

Kurskompendium - introduktion till R

Version 0.99

Innehåll

Introduktion.....	10
Basics.....	11
R-konsolen	11
RStudio	13
Hjälpfunktioner	14
Utforska data.....	15
Visualisering.....	16
Ett första diagram.....	16
Ladda data	16
Skapa en ggplot.....	17
En grafikmall	18
Övningar.....	18
aes()-funktionen och mappings.....	18
Övningar.....	24
Vanliga problem.....	24
Facets.....	25
Övningar.....	27
“Geometrisk” objekt - geoms	27
Övningar.....	32
Statistiska transformationer	33
Övningar.....	37
Positionering.....	37
Övningar.....	45
Koordinatsystem	45
Övningar.....	50
“The layered grammar of graphics”	50
Workflow: Basics.....	52
Skriva kod.....	52
Anropa funktioner.....	52
Övningar.....	53
Transformerering av data	54
Introduktion	54

Förberedelser.....	54
dplyr grunder.....	55
Filtrera rader/poster med filter().....	56
Övningar.....	59
Arrangera poster med arrange().....	59
Övningar.....	60
Välj kolumner med select()	60
Övningar.....	63
Lägg till flera variabler med hjälp av mutate().....	64
Funktioner och operatorer att användas med mutate()	66
Övningar.....	68
Grupperade summeringar med summarise()	68
Kombinera multipla operationer med the pipe.....	69
Missing values.....	71
Antal (counts)	72
Användbara summeringsfunktioner.....	75
Gruppera med flera variabler	80
Av-gruppera.....	82
Övningar.....	82
Grupperade beräkningar och filtreringar	82
Övningar.....	84
Arbetsflöde: scripts	85
Att köra kod	85
Diagnostics	86
Praktik.....	86
Explorativ analys av data	87
Introduktion	87
Frågor.....	87
Variation.....	88
Visualisera fördelningar	88
Outliers.....	92
Övningar.....	95
Missing values.....	95
Samvarians - covariance.....	98

En kategorisk och en kontinuerlig variabel.....	98
Övningar	105
Två kategoriska variabler.....	105
Två kontinuerliga variabler	107
ggplot2 calls	111
Att lära mer	114
Arbetsflöde: Projekt	115
Vad är viktigt?	115
Vart lever analysen? Arbetsbiblioteket.....	116
Sökvägar och bibliotek/mappar	116
RStudio projects VIKTIGT!.....	116
Sammanfattningsvis	119
Wrangle - att brottas med data.....	120
Introduktion	120
Tibbles	121
Skapa tibbles	121
Tibbles vs data frames	122
Utskrift (Printing).....	122
Urval	124
Använda äldre kod	124
Importera data	126
Jämförelse med base R	128
Övningar.....	128
Omvandla vektorer	128
Numeriska variabler.....	130
Textsträngar	131
Kategoriska data	132
Datum och tid	133
Övningar	134
Hur readr "plockar isär" (parse) en fil	134
Problem	135
Skriva till en fil.....	135
Andra datatyper	136
Städa data (tidy data)	137

Introduktion	137
Spreading and gathering.....	140
Gathering.....	140
Spreading.....	142
Övningar.....	143
Separera och förena.....	143
Separate.....	144
Unite.....	145
Övningar.....	146
Missing values.....	146
Övningar.....	149
En case study.....	149
Övningar.....	154
Non-tidy data.....	154
Relationer mellan datamängder.....	155
Introduktion	155
nycflights13	155
Keys.....	157
<i>Mutating joins</i>	159
Att förstå <i>joins</i>	161
Inner join.....	162
Outer joins	162
Duplicate keys.....	163
Definiera nyckel-kolumner (<i>key columns</i>).....	164
Övningar	166
<i>Filtering joins</i>	167
Problem med <i>joins</i>	170
Set operations.....	170
Textsträngar (strings)	172
Introduktion	172
Basics.....	172
stringr-funktioner.....	173
Kombinera strängar.....	173
Extrahera delar av textsträngar (subsetting)	174

Övningar	175
Matcha textmönster med hjälp av <i>regular expressions</i>	175
Basic matches	175
Ankare (Anchors)	176
Tecken-klasser	178
Repeterande tecken	178
Verktyg	181
Upptäck matchningar	181
Övningar	183
Extrahera matchningar	184
Gruppereade matchningar	186
Ersätt matchningar	186
Dela upp textsträngar	187
Hitta matchningar	189
Andra typer av mönster	189
Andra typer av regexps	194
stringi	194
Kategoriska variabler (Factors)	195
Introduktion	195
Skapa factors	195
General Social Survey	197
Modifiera ordningen av factors	199
Modifiera factor levels	205
Datum och tidsformat	209
Introduktion	209
Skapa datum och tid	209
Från textsträngar	210
Från enskilda komponenter	210
Från andra datum/tid-format	214
Övningar	215
Datum/tid-komponenter	215
Extrahera komponenter	215
Avrundning	216
Ange komponenter	217

Tidsintervall	218
Varaktighet.....	218
Perioder	219
Intervall	221
Summering.....	222
Programmering i R	223
Introduktion	223
Pipes	224
Funktioner.....	225
När bör du skapa en funktion?	225
Funktioner är till för människor och datorer	227
Villkorlig exekvering	228
Villkor	228
Multipla villkor	229
Funktionsargument.....	229
Kolla värden.....	230
punkt-punkt-punkt.....	231
Returnera värden	232
Explicita return-uttryck	232
Skriva funktioner med hjälp av pipes.....	233
Environment (svårt hitta ett svenskt uttryck)	234
Vektorer	235
Introduktion	235
Atomic vectors	236
Logical vectors	236
Numeriska vektorer.....	236
Character.....	237
Använda atomic vectors.....	237
Scalars och recycling	239
Rekursiva vektorer (lists).....	242
Attribut	245
Förstärkta vektorer (Augmented vectors)	245
Itereringar	248
For-loops.....	248

Variationer av for-loopar.....	249
Modifiera ett existerande objekt.....	249
När längden på output inte är känd.....	250
När sekvensens längd är okänd	251
for-loops vs. Funktioner.....	252
Mappningsfunktioner	254
Genvägar.....	255
Hantera errors	256
Introduction.....	260
Modellering: basics.....	261
En enkel modell.....	261
Visualisera modeller.....	263
Prediktioner	263
Residualer.....	264
Formler och modell-familjer.....	266
Kategoriska variabler	267
Interaktioner	269
Interaktioner (två kontinuerliga variabler)	272
Transformationer	275
Missing values.....	276
Andra modell-familjer.....	277
Bygga modeller i R.....	278
Varför är diamanter av låg kvalitet dyrare?.....	278
Sambandet pris och karat.....	280
En mer komplicerad modell	286
Vad är det som påverkar antalet dagliga flighter?.....	289
Veckodag.....	290
Lördags-effekten	295
Lära mer om modellering.....	301
Grafik för att kommunicera.....	302
Etiketter (labels).....	302
Annoteringar	305
Scales	308
Axis ticks och teckenförklaringar	310

Teckenförklaringens (legend) layout	312
Byta ut en scale	315
Zooming.....	324
Teman (themes)	329
Spara graferna	330

Introduktion

I denna introduktionskurs kommer du att bli hyggligt bekväm med att hantera de viktigaste verktygen i R och det modernare gränssnittet *Rstudio*. Som alltid bygger det dock på att du själv brottas med programmet och allteftersom lär dig genom att göra misstag.

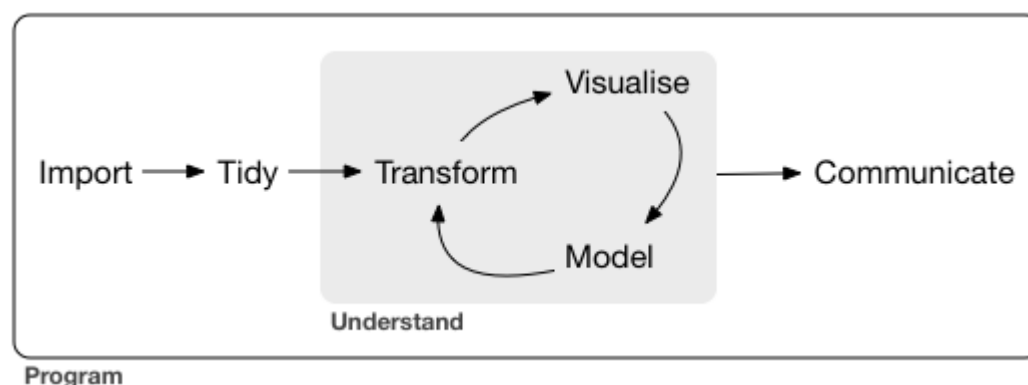
Kursen är grovt tänkt att genomföras i tre (eventuellt fyra) workshops uppdelade ungefär så här:

1. visualisering och transformering av data, dvs fram till och med avsnitt 8, *Workflow: projects*.
2. brottas med data; importera och städa data, hur handskas med olika datatyper, dvs fram till och med avsnitt 16, *Datum och tidsformat*.
3. programmering i R; om funktioner i R, om modellering och kommunikering.

Vi får se hur detta upplägg håller. Eventuellt kan vi ordna ytterligare en workshop. Tanken är att du med kompendiet som stöd på förhand går igenom avsnitten som tas upp på kommande workshop. Det blir alltså inga större föreläsningar utan workshopen kommer att lägga tonvikten vid sådant som deltagarna stött på problem med och att vi då löser dem tillsammans.

Kompendiet bygger på Hadley Wickham's och Garrett Grolemond's bok *R for Data Science* som också finns tillgänglig på internet (<https://r4ds.had.co.nz/index.html>). I själva verket följer innehållet ganska slaviskt Wickham's och Grolemeund's bok. Jag har emellertid utelämnat vissa delar som kräver mera tid samt delar som är mer av karaktären statistisk teori, för att på så sätt fokusera på själva programmet R/RStudio och förmedla grunderna i att använda dessa program praktiskt. Den som är sugen på mer rekommenderas varmt att kika i grundboken. Anledningen till att använda boken som ett underlag för kompendiet äär att den är verkligen en pedagogisk genomgång av hur man använder det "moderna R" - Wickham har varit en drivkraft bakom att utveckla det från början tämligen svåröverskådliga och inkonsistenta programspråket till ett mer konsistent och därmed sänkt inlärningströskeln. Det har han gjort genom att utveckla och modifiera programmeringsspråket samt utvecklat en rad packages för att hantera data på ett effektivare och möjligen modernare sätt än tidigare.

Innehållet i kompendiet följer en viss logik, enligt nedanstående figur:



Figuren beskriver datahantering och -analys som en process som omfattar

1. Importera data
2. Rensa data

3. Transformera data till ändamålsenliga arbets-data - reproducerbarhet!
4. Visualisera, utforska data för överblick och förståelse
5. Modellera data
6. Och att kommunicera resultaten

Punkterna 1-2 handlar om att importera och manipulera data så att det blir möjligt att analysera dem. Punkterna 3-5 syftar till att få en överblick och förstå datamängden och hur olika komponenter hänger samman med varandra. Punkt 6 handlar om att kommunicera resultaten från analysen.

Vi kommer att beröra samtliga delar mer och mindre. Till att börja med dyker vi in i punkterna 3-5 för att snabbt komma in i programmeringsspråket och känna på olika verktyg för visualisering, modellering och transformering av data. Därefter behandlar vi punkterna 1-2 och brottas då med verktyg för att anpassa rådata till arbets-data och i den avslutande delen fördjupar vi valda delar av 1-5 samt berör punkt 6.

Men först lite basal R/RStudio-hantering.

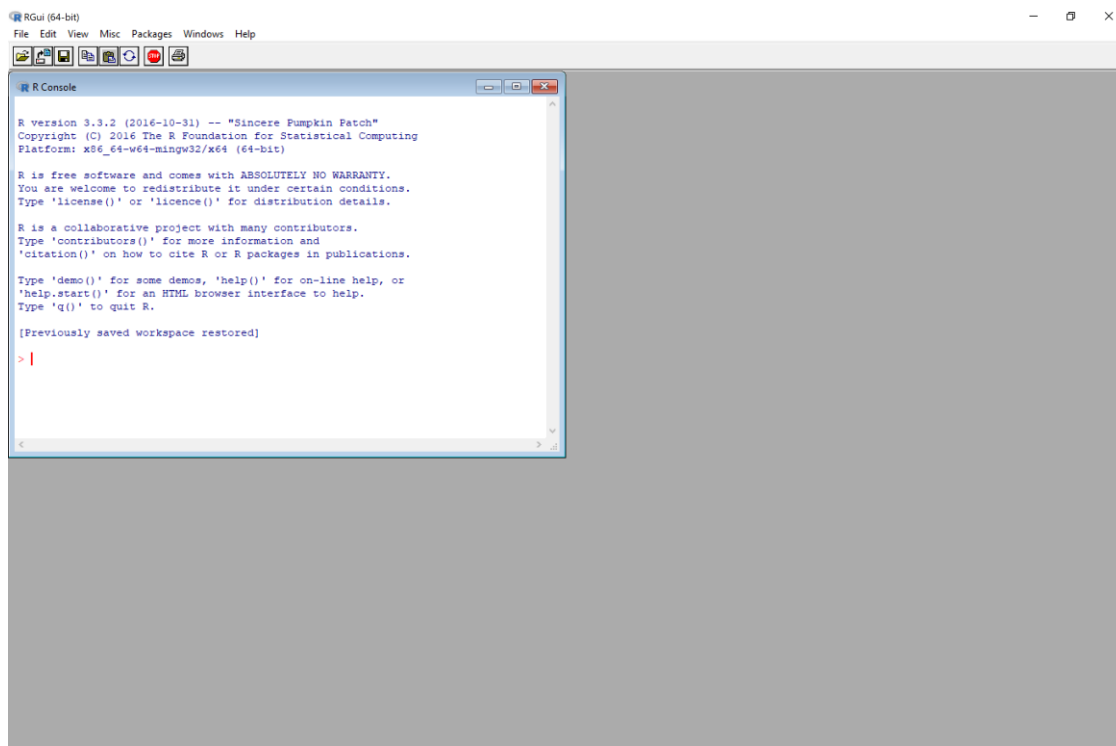
Basics

I detta avsnitt ska vi väldigt översiktligt nosa på R-konsolen som självständigt program och använda det som en kalkylator. Därefter ska vi bekanta oss med RStudio och börja uppskatta finessen med att ha ett separat gränssnitt. Vi ska även titta på några olika sätt att få hjälp när du kör fast – för det kommer du att göra.

R-konsolen

R fungerar som ett självständigt program och man kan i princip göra alla beräkningar och bearbetningar direkt i programmets eget gränssnitt – konsolen. I det här avsnittet ska vi bekanta oss översiktligt med konsolen.

Starta upp R. nedanstående skärmbild kommer upp. I konsolen ses en standardtext som visar vilken version av R som körs, en deklaration om R samt några tips om hjälp-funktioner.



Menyraden innehåller sedvanliga funktioner och länkar. Vi ska inte gå igenom dessa utan bara känna på hur det är att arbeta i detta gränssnitt. Låt oss börja med att titta på några inbyggda R-demon. Skriv

`demo()`

vid prompten (`>`) och tryck ENTER. Då kommer en "output"-skärm fram vilken visar vilka tillgängliga demon som finns. Låt oss titta på några exempel på R:s grafiska möjligheter. Det gör man genom att ange `demo(graphics)`. Det kan man förstås skriva ut men ett smartare sätt är att unyttja att R håller reda på tidigare kommandon och genom att istället trycka "uppåt"-pilen på tangentbordet kommer det senaste kommandot att synas på skärmen. Med hjälp av vänsterpilen flyttar du markören innanför parentesen och skriver *graphics* så att kommandot nu är

`demo(graphics)`

Tryck ENTER. I konsolen ses nu en bekräftelse på att R har laddat grafik-demot. Tryck ENTER en gång till för att köra igång demot. Då förbereds R genom att ladda in nödvändiga packeages och data-set. Längst ner står "Waiting to confirm page change...". R är redo att visa olika grafiska outputs. Högst upp på output-skärmen uppmanas vi att trycka ENTER för att bekräfta fortsättningen. Då vi gör det visas ett nytt diagram för varje gång ENTER trycks ned. I konsolen visas den syntax R använt för att framställa diagrammen. Efter ett antal ENTER händer inget mer. Vi har nått slutet på demon och det markeras på konsolen genom att prompten (`>`) nu blivit röd. R är nu redo för nya kommandon.

Låt oss använda R som en kalkylator. Man skriver då in de beräkningar man vill utföra och trycker ENTER. Till exempel:

`2+3`

`4*7`

```
9/4
log(4)
3^2
sqrt(6)
```

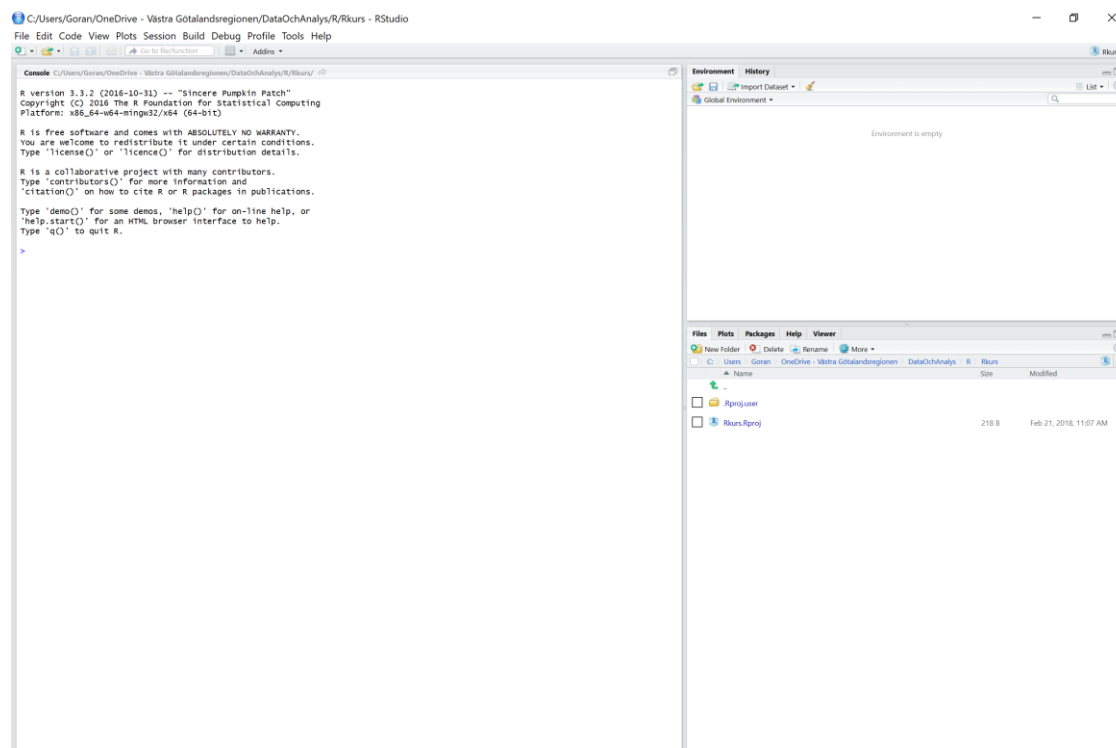
även om R-konsolen är fullt funktionell är användningen en mödosam process, speciellt då man använder mer komplexa script för sina sessioner. Det blir mycket tangent-tryckningar och ofta omständligt att felsöka scripten man jobbar med. Därför bör man använda något av de modernare gränssnitt som utvecklats. Det kanske mest använda är RStudio, vilket vi nu går över till. Skriv

```
q()
```

i konsolen, tryck ENTER och välj att inte spara. Tryck OK.

RStudio

RStudio är en integrerad utvecklingsmiljö (IDE, integrated development environment) för R programmering. När du öppnar programmet möts du av två dominerande ytor, konsolen och resultatytan nere till höger (*console* och *output*).



Konsolen är identisk med gränssnittet i det nativa R-gränssnittet som vi nosade på ovan.

Innan vi börjar laborera med RStudio behöver vi emellertid installera en modul, eller ett paket (package), som innehåller funktioner, data och dokumentation nödvändiga för att genomföra kursen. Sådana paket (packages) utgör en viktig del av den breda funktionaliteten i R och vi ska komma tillbaka

till dessa lite senare. Men för nu nöjer vi oss med att installera en modul som heter `tidyverse` och är egentligen en samling av andra moduler för att förenkla och effektivisera datahanteringen.

Det finns ett par olika sätt att installera en modul som *tidyverse*. För tillfället ska vi installera *tidyverse* med en enkel rad med kod. I konsolen skriv

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

och tryck ENTER.

Tidyverse är en samling moduler vilka bygger på samma konsistenta programlogik och utnyttjar kapaciteten i R optimalt för att hantera data. För kursen täcker tidyverse behovet av moduler som inte finns med i basversionen av R. Nästan. Vi ska ladda ned ytterligare tre moduler som innehåller data till övningsexemplen längre fram. Så på samma sätt som tidigare, skriv

```
install.packages(c("nycflights13", "gapminder", "Lahman"))
```

Och tryck ENTER.

Dessa moduler innehåller data över flygtrafik, global utveckling och baseball-data.

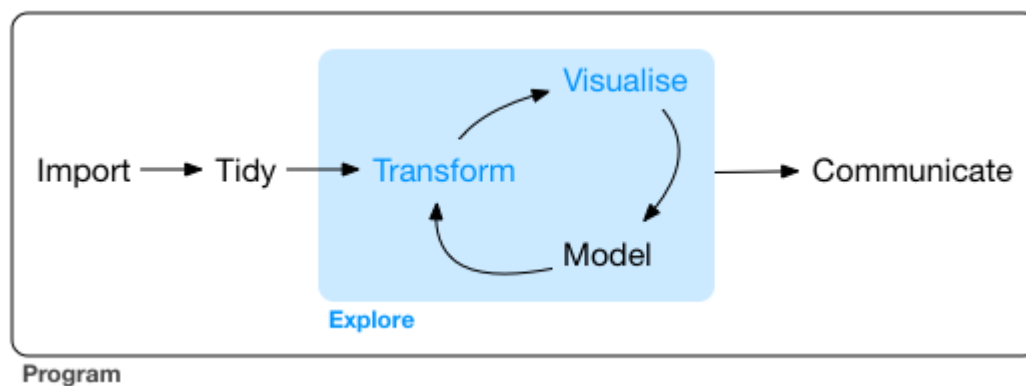
Hjälpfunktioner

R och RStudio innehåller flera inbyggda möjligheter att få hjälp när man kör fast, vilket inte sällan blir fallet. Förutom dessa måste nämnas möjligheten att googla fram en lösning på problemet. Det kan gälla alltifrån tämligen ospecifika frågor om t.ex. en viss funktion till att förstå vad ett felmeddelande betyder. Att kombinera sökfrasen med "R" räcker i allmänhet för att resultatet ska vara tillräckligt R-specifikt. Google är förvånansvärt effektivt för att få klarhet i just felmeddelanden vilka ofta är ganska kryptiska.

Om Google inte ger svaret bör du försöka med Stackoverflow (<http://stackoverflow.com/>) och i sökfrasen inkludera [R] för att begränsa sökningen till R-specifika svar.

Utforska data

Målet med detta avsnitt är att du så snabbt som möjligt ska komma på banan med några grundläggande verktyg/funktioner för att utforska (explore) en datamängd, dvs. att få en översikt över vilka data som finns, pröva preliminära hypoteser, testa dem och allteftersom lära känna data ordentligt. Det handlar alltså om den mellersta fasen i nedanstående bild - fr.a att Visualisera och transformera data för överblick och förståelse.



För detta ändamål ska vi använda ett antal verktyg. Det är verktyg för att *visualisera* datastrukturer, att *transformera* data för att kunna undersöka associationer och (preliminärt) *formulera hypoteser* för vidare analys.

I avsnittet finns även ett par stycken som handlar om arbetsflöden i R - "good practice" för att skriva och organisera R-kod.

