bude hodnota tohoto argumentu `TRUE`, bude počet chybějících hodnot vydělen počtem všech hodnot v proměnné.

```

count_na <- function(x, relative = FALSE) {
  na_count <- sum(is.na(x))

  if(relative){
    na_count <- na_count / length(x)
  }

  return(na_count)
}

```

Všimněme si, že oproti předchozí verzi, jsme v naší funkci udělali několik změn. Zaprvé jsme přidali argument `relative`, jehož výchozí hodnotu jsme nastavili na `FALSE` (ve výchozím nastavení tedy funkce počítá absolutní počet chybějících hodnot). Dále jsme přidali blok začínající funkcí `if()`. Tato funkce zkontroluje, jestli je hodnota argumentu `relative` rovná `TRUE` a pokud ano, vydělí počet chybějících hodnot celkovou délkou proměnné `x`. Funkci používáme tak, jak jsme zvyklí:

```

countries %>%
  summarise(across(.cols = everything(),
                  .fns = count_na, relative = TRUE)) %>%
  pivot_longer(cols = everything(),
              values_to = "missings_count")

```

```

# A tibble: 17 x 2
  name                missings_count
  <chr>                <dbl>
1 country              0
2 code                 0
3 gdp                  0.0789
4 population           0.0263
5 area                 0
6 eu_member            0
7 postsoviet           0
8 life_exp             0.0263
9 uni_prc              0.0789
10 poverty_risk        0.132
11 material_dep         0.132
12 hdi                  0
13 foundation_date     0
14 maj_belief           0
15 dem_index            0.0263
16 di_cat              0.0263

```

27.2 Graf pro likertovské položky

Komplexnějším příkladem vlastní funkce je vytvoření grafu pro baterii likertovských položek. Přestože vytvoření takového grafu je pomocí `ggplot2` možné, jde o poměrně zdlouhavý proces. Jako příklad si můžeme ukázat vizualizaci položek týkajících se postojů veřejnosti o válce na Ukrajině. Data pochází z dotazníkového šetření Centra pro průzkum veřejného mínění z března 2022. Data jsou rozdělena do dvou dataframů. První z nich, `ukraine`, obsahuje odpovědi respondentů a druhý, `ukraine_labels`, obsahuje zjednodušené znění jednotlivých položek.

```
ukraine <- read_rds("data-raw/ukraine.rds")
ukraine_labels <- read_rds("data-raw/ukraine_labels.rds")
```

Pro vytvoření grafu popisující postoje občanů k různým formám zapojení zemí do války vypadá následovně:

```
library(RColorBrewer)
library(scales)

likert_palette <- c("grey70", brewer.pal(4, "RdYlGn"))

ukraine %>%
  select(starts_with("PL_5")) %>%
  pivot_longer(cols = everything(),
               names_to = "item",
               values_to = "response") %>%
  left_join(ukraine_labels, by = "item") %>%
  count(label, response) %>%
  filter(!is.na(response)) %>%
  group_by(label) %>%
  mutate(freq = n / sum(n),
         freq_label = percent(freq, accuracy = 1),
         positive = sum(freq[response %in% c("rozhodně by mělo", "spíše by mělo") ])) %>%
  ungroup() %>%
  mutate(label = fct_reorder(label, positive),
         response = fct_rev(response)) %>%
  ggplot(aes(x = freq, y = label, label = freq_label, fill = response)) +
  geom_col() +
  geom_text(position = position_stack(vjust = 0.5),
           color = "white",
```

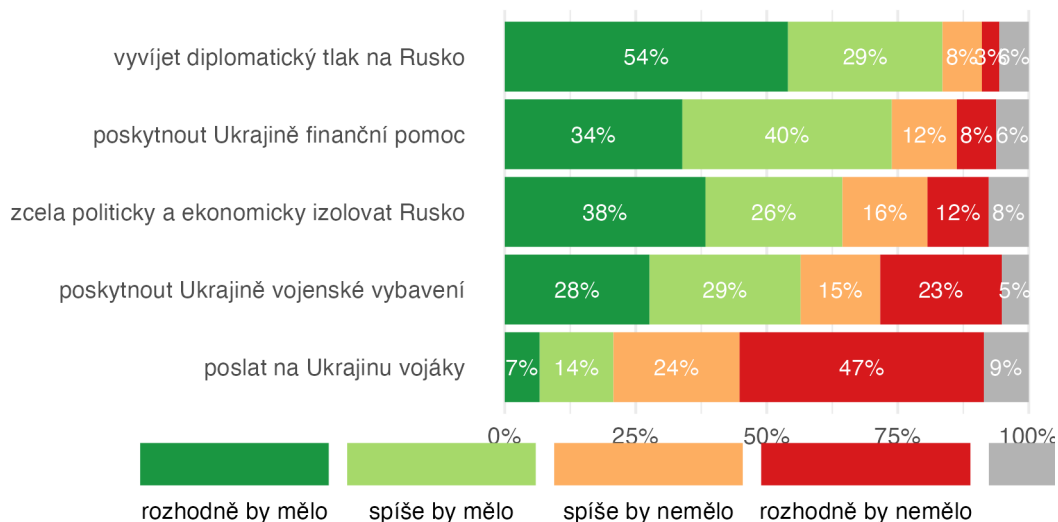


```

    size = 3) +
scale_x_continuous(labels = percent_format()) +
scale_fill_manual(values = likert_palette) +
labs(x = element_blank(),
     y = element_blank(),
     fill = element_blank(),
     title = "Jaké kroky by podle vás mělo podniknout mezinárodní společenství tváří v t
theme_minimal() +
theme(legend.position = c(0.3, -0.17),
     panel.grid.major.y = element_blank(),
     plot.title.position = "plot",
     plot.margin = unit(c(0,0,3.5,0), 'lines')) +
guides(fill = guide_legend(label.position = "bottom",
                          keywidth = 5,
                          reverse = TRUE,
                          direction = "horizontal"))

```

Jaké kroky by podle vás mělo podniknout mezinárodní společenství válce na Ukrajině? Mělo by...



Uf... asi si dokážeme představit, že použít takto monstrózní kus kódu opakovaně je nejen otravné, ale i recept na to něco zkazit. Naštěstí pro nás, většina kódu zůstane při každém použití stejná. Jediné, co se bude měnit jsou 1) použitá data, 2) barevná paleta, 3) odpovědi, podle kterých jsou položky seřazeny, 4) název a 5) dataframe obsahující znění položek. Vytvořme si funkci `plot_likert`, která tento graf udělá za nás:

```

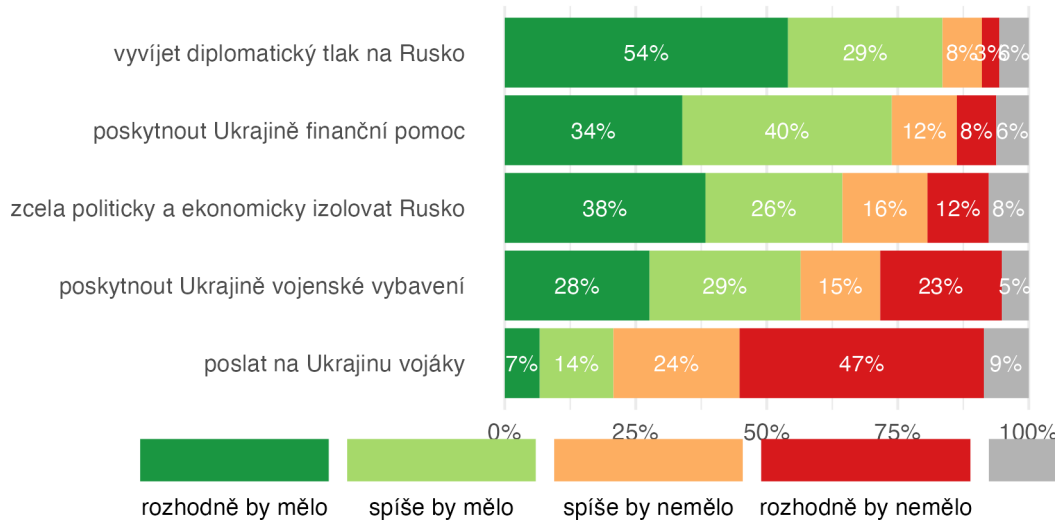
plot_likert <- function(data, color_palette, order_by, title, var_labels) {
  data %>%
    pivot_longer(cols = everything(),
                 names_to = "item",
                 values_to = "response") %>%
    left_join(var_labels, by = "item") %>%
    count(label, response) %>%
    filter(!is.na(response)) %>%
    group_by(label) %>%
    mutate(freq = n / sum(n),
           freq_label = percent(freq, accuracy = 1),
           positive = sum(freq[response %in% order_by ])) %>%
    ungroup() %>%
    mutate(label = fct_reorder(label, positive),
           response = fct_rev(response)) %>%
    ggplot(aes(x = freq, y = label, label = freq_label, fill = response)) +
    geom_col() +
    geom_text(position = position_stack(vjust = 0.5),
              color = "white",
              size = 3) +
    scale_x_continuous(labels = percent_format()) +
    scale_fill_manual(values = color_palette) +
    labs(x = element_blank(),
         y = element_blank(),
         fill = element_blank(),
         title = title) +
    theme_minimal() +
    theme(legend.position = c(0.3, -0.17),
          panel.grid.major.y = element_blank(),
          plot.title.position = "plot",
          plot.margin = unit(c(0,0,3.5,0), 'lines')) +
    guides(fill = guide_legend(label.position = "bottom",
                               keywidth = 5,
                               reverse = TRUE,
                               direction = "horizontal"))
}

```

Definice naší funkce vypadá téměř identicky jako původní kód, pět výše zmíněných částí jsme strategicky nahradili argumenty naší funkce. Jakmile je funkce vytvořené, je možné jí aplikovat na data. Kód pro vytvoření grafu se smrknul z třiceti řádků na šest:

```
ukraine %>%
  select(starts_with("PL_5")) %>%
  plot_likert(color_palette = likert_palette,
             order_by = c("rozhodně by mělo", "spíše by mělo"),
             title = "Jaké kroky by podle vás mělo podniknout mezinárodní společenství tv
             var_labels = ukraine_labels)
```

Jaké kroky by podle vás mělo podniknout mezinárodní společenství válce na Ukrajině? Mělo by...