Visualisering

Att börja jobba med visualiseringsverktygen i R är ett bra sätt att lära sig R. Man får snabbt synlig feedback samtidigt som man får en känsla för programspråket.

Det finns ett omfattande grundläggande grafikhanteringssystem i R. Nackdelen är att det har en tämligen hög inlärningströskel, åtminstone då man vill framställa mer avancerade grafer. Vi ska därför använda en modernare modul, kallad `ggplot2`, vilket blivit populärt genom sin något lägre inlärningströskel och flexibilitet.

Ett första diagram

Vi börjar med att ladda in modulen `tidyverse`:

```
library(tidyverse)
```

Denna modul innehåller flera andra moduler som är användbara för att transformera och visualisera data. Låt oss göra ett diagram för att besvara frågan om bilar med större motorer förbrukar mer bränsle än bilar med mindre motorer. Tja, det kanske är uppenbart men hur ser sambandet ut mer i detalj? Är sambandet positivt eller negativt? Linjärt eller icke-linjärt?

Ladda data

Inbyggt i R finns ett antal färdiga dataset som man kan utnyttja för att lära sig olika moduler eller pröva olika analysmodeller. Bland annat finns i modulen `ggplot2` (vilken laddas med `tidyverse`) ett dataset som innehåller information om olika bilmodeller, kallat `mpg`. Det är en *data frame*, i princip en tabell, med variabler i kolumner och de olika bilmodellerna i raderna. Vi kan se närmare på tabellen genom att ladda tabellen:

```
mpg
```

Det finns ett par olika sätt att få en överblick över datastrukturen.

1. Genom att helt enkelt ange namnet på datamängden (`mpg`) i konsolen och trycka ENTER.
2. Genom att klicka på pil-markören i rutan "Environment" uppe till höger. Då listas de 11 variablerna och beskriver typ av variabel (`chr` = text; `num` = numerisk kontinuerlig; `int` = numerisk och diskret) samt de första värdena för respektive variabel. Notera att data-namnet kompletteras med en kort beskrivning av storleken av datamängden.
3. Genom att dubbelklicka på data-namnet i "Environment" upp till höger. Då öppnas tabellen som en flik uppe till vänster, i editorn.

Två av variablerna, `displ` och `hwy`, beskriver motorstorleken i liter resp bränsleeffektiviteten uttryckt i miles per gallon.

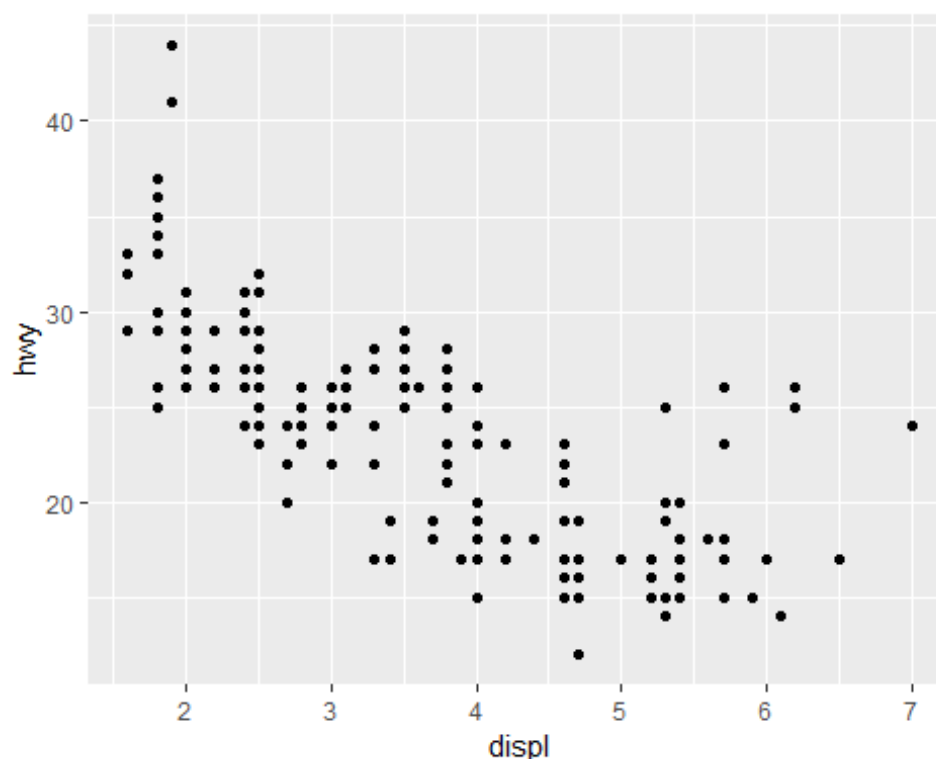
För att veta mer om datasetet kör man kommandot `?mpg`. Det öppnar en hjälp-sida där datasetet beskrivs närmare, definitioner av variabler etc.

Skapa en ggplot

För att beskriva sambandet mellan motorstorlek och bränsleeffektivitet grafiskt sätter vi motorstorleken på x-axeln och bränsleeffektiviteten på y-axeln (När du kör kod - inkl nedanstående - får du ofta en hel del information förutom önskat output, t.ex. "warnings", "conflicts". Vi återkommer till nyttan med dessa.):

```
library(tidyverse)

ggplot(data = mpg, aes(x = displ, y = hwy)) +
  geom_point()
```



Grafen visar som förväntat ett negativt samband mellan motorstorlek och bränsleeffektivitet - större motorer konsumerar mer bränsle.

Med *ggplot2* börjar man med att konstruera en graf genom att ange *funktionen* `ggplot()`. Denna funktion skapar ett koordinatsystem till vilket man kan addera olika lager (layers). Det första argumentet i `ggplot()` anger vilken datamängd som ska användas.

Man kompletterar grafen genom att lägga till ett eller flera lager till `ggplot()`. I detta fall lägger vi till ett lager med hjälp av funktionen `geom_point()`. Denna funktion lägger till ett lager av punkter till koordinatsystemet, vilket skapar en scatterplot. Det finns en mängd olika typer av `geom_*` funktioner med vars hjälp man kan skapa en mängd olika grafer vilket vi ska återkomma till.

Varje `geom_*` funktion använder ett *mapping-argument*. Det definierar hur variablerna mappas till visuella egenskaper. Varje mapping-argument är alltid kopplade till en `aes()`-funktion som med hjälp av x och y-argumenten specificerar de variabler som ska plottas i xy-systemet.

En grafikmall

Exemplet visar den principiella uppbygganden av en graf i ggplot2. Man använder olika funktioner och argument för att precisera hur grafen ska byggas upp.

```
ggplot(data = <DATA>) + <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

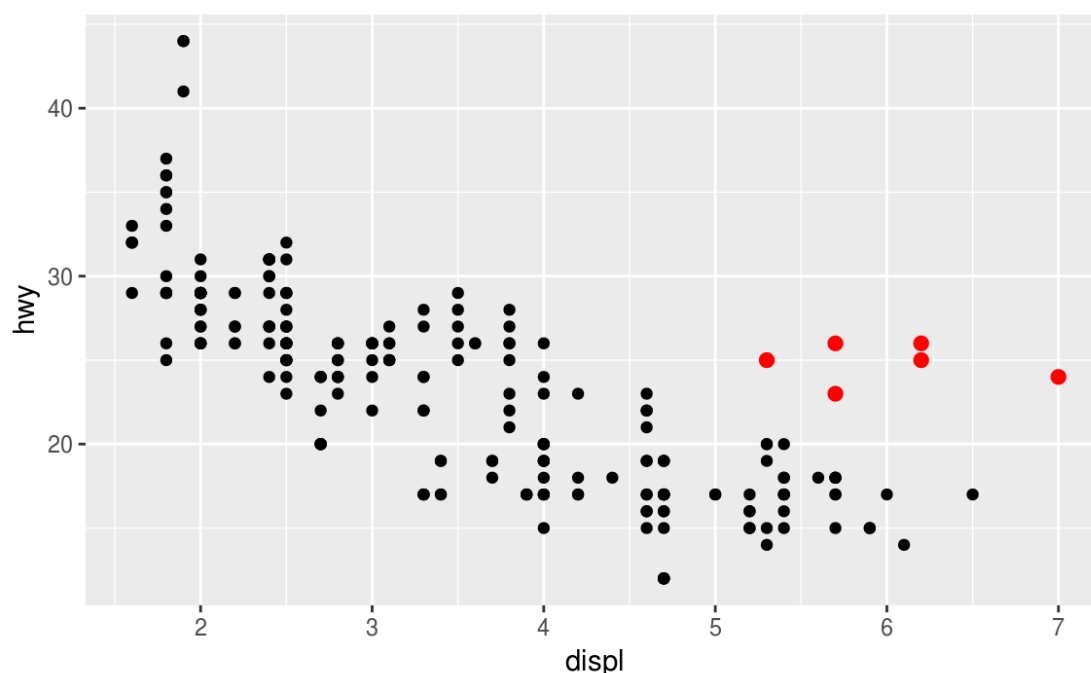
Vi ska fortsätta med att bygga ut denna mall för att framställa olika typer av grafer men först några övningar.

Övningar

1. Kör `ggplot(data = mpg)`. Vad ser du?
2. Hur många rader finns i `mpg`? Hur många kolumner?
3. Vad innehåller variabeln `drv` för information? Tips: använd hjälpfunktionen `?mpg`.
4. Framställ en scatterplot som beskriver sambandet mellan `hwy` och `cyl`.

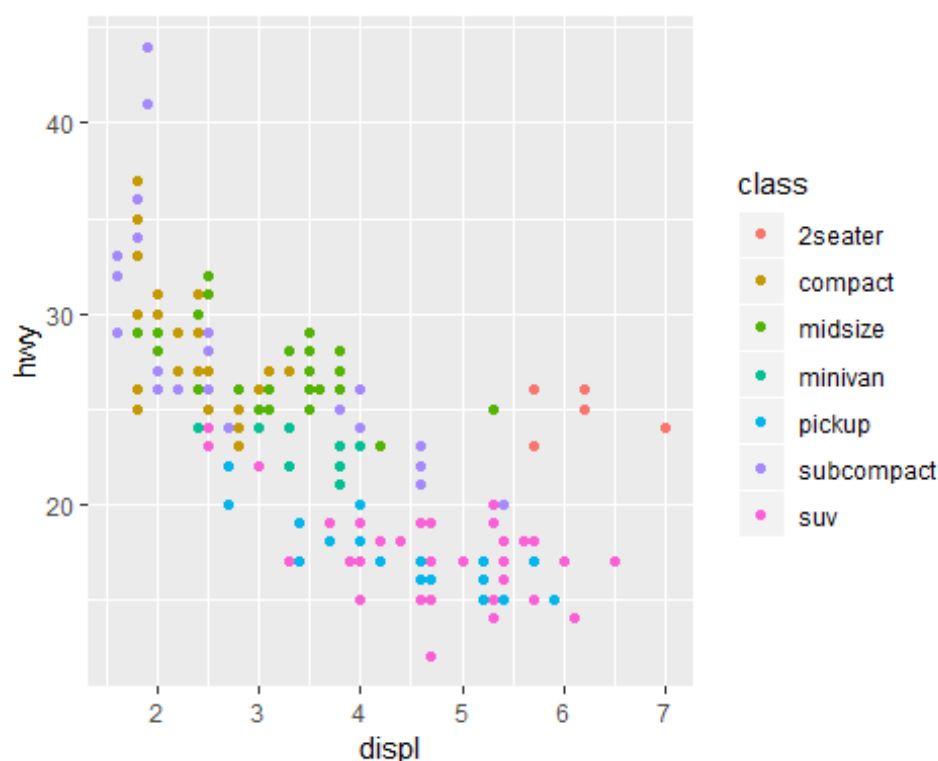
`aes()`-funktionen och mappings

I grafen nedan förefaller en liten grupp av bilar att ha högre bränsleeffektivitet än förväntat vid en linjär trend (inringade).



Är det en särskild grupp av bilar? Vi kan undersöka det genom att använda variabeln `class`. Den innehåller information om biltyper kategoriserade som *compact*, *midsize* etc. Vi kan lägga till en tredje variabel till en tvådimensionell scatterplot genom att mappa den till ett estetiskt argument, en *aesthetic*. Det är en visuell egenskap hos ett objekt i grafen och inkluderar storlek, form eller färg hos objektet, t.ex. punkterna i grafen. Man kan förändra dessa egenskaper genom att ange olika värden för *aesthetics*. Detta kan vi utnyttja för att gruppera sambandet mellan bränsleeffektivitet och motorstorlek efter biltyp. Vi använder variabeln `class` för att färga punkterna i grafen efter biltyp.

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



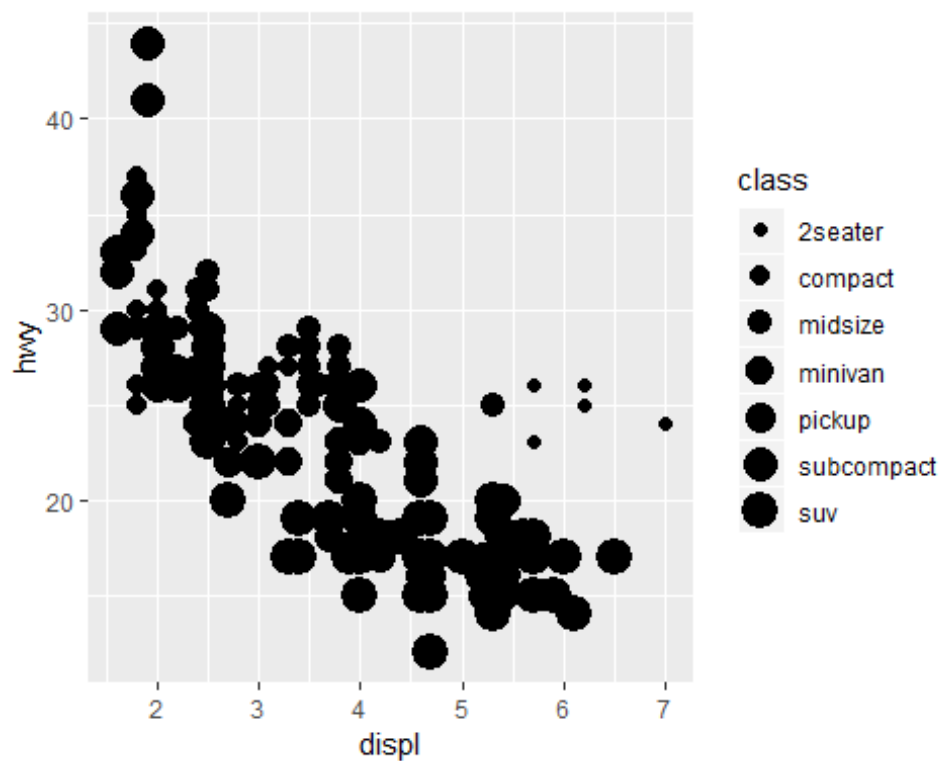
För att mappa en *aesthetic* till en variabel ska man associera namnet på *aesthetic* till variabeln och göra det *innanför* parenteserna i funktionen `aes()`. `ggplot2` kommer då att fördela en unik nivå av *aesthetic* (i detta fall en unik färg) till varje unikt värde på variabeln. Denna process kallas i `ggplot2` för *scaling*. `ggplot2` kommer också automatiskt att göra en teckenförklaring (*legend*) som förklarar vilka nivåer som är associerade med vilka variabelvärden.

Färgerna påvisar att flera av de extremare värdena är tvärsitsiga bilar, i själva verket sportbilar (stira motorer men relativt lätta).

Vi mappade classtill färg (`color`), men vi kan även mappa `class` till andra *aesthetics*, t.ex. storlek (`size`). I detta fall skulle punkt-storleken visa vilken *class* observationen tillhör. Vi får en varning här eftersom det oftast är en sämre idé att mappa en nominal variabel till en icke-nominal skala, dvs en skala som är rangordnad (som t.ex. `size`).

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, size = class))
```

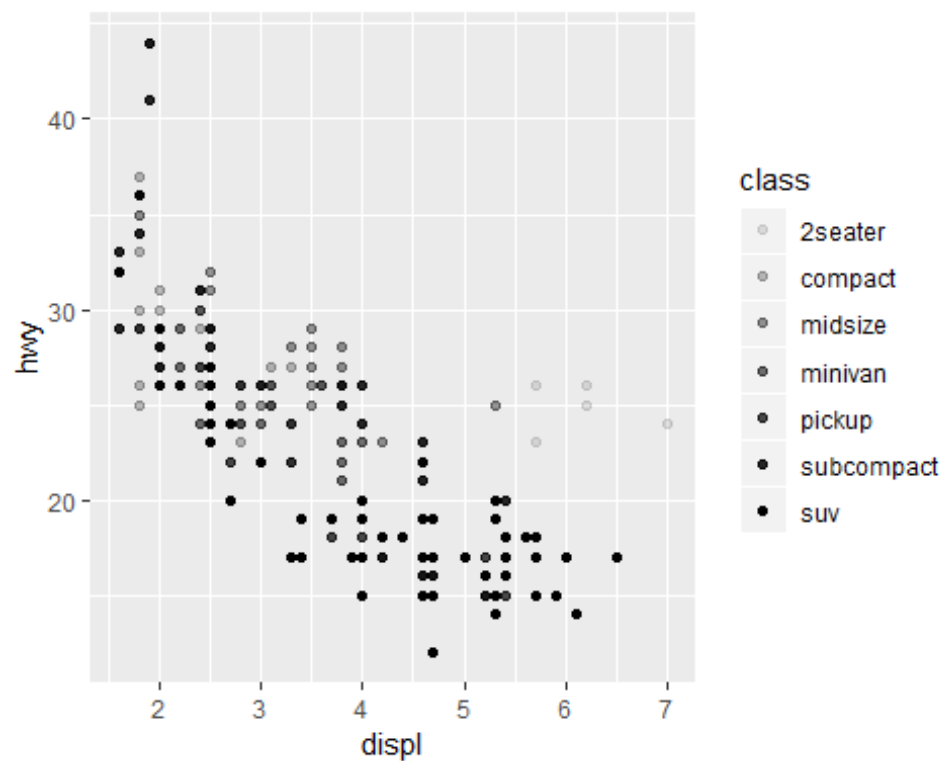
```
## Warning: Using size for a discrete variable is not advised.
```



Eller så kan vi mappa till alpha för att kontrollera graden av genomskinlighet,

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, alpha = class))
```

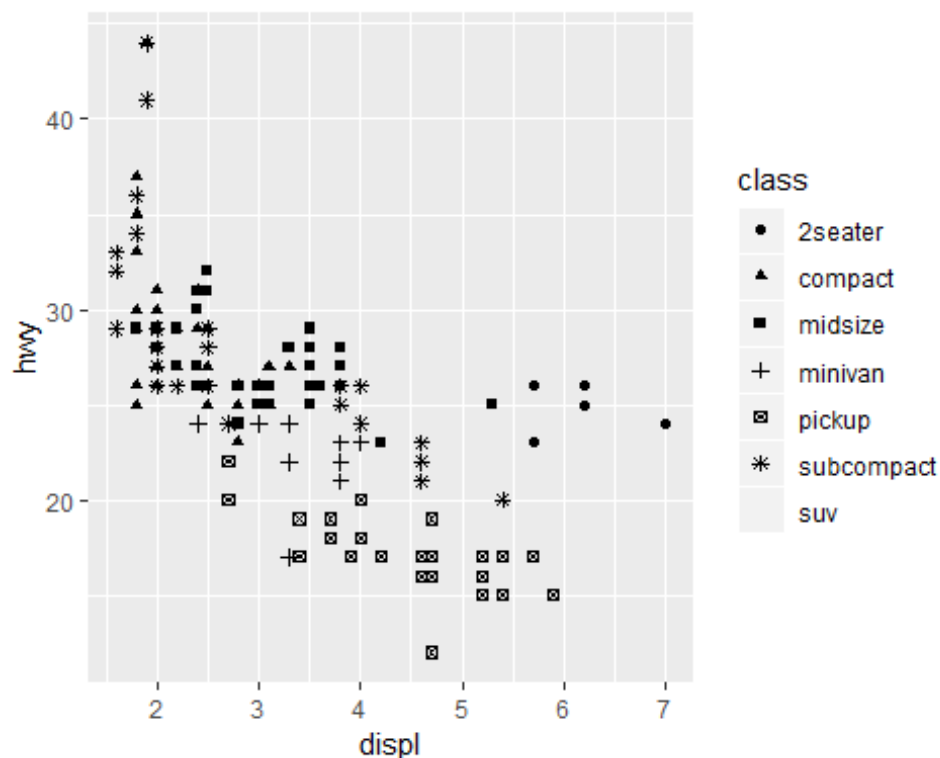
```
## Warning: Using alpha for a discrete variable is not advised.
```



eller till punkternas typ (shape):

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```

Warning: The shape palette can deal with a maximum of 6 discrete values
 ## because more than 6 becomes difficult to discriminate; you have 7.
 ## Consider specifying shapes manually if you must have them.
 ## Warning: Removed 62 rows containing missing values (geom_point).

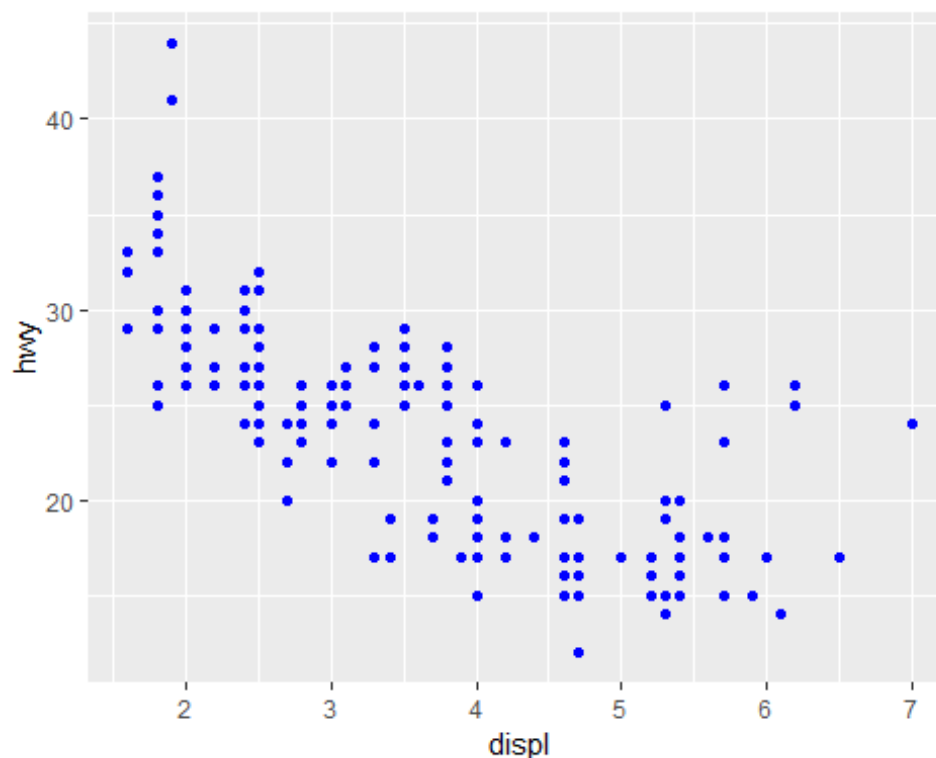


För varje *aesthetic* används `aes()` för att associera namnet på *aesthetic* med variabeln vi vill visualisera. `aes()` samlar ihop samtliga mappningar som används i ett lager (*layer*) och använder dem i lagrets mappnings-argument. Notera att `x` och `y` är i sig själva *aesthetics*, alltså visuella karakteristika som används för att visa information om data (positionen i koordinatsystemet).

När du väl mappat en *aesthetic* tar `ggplot2` hand om resten. Det väljer en rimlig skala att använda, skapar en förklaring (*legend*) till mappningen mellan kategorier och värden.

Vi kan även bestämma *aesthetic*-egenskaper manuellt. Exempelvis kan vi göra alla punkter blå:

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy), color = "blue")
```



I detta fall ger inte färgen blå någon information om en viss variabels karakteristika utan påverkar endast hur diagrammet framträder. Vi gör det genom att sätta den estetiska egenskapen som ett särskilt argument till `geom_*`-funktionen, dvs utanför `aes()`. Du behöver välja en nivå som är meningsfull för en sådan *aesthetic*:

- namnet på en färg som en textsträng (*chr*)
- storleken på en punkt i mm.
- formen på en punkt som ett nr, enligt figuren nedan

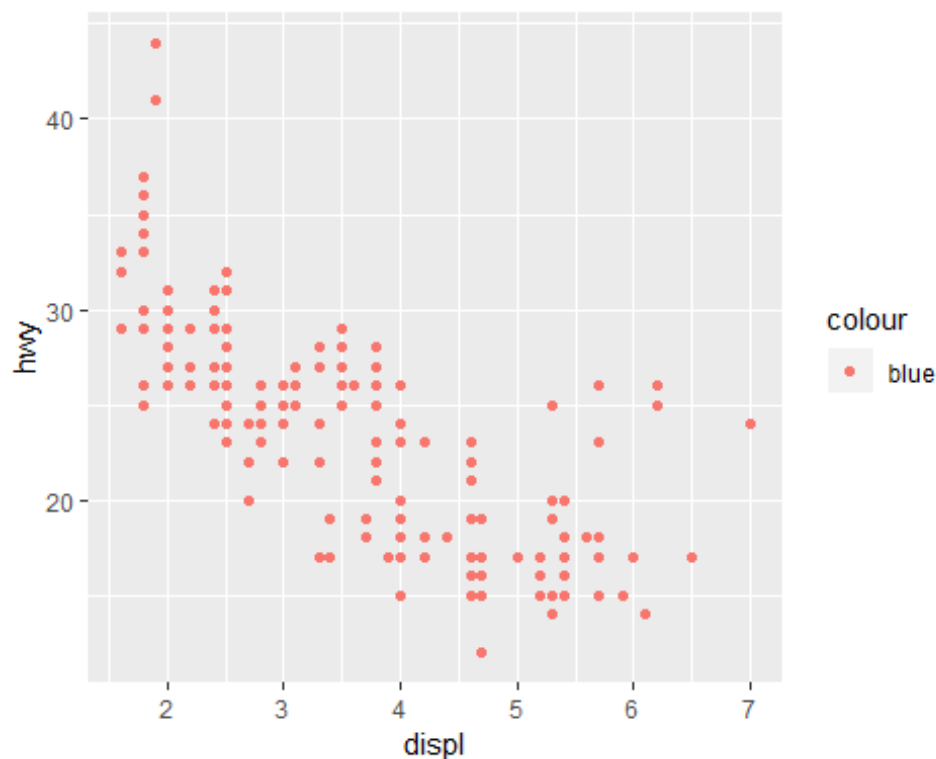
□ 0	✕ 4	⊕ 10	■ 15	■ 22
○ 1	▽ 6	⊗ 11	● 16	● 21
△ 2	⊠ 7	⊞ 12	▲ 17	▲ 24
◇ 5	✱ 8	⊗ 13	◆ 18	◆ 23
⊥ 3	⊞ 9	⊞ 14	● 19	● 20

R har 25 inbyggda former som identifieras med ett nummer 0-24. De ihåliga formerna (0-14) har en ram som bestäms av `colour`; de helfyllda formerna (15-20) fylls med `colour`; formerna 21-24 har en ram (`border`) bestämd av `colour` och fyllning definierad med `fill`. Se `?points` för detaljer.