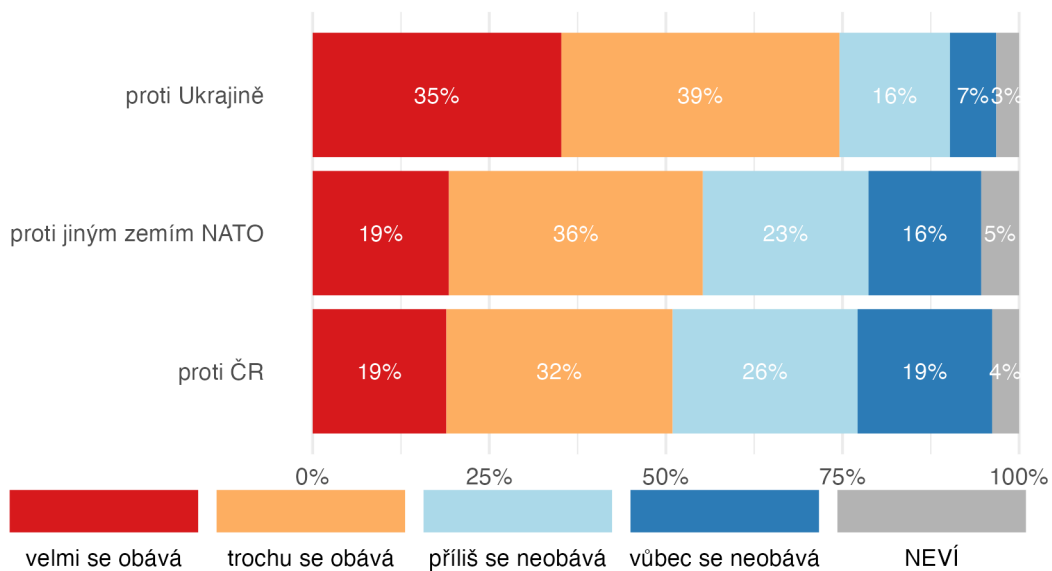


Naší funkci můžeme stejně jednoduše aplikovat na druhou baterii v datech, týkající se vnímané hrozby použití atomových zbraní:

```
likert_palette2 <- c("grey70", rev(brewer.pal(4, "RdYlBu")))

ukraine %>%
  select(starts_with("PL_6")) %>%
  plot_likert(color_palette = likert_palette2,
             order_by = c("velmi se obává", "trochu se obává"),
             var_labels = ukraine_labels,
             title = "Obáváte se, že Rusko může použít jaderné zbraně...")
```

Obáváte se, že Rusko může použít jaderné zbraně...



Jak je vidět, vytvořením vlastní funkce si můžeme výrazně ulehčit práci, nemluvě o tom, že tím náš kód uděláme čitelnější a robustnější.

28 For loops (cykly)

V průběhu analýzy dat se čas od času dostaneme do situace, kdy bude třeba opakovaně vykonávat určitý úkon, ať už se jedná o import velkého množství datasetů, vytvoření grafu pro každou proměnnou v dataframě nebo odhad intervalů spolehlivosti pomocí bootstrappingu. U všech těchto úkonů by bylo zdlouhavé a nepraktické aplikovat funkce ručně. Naštěstí pro nás, počítače jsou velmi dobré v opakování.

Nástroj, pomocí kterého docílíme výše zmíněného, se nazývá *For loop* (For cyklus).

28.1 Kdo je členem gangu?

Základní aplikaci For cyklu si představím na jednoduchém příkladu. Naším cílem bude vyjmenovat jednotlivé členy Scoobyho gangu. Nejdříve si vytvoříme objekt obsahující jména členů:

```
gang <- c("Fred", "Velma", "Daphne", "Shaggy", "Scooby")
```

Bez For cyklů musíme členy vyjmenovat ručně. Využijeme k tomu funkci `paste()`, pomocí které spojíme jméno člena/ky s větou *“is a member!”* :

```
paste(gang[1], "is a member!")
```

```
[1] "Fred is a member!"
```

```
paste(gang[2], "is a member!")
```

```
[1] "Velma is a member!"
```

```
paste(gang[3], "is a member!")
```

```
[1] "Daphne is a member!"
```

```
paste(gang[4], "is a member!")
```

```
[1] "Shaggy is a member!"
```

```
paste(gang[5], "is a member!")
```

```
[1] "Scooby is a member!"
```

Toto řešení je dost nepraktické, protože opakovaně kopírujeme stejný kód. Přitom jedině, co se ve funkcích výše mění, je pořadí člena. Elegantnější alternativou je již zmiňovaný for cyklus. For cyklus lze v R aplikovat (minimálně) dvěma způsoby.

Prvním způsobem je explicitní *for loop*. Explicitní *for loop* začíná funkcí `for()`, následovanou složenými závorkami. Argumenty `for()` funkce mají speciální podobu, v našem případě bude funkce vypadat následovně: `for(name in gang)`. Tímto říkáme, že chceme aplikovat nějakou funkci na každý element (zde zvaný `name`) objektu `gang`. Můžeme si přitom zvolit jakékoli označení pro jednotlivé elementy, které chceme. Zde používáme `name`, ale stejně tak bychom mohli použít například `for(i in gang)` nebo `for(pesky_child in gang)`. Následují složené závorky, definující, jaké funkce se na každý element mají aplikovat. Celý *for loop* by našem případě vypadal následovně:

```
for(name in gang) {  
  print(paste(name, "is a member!"))  
}
```

```
[1] "Fred is a member!"
```

```
[1] "Velma is a member!"
```

```
[1] "Daphne is a member!"
```

```
[1] "Shaggy is a member!"
```

```
[1] "Scooby is a member!"
```

Jak je vidět, nemusíme již kopírovat funkci `paste()` pětkrát za sebou. For cyklus se o to postará za nás. Výhodou explicitních *for loops* je, že se aplikují téměř identicky v každém programovacím jazyce. Pokud se tedy seznámíte s for cykly v R, můžete je jednoduše aplikovat i Pythonu nebo Julii.

Nevýhodou explicitních for cyklů, že jsou relativně květnaté - jejich zápis je delší, než je nezbytně nutné. Alternativou jsou funkce z balíčku `purrr`, který je součástí Tidyverse. Tyto funkce také aplikují různé druhy cyklů, jejich zápis je ale kompaktnější. Hlavní funkcí je zde `map()`, která má dva argumenty. Prvním argumentem je `.x`, objekt přes jehož elementy chceme iterovat. Druhým argumentem jsou funkce, které chceme aplikovat:

```
map(.x = gang, ~paste(.x, "is a member!"))
```

```
[[1]]  
[1] "Fred is a member!"  
  
[[2]]  
[1] "Velma is a member!"  
  
[[3]]  
[1] "Daphne is a member!"  
  
[[4]]  
[1] "Shaggy is a member!"  
  
[[5]]  
[1] "Scooby is a member!"
```

Jak je vidět, funkce `map()` zabírá méně prostoru, než explicitní *for loop* (a má další výhody, které zmíním za chvíli). Výsledkem této funkce je objekt typu *list*. Protože ale víme, že výsledkem naší funkce je věta, můžeme využít funkci `map_chr()`, jejímž výsledkem je *character* vektor:

```
map_chr(.x = gang, ~paste(.x, "is a member!"))
```

```
[1] "Fred is a member!" "Velma is a member!" "Daphne is a member!"  
[4] "Shaggy is a member!" "Scooby is a member!"
```

Obdobně bychom mohli využít funkce `map_dbl()` pokud je výsledkem desetinné číslo, `map_int()` pro celé číslo nebo `map_lgl()` pokud je výsledkem logický vektor.

28.2 Průměr každé numerické proměnné

Vyzkoušejme si nyní o něco praktičtější příklad. Naším cílem bude spočítat průměr každé numerické proměnné v datasetu `countries`. Začneme tím, že si vyfiltrujeme pouze numerické proměnné.

```
countries_numeric <- select(countries, where(is.numeric))
```

Nejdříve spočítáme průměry proměnných pomocí explicitního for cyklu. Prvním krokem je vytvořením prázdného vektoru, do kterého uložíme výsledky. Toho docílíme pomocí funkce `vector()`. Tento krok není nezbytně nutný, jedná se ale o dobrou praxi, protože urychlí výpočet. Poté definujeme *for loop* samotný. Nejdřív musíme získat pořadí jednotlivých proměnných v dataframe, a to pomocí funkce `seq_along()`. Ta vytvoří vektor čísel od 1 po hodnotu rovnou počtu elementů v objektu, přes iterujeme. Náš dataframe obsahuje devět proměnných, `seq_along(countries_numeric)` tedy vytvoří řadu celých čísel od 1 do 9. Uvnitř for cyklu spočítáme průměr i-té proměnné a výsledek uložíme jako i-tý element předpřipraveného objektu `countries_means`:

```
countries_means <- vector("numeric", length = ncol(countries_numeric))

for (i in seq_along(countries_numeric)) {
  countries_means[i] <- mean(countries_numeric[[i]], na.rm = TRUE)
}

countries_means
```

```
[1] 4.846008e+05 1.675474e+07 1.560186e+05 7.957838e+01 2.914857e-01
[6] 2.403030e-01 1.798788e-01 8.739474e-01 7.639189e+00
```

Výsledkem je devět průměrů pro devět proměnných. Zde začínáme vidět, že explicitní for cykly mohou být poněkud krkolomné. Je třeba předvytvořit vektor pro výsledky a musíme pracovat s pořadím proměnných. Vyzkoušejme si stejný úkol pomocí funkcí z balíčku `purrr`.