Övningar

1. Vad är fel med nedanstående kod? Varför är inte punkterna blå?

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy, color = "blue"))
```



2. Vilka variabler i `mpg` är kategoriska? Vilka är kontinuerliga? Hur avgör man detta om man kör `mpg`? (Tips: `?mpg`)
3. Mappa en kontinuerlig variabel till `color`, `size` och `shape`. Hur beter sig dessa *aesthetics* jämfört med om man mappar med en kategorisk variabel?
4. Vad händer om du mappar samma variabel till flera *aesthetics*?
5. Vad gör `stroke`? För vilka shapes är den relevant? (Tips: `?geom_point`)
6. Vad händer om du mappar en *aesthetic* till något annat än en variabel, t.ex. `aes(colour = displ < 5)`?

Vanliga problem

När du stöter på problem - och det kommer du att göra! - börja med att kolla koden. R är extremt petigt med tecknen.

Ett vanligt misstag är att en av två parenteser saknas - kolla att varje `(` motsvaras av `)` och varje `"` med ett annat `"`.

Ibland kör man en kod men inget händer. Kolla då vänsterkanten i konsolen - om raden börjar med ett + innebär det att R inte tycker du kör ett komplett uttryck utan väntar på nästa input. Det är enkelt att komma loss från konsolen genom att trycka ESCAPE och komplettera koden.

Ett vanligt problem när man kör *ggplot2* är att sätta + på fel ställe, dvs på ny rad istället för att avsluta den tidigare raden:

```
ggplot(data = mpg)
+ geom_point(mapping = aes(x = displ, y = hwy))
```

Det kommer inte att fungera.

Om du fortfarande sitter fast, prova hjälpfunktionen. Kör `?function_name` i konsolen eller markera funktionsnamnet i koden och tryck F1. I slutet av hjälpavsnittet finns (oftast) ett antal exempel som kan vara klargörande om själva texten känns överväldigande.

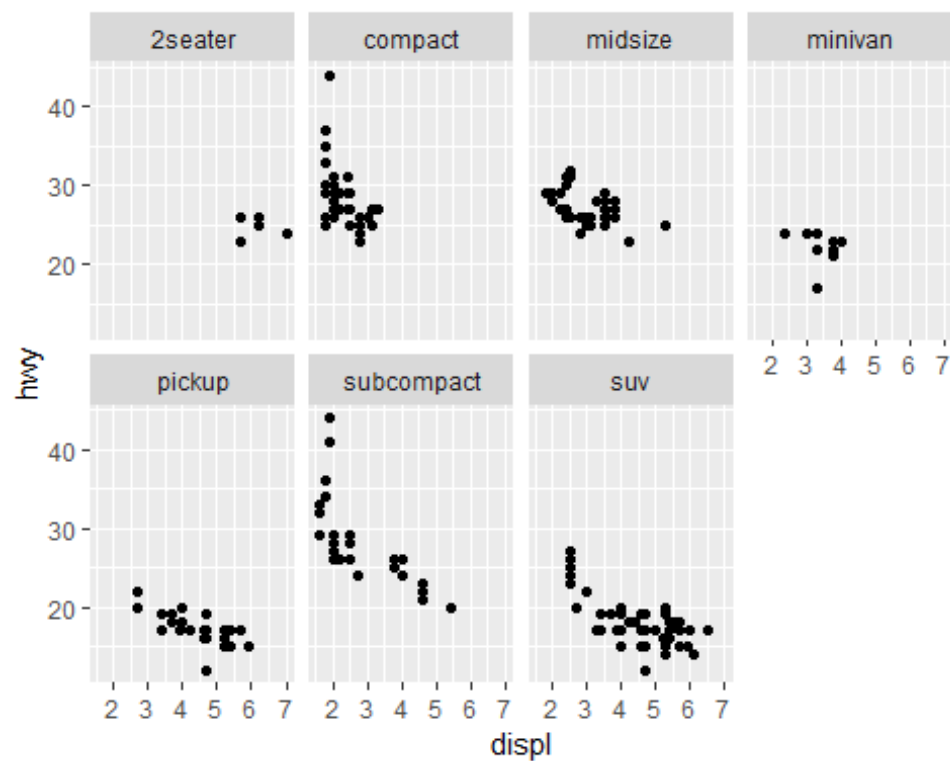
Om inte heller det hjälper, läs **error-meddelandet**. Ibland finns svaret där men ofta är det svårt att förstå. Kopiera felmeddelandet och googla det! Oftast finns någon därute som gjort exakt samma fel och fått hjälp online.

Facets

Man kan alltså addera ytterligare variabler via *aesthetics*. Ett annat sätt, särskilt med kategoriska variabler, är att dela upp grafen i **facets** - ungefär "delgrafer".

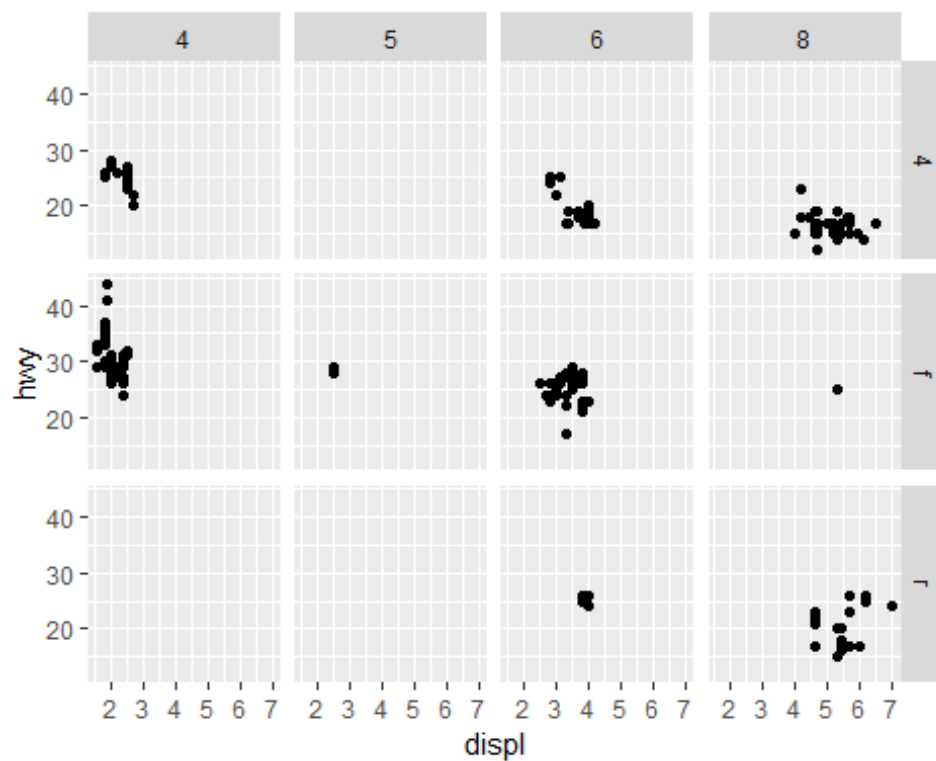
Använd `facet_wrap()`. Det första argumentet ska vara en "formel" vilket du skapar med `~` följd av variabelnamnet ("formel" är i R namnet på en *data-struktur*, inte en synonym för "ekvation"). Variabeln måste vara diskret, t.ex.:

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```



Om man vill använda två variabler för att dela upp grafen fungerar `facet_grid()`. Det första argumentet är också en formel som nu innehåller de två variabelnamnen separerade med ett tilde `~`:

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy)) +  
  facet_grid(drv ~ cyl)
```



Om du föredrar att inte dela upp grafen i rader/kolumner kan du använda `.` istället för variabelnamnet, t.ex. `facet_grid(~ cyl)`.

Övningar

1. Vad händer om man använder `facet_*`() på en kontinuerlig variabel?
2. Vilka grafer får du med följande kod? Vad gör `.`?

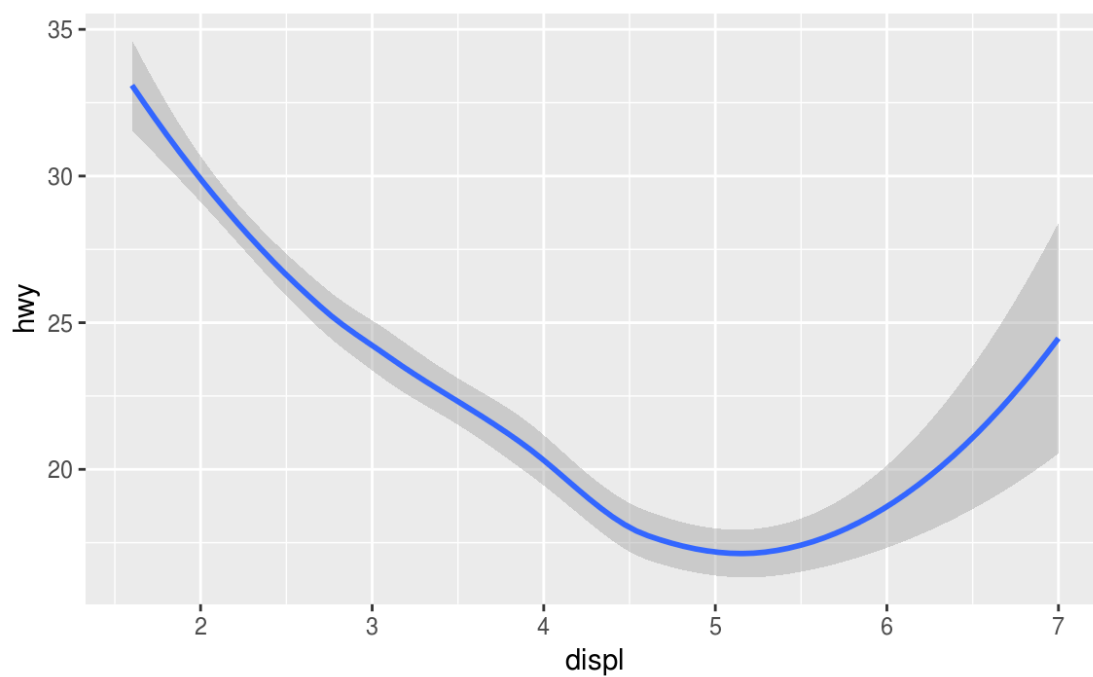
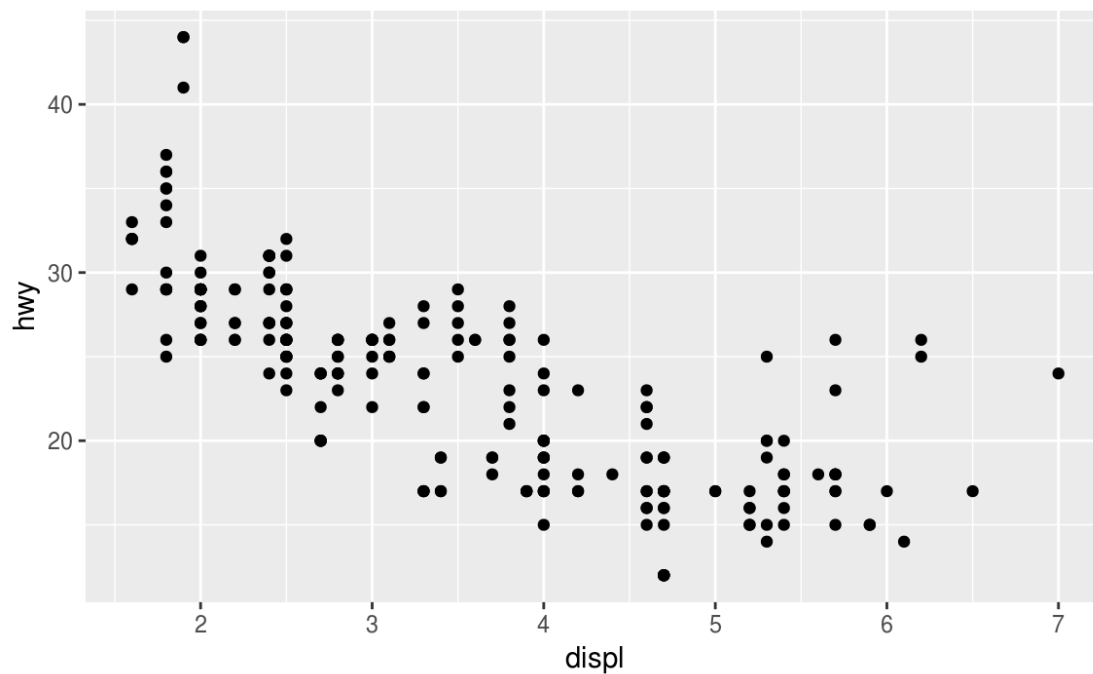
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ .)
```

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(. ~ cyl)
```

3. Läs `facet_wrap`. Vad gör `nrow`? Vad gör `ncol`?

“Geometriska” objekt - geoms

På vilket sätt är nedanstående två diagram lika?



Det är samma x-och y-variabler. Men representeras med olika visuella objekt vilka i ggplot2 syntax kallas **geoms**.

En **geom** är det geometriska objekt som används för att representera data. T.ex. används i det första diagrammet ovan `geom_point()` och i det andra `geom_smooth()`. För att ändra *geom* lägger man till ett `geom_*` till `ggplot()`. Till exempel, de två inledande diagrammen ovan gjordes med

```
ggplot(data = mpg) +  
  geom_point(mapping = aes(x = displ, y = hwy))
```

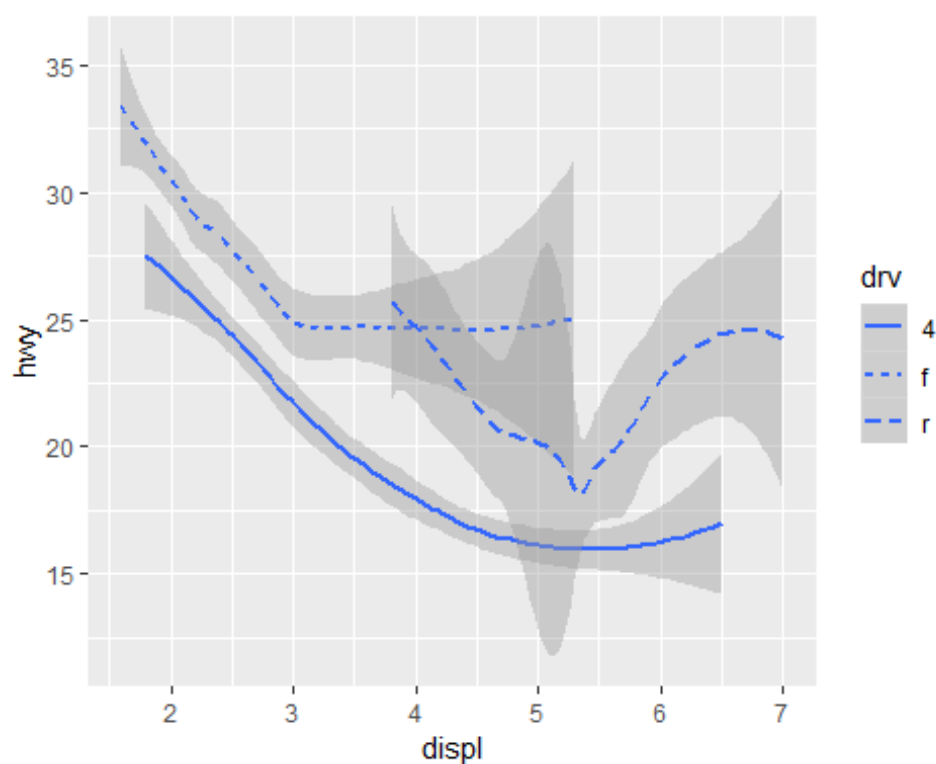
```
ggplot(data = mpg) +  
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

Prova.

Varje `geom_*()` tar ett `mapping`-argument. Men, alla typer av *aesthetics* fungerar inte med alla *geoms*. Man kan sätta en `shape` till en punkt (`point`), men man kan inte sätta en `shape` till en linje. Däremot kan man använda `linetype` för att definiera *typen* av linje. `geom_smooth()` kommer att rita en annorlunda linje för varje linje-typ, för varje unikt värde på variabeln som mappas till linje-typ:

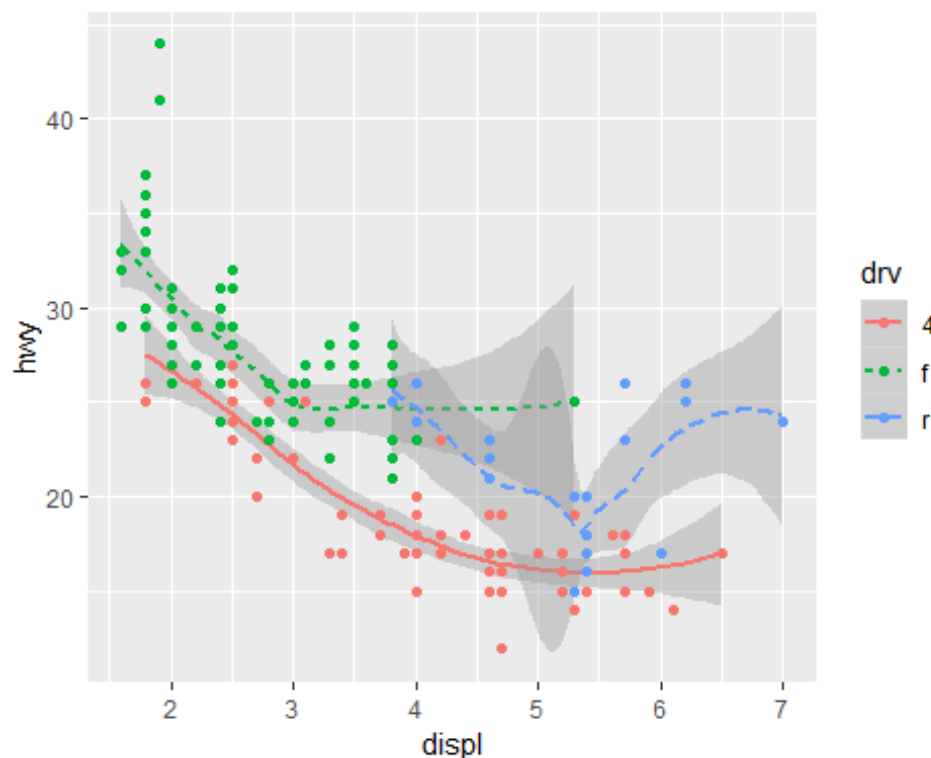
```
ggplot(data = mpg) +  
  geom_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Prova.

`geom_smooth()` delar data i tre delar beroende av värdet på `drv` (4 = fyrhjulsdrift, f=framhjulsdriven, r = bakhjulsdrift). Vi kan göra diagrammet tydligare genom att lägga till ett *punkt-lager* och använda `colour` för att särskilja `drv`.



Återkommer till detta strax.

`ggplot2` innehåller fler än 30 olika *geoms* och dessutom finns en rad *extension packs* (se <https://www.ggplot2-exts.org>). Det finns ett *cheat sheet* du kan ladda ned vilket ger en bra översikt över `ggplot2` (hämta från <http://rstudio.com/cheatsheets>).

Många *geoms* använder ett enda geometriskt objekt för att visa många data-rader. För sådan *geoms* kan du använda group-aesthetic för att rita flera objekt till en kategorisk variabel - `ggplot2` ritas då ett separat objekt för varje unik kategori av variabeln. I praktiken gör `ggplot2` detta automatiskt för sådana *geoms* närhelst man mappar en diskret variabel (som i t.ex. `linetype` ovan). Prova nedanstående kod:

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))

ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, group = drv))

ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, color = drv),
    show.legend = FALSE)
```

För att visa flera *geoms* i samma graf, lägg till dessa *geoms* till `ggplot`. Alltså:

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

Men, detta medför att vi duplicerar kod. Du kan undvika det genom att skriva en uppsättning mappings till ggplot som då kommer att hantera dessa som *globala* mappings. Alltså:

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
```

vilket ger samma graf som i föregående kod-chunk.