Narozdíl od explicitního for cyklu není třeba předvytvářet vektor pro výsledky, `map()` a příbuzné funkce to za nás udělají automaticky. také není třeba řešit pořadí proměnných pomocí `seq_along()`. Stačí nám tedy aplikovat funkci `map_dbl()` (protože výsledkem bude desetinné číslo) následovně:

```
map_dbl(.x = countries_numeric, ~mean(.x, na.rm = TRUE))
```

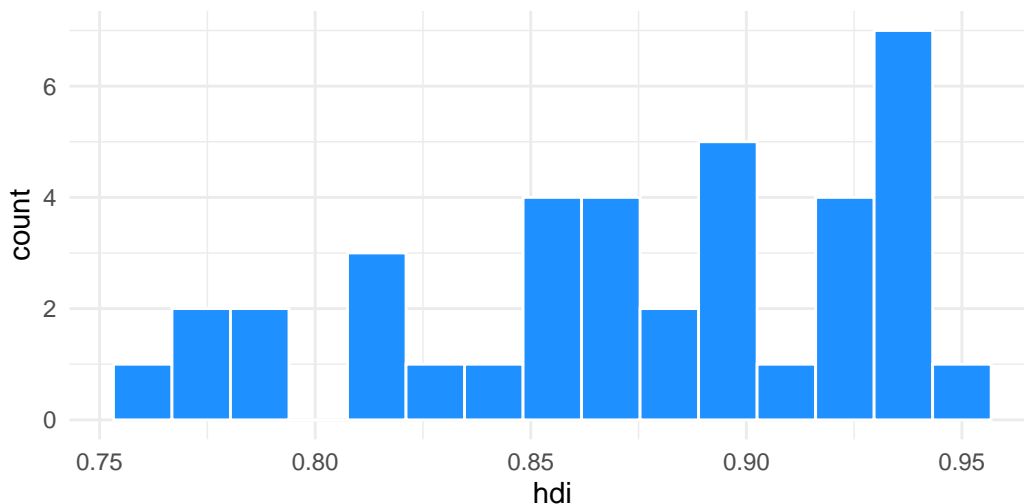
```
      gdp  population      area  life_exp  uni_prc poverty_risk
4.846008e+05 1.675474e+07 1.560186e+05 7.957838e+01 2.914857e-01 2.403030e-01
material_dep      hdi  dem_index
1.798788e-01 8.739474e-01 7.639189e+00
```

Jak vidíme, tato funkce je mnohem kompaktnější a navíc zachovává názvy původních proměnných. Jinak jsou výsledky identické.

28.3 Histogram pro každou numerickou proměnnou

Naším dalším cílem bude vytvoření histogramu pro každou proměnnou datasetu `countries_numeric`. Graf pro jednu proměnnou můžeme vytvořit tak, jak jsme si ukázali v předchozích kapitolách:

```
countries_numeric %>%  
  ggplot(aes(x = hdi)) +  
  geom_histogram(bins = 15, fill = "dodgerblue", color = "white") +  
  theme_minimal()
```

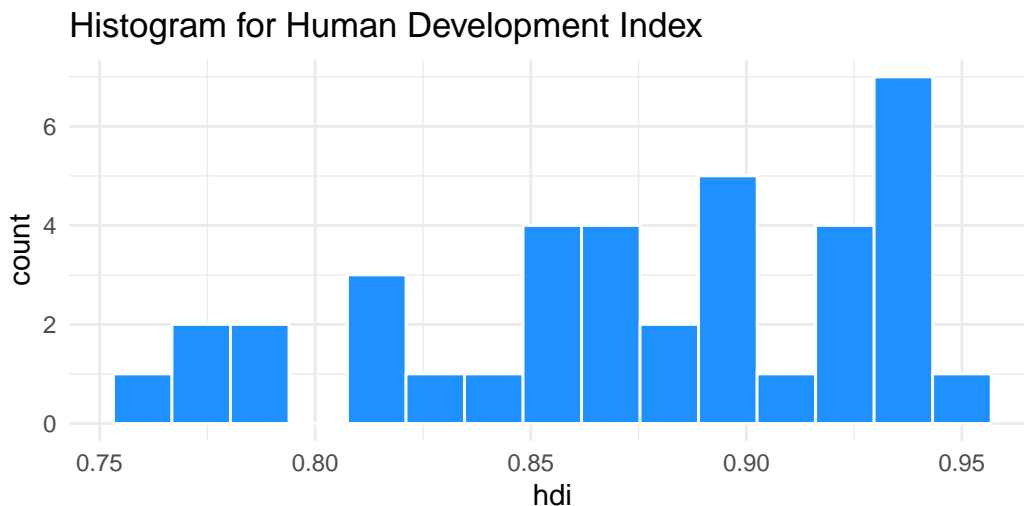


Co kdychom chtěli efektivně vytvořit tento graf pro všechny proměnné. S využitím funkce `map()` je to jednoduché. Jedinou změnou oproti předchozím příkladům bude využití složených závorek, které nám umožní aplikovat více funkcí najednou uvnitř jednoho for cyklu:

```
countries_histograms <- map(.x = countries_numeric,  
  ~{countries_numeric %>%  
    ggplot(aes(x = .x)) +  
    geom_histogram(bins = 15, fill = "dodgerblue", color = "white") +  
    theme_minimal()  
  })
```