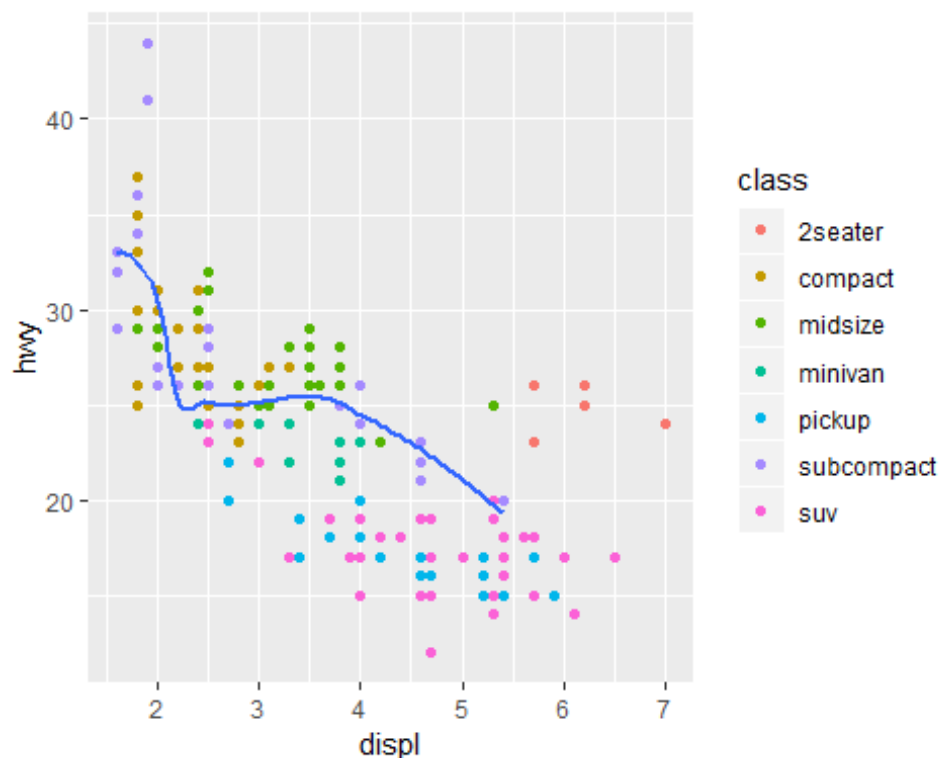
Om man placerar *mapping* i en *geom* kommer ggplot2 att överordna den globala mappningen **enbart för denna geom**. Detta gör det möjligt att ha skilda *aesthetics* i olika *geoms*.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth()
```

Du kan använda samma idé för att spec:a olika data för olika lager. I nästa exempel visar `geom_smooth()` ett subset av data, nämligen "subcompact cars". Det lokala data-argumentet i `geom_smooth()` överordnar dessa data över det globala datasetet i `ggplot()` *enbart* i `geom_smooth()`-lagret.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth(data = filter(mpg, class == "subcompact"), se = FALSE)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



(Vi återkommer till `filter`-funktionen. För nu konstaterar vi bara att funktionen filtrerar ut "subcompact"-klassen).

Övningar

1. Vilken *geom* skulle du använda för att rita ett linjediagram? En boxplot? Ett histogram? Ett areadiagram?
2. Kör nedanstående kod i huvudet och tänk ut hur output kommer att se ut. Kör sedan koden i R och kolla hur det blev:

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, color = drv)) +  
  geom_point() +  
  geom_smooth(se = FALSE)
```

3. Vad gör `show.legend = FALSE`? Vad händer om man tar bort den?
4. Vad gör argumentet `se = FALSE` i `geom_smooth()`?
5. Kommer nedanstående två kod-avsnitt ("chunks") att generera samma graf?

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +  
  geom_point() +  
  geom_smooth()  
  
ggplot() +
```

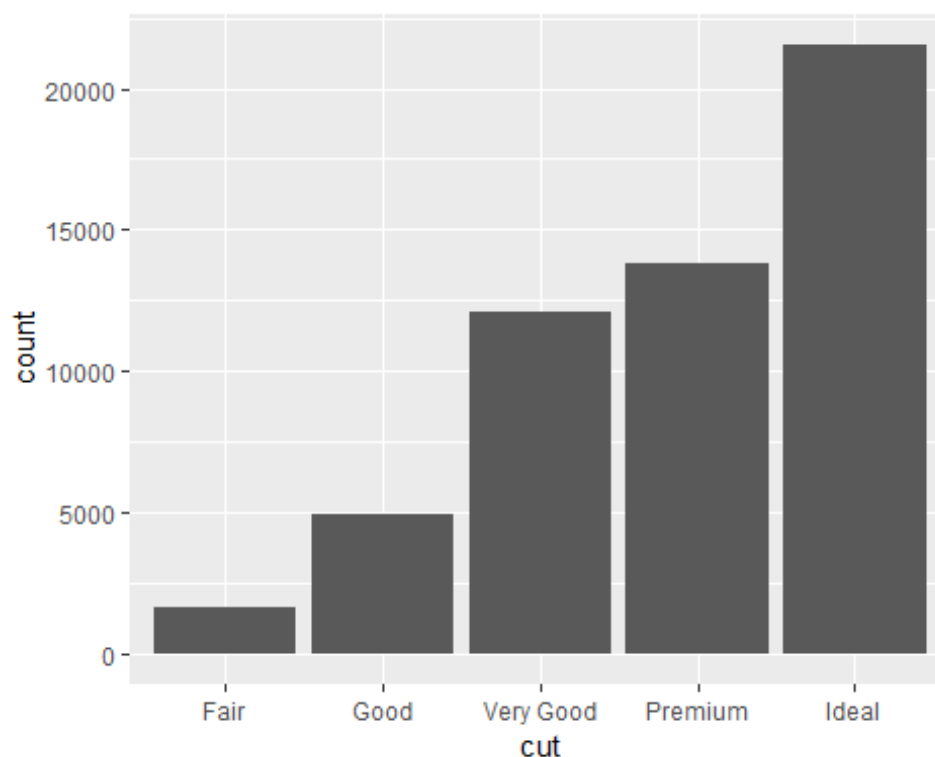


```
geom_point(data = mpg, mapping = aes(x = displ, y = hwy)) +  
geom_smooth(data = mpg, mapping = aes(x = displ, y = hwy))
```

Statistiska transformationer

Låt oss kika närmare på ett enkelt stapeldiagram, skapat med `geom_bar()`. Exemplet visar antalet diamanter i datasetet `diamonds`, grupperat efter kategorier i variabeln `cut`. `diamonds` finns inbyggt i `ggplot2` och innehåller information om ungefär 54 000 diamanter och deras `price`, `carat`, `color`, `clarity` och `cut` för varje diamanter. Stapeldiagrammet visar att det finns fler diamanter med högre `cut`-kvalitet än lägre.

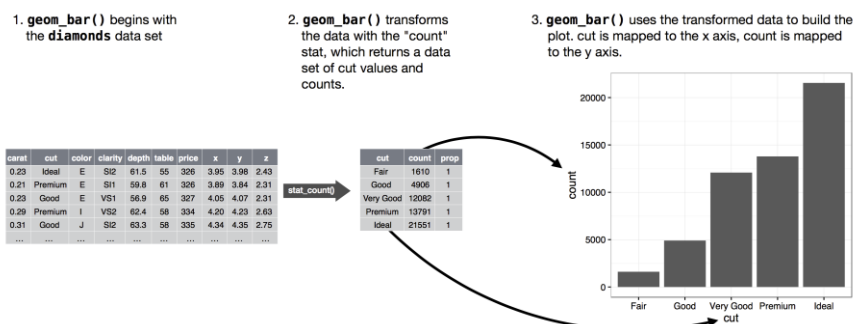
```
ggplot(data = diamonds) +  
geom_bar(mapping = aes(x = cut))
```



På x-axeln visas `cut` och på y-axeln visas antalet diamanter. Men *antal* är ju inte en variabel i datasetet - var kommer den ifrån? Många grafer använder "rå-data" från datamängden direkt medan andra, typ stapeldiagram, beräknar nya värden som plottas:

- *stapeldiagram*, *histogram* och *frekvenspolygoner* delar upp data i "bins", och räknar sedan antalet observationer i varje uppdelat "fack" (= bin)
- *smoothers* anpassar en statistisk modell till data och plottar sedan de predicerade värdena
- *boxplots* beräknar robust summa-statistik av fördelningen av värden och plottar sedan en speciellt formaterad box.

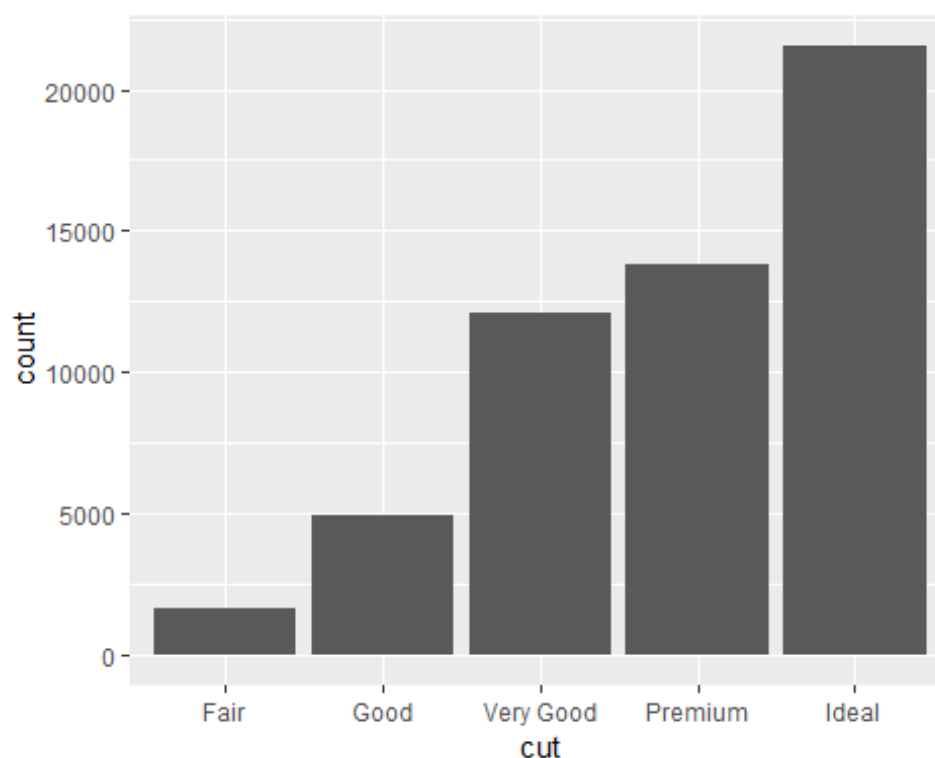
Algoritmen för hur nya värden beräknas kallas för en **stat** (förkortning för *statistical transformation*). I nedanstående figur visas hur den processen fungerar för `geom_bar()`.



Vilken statsom är default för vilken *geom* framgår av hjälp-funktionen. Från t.ex. `?geom_bar()` framgår att default för stat är *count* vilket innebär att `geom_bar()` använder `stat_count()`.

Geoms och *stats* är generellt utbytbara. Man kan alltså reproducera ovanstående graf genom att använda `stat_count()` istället för `geom_bar()`.

```
ggplot(data = diamonds) +
  stat_count(mapping = aes(x = cut))
```



Detta fungerar eftersom varje *geom* har en default *stat* och varje *stat* har ett default *geom*. Det finns tre skäl till att vilja använda ett specifikt *stat*:

1. om du vill definiera ett annat *stat* än default. I följande exempel är *stat* ändrat från *count* till *identity* vilket innebär att *ggplot()* använder data som redan finns i tabellen snarare än att räkna antalet rader (=observationer):

```
library(tidyverse)
```

```
demo <- tribble(
```

```
  ~cut, ~freq,
```

```
  "Fair", 1610,
```

```
  "Good", 4906,
```

```
  "Very good", 12082,
```

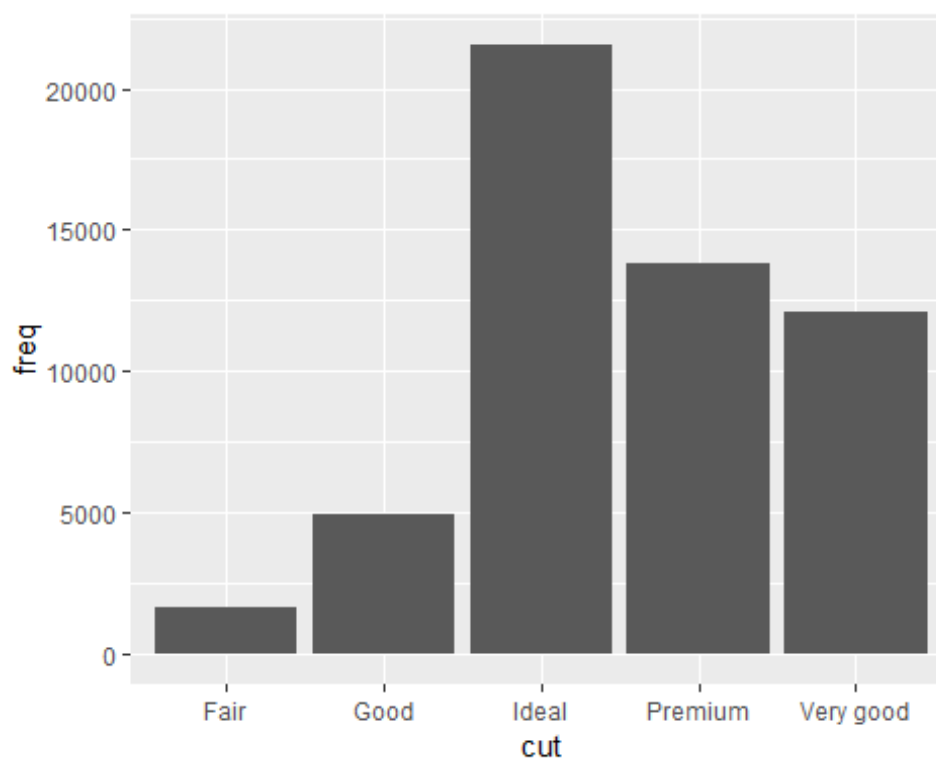
```
  "Premium", 13791,
```

```
  "Ideal", 21551
```

```
)
```

```
ggplot2::ggplot(data = demo)+
```

```
  geom_bar(mapping = aes(x = cut, y = freq), stat = "identity")
```



```
demo
```

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

```
##   cut    freq
```

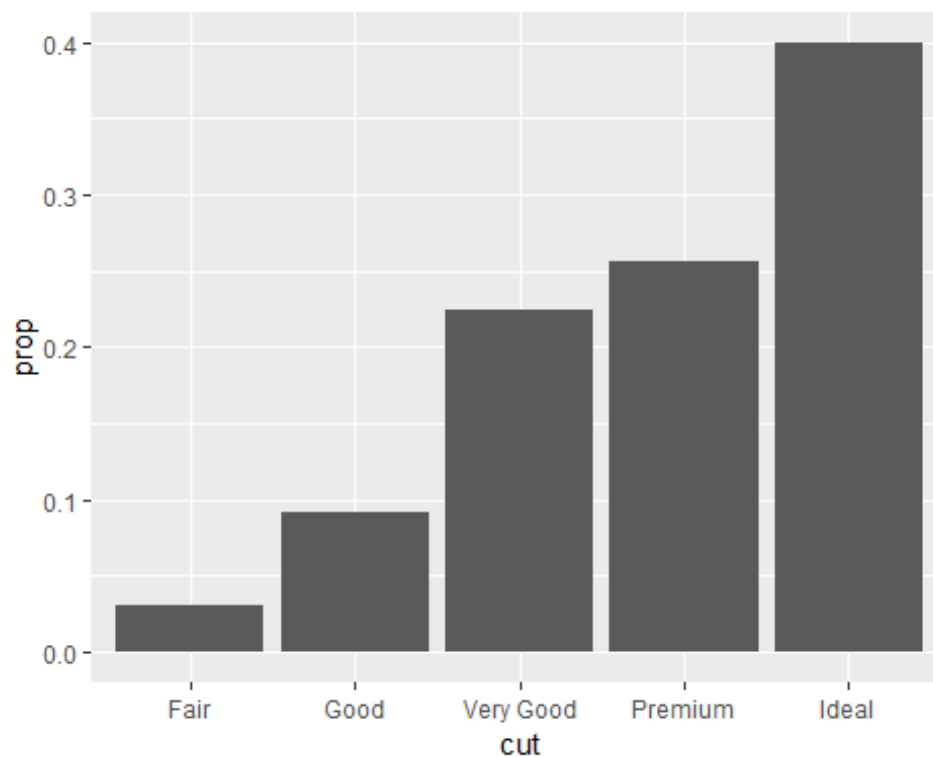
```
##   <chr> <dbl>
```

```
## 1 Fair  1610
```

```
## 2 Good    4906
## 3 Very good 12082
## 4 Premium 13791
## 5 Ideal   21551
```

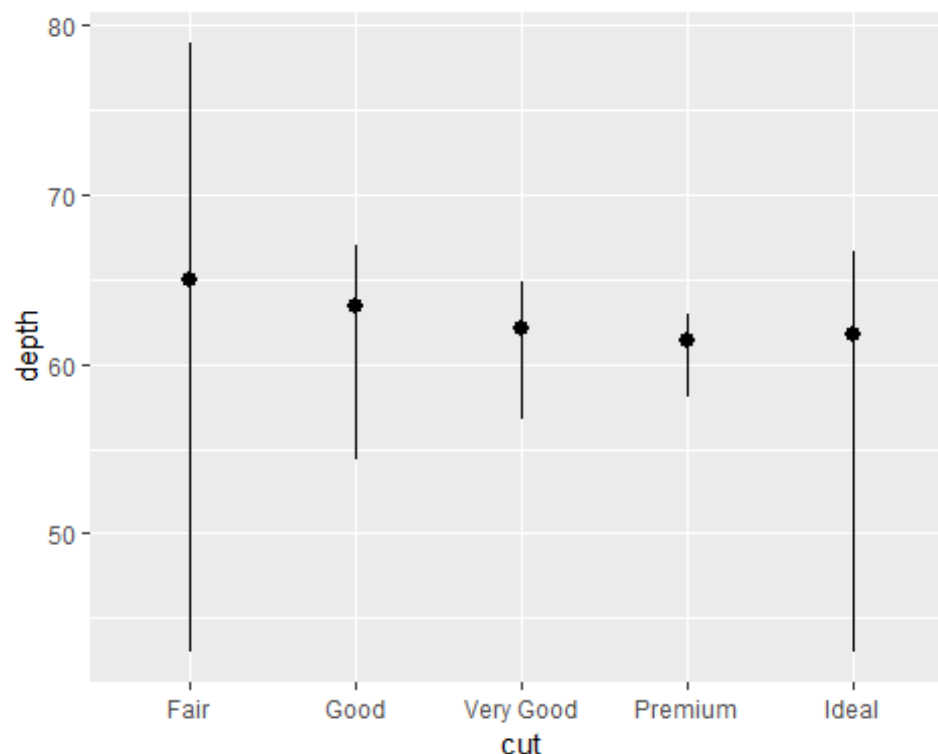
2. om du vill använda en annan *aesthetic* snarare än som transformerad variabel. T.ex. för att visa ett stapeldiagram med andelar snarare än *count*:

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop.., group = 1))
```



Kika i hjälpfunktionen för *geoms* under rubriken “Computed variables” för att se vilka variabler som beräknas för en viss *geom*. 3. om du vill förtydliga den statistiska transformationen som görs. Du kan t.ex. använda `stat_summary()` som sammanfattar y-värdet för varje unikt x-värde:

```
ggplot(data = diamonds) +
  stat_summary(
    mapping = aes(x = cut, y = depth),
    fun.ymin = min,
    fun.ymax = max,
    fun.y = median
  )
```



Det finns över 20 *stats* i *ggplot2*. varje *stat* är en funktion så man kan få hjälp på vanligt sätt, ex `?stat_bin`. För en komplett lista på tillgängliga *stats*, se [ggplot2 cheatsheet](#).

Övningar

1. Vilken default *geom* är associerad med `stat_summary()`? Hur kan du skriva om föregående kod och använda den *geom*-funktionen istället?
2. Vilka variabler skapar `stat_smooth()`? Vilka parametrar styr vad som sker då man använder denna *geom*?
3. I stapeldiagrammet med andelar behövdes `group = 1`? Varför det? Med andra ord, vad är problemet med nedanstående två kod avsnitt?

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop..))

ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = color, y = ..prop..))
```

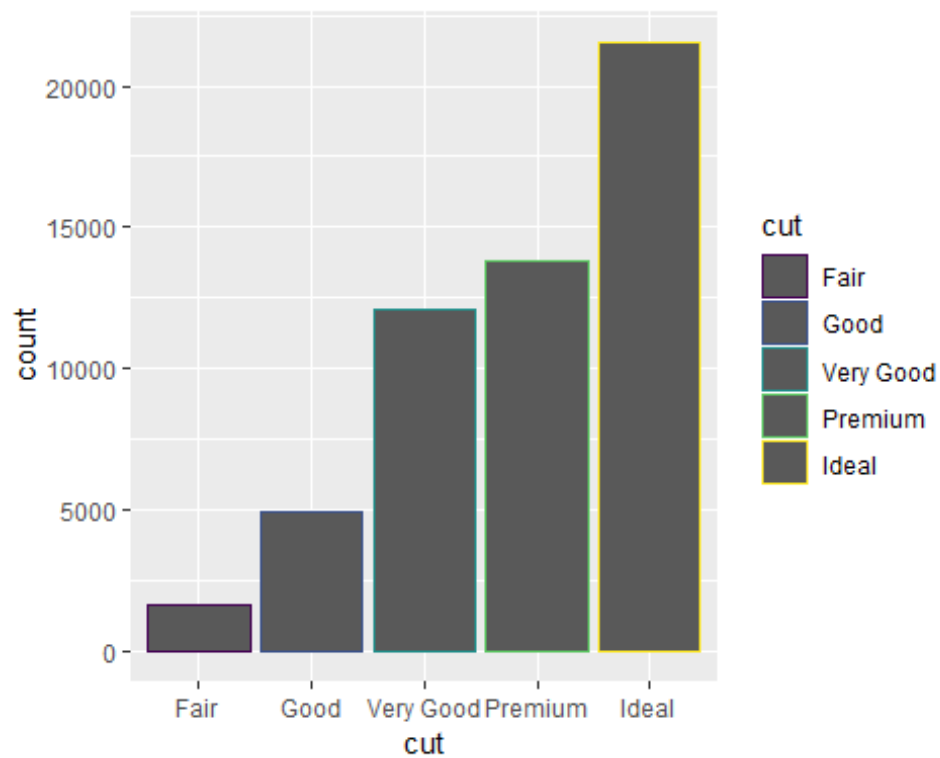
Positionering

En ytterligare komponent är associerad med stapeldiagram. Du kan färga ett stapeldiagram med antingen *colour-aesthetic* eller med argumentet `fill`.

```
library(tidyverse)
```

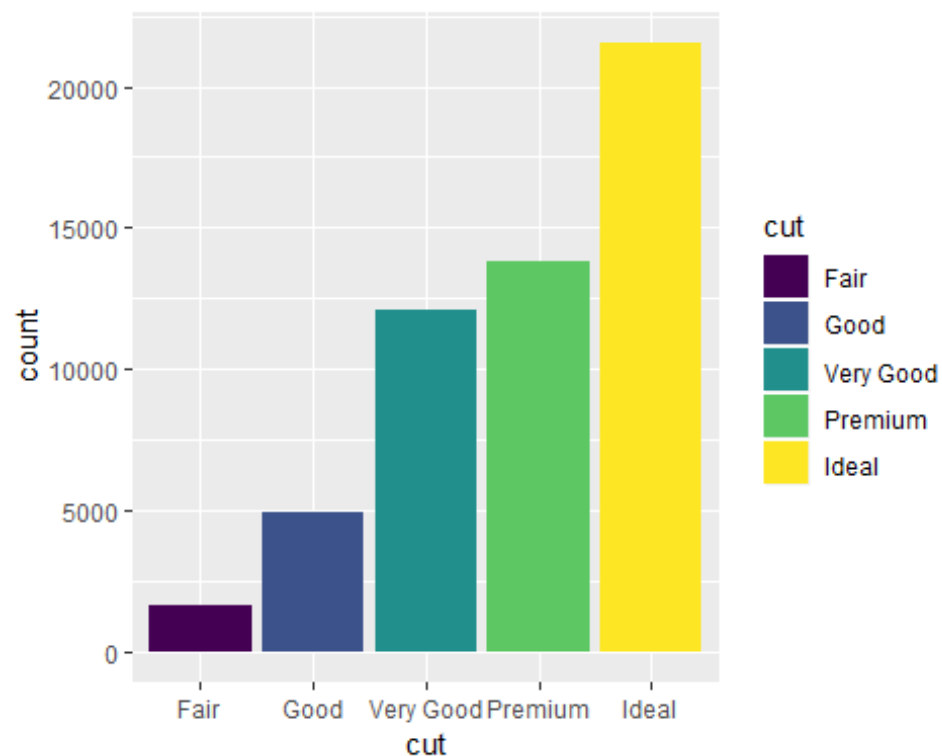
```
ggplot(data = diamonds) +
```

```
  geom_bar(mapping = aes(x = cut, colour = cut))
```



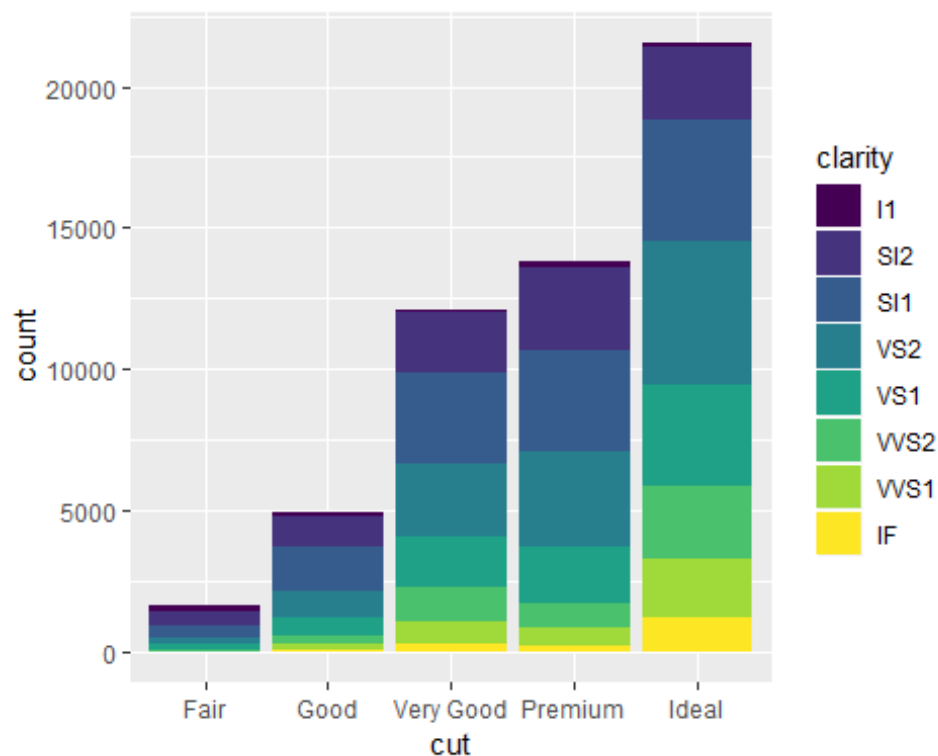
```
ggplot(data = diamonds) +
```

```
  geom_bar(mapping = aes(x = cut, fill = cut))
```



Notera vad som händer då du mappar fill till en annan variabel, t.ex. clarity: staplarna blir automatiskt lagrade, *stacked*. Varje färgad rektangel representerar en kombination av cut och clarity.

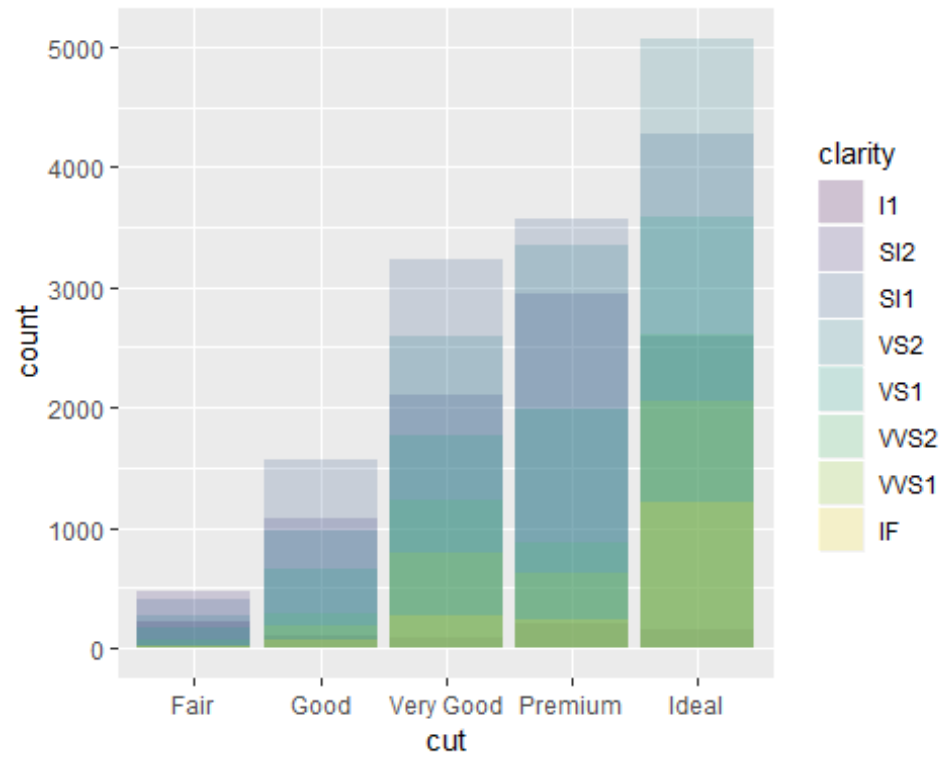
```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, fill = clarity))
```



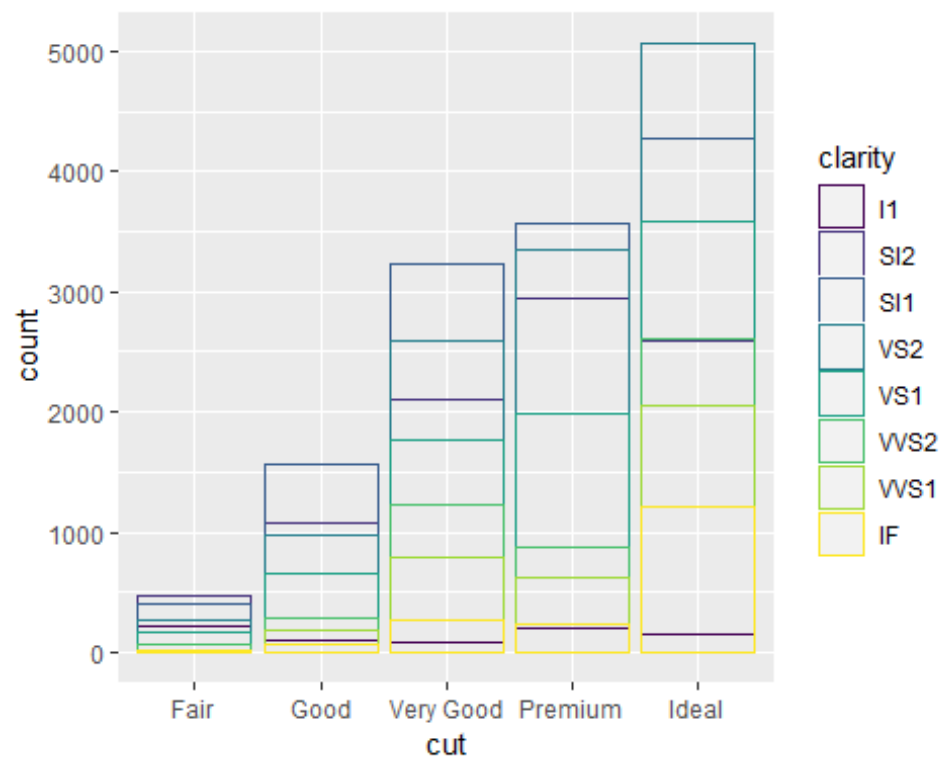
Lagringen utförs automatiskt av *the position adjustment* vilken är specificerad i argumentet `position`. Om du inte vill ha ett lagrat stapeldiagram kan du välja ett av tre andra alternativ: "identity", "dodge" och "fill".

- `position = "identity"` placerar varje objekt exakt där det hamnar i sin specifika kontext, som kan skifta beroende på val av *geom*. Det är inte särskilt användbart i ett stapeldiagram eftersom objekten kommer att överlappa varandra. Det kan vi se genom att antingen göra staplarna delvis transparenta eller helt transparenta genom argumentet `fill = NA`.

```
ggplot(data = diamonds, mapping = aes(x = cut, fill = clarity)) +  
  geom_bar(alpha = 1/5, position = "identity")
```

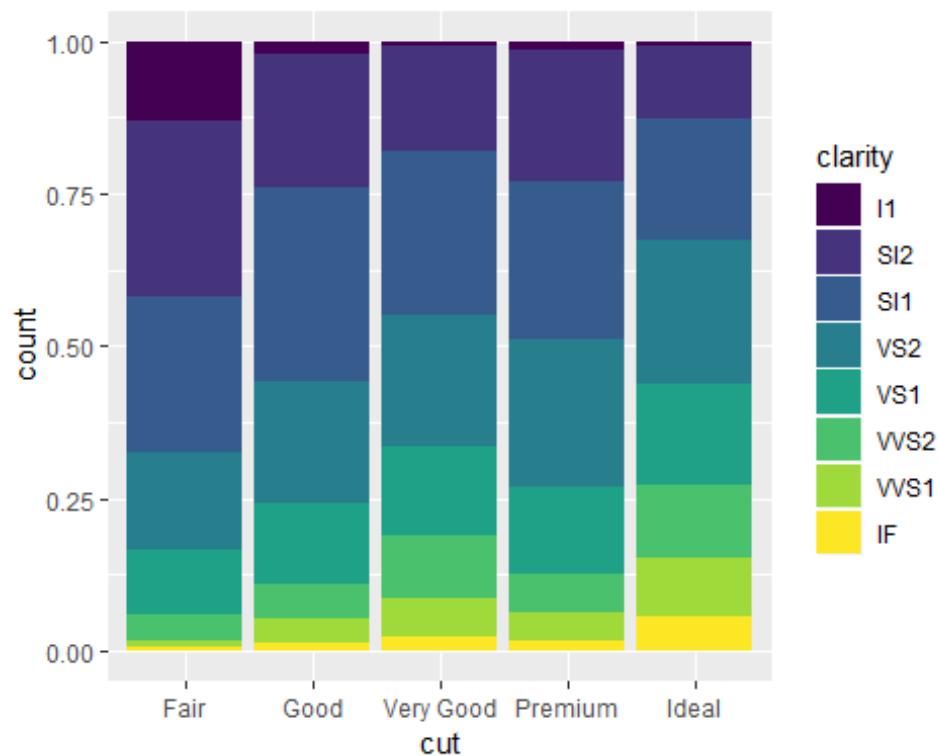



```
ggplot(data = diamonds, mapping = aes(x = cut, colour = clarity)) +  
  geom_bar(fill = NA, position = "identity")
```



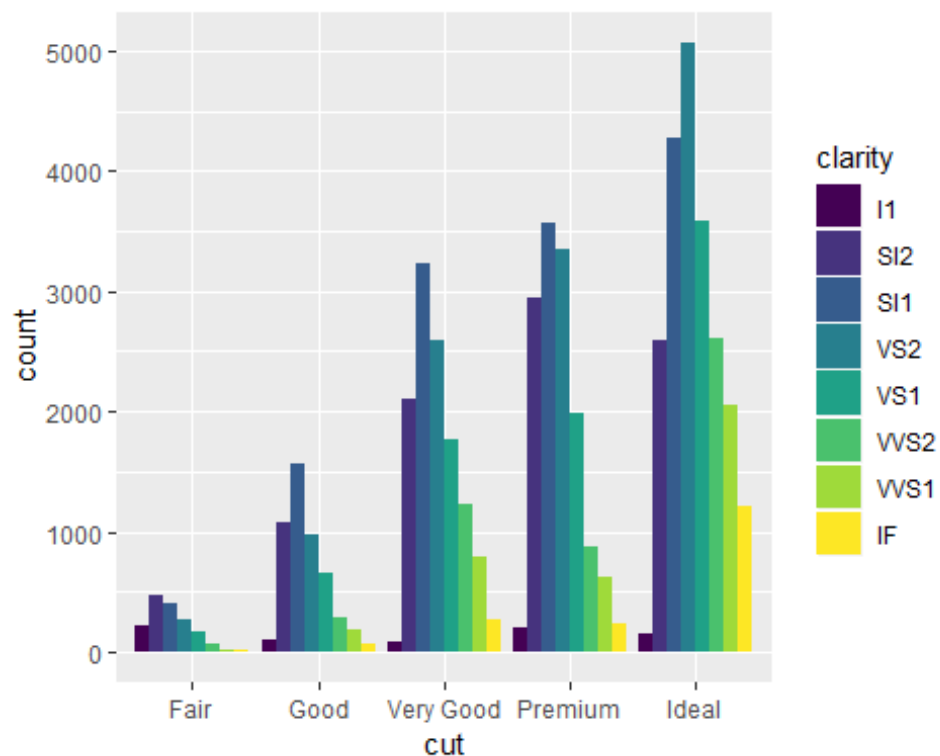
- `position = "fill"` fungerar som *stacking* men varje kategori får samma höjd. Det gör det enklare att jämföra andelar över kategorierna.

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, fill = clarity), position = "fill")
```



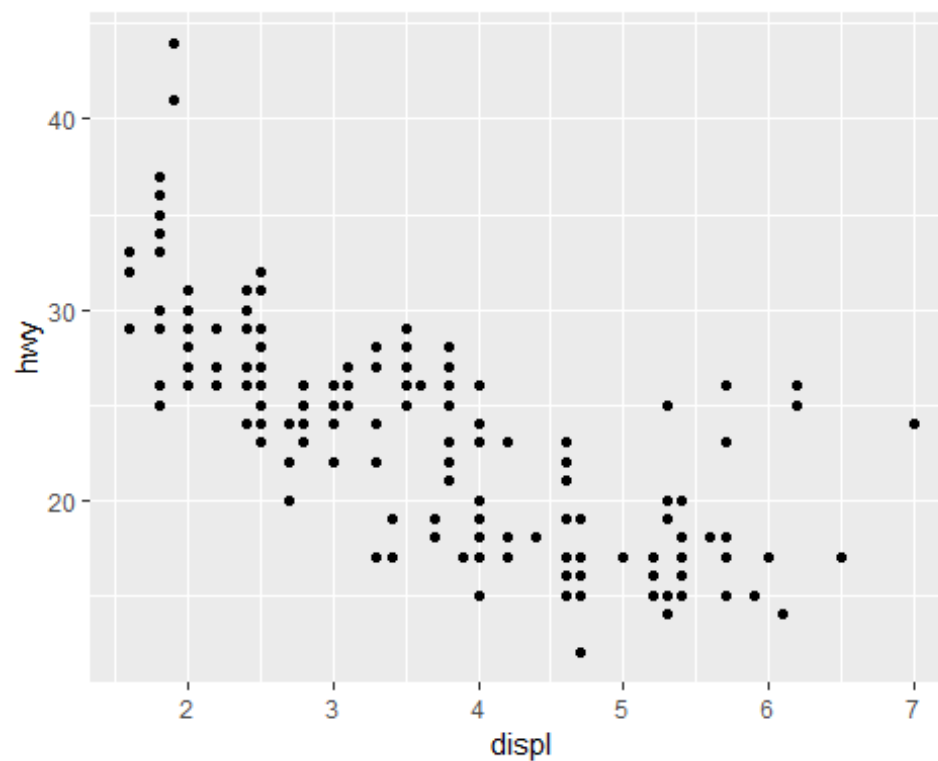
- `position = "dodge"` placerar överlappande objekt direkt vid sidan av varandra, vilket gör det lättare att jämföra individuella värden.

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut, fill = clarity), position = "dodge")
```



Det finns en annan typ av positionering som inte är relevant för stapeldiagram men som kan vara mycket användbar för scatterplots. Minns den första scatterplotten:

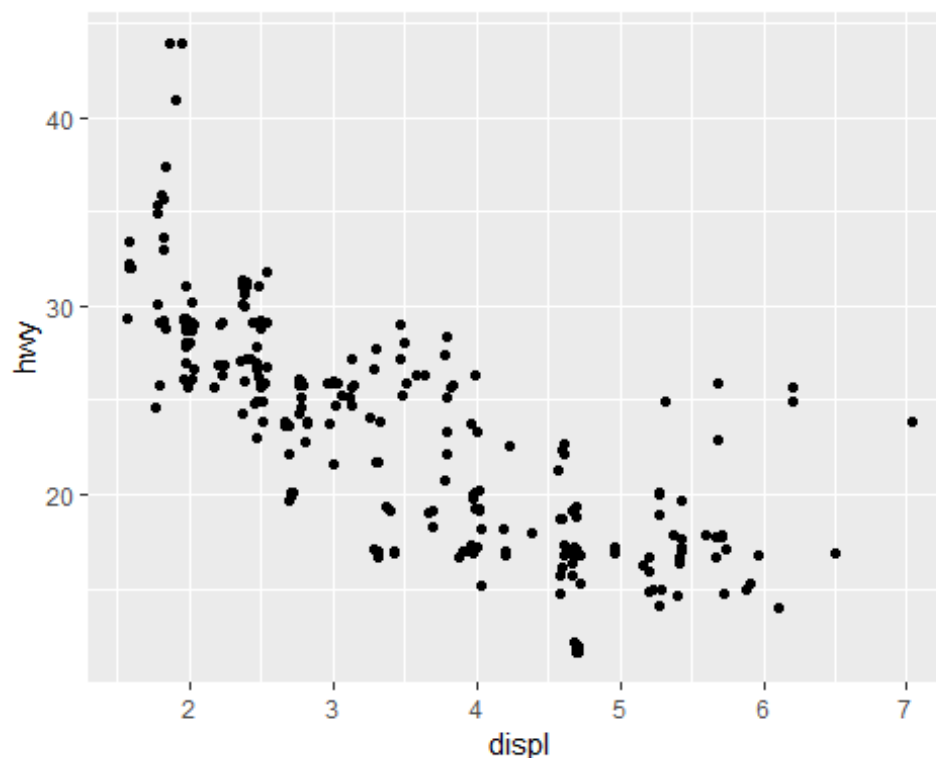
```
ggplot(data = mpg, aes(x = displ, y = hwy)) +  
  geom_point()
```



I den finns endast 126 punkter fastän det finns 234 observationer i data. Värdena på `hwy` och `displ` är avrundade så punkterna framträder på ett koordinatsystem (*grid*) och många punkter överlappar varandra, *overplotting*. Det är svårt att utifrån grafen se om punkterna är fördelade jämnt över grafen eller om en punkt döljer 109 värden.

Du kan undvika detta med hjälp av en *positionering* som kallas *jitter*. `position = "jitter"` adderar lite slumpmässigt *noise* till varje punkt. Det sprider ut punkterna eftersom det är osannolikt att två punkter tilldelas samma mängd *noise*.

```
ggplot(data = mpg, aes(x = displ, y = hwy)) +  
  geom_point(position = "jitter")
```



Eftersom detta är en så användbar positionering finns en genväg för `geom_point(position = "jitter")`: `geom_jitter()`.

Kolla gärna igenom hjälp-sidorna för dessa positioneringar: `?position_dodge`, `?position_fill`, `?position_identity`, `?position_jitter` och `?position_stack`.

Övningar

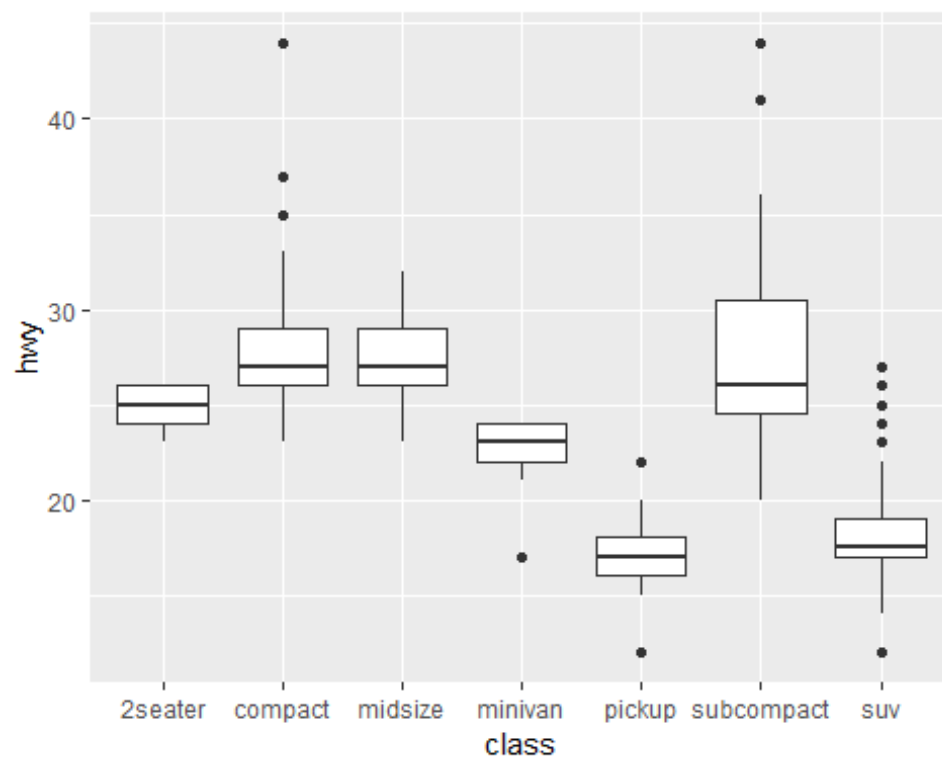
1. Vilka parametrar till `geom_jitter()` kontrollerar mängden *jitter*?
2. Jämför `geom_jitter()` med `geom_count()`. 3. Vilken är default positionering i `geom_boxplot()`?

Koordinatsystem

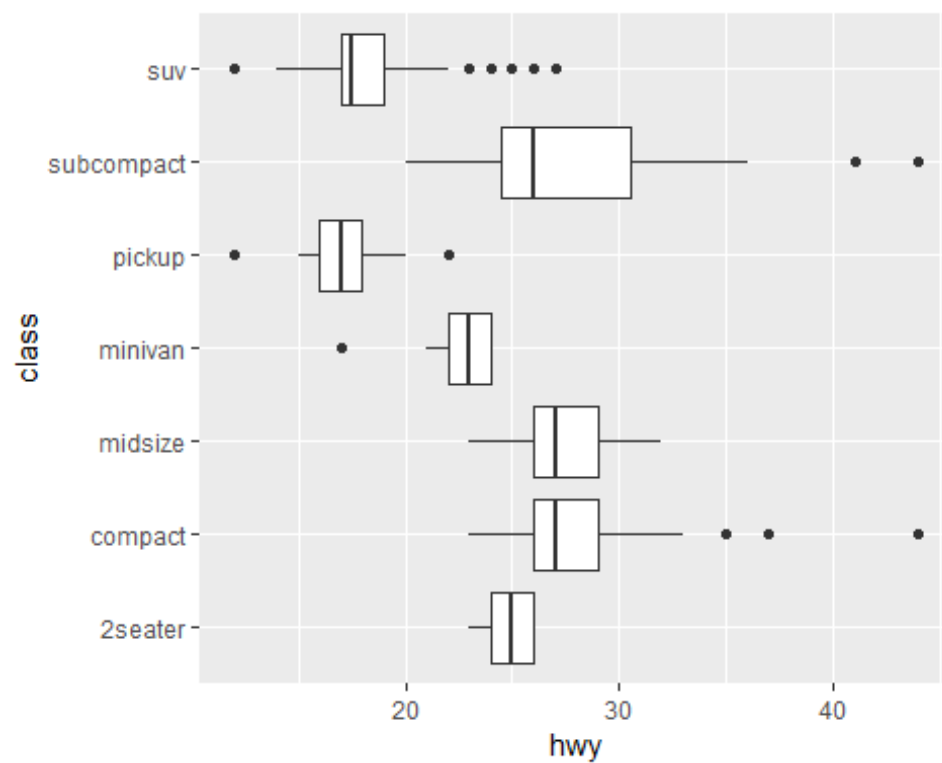
Koordinatsystem är förmodligen den mest komplicerade delen av ggplot2. Default är *Cartesian* där x- och y- positionerna bestämmer positionen för varje punkt. Det finns emellertid ett antal andra koordinatsystem som är användbara för olika ändamål.

- `coord_flip()` växlar x- och y-axlarna. Detta är användbart t.ex. om man vill ha horisontella boxplots eller om man har långa etiketter på x-axeln.

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +  
  geom_boxplot()
```