Kód pro vytvoření grafu je téměř identický, pouze název proměnné jsme nahradili generickým argumentem `.x`. Výsledkem je list, obsahující devět grafů. Naše grafy ovšem postrádají nadpis, který by identifikoval, kterou proměnnou reprezentují. Naším dalším cílem tedy bude přidání popisků obsahující název proměnné. Pro jeden konkrétní graf bychom graf s popiskem vytvořili následovně:

```
countries_numeric %>%
  ggplot(aes(x = hdi)) +
  geom_histogram(bins = 15, fill = "dodgerblue", color = "white") +
  theme_minimal() +
  labs(title = "Histogram for Human Development Index")
```



Pro vytvoření podobného grafu bude třeba iterovat přes dvě objekty. Prvním objektem je proměnná, pro kterou chceme histogram vytvořit, druhým je vektor názvů, které chceme v grafech použít. K tomu využijeme funkci `map2()`. Ta funguje velmi podobně, jako nám již známá funkce `map()`, ale kromě argumentu `.x` má i druhý `.y`, čímž nám umožňuje dosadit do cyklu dva různé objekty. Pro vytvoření grafů s popisky pro každou z proměnných nejdříve vytvoříme vektor obsahující názvy všech proměnných a poté aplikujeme funkci `map2()`:

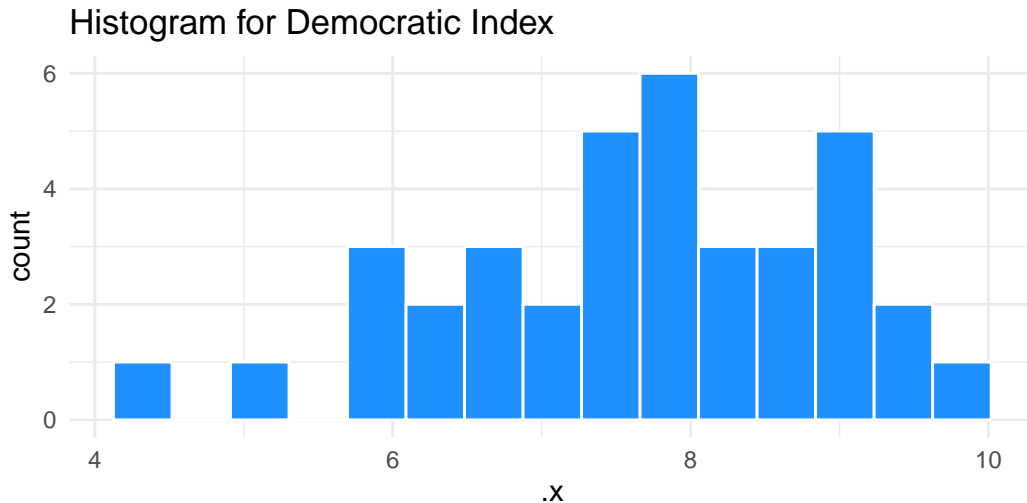
```
countries_names <- c("Gross Domestic Product",
  "Population",
  "Area",
  "Life Expectancy",
  "Proportion of People with University Degree",
  "Proportion of People at the risk of Poverty",
  "Proportion of Materialy Deprived",
  "Human Development Index",
  "Democratic Index")

countries_histograms <- map2(.x = countries_numeric,
  .y = countries_names,
  ~{countries_numeric %>%
```

```
ggplot(aes(x = .x)) +  
  geom_histogram(bins = 15, fill = "dodgerblue", color = "white") +  
  theme_minimal() +  
  labs(title = paste("Histogram for", .y))}}
```

A je to! Graf pro jednotlivé proměnné můžeme zobrazit jejich zavoláním:

```
countries_histograms$dem_index
```



💡 Iterace více než dvou objektů

Funkce `map2()` nám umožňuje iterativně aplikovat funkce na dva objekty najednou. Co kdybychom ale chtěli iterovat přes tři, čtyři nebo více objektů? Právě k tomu slouží funkce `pmap()`. Funkce `pmap()` funguje trochu odlišně oproti klasickému `map()`. Prvním argumentem je list objektů, přes které chceme iterovat. Každému z těchto objektů je přiděleno kódové označení - první objekt dostane označení `..1`, druhý `..2`, třetí `..3` a tak dále. Druhým argumentem jsou poté funkce, které chceme aplikovat. Pokud bychom chtěli předchozí úkol vyřešit pomocí `pmap()` místo `map2()`:

```

countries_histograms <- pmap(list(countries_numeric, #..1
                                countries_names), #..2
~{countries_numeric %>%
  ggplot(aes(x = ..1)) +
  geom_histogram(bins = 15,
                 fill = "dodgerblue",
                 color = "white") +
  theme_minimal() +
  labs(title = paste("Histogram for", ..2))})