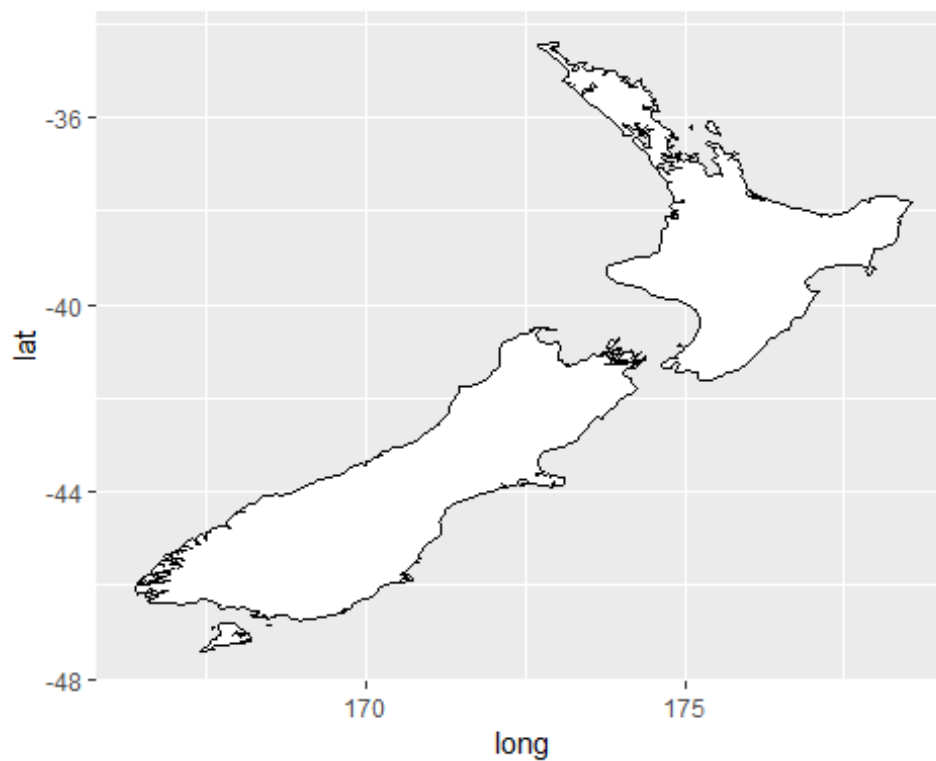
```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +  
  geom_boxplot() +  
  coord_flip()
```



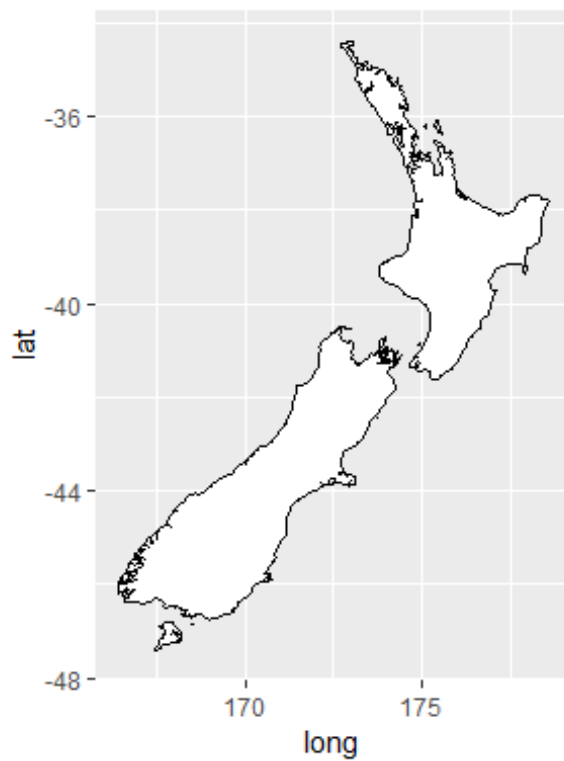
- `coord_quickmap()` sätter korrekt *aspects* för kartor. detta är förstås viktigt om man plottar geo-data med ggplot2:

```
nz <- map_data("nz")
```

```
ggplot(nz, aes(long, lat, group = group)) +  
  geom_polygon(fill = "white", colour = "black")
```



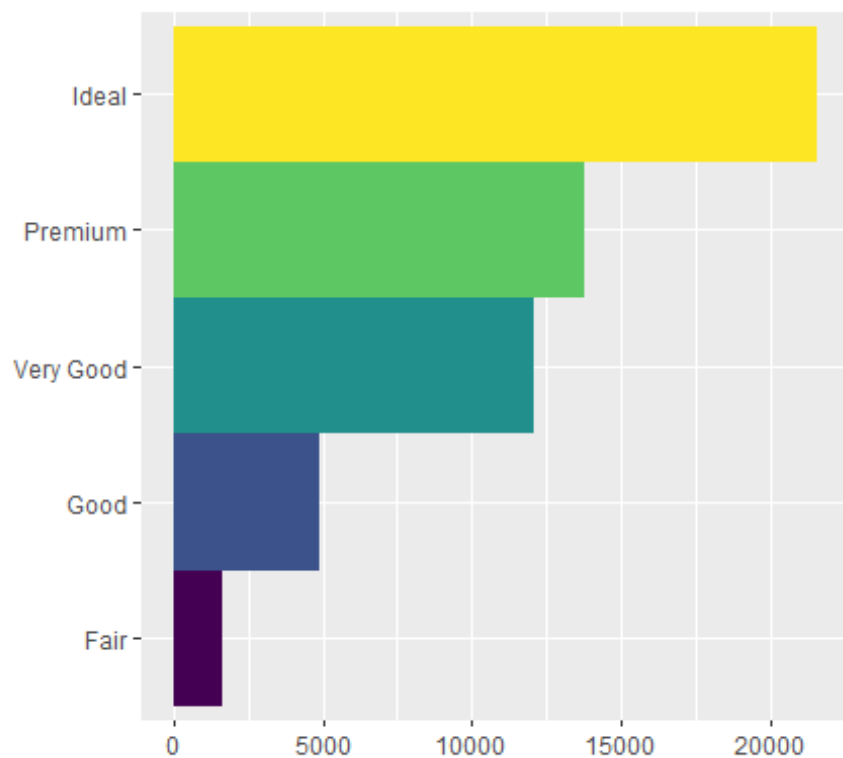
```
ggplot(nz, aes(long, lat, group = group)) +  
  geom_polygon(fill = "white", colour = "black") +  
  coord_quickmap()
```



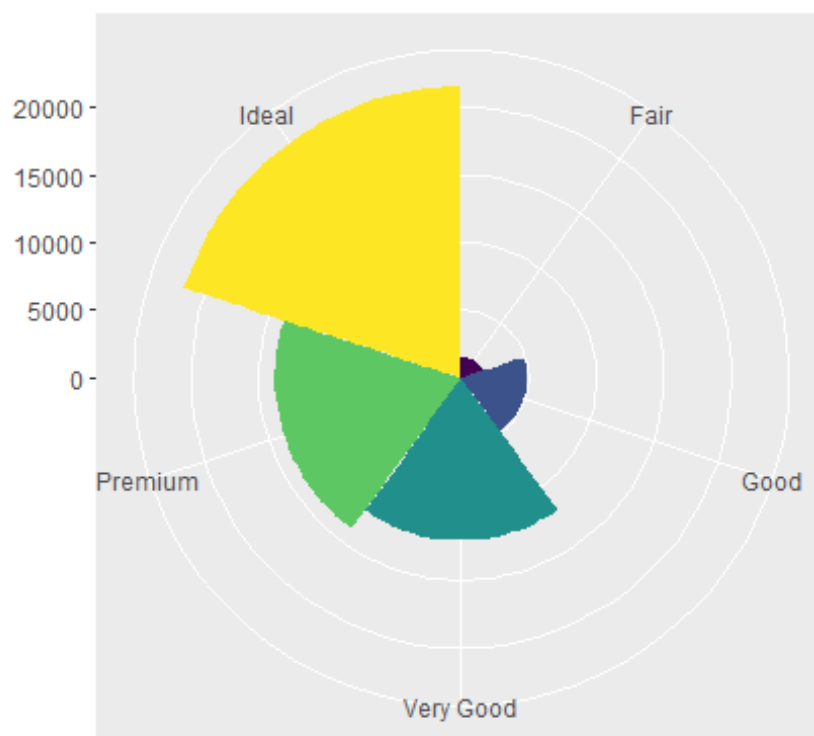
- `coord_polar()` används för polära koordinater.

```
bar <- ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = cut),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)

bar + coord_flip()
```

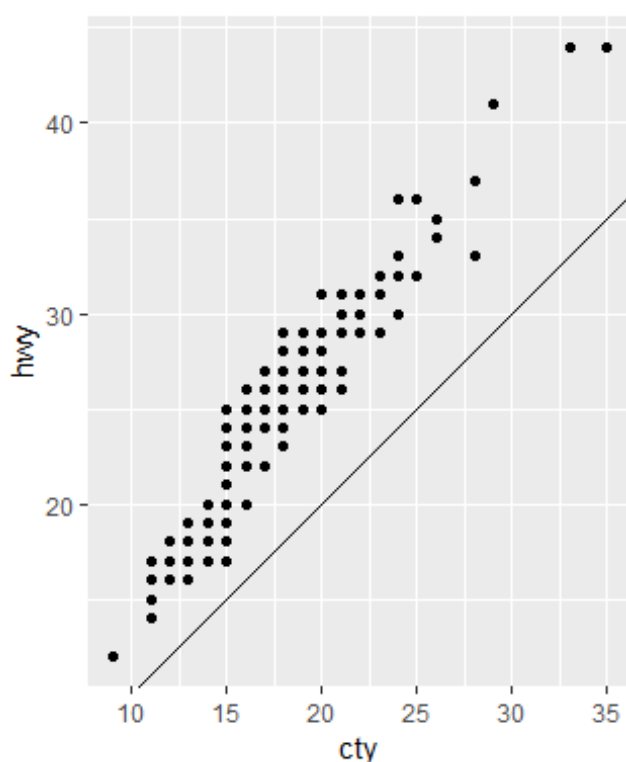
```
bar + coord_polar()
```



Övningar

1. Gör om ett lagrat (*stacked*) stapeldiagram till en pie-chart med hjälp av `coord_polar()`.
2. Vad gör `labs()`?
3. Vad är skillnaden mellan `coord_quickmap()` och `coord_map()`?
4. Vad säger scatterplotten nedan om relationen mellan *city* och *highway mpg*? Varför är `coord_fixed()` viktig? Vad gör `geom_abline()`?

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +  
  geom_point() +  
  geom_abline() +  
  coord_fixed()
```



“The layered grammar of graphics”

Som förhoppningsvis framgår av ovanstående exempel finns i *ggplot2* en slags grammatik som bygger på att man adderar olika *lager* till en kod eller script för att göra ett diagram. Vi har gått igenom sju parametrar vilka tillsammans definierar diagrammet:

1. `data`
2. `geom_*`
3. `mappings`
4. `stat`

5. `position`
6. `coord_*()`
7. `facet_*()`

I praktiken behöver man sällan ange samtliga sju eftersom ggplot2 har fungerande defaults för samtliga utom data, mappings och geoms.

De sju parametrarna utgör "grafikens grammatik", ett formellt system för att bygga upp grafer. Du startar med ett dataset och transformerar det till den information du vill visa (görs med en *stat*).

1. Begin with the **diamonds** data set

2. Compute counts for each cut value with **stat_count()**.

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	Si2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75

stat_count()

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

Sedan väljer du ett *geom* för att representera varje observation i det transformerade data. Du mappar sedan varje variabelvärde till en *aesthetic*.

3. Represent each observation with a bar.

4. Map the **fill** of each bar to the **..count..** variable.

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	Si2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75

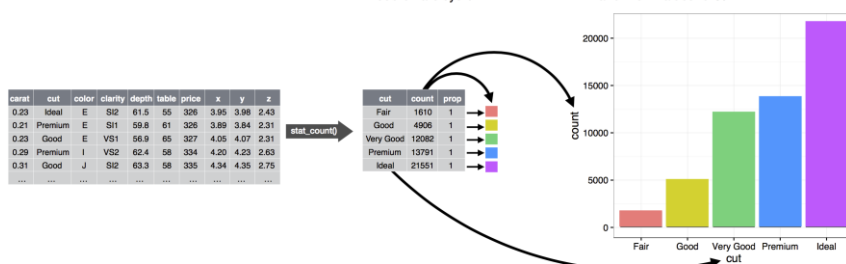
stat_count()

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

Därefter väljer du ett koordinatsystem att placera *geoms* i. Du använder positionen av dessa objekt för att visa värdena av x- och y-variablerna. Här är grafen i sig komplett men du kan justera positionerna av *geoms* inom koordinatsystemet (*position*) eller dela upp grafen i del-grafer (*faceting*). Du kan utvidga grafen med ytterligare lager.

5. Place geoms in a cartesian coordinate system.

6. Map the y values to **..count..** and the x values to **cut**.



Du kan använda denna metod för att bygga i princip vilken graf som helst.

Workflow: Basics

Skriva kod

All kod som tilldelar ett värde till ett objekt, *assignment statements*, har samma form i R:

```
object_name <- value
```

Det utläses "object_name får ett värde". En tilldelning av värde markeras med `<-`. Det är inte särskilt bekvämt att skriva. Man kan även använda `=` men rutinerade R-användare rekommenderar av olika skäl att hålla fast vid `<-`. Här kan du använda Rstudios snabbtangenter `Alt + -` (minustecknet).

Objektnamn måste börja med en bokstav och kan endast innehålla bokstäver, siffror och `_` samt `.` (punkter). Du kan visa ett objekt genom att ange dess namn:

```
x=12
x
## [1] 12
```

Rstudio innehåller många stödfunktioner. Ge ett långt namn till ett objekt och visa det:

```
detta_är_ett_riktigt_långt_namn <- 2.5
```

Om du nu börjar skriva objektnamnet (detta) och trycker TAB så föreslår Rstudio de objekt som börjar med "detta". Tryck ENTER för att välja det.

Om du gjorde fel och objektet istället skulle vara 3.5: börja skriv objektets namn och tryck `Ctrl + uppåtpil`. Då listas de senaste kommandon du använt som börjar med de tecken du matat in nyss, i exemplet "detta". Använd piltangenterna för att hitta rätt kommandorad och justera till 3.5.

Observera att R är känsligt för versaler/gemener - viktigt att vara exakt då man skriver kommandon, annars kommer ett felmeddelande.

Anropa funktioner

R innehåller en mängd inbyggda funktioner vilka kan användas genom att ange deras namn och argument:

```
function_name(arg1 = värde1, arg2 = värde2, ...)
```

`seq()` är en funktion som bildar regelbundna sekvenser av sifferföljder. Börja skriv "se" och tryck TAB så kommer en lista på objekt som börjar på "se". Tryck `q` för att reducera antalet alternativ, eller använd `upp/nerpil` för att välja. Notera att det kommer upp en *tooltip* som förklarar mer om objektet, i detta fall funktionen `seq()` och vad funktionen åstadkommer. Om man trycker F1 kommer en detaljerad hjälp upp till rutan nedtill höger. Tryck TAB en gång till och Rstudio kompletterar objektet, i detta fall med

parenteser (som markerar att objektet är en funktion). Prova med argumenten 1, 10, och tryck ENTER, alltså

```
seq(1, 10)
```

Skriv följande textrad och notera att du får samma stöd med citat-tecknen i par:

```
X <- "hello world"
```

Det är viktigt att citat-tecken och parenteser kommer i par. Lätt att missa och om det inträffar kommer R dels att ge ett error, dels att visa fortsättnings-tecknet “+” i konsolen nedtill vänster:

```
X <- "hello
## Error: <text>:1:6: unexpected INCOMPLETE_STRING
## 1: X <- "hello
## 2:
##      ^
```

+ -tecknet i början på raden i konsolen talar om för dig att R väntar på mer input, oftast beroende på att du missat ett " eller).

Åter till `seq()`:

```
Y <- seq(1, 10, length.out = 5)
Y
## [1] 1.00 3.25 5.50 7.75 10.00
```

Om man omger hela uttrycket med parenteser då du skriver in det printar R resultatet direkt, utan att behöva ange variabelnamnet.

```
(Y <- seq(1, 10, length.out = 5))
## [1] 1.00 3.25 5.50 7.75 10.00
```

Kolla nu in miljö-fönstret uppe till höger.

Här finns samtliga skapade objekt.

Övningar

1. Justera de följande kod-raderna så att de funkar:

```
ggplot(dota = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

fliter(mpg, cyl = 8)
filter(diamond, carat > 3)
```

2. Tryck Alt + Shift + K. Vad händer? Hur kan du nå motsvarande kommandon via menyerna?

Transformerings av data

Introduktion

Visualisering är en viktig del i att förstå innehållet i datamängder men det är sällsynt att du får en datafil som är organiserad exakt som du vill ha den.

Ofta behöver du skapa några nya variabler eller summeringar, kanske du bara vill byta namn på variablerna eller ordna om observationerna för att förenkla hanteringen. I detta kapitel går vi igenom de viktigaste verktygen för att ordna data som du vill ha dem. Vi gör det med hjälp av modulen `dplyr` och ett dataset som innehåller information om flighter från New York City 2013.

Förberedelser

Vi börjar med att installera modulen `nycflights13`:

```
install.packages("nycflights13")
```

Vi laddar in modulerna

```
library(nycflights13)
library(tidyverse)
```

När man laddar in `tidyverse` kommer ett antal felmeddelanden/varningar som talar om att `dplyr` skriver över några av funktionerna i *base R*, t.ex. `filter()` och `lag()`. Det är helt OK.

Datasetet vi ska använda finns i modulen `nycflights13` och kallas `flights`. Du kan få information om innehållet genom hjälpfunktionen

```
?flights
```

Det är en *dataframe* som innehåller information om samtliga 336 776 flighter som lyfte från NYC airport 2013. Du kan få en bild av hur tabellen är uppbyggd genom att skriva *tabellnamnet*

```
flights
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>    <dbl>   <int>
## 1 2013     1     1    517           515         2     830
## 2 2013     1     1    533           529         4     850
## 3 2013     1     1    542           540         2     923
## 4 2013     1     1    544           545        -1    1004
## 5 2013     1     1    554           600        -6     812
## 6 2013     1     1    554           558        -4     740
```

```
## 7 2013 1 1 555 600 -5 913
## 8 2013 1 1 557 600 -3 709
## 9 2013 1 1 557 600 -3 838
## 10 2013 1 1 558 600 -2 753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

Det som syns i output är en s.k. tibble, vilket är en modifierad tabell som visar de översta tabellraderna och de kolumner som får plats i förnstret. Du kan se hela tabellen genom kommandot

View(flights)

Vilket öppnar hela tabellen i *editorn* uppe till vänster. *Tibbles* är en speciell form av datatabell som fungerar lite effektivare i tidyverse och vi återkommer till detaljerna senare.

Notera raden under variabelnamnen med de tre bokstäverna inom <...>. Dessa bokstäver beskriver data-typen för varje variabel:

- Int = ental (integer)
- Dbl = reella tal (doubles)
- Chr = textsträng (character)
- Dttm = datum/tid (date and time)

Det finns ytterligare tre data-typer som vi återkommer till:

- Lgl = logisk (logical; TRUE/FALSE)
- Fctr = kategorisk (factor)
- Date = datum

dplyr grunder

Vi ska jobba med fem nyckelfunktioner i dplyr som klarar de flesta manipulationer av rådata:

-filter() används för att välja ut poster baserat på variabelvärden -arrange() används för att förändra ordningsföljden av posterna i en tabell -select() används för att välja ut variabler -mutate() används för att skapa nya variabler -summarise() används för att summera eller aggregera poster

Dessa funktioner används ofta tillsammans med group_by() som är en funktion vilken grupperar data och utnyttjar någon eller flera av de fem funktionerna på varje grupp snarare än på hela datamängden.

Dessa funktioner fungerar på liknande sätt:

1. Det första argumentet är namnet på datamängden/tabellen
2. De följande argumenten beskriver vad som ska göras med tabellen, genom att inkludera variabelnamnen

3. Resultet är en ny datatabell

Sammantaget gör dessa egenskaper det relativt lätt att länka samman flera enkla steg till en mer komplex kedja av manipulationer. Låt oss se hur det fungerar praktiskt.

Filtrera rader/poster med `filter()`

`filter()` låter dig välja ut en delmängd av posterna baserat på variabelvärden. De första argumentet är namnet på datamängden/tabellen. De övriga argumenten är uttryck som preciserar vad filtret ska göra. Vi börjar med att välja ut samtliga flighter som skedde den 1 januari.

```
filter(flights, month == 1, day == 1)

## # A tibble: 842 x 19
##   year month  day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>    <dbl>   <int>
## 1 2013     1     1     517             515         2     830
## 2 2013     1     1     533             529         4     850
## 3 2013     1     1     542             540         2     923
## 4 2013     1     1     544             545        -1    1004
## 5 2013     1     1     554             600        -6     812
## 6 2013     1     1     554             558        -4     740
## 7 2013     1     1     555             600        -5     913
## 8 2013     1     1     557             600        -3     709
## 9 2013     1     1     557             600        -3     838
##10 2013     1     1     558             600        -2     753
## # ... with 832 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

När koden körs filtreras `flights` och en ny tabell skapas. Dplyr förändrar aldrig ursprungsdata så om du vill spara resultatet för vidare bearbetning behöver du tillägna det ett namn, t.ex.:

```
jan1 <- filter(flights, month == 1, day == 1)
```

Jämförelser

För att använda filtrering effektivt bör du känna till hur jämförelseoperatorerna används. Dessa är de etablerade: `>`, `>=`, `<`, `<=`, `!=` och `==`. Det är lätt att man råkar använda `=` istället för `==` när man vill testa för exakt likhet. Prova

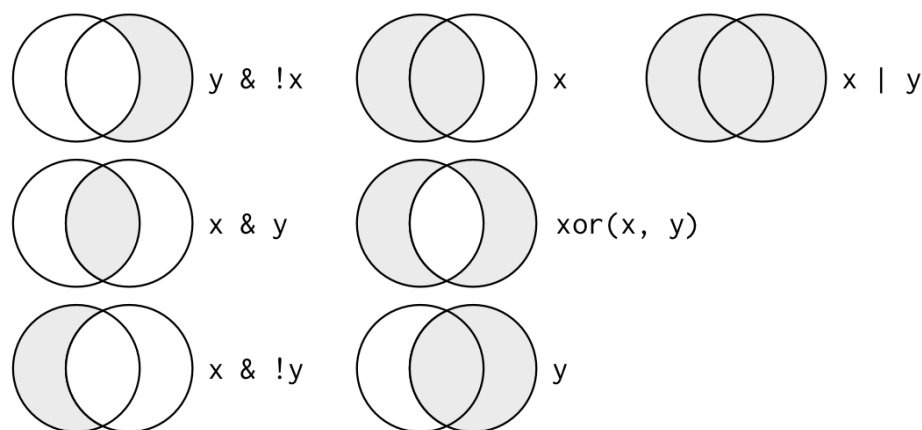
```
filter(flights, month = 1)
```


Logiska operatorer

Man kan förstås använda multipla argument för filtrering. Sedvanliga Booleanska operatorer fungerar, dvs

- `&` för "och"
- `|` för "eller"
- `!` för "icke"

Se figuren nedan.



Följande kod filtrerar fram samtliga fligheter som lyfte under november eller december:

```
filter(flights, month == 11 | month == 12)

## # A tibble: 55,403 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>    <dbl>   <int>
## 1 2013   11     1     5         2359           6     352
## 2 2013   11     1    35         2250        105    123
## 3 2013   11     1   455           500         -5    641
## 4 2013   11     1  539           545         -6    856
## 5 2013   11     1  542           545         -3    831
## 6 2013   11     1  549           600        -11    912
## 7 2013   11     1  550           600        -10    705
## 8 2013   11     1  554           600         -6    659
## 9 2013   11     1  554           600         -6    826
## 10 2013   11     1  554           600         -6    749
## # ... with 55,393 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

Viktigt att ange kompletta argument! Det kan t.ex. vara förvirrande att `filter(flights, month == 11 | 12)` kommer att ange samtliga fligheter i januari, inte i november eller december. Det hänger samman med att uttrycket `11|12` är ett logiskt argument och utvärderas till sant, dvs TRUE. *I ett numeriskt sammanhang som här blir TRUE = 1 dvs tolkas som månaden 1, alltså januari!*

Ett användbart snabbkommando för detta filtreringsproblem är `x %in% y`. detta söker ut samtliga poster där x är något av värdena i y. För att filtrera fram samtliga fligheter under nov-dec kan vi då skriva:

```
nov_dec <- filter(flights, month %in% c(11, 12))
```

Närhelst du ska använda mer komplexa uttryck för att filtrera data bör du överväga att göra dem till explicita, ev temporära, variabler istället. Det gör det lättare att kontrollera eller felsöka kod. Återkommer strax till hur nya variabler skapas.