```

28.4 Bootstrapping

Posledním příkladem využití for cyklů, který si zde ukážeme, je *bootstrapping*. Většina čtenářů pravděpodobně ví, že výzkumníci většinou nemají k dispozici data o celé populaci, kterou studují. Místo toho se musíme spokojit pouze s jejím vzorkem. Žádný vzorek ale svým složením nekopíruje dokonale populaci, ze které byl získaný. Tyto odchylky ve složení vzorku vedou k odchylkám v našich výsledcích. Učebnicovým příkladem jsou předvolební výzkumy - pokud ve vzorku voličů podporuje určitou politickou stranu například 5 procent respondentů, reálná podpora strany v populaci je zpravidla něco mezi 3 a 8 procenty. Tyto odchylky od skutečné hodnoty (takzvaná výběrová chyba) jsou náhodné a velká část statistiky je věnována jejímu vyčíslení. Protože zde nechceme příliš zabíhat do statistické teorie, doporučíme zájemcům o více informací učebnici [Learning Statistics with R](#) od Danielle Navarro. My si zde ukážeme jeden ze způsobů, jak tuto náhodnou výběrovou chybu vyčíslit, a to *bootstrapping*.

Metoda bootstrappingu je založená na vytváření nových vzorků pomocí opakovaného náhodného vytahování pozorování z našich původních dat. Pozorování jsou vytahována s opakováním, jedno pozorování se tedy může do nového vzorku dostat více než jednou. Tímto způsobem můžeme za určitých podmínek zodpovědět otázku “*Jak by se naše výsledky lišily, kdybychom nasbírali trochu odlišná data?*”. R nabízí několik balíčků pro aplikaci bootstrappingu, jako jsou `boot`, `rsample`, `coin` nebo `infer`. Jednoduchý bootstrapping je ale možné aplikovat pomocí for cyklů.

Bootstrapping probíhá ve dvou krocích. V prvním kroku vytvoříme bootstrapový vzorek pomocí náhodného výběru s opakováním z původních dat (je přitom nezbytné, aby nový vzorek měl stejný počet pozorování, jako ten původní). Poté spočítáme statistiku, pro kterou chceme vyčíslit náhodnou výběrovou chybu (například průměr). Tyto dva kroky mnohokrát opakujeme, čímž dostaneme vektor statistik, spočítaný na vzorcích s trochu odlišným složením. Tyto hodnoty představují odhad výběrové distribuce výběrové statistiky.

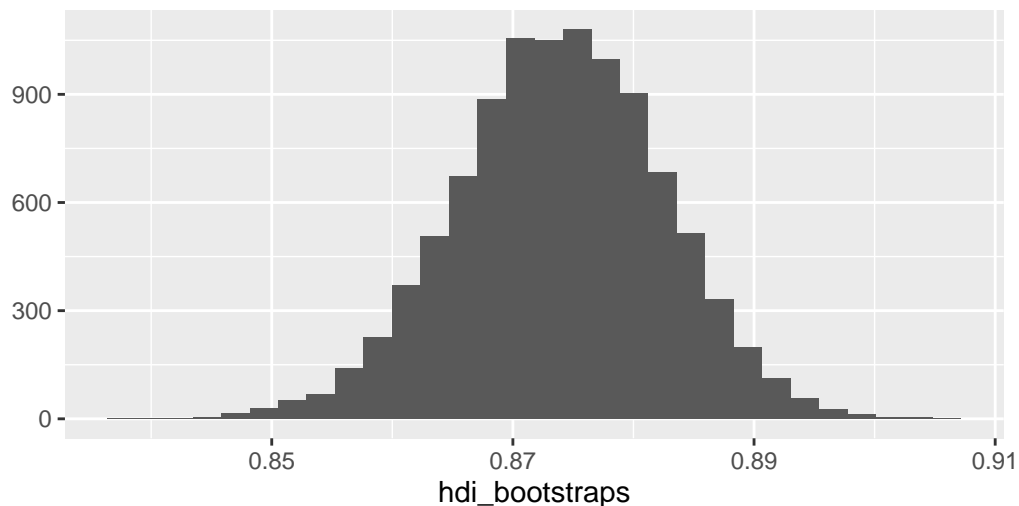
Jako praktický příklad zkusme vyčíslit výběrovou chybu průměru proměnné `hdi` v datasetu `countries`. Začneme aplikací funkce `map_dbl()`. To použijeme, protože víme, že jednotlivé

průměry jsou čísla a výsledkem tedy bude numerický vektor. Prvním argumentem není žádný konkrétní objekt, místo toho se jedná o vektor celých čísel od 1 po hodnotu určující počet bootstrapových vzorků, které chceme vytvořit. My budeme chtít vytvořit 10 000 vzorků, prvním argumentem tedy bude `1:10000`. Druhým argumentem je výpočet bootstrapové statistiky samotné. K tomu nejdříve využijeme funkce `sample()`, pomocí které vytvoříme bootstrapový vzorek. Dáme si přitom pozor nato, abychom nastavili argument `replace = TRUE`, čímž zajistíme, že výběr probíhá s opakováním (jinak bychom skončili pokaždé se stejným vzorkem). Poté už jen stačí spočítat požadovanou statistiku, v našem případě průměr. Celá funkce funkce provádí následující - pro každý krok od prvního po desetitisící vytvoř nový bootstrapový vzorek a spočítej jeho průměr. Výsledek uložíme do objektu `hdi_bootstraps`:

```
hdi_bootstraps <- map_dbl(1:10000,  
  ~{  
    sample <- sample(countries$hdi, replace = TRUE)  
    mean(sample)  
  })
```

Výsledný vektor 10 000 průměrů představuje odhad výběrové distribuce průměrů proměnné `hdi`. Můžeme jí graficky zobrazit:

```
qplot(hdi_bootstraps)
```



Můžeme spočítat standardní odchylku výběrových průměrů (tedy standardní chybu):

```
sd(hdi_bootstraps)
```

```
[1] 0.008568953
```

A nakonec můžeme spočítat 95% intervalový odhad průměru hdi:

```
quantile(hdi_bootstraps, probs = c(0.025, 0.975))
```

```
      2.5%      97.5%  
0.8568421 0.8900066
```

A je to! *Bootstrapping* představuje jednoduchý způsob vyčíslení náhodné výběrové chyby, a to zejména pokud pracujeme s většími vzorky s jednoduchou strukturou. Čtenáři by ovšem měli mít na paměti dvě věci. Zaprvé, bootstrapované hodnoty představují pouze odhad výběrové distribuce - čím více simulací, tím přesnější odhad bude. Zadruhé, *bootstrapping* je stejně jako všechny ostatní statistické nástroje založený na sadě předpokladů, které zaručují jeho správnou funkci. Pokud je některý z těchto předpokladů výrazně porušený, naše výsledky se stanou velmi nepřesnými. Čtenáři by si měli důkladně nastudovat potřebnou teorii, než aplikují bootstrapping v praxi.

💡 R rozumí vědecké notaci

Protože R rozpoznává vědeckou notaci, je možné nahradit 10000 výrazem `1e4`. Výsledná funkce tedy může vypadat následovně:

```
hdi_bootstraps <- map_dbl(1:1e4,  
  ~{  
    sample <- sample(countries$hdi, replace = TRUE)  
    mean(sample)  
  })
```

29 Co dál?

Pokud jste dočetli až sem, znamená to, že máte za sebou vše nezbytné pro to začít s analýzou kvantitativních dat v R. Možnosti práce s R jsou ovšem široké a mnoho dalšího vás ještě čeká. V této, již poslední, kapitole vám proto dáme pár tipů, kam se vaše cesta kvantitativní analýzou dat může ubírat.

29.1 Statistika

Cesta, která se přirozeně nabízí, je dále se vzdělávat v oblasti statistiky a statistického modelování. Zájemcům o studium statistiky doporučujeme začít knihou [Learning Statistics with R](#) od Danielle Navarro, která vás přátelským způsobem provede úvodem do statistické teorie. Navázat na ní můžete s [Regression and Other Stories](#) od Andrewa Gelmana, Jennifer Hill a Aki Vehtariho, knihou která poskytne cenné rady týkající se statistického modelování jak začátečníkům, tak pokročilým. Z trochu jiného soudku je [Tidy Modelling with R](#) od Maxe Kuhna a Julie Silge, text zaměřený na prediktivní modelování a *“machine learning”*.

29.2 Vizualizace dat

Ti z vás, kteří našli svůj zájem ve vizualizaci dat jistě ocení knihu [Fundamentals of Data Visualization](#) od Clause Wilkea. Ta vás provede teorií a užitečnými tipy pro vytváření hezky vypadajících a zároveň efektivních grafů. Naleznete v ní vše od návodů pro vizualizaci různých druhů proměnných až po tipy pro vytváření barevných palet. Pro hlubší pochopení fungování balíčku `ggplot2` a gramatiky grafů doporučujeme [ggplot2: elegant graphics for data analysis](#) od Hadleyho Wickhama, Danielle Navarro, and Thomase Lin Pedersena.

29.3 Organizace práce

Poslední a neméně důležitou oblastí je organizace práce. Ti z vás, kteří mají zkušenosti z prací v týmech a na dlouhodobých projektech jistě chápou jak nezbytné je mít dobře nastavené systémy pro sdílení a zálohování dat, analýz a výsledků. Užitečné tipy pro toto vše nabízí [The Plain Person's Guide to Plain Text Social Science](#) od Keirana Healyho. Pro moderního

analytika je nezbytný především verzovací program Git, se kterým vás seznámí [Happy Git and GitHub for the useR](#) od Jennifer Bryan.

29.4 Na vše ostatní je tu **Big Book of R**

Pokud vás žádné z předchozích témat nezaujalo, nemusíte smutnit. [Big Book of R](#) je ultimátní kniha knih spravovaná Oscarem Baruffem. V ní naleznete odkazy na zdroje týkající se nepřehledného množství témat. Na své si přijdou všichni, od fanoušku analýzy sociální sítí po finanční analytiku a datové žurnalisty. Rozhodně doporučujeme si najít klidnou půl hodinu a knihu si projít.

A to je vše! Doufáme, že se vám naše kniha líbila a přejeme mnoho zdaru s analýzami. :-)