Missing

Ett viktigt drag i R är hanteringen av missing values (NA - *not availables*) vilka representerar okända värden och är därför "smittsamma" - nästan varje behandling av data som innehåller NA kommer också att bli NA. Prova:

```
NA > 5
## [1] NA

10 == NA
## [1] NA

NA + 10
## [1] NA

NA / 2
## [1] NA
```

Om du vill avgöra om ett värde är NA kan du använda `is.na()`:

```
x <- NA
is.na(x)
## [1] TRUE
```

`filter()` inkluderar endast poster där villkoret är sant (TRUE). Det exkluderar således alla FALSE men också alla NA. Om man vill få med NA i filtreringen behöver de ingå explicit i villkoret:

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)

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

```
## <dbl>
## 1 3

filter(df, is.na(x) | x > 1)

## # A tibble: 2 x 1
##   x
## <dbl>
## 1 NA
## 2 3
```

Övningar

- Hitta samtliga flighter
 - med en försenad ankomst mer än 2 timmar.
 - Lyfte med destination Houston
 - Avgick under juli, aug, september
 - Ankom med mer än 2 timmars försening men lyfte i tid
- Ett annat filter-verktyg är `between()`. Hur fungerar det? Kan man förenkla filtreringarna i punkt 1 med hjälp av `between()`?
- Hur många flighter saknar data för `dep_time`?

Arrangera poster med `arrange()`

`arrange()` fungerar på liknande sätt som `filter()` men istället för att filtrera poster så ändrar man deras inbördes ordning. Funktionen tar en data-frame och en eller flera kolumner för att ordna posterna. Om du använder mer än en kolumn kommer varje ytterligare kolumn att användas för att skapa brytpunkter i de följande kolumnerna. Prova

```
arrange(flights, year, month, day)

## # A tibble: 336,776 x 19
##   year month day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int> <int>         <int>    <dbl>    <int>
## 1 2013     1   1   517           515         2     830
## 2 2013     1   1   533           529         4     850
## 3 2013     1   1   542           540         2     923
## 4 2013     1   1   544           545        -1    1004
## 5 2013     1   1   554           600        -6     812
## 6 2013     1   1   554           558        -4     740
## 7 2013     1   1   555           600        -5     913
## 8 2013     1   1   557           600        -3     709
## 9 2013     1   1   557           600        -3     838
## 10 2013     1   1   558           600        -2     753
```

```
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

Man kan använda `desc()` för att sortera posterna i sjunkande ordning:

```
arrange(flights, desc(arr_delay))
```

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>      <int>    <dbl>   <int>
## 1 2013     1     9     641        900     1301   1242
## 2 2013     6    15    1432       1935     1137   1607
## 3 2013     1    10    1121       1635     1126   1239
## 4 2013     9    20    1139       1845     1014   1457
## 5 2013     7    22     845       1600     1005   1044
## 6 2013     4    10    1100       1900      960   1342
## 7 2013     3    17    2321        810      911    135
## 8 2013     7    22    2257        759      898    121
## 9 2013    12     5     756       1700      896   1058
## 10 2013     5     3    1133       2055      878   1250
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

Övningar

1. Hur kan du använda `arrange()` för att sortera alla missing values? (minns funktionen `is.na()`)
2. Sortera `flights` för att identifiera de mest försenade flighterna. Gör samma för de som lyfte tidigast.
3. Sortera `flights` för att identifiera de snabbaste flighterna.
4. Vilka flighter flög längst? Kortast?

Välj kolumner med `select()`

Många gånger får man rådata med mängder av variabler och behöver egentligen endast ett fåtal. Funktionen `select()` ger möjlighet att snabbt avgränsa datamängden till de variabler man vill använda, baserat på variabelnamnen.

Välj kolumner med hjälp av deras namn:

```
select(flights, year, month, day)
```

```
## # A tibble: 336,776 x 3
##   year month  day
##   <int> <int> <int>
## 1 2013     1     1
## 2 2013     1     1
## 3 2013     1     1
## 4 2013     1     1
## 5 2013     1     1
## 6 2013     1     1
## 7 2013     1     1
## 8 2013     1     1
## 9 2013     1     1
## 10 2013    1     1
## # ... with 336,766 more rows
```

Välj samtliga kolumner fr.o.m. year t.o.m. day:

```
select(flights, year:day)

## # A tibble: 336,776 x 3
##   year month  day
##   <int> <int> <int>
## 1 2013     1     1
## 2 2013     1     1
## 3 2013     1     1
## 4 2013     1     1
## 5 2013     1     1
## 6 2013     1     1
## 7 2013     1     1
## 8 2013     1     1
## 9 2013     1     1
## 10 2013    1     1
## # ... with 336,766 more rows
```

Välj samtliga kolumner utom kolumnerna fr.o.m. year t.o.m. day:

```
select(flights, -(year:day))

## # A tibble: 336,776 x 16
##   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
##   <int>      <int>      <dbl> <int>      <int>      <dbl>
## 1    517        515         2    830        819         11
## 2    533        529         4    850        830         20
## 3    542        540         2    923        850         33
## 4    544        545        -1   1004       1022        -18
```

```
## 5 554 600 -6 812 837 -25
## 6 554 558 -4 740 728 12
## 7 555 600 -5 913 854 19
## 8 557 600 -3 709 723 -14
## 9 557 600 -3 838 846 -8
## 10 558 600 -2 753 745 8
## # ... with 336,766 more rows, and 10 more variables: carrier <chr>,
## # flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
## # distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dtm>
```

Det finns ett antal hjälpfunktioner som kan användas med `select()`:

- `starts_with("abc")`: matchar namn som börjar med "abc".
- `ends_with("xyz")`: matchar namn som slutar med "xyz".
- `contains("ijk")`: Matchar namn som innehåller "ijk".
- `matches("(.)\\1")`: väljer variabler som matchar ett *regular expression*. Detta exempel matchar varje variabel som innehåller upprepade tecken. Vi ska kika mer på *regular expressions* senare.
- `num_range("x", 1:3)` matchar x1, x2 and x3.

Använd `?select` för mer information.

`select()` kan användas för att ändra variabelnamn men är sällan en lämplig funktion för detta eftersom `select()` droppar alla de variabler som inte nämns explicit. Använd istället `rename()`:

```
rename(flights, tail_num = tailnum)

## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>    <dbl>   <int>
## 1 2013     1     1    517           515         2     830
## 2 2013     1     1    533           529         4     850
## 3 2013     1     1    542           540         2     923
## 4 2013     1     1    544           545        -1    1004
## 5 2013     1     1    554           600        -6     812
## 6 2013     1     1    554           558        -4     740
## 7 2013     1     1    555           600        -5     913
## 8 2013     1     1    557           600        -3     709
## 9 2013     1     1    557           600        -3     838
## 10 2013     1     1    558           600        -2     753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## # arr_delay <dbl>, carrier <chr>, flight <int>, tail_num <chr>,
## # origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## # minute <dbl>, time_hour <dtm>
```

Ett annat alternativ är att använda hjälpfunktionen `everything()`. Detta kan vara praktiskt då du vill flytta vissa variabler t.ex. till början av datamängden:

```
select(flights, time_hour, air_time, everything())

## # A tibble: 336,776 x 19
##   time_hour      air_time year month  day dep_time sched_dep_time
##   <dtm>         <dbl> <int> <int> <int> <int>      <int>
## 1 2013-01-01 05:00:00   227  2013     1     1   517         515
## 2 2013-01-01 05:00:00   227  2013     1     1   533         529
## 3 2013-01-01 05:00:00   160  2013     1     1   542         540
## 4 2013-01-01 05:00:00   183  2013     1     1   544         545
## 5 2013-01-01 06:00:00   116  2013     1     1   554         600
## 6 2013-01-01 05:00:00   150  2013     1     1   554         558
## 7 2013-01-01 06:00:00   158  2013     1     1   555         600
## 8 2013-01-01 06:00:00    53  2013     1     1   557         600
## 9 2013-01-01 06:00:00   140  2013     1     1   557         600
## 10 2013-01-01 06:00:00   138  2013     1     1   558         600
## # ... with 336,766 more rows, and 12 more variables: dep_delay <dbl>,
## #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
## #   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, distance <dbl>,
## #   hour <dbl>, minute <dbl>
```

Övningar

1. Vad händer om du använder ett variabelnamn flera gånger i `select()`?
2. Vad åstadkommer hjälpfunktionen `one_of()`? Varför kan den vara till nytta tillsammans med nedanstående vektor?

```
vars <- c("year", "month", "day", "dep_delay", "arr_delay")
```

3. Blir du förvånad över resultatet av följande kod? Hur kan hjälpfunktionerna till `select()` hantera dessa "by default"? Hur kan du ändra "default"?

```
select(flights, contains("TIME"))

## # A tibble: 336,776 x 6
##   dep_time sched_dep_time arr_time sched_arr_time air_time
##   <int>      <int>      <int>      <int>      <dbl>
## 1   517        515    830        819    227
## 2   533        529    850        830    227
## 3   542        540    923        850    160
## 4   544        545   1004       1022    183
## 5   554        600    812        837    116
## 6   554        558    740        728    150
## 7   555        600    913        854    158
```

```
## 8 557 600 709 723 53
## 9 557 600 838 846 140
## 10 558 600 753 745 138
## # ... with 336,766 more rows, and 1 more variable: time_hour <dtm>
```

Lägg till flera variabler med hjälp av `mutate()`

Förutom att välja ut variabler behöver man ofta lägga till nya variabler vilka är funktioner av befintliga variabler. Det är vad funktionen `mutate()` gör.

`mutate()` lägger alltid till de nya variablerna sist i datamängden. Vi börjar med att reducera antalet variabler så att kolumnerena blir mer lättöverskådliga:

```
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60
)

## # A tibble: 336,776 x 9
##   year month   day dep_delay arr_delay distance air_time gain speed
##   <int> <int> <int>   <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 2013     1     1         2      11    1400     227     9  370.
## 2 2013     1     1         4      20    1416     227    16  374.
## 3 2013     1     1         2      33    1089     160    31  408.
## 4 2013     1     1        -1     -18    1576     183   -17  517.
## 5 2013     1     1        -6     -25     762     116   -19  394.
## 6 2013     1     1        -4      12     719     150    16  288.
## 7 2013     1     1        -5      19    1065     158    24  404.
## 8 2013     1     1        -3     -14     229      53   -11  259.
## 9 2013     1     1        -3      -8     944     140    -5  405.
## 10 2013     1     1        -2       8     733     138    10  319.
## # ... with 336,766 more rows
```

Notera att du kan referera till de nya kolumnerna:

```
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
```



```

gain_per_hour = gain / hours
)

## # A tibble: 336,776 x 10
##   year month   day dep_delay arr_delay distance air_time gain hours
##   <int> <int> <int>   <dbl>   <dbl>   <dbl>   <dbl> <dbl> <dbl>
## 1 2013     1     1     2     11    1400    227    9 3.78
## 2 2013     1     1     4     20    1416    227   16 3.78
## 3 2013     1     1     2     33    1089    160   31 2.67
## 4 2013     1     1    -1    -18    1576    183  -17 3.05
## 5 2013     1     1    -6   -25     762    116 -19 1.93
## 6 2013     1     1    -4     12     719    150   16 2.5
## 7 2013     1     1    -5     19    1065    158   24 2.63
## 8 2013     1     1    -3    -14     229     53  -11 0.883
## 9 2013     1     1    -3     -8     944    140   -5 2.33
## 10 2013     1     1    -2      8     733    138   10 2.3
## # ... with 336,766 more rows, and 1 more variable: gain_per_hour <dbl>

```

Om du vill endast behålla de nya variablerna kan du använda `transmute()`:

```

transmute(flights,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)

## # A tibble: 336,776 x 3
##   gain hours gain_per_hour
##   <dbl> <dbl>   <dbl>
## 1    9 3.78      2.38
## 2   16 3.78      4.23
## 3   31 2.67     11.6
## 4  -17 3.05     -5.57
## 5  -19 1.93     -9.83
## 6   16 2.5      6.4
## 7   24 2.63     9.11
## 8  -11 0.883    -12.5
## 9   -5 2.33     -2.14
## 10  10 2.3      4.35
## # ... with 336,766 more rows

```

Funktioner och operatorer att användas med `mutate()`

Den grundläggande egenskapen hos denna funktion är att den måste vektoriseras, dvs dess argument måste vara en vektor av värden (input) och den resulterar i en vektor av nya värden (output), lika många som input.

Det finns alldeles för många funktioner som man kan använda tillsammans med `mutate()` för att kunna lista dem här. Men de nedanstående är en uppsättning som är de vanligast förekommande:

- Aritmetiska operatorer: `+`, `-`, `*`, `/`, `^`. Dessa vektoriseras genom "recycling rules". Om en parameter är kortare än en annan kommer den att automatiskt förlängas till samma längd som den andra vektorn. Detta är mycket användbart då den ena parametern består av ett fixerat värde, t.ex. `air_time/60` eller `hours * 60 + minute`.
- Aritmetiska operatorer är också användbara tillsammans med aggregeringsfunktioner som vi ska kika på senare. Till exempel beräknar `x / sum(x)` andelen av en total; `y - mean(y)` beräknar avvikelser från medelvärdet.
- Modulär aritmetik: `%/%` (heltalsdivision) och `%%` (rest). Detta är en användbar funktion för att dekomponera ett heltal. Till exempel:

```
transmute(flights,
  dep_time,
  hour = dep_time %/% 100,
  minute = dep_time %% 100
)

## # A tibble: 336,776 x 3
##   dep_time hour minute
##   <int> <dbl> <dbl>
## 1    517     5     17
## 2    533     5     33
## 3    542     5     42
## 4    544     5     44
## 5    554     5     54
## 6    554     5     54
## 7    555     5     55
## 8    557     5     57
## 9    557     5     57
## 10   558     5     58
## # ... with 336,766 more rows
```

- Logaritmer: `log()`, `log2()`, `log10()`.
- Offsets: `lead()` and `lag()` gör det möjligt att referera till värden tidigare eller senare i vektorn. Till exempel beräkna löpande skillnader (`x - lag(x)`) eller att identifiera poster där värden förändras (`x != lag(x)`).

```
(x <- 1:10)
```

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

```
lag(x)
```

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

```
lead(x)
```

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

- Kumulativa “aggregat”: R innehåller funktioner för löpande summor, produkter, minimum, maximum: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`; och `dplyr` innehåller `cummean()` för kumulativa medelvärden.

```
x
```

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

```
cumsum(x)
```

```
## [1] 1 3 6 10 15 21 28 36 45 55
```

```
cummean(x)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

- Logiska jämförelser (se ovan): `<`, `<=`, `>`, `>=`, `!=`. Tips: om du skapar en komplex sekvens av logiska operatorer är det ofta en god idé att lagra delvärden som nya variabler - det gör det lättare att kolla så att varje steg fungerar som det ska.
- Ranking: det finns ett antal ranking-funktioner av vilka `min_rank()` är mest använd. Den gör den vanligaste typen av rankning, dvs första, andra, tredje osv. Default är att tilldela det lägsta värdet den lägsta rankningen. Man kan använda `desc(x)` för att tilldela det största värdet den lägsta rankningen:

```
y <- c(1, 2, 2, NA, 3, 4)
```

```
min_rank(y)
```

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

```
min_rank(desc(y))
```

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

Om `min_rank()` inte är det du söker, prova `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()`, `ntile()`. Se respektive hjälpsida för mer info.

```
row_number(y)
```

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

```
dense_rank(y)
```

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

percent_rank(y)

## [1] 0.00 0.25 0.25 NA 0.75 1.00

cume_dist(y)

## [1] 0.2 0.6 0.6 NA 0.8 1.0
```

Övningar

1. `dep_time` och `sched_dep_time` kan vara mer begripliga att se på men svårare att räkna med eftersom de inte innehåller kontinuerliga värden. Konvertera dem till mer användbara värden som antal minuter efter midnatt.
2. Jämför `air_time` med `arr_time - dep_time`. Vad förväntar du dig att se? Vad ser du? Vad behöver göras för att se det du förväntade dig?
3. Identifiera de 10 mest försenade flighterna genom att använda en rankningsfunktion.
4. Vad returneras av `1:3 + 1:10`? Varför?
5. Vilka trigonometriska funktioner finns i R?

Grupperade summeringar med `summarise()`

Den sista nyckelfunktionen är `summarise()`. Den slår samman en datamängd till en enda rad. Prova:

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))

## # A tibble: 1 x 1
##   delay
##   <dbl>
## 1 12.6
```

(Vi återkommer till `na.rm = TRUE` alldeles strax.)

`summarise()` är inte särskilt användbart såvida vi inte samtidigt använder `group_by()`. Denna funktion förändrar "the unit of analysis" från hela datamängden till grupper av data. När du använder dplyr-funktionerna på en grupperad datamängd kommer funktionerna att automatiskt tillämpas på varje grupp. Så när vi till exempel tillämpar exakt samma kod som ovan på data som är grupperade per dag så får vi den genomsnittliga förseningen per dag. Prova:

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##   year month   day delay
##   <int> <int> <int> <dbl>
```

```
## 1 2013 1 1 11.5
## 2 2013 1 2 13.9
## 3 2013 1 3 11.0
## 4 2013 1 4 8.95
## 5 2013 1 5 5.73
## 6 2013 1 6 7.15
## 7 2013 1 7 5.42
## 8 2013 1 8 2.55
## 9 2013 1 9 2.28
## 10 2013 1 10 2.84
## # ... with 355 more rows
```

`group_by()` och `summarise()` är tillsammans de verktyg du kommer att använda mest då du arbetar med `dplyr`, nämligen grupperade summeringar. Men innan vi går vidare med detta behöver vi kika på “the pipe” (hur översätter man detta?).

Kombinera multipla operationer med the pipe

Antag att vi vill undersöka relationen mellan avståndet och den genomsnittliga förseningen för varje destination. Vi kan skriva följande kod:

```
by_dest <- group_by(flights, dest)

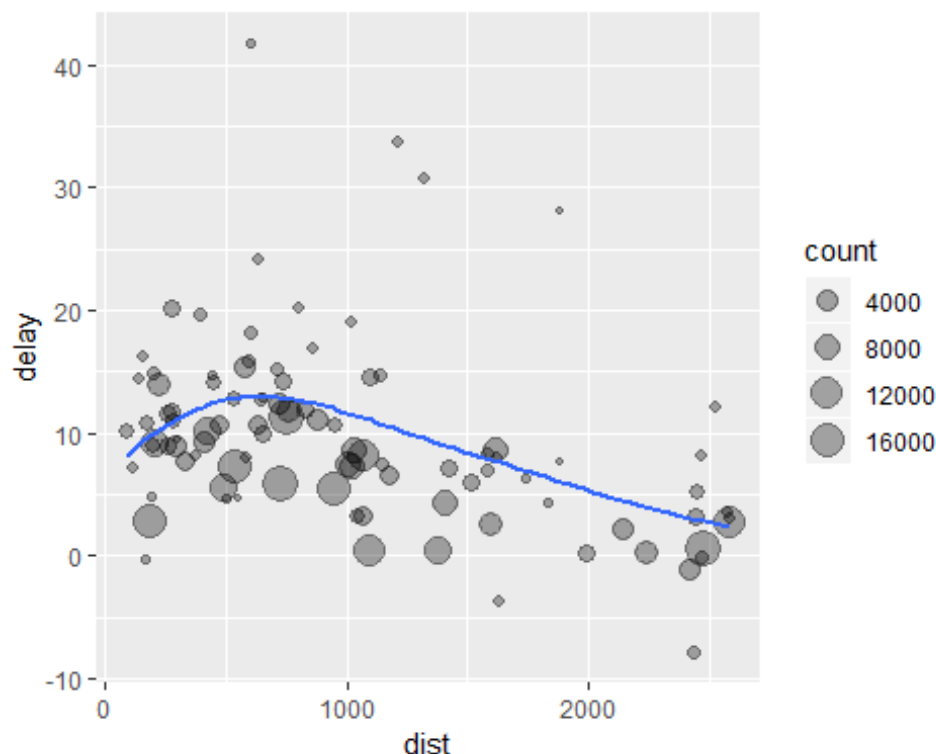
delay <- summarise(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
)

delay <- filter(delay, count > 20, dest != "HNL")
```

Det verkar som förseningen ökar upp till ett avstånd runt 750 miles och sedan minskar. Kanske är det så att på en längre flygning finns större förutsättningar att hämta in en försening? Vi kan visualisera data genom:

```
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Dessa data förbereds i tre steg: 1. Gruppera flighter per destination. 2. Summera avstånd, genomsnittlig försening och antal flighter. 3. Filtrera bort brus och flighter till Honolulu, som ligger mer än dubbelt så långt bort som den näst längst bort.

Denna kod är lite omständlig att skriva eftersom vi behöver tilldela varje intermediär datamängd ett namn fastän vi inte är intresserade av själva datamängden i sig. Istället kan man skriva om koden med hjälp av "the pipe", %>% (snabbtangenter Ctrl+Shift+m):

```
delays <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")
```

Detta fokuserar transformeringarna istället för det som transformeras, vilket gör koden lättare att läsa. Du kan läsa ut koden som en serie av länkade uppmaningar: "gruppera, sedan summera, sedan filtrera". Man kan alltså använda "the pipe" för att skriva om multipla operationer så att de kan läsas från vänster till höger, uppifrån och ned.

Detta har blivit en ganska central del i det modernare R och rekommenderas starkt. MEN, det finns svagheter. Ett viktigt undantag är ggplot2 som skrevs innan "the pipe"-grammatiken utvecklades. Nästa generation av ggplot2 är dock under stark utveckling. Den modulen kallas ggvis (i skrivande stund

version 0.4) och förutom att integrera “the pipe” är den tänkt att också integrera interaktiva diagram. Läs mer på Rstudios hemsida <http://ggvis.rstudio.com>.

Missing values

Vi använde tidigare argumentet `na.rm()`. Vad händer om vi inte gör det? Pröva:

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##   year month  day mean
##   <int> <int> <int> <dbl>
## 1 2013     1    1  NA
## 2 2013     1    2  NA
## 3 2013     1    3  NA
## 4 2013     1    4  NA
## 5 2013     1    5  NA
## 6 2013     1    6  NA
## 7 2013     1    7  NA
## 8 2013     1    8  NA
## 9 2013     1    9  NA
## 10 2013    1   10  NA
## # ... with 355 more rows
```

En massa värden blir missing (NA)! Det blir så därför att de olika summeringsfunktionerna lyder under “the rule of missing values”: om det finns ett eller fler NA i input kommer output att vara ett NA. Argumentet `na.rm()` flyttar bort alla missing values (NA) innan beräkningen görs:

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE))

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##   year month  day mean
##   <int> <int> <int> <dbl>
## 1 2013     1    1 11.5
## 2 2013     1    2 13.9
## 3 2013     1    3 11.0
## 4 2013     1    4  8.95
## 5 2013     1    5  5.73
```

```
## 6 2013 1 6 7.15
## 7 2013 1 7 5.42
## 8 2013 1 8 2.55
## 9 2013 1 9 2.28
## 10 2013 1 10 2.84
## # ... with 355 more rows
```

I detta fall där NA representerade inställda flyg, kan vi även genomföra en summering genom att först ta bort samtliga inställda flyg (= NA). Vi skapar ett nytt dataset för kommande exempel:

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##   year month  day mean
##   <int> <int> <int> <dbl>
## 1 2013     1     1 11.4
## 2 2013     1     2 13.7
## 3 2013     1     3 10.9
## 4 2013     1     4  8.97
## 5 2013     1     5  5.73
## 6 2013     1     6  7.15
## 7 2013     1     7  5.42
## 8 2013     1     8  2.56
## 9 2013     1     9  2.30
## 10 2013    1    10  2.84
## # ... with 355 more rows
```

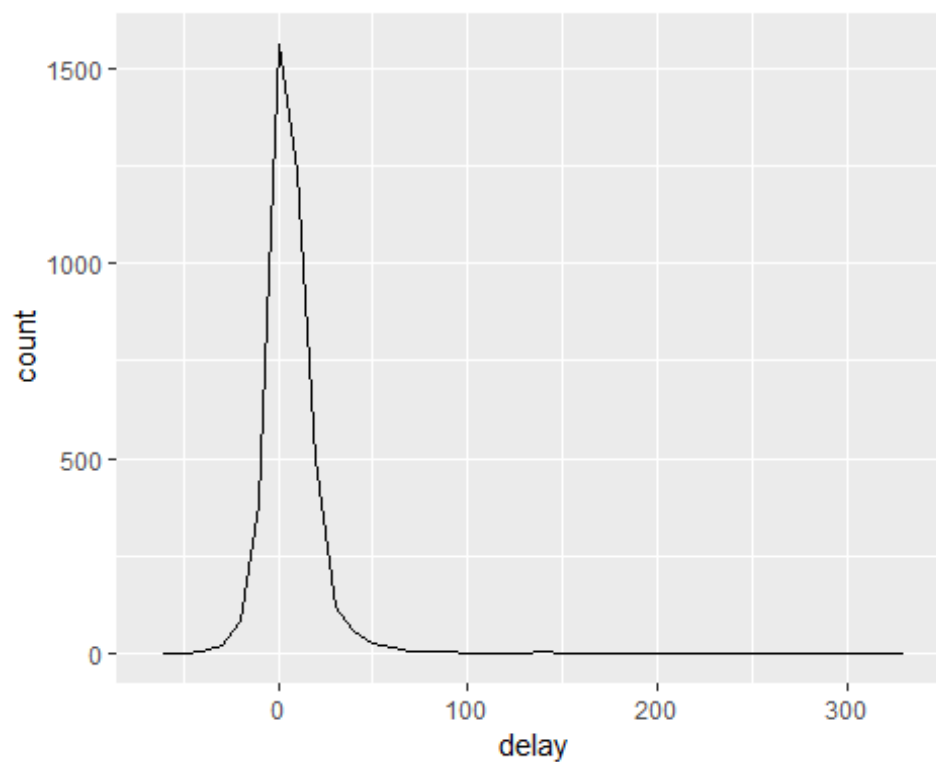
Antal (counts)

Då man gör en summering är det ofta en bra idé att inkludera antalet observationer (vilket görs med funktionen `n()`) liksom antalet missing values (`sum(!is.na(x))`). På det sättet är det lätt att kolla så att man t.ex. inte drar slutsatser på flygplanen (identifierade genom deras tail numbers) med längst genomsnittlig försening:

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay)
```



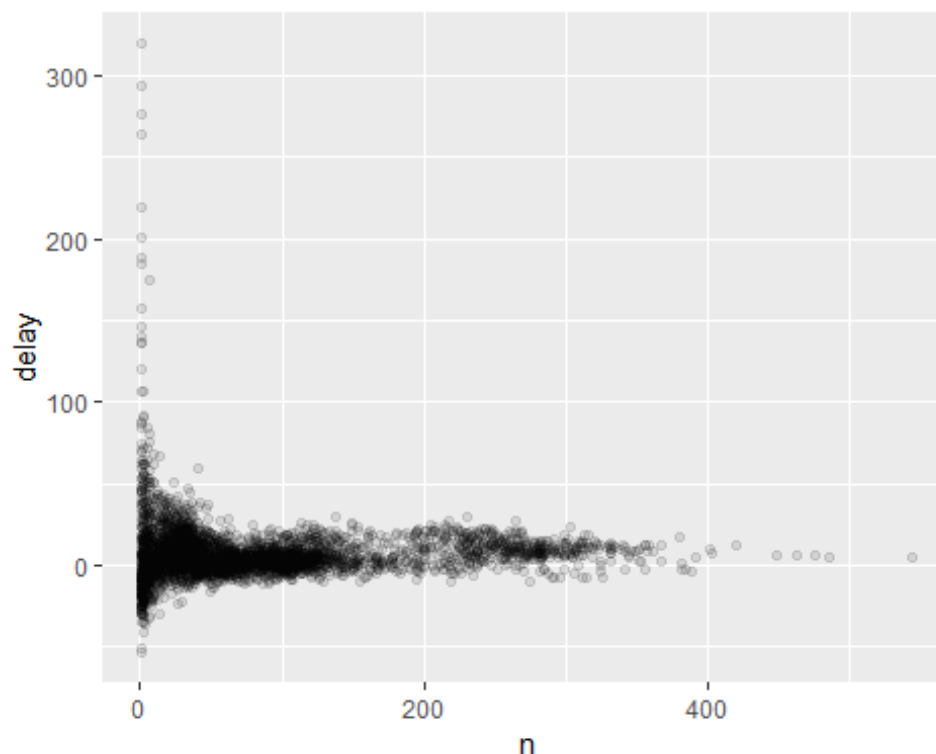
```
)
ggplot(data = delays, mapping = aes(x = delay)) +
  geom_freqpoly(binwidth = 10)
```



OK, det finns alltså plan med en genomsnittlig försening med mer än 5 timmar! Nja, det är lite mer komplicerat än så. Vi får mer information genom att göra en scatterplot över antalet flighter vs. genomsnittlig försening:

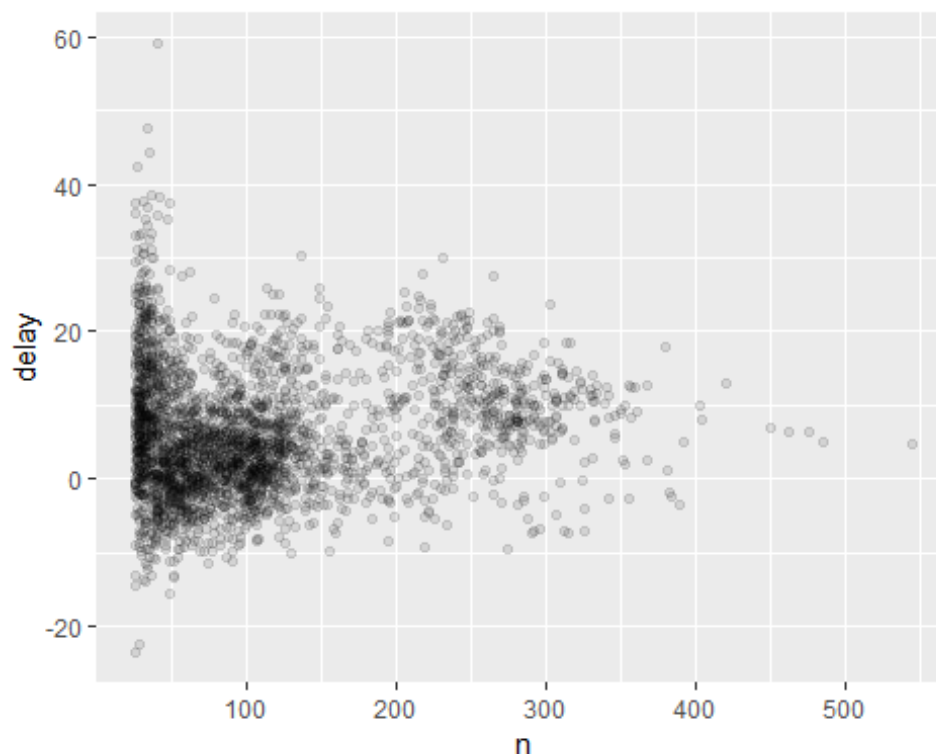
```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

ggplot(data = delays, mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



Inte helt överraskande är det mycket större variation i genomsnittlig försening då flighterna är få. Scatterplotten är karakteristisk: då man plottar genomsnittsvärden (eller andra summeringar) mot gruppstorlek kommer variationen att minska då urvalsstorleken ökar. I dessa fall är det ofta användbart att filtrera ut grupperna med minst antal observationer vilket gör att man tydligare ser ev mönster och mindre av extrema värden. Detta åstadkommer vi med följande kod. Notera hur `ggplot2`-kod integreras i `dplyr` flödet. Visserligen lite retfullt att byta från `%>%` till `+` men det funkar:

```
delays %>%
  filter(n > 25) %>%
  ggplot(mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



Tips: ett användbart kortkommando är Ctrl+Shift+P. Då återanvänds det senaste kodavsnittet (i Rstudio-språk "the chunk") och sänds från editorn till konsolen. Detta är smidigt när man t.ex. prövar olika värden på `n` i exemplet. Du skickar hela kodavsnittet med hjälp av Ctrl+Enter, sedan modifierar du värdet på `n` och återanvänder det genom Ctrl+Shift+P.

Användbara summeringsfunktioner

Man kan komma långt enbart med medelvärden, antal och summa men R innehåller många fler summeringsfunktioner: - *Medianvärdet* beräknas med `median(x)`. Det är ibland smidigt att använda tillsammans med verktyg för att välja ut delar av ett dataset, *subsetting* (kommer senare). Prova:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    avg_delay1 = mean(arr_delay),
    avg_delay2 = mean(arr_delay[arr_delay > 0]) # the average positive delay
  )

## # A tibble: 365 x 5
## # Groups:   year, month [12]
##   year month  day avg_delay1 avg_delay2
##   <int> <int> <int>    <dbl>    <dbl>
## 1 2013     1     1    12.7     32.5
```

```
## 2 2013 1 2 12.7 32.0
## 3 2013 1 3 5.73 27.7
## 4 2013 1 4 -1.93 28.3
## 5 2013 1 5 -1.53 22.6
## 6 2013 1 6 4.24 24.4
## 7 2013 1 7 -4.95 27.8
## 8 2013 1 8 -3.23 20.8
## 9 2013 1 9 -0.264 25.6
## 10 2013 1 10 -5.90 27.3
## # ... with 355 more rows
```

- *Spridningsmått*: `sd(x)`, `IQR(x)`, `mad(x)`, för *standardavvikelsen*, *interkvartil-avståndet* (interquartile range) resp *median absolute deviation*.
- *Ranking-mått*: `min(x)`, `quantile(x, 0.25)`, `max(x)`. Kvantiler är en generalisering av medianen. Så kommer t.ex. `quantile(x, 0.25)` att identifiera det värde som är större än 25% av värdena men mindre än de resterande 75% .

När lyfter den första respektive sista flighten per dag?

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first = min(dep_time),
    last = max(dep_time)
  )
```

```
## # A tibble: 365 x 5
## # Groups:   year, month [12]
##   year month  day first last
##   <int> <int> <int> <dbl> <dbl>
## 1 2013     1     1 517 2356
## 2 2013     1     2  42 2354
## 3 2013     1     3  32 2349
## 4 2013     1     4  25 2358
## 5 2013     1     5  14 2357
## 6 2013     1     6  16 2355
## 7 2013     1     7  49 2359
## 8 2013     1     8 454 2351
## 9 2013     1     9   2 2252
## 10 2013    1    10   3 2320
## # ... with 355 more rows
```

- *Positionsmått*: `first(x)`, `nth(x, 2)`, `last(x)`. Dessa motsvarar `x[1]`, `x[2]`, och `x[length(x)]` men gör det möjligt att ha ett default-värde ifall den positionen inte existerar, t.ex. om du försöker få det tredje

värdet i en vektor med endast två värden. Vi kan t.ex. identifiera den första resp sista avgången per dag:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first_dep = first(dep_time),
    last_dep = last(dep_time)
  )

## # A tibble: 365 x 5
## # Groups:   year, month [12]
##   year month   day first_dep last_dep
##   <int> <int> <int>   <int>   <int>
## 1 2013     1     1     517     2356
## 2 2013     1     2      42     2354
## 3 2013     1     3      32     2349
## 4 2013     1     4      25     2358
## 5 2013     1     5      14     2357
## 6 2013     1     6      16     2355
## 7 2013     1     7      49     2359
## 8 2013     1     8     454     2351
## 9 2013     1     9       2     2252
## 10 2013     1    10       3     2320
## # ... with 355 more rows
```

Dessa funktioner är komplementära till att filtrera efter rank. Om du filtrerar får du samtliga variabler för de poster som uppfyller filtreringsvillkoren, med varje post på en egen rad:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  mutate(r = min_rank(desc(dep_time))) %>%
  filter(r %in% range(r))

## # A tibble: 770 x 20
## # Groups:   year, month, day [365]
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>       <int>    <dbl>   <int>
## 1 2013     1     1     517         515         2     830
## 2 2013     1     1     2356         2359        -3     425
## 3 2013     1     2      42         2359        43     518
## 4 2013     1     2     2354         2359        -5     413
## 5 2013     1     3      32         2359        33     504
## 6 2013     1     3     2349         2359       -10     434
## 7 2013     1     4      25         2359        26     505
```

```
## 8 2013 1 4 2358 2359 -1 429
## 9 2013 1 4 2358 2359 -1 436
## 10 2013 1 5 14 2359 15 503
## # ... with 760 more rows, and 13 more variables: sched_arr_time <int>,
## # arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## # origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## # minute <dbl>, time_hour <dtm>, r <int>
```

- *Antal*: `n()`, som inte tar några argument och returnerar storleken på gruppen. För att räkna non-missing värden använder du `sum(!is.na(x))`. För att beräkna antalet unika värden i en datamängd använde du `n_distinct(x)`: Vilka destinationer har flest carriers?

```
not_cancelled %>%
  group_by(dest) %>%
  summarise(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))

## # A tibble: 104 x 2
##   dest carriers
##   <chr>   <int>
## 1 ATL     7
## 2 BOS     7
## 3 CLT     7
## 4 ORD     7
## 5 TPA     7
## 6 AUS     6
## 7 DCA     6
## 8 DTW     6
## 9 IAD     6
## 10 MSP    6
## # ... with 94 more rows
```

Eftersom antalsberäkningar är så användbara finns det i `dplyr` en enkel hjälpfunktion om allt du vill göra är att beräkna antalet:

```
not_cancelled %>%
  count(dest)

## # A tibble: 104 x 2
##   dest    n
##   <chr> <int>
## 1 ABQ   254
## 2 ACK   264
## 3 ALB   418
## 4 ANC    8
```

```
## 5 ATL 16837
## 6 AUS 2411
## 7 AVL 261
## 8 BDL 412
## 9 BGR 358
## 10 BHM 269
## # ... with 94 more rows
```

Du kan också använda en viktningsvariabel. Du kan t.ex. använda en sådan för att beräkna antalet miles som ett plan flög:

```
not_cancelled %>%
  count(tailnum, wt = distance)

## # A tibble: 4,037 x 2
##   tailnum     n
##   <chr>   <dbl>
## 1 D942DN  3418
## 2 N0EGMQ 239143
## 3 N10156 109664
## 4 N102UW  25722
## 5 N103US  24619
## 6 N104UW  24616
## 7 N10575 139903
## 8 N105UW  23618
## 9 N107US  21677
## 10 N108UW 32070
## # ... with 4,027 more rows
```

Antal och andelar av logiska utfall: `sum(x > 10)`, `mean(y == 0)`. När logiska operatorer används på numeriska värden "översätts" TRUE till 1 och FALSE till 0. Detta gör `sum()` och `mean()` väldigt användbara: `sum(x)` ger antalet TRUES i x, och `mean(x)` ger andelen:

Hur många flighter lyfte innan kl 5 på morgonen?

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(n_early = sum(dep_time < 500))

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##   year month   day n_early
##   <int> <int> <int>   <int>
## 1 2013     1     1     0
## 2 2013     1     2     3
```

```
## 3 2013 1 3 4
## 4 2013 1 4 3
## 5 2013 1 5 3
## 6 2013 1 6 2
## 7 2013 1 7 2
## 8 2013 1 8 1
## 9 2013 1 9 3
## 10 2013 1 10 3
## # ... with 355 more rows
```

Hur stor andel av flighterna var försenade mer än en timme?

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(hour_perc = mean(arr_delay > 60))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [12]
##   year month   day hour_perc
##   <int> <int> <int>   <dbl>
## 1 2013     1     1  0.0722
## 2 2013     1     2  0.0851
## 3 2013     1     3  0.0567
## 4 2013     1     4  0.0396
## 5 2013     1     5  0.0349
## 6 2013     1     6  0.0470
## 7 2013     1     7  0.0333
## 8 2013     1     8  0.0213
## 9 2013     1     9  0.0202
## 10 2013     1    10  0.0183
## # ... with 355 more rows
```

Gruppera med flera variabler

När du använder fler variabler för att gruppera ett dataset erhålls en delsumma för varje nivå i grupperingen. Det gör det enkelt att rulla upp en datamängd som ett sätt att förstå hur den är uppbyggd:

```
daily <- group_by(flights, year, month, day)
(per_day <- summarise(daily, flights = n()))

## # A tibble: 365 x 4
## # Groups:   year, month [12]
##   year month   day flights
##   <int> <int> <int>   <int>
```



```
## 1 2013 1 1 842
## 2 2013 1 2 943
## 3 2013 1 3 914
## 4 2013 1 4 915
## 5 2013 1 5 720
## 6 2013 1 6 832
## 7 2013 1 7 933
## 8 2013 1 8 899
## 9 2013 1 9 902
## 10 2013 1 10 932
## # ... with 355 more rows

(per_month <- summarise(per_day, flights = sum(flights)))
```

```
## # A tibble: 12 x 3
## # Groups:   year [1]
##   year month flights
##   <int> <int> <int>
## 1 2013 1 27004
## 2 2013 2 24951
## 3 2013 3 28834
## 4 2013 4 28330
## 5 2013 5 28796
## 6 2013 6 28243
## 7 2013 7 29425
## 8 2013 8 29327
## 9 2013 9 27574
## 10 2013 10 28889
## 11 2013 11 27268
## 12 2013 12 28135
```

```
(per_year <- summarise(per_month, flights = sum(flights)))

## # A tibble: 1 x 2
##   year flights
##   <int> <int>
## 1 2013 336776
```

Men var försiktig: Detta fungerar för summor och antal men inte för viktade medelvärden eller för rankningsmått, typ median. Med andra ord, summan av gruppvisa summor är totalsumman, men medianen för gruppvisa medianer är inte lika med medianen för hela datamängden.

Av-gruppera

OM du behöver ta bort grupperingen och fortsätta med avgrupperade data använd `ungroup()`.

```
daily %>%
  ungroup() %>%      # no longer grouped by date
  summarise(flights = n()) # all flights

## # A tibble: 1 x 1
##   flights
##   <int>
## 1 336776
```

Övningar

1. Föreslå en annan ansats som ger samma resultat som `not_cancelled %>% count(dest)` och `not_cancelled %>% count(tailnum, wt = distance)` utan att använda `count()`.
2. Vår definition på inställda flyg (`is.na(dep_delay) | is.na(arr_delay)`) är något suboptimal. Varför? Vilken är den viktigaste kolumnen?
3. Betrakta antalet inställda flighter per dag. Finns ett mönster? Är andelen inställda flighter relaterad till genomsnittlig försening?
4. Vilket plan har de största förseningarna? Går det att särskilja effekterna från "dåliga" flygplatser och "dåliga" flyg? Varför/Varför inte? Ledtråd: fundera på `flights %>% group_by(carrier, dest) %>% summarise(n())`.
5. Vad gör argumentet `sort` i `count()`? När kan du ha nytta av det?

Grupperade beräkningar och filtreringar

Gruppering är mest användbart samtidigt med `summarise()`, men det underlättar också vid beräkningar med hjälp av `mutate()` och `filter()`:

- Identifiera flighterna med de största förseningarna:

```
flights_sml %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 10)

## # A tibble: 3,306 x 7
## # Groups:   year, month, day [365]
##   year month  day dep_delay arr_delay distance air_time
##   <int> <int> <int>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 2013     1     1     853     851     184     41
## 2 2013     1     1     290     338    1134    213
```

```
## 3 2013 1 1 260 263 266 46
## 4 2013 1 1 157 174 213 60
## 5 2013 1 1 216 222 708 121
## 6 2013 1 1 255 250 589 115
## 7 2013 1 1 285 246 1085 146
## 8 2013 1 1 192 191 199 44
## 9 2013 1 1 379 456 1092 222
## 10 2013 1 2 224 207 550 94
## # ... with 3,296 more rows
```

- Identifiera alla destinationer med mer än 365 flighter:

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
popular_dests

## # A tibble: 332,577 x 19
## # Groups:   dest [77]
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>    <dbl>   <int>
## 1 2013     1     1    517             515         2     830
## 2 2013     1     1    533             529         4     850
## 3 2013     1     1    542             540         2     923
## 4 2013     1     1    544             545        -1    1004
## 5 2013     1     1    554             600        -6     812
## 6 2013     1     1    554             558        -4     740
## 7 2013     1     1    555             600        -5     913
## 8 2013     1     1    557             600        -3     709
## 9 2013     1     1    557             600        -3     838
## 10 2013     1     1    558             600        -2     753
## # ... with 332,567 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

- Standardisera för att beräkna grupp-mått:

```
popular_dests %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
  select(year:day, dest, arr_delay, prop_delay)

## # A tibble: 131,106 x 6
## # Groups:   dest [77]
##   year month   day dest arr_delay prop_delay
```

```
##   <int> <int> <int> <chr>   <dbl>   <dbl>
## 1 2013    1    1 IAH      11 0.000111
## 2 2013    1    1 IAH      20 0.000201
## 3 2013    1    1 MIA      33 0.000235
## 4 2013    1    1 ORD      12 0.0000424
## 5 2013    1    1 FLL      19 0.0000938
## 6 2013    1    1 ORD       8 0.0000283
## 7 2013    1    1 LAX       7 0.0000344
## 8 2013    1    1 DFW      31 0.000282
## 9 2013    1    1 ATL      12 0.0000400
## 10 2013   1    1 DTW      16 0.000116
## # ... with 131,096 more rows
```

- Funktioner som fungerar bäst för grupperade data (t.ex. summarise för summeringar) betecknas "window functions". Du kan läsa mer om dem i en vignette till dplyr: `vignette("window-functions")`

Övningar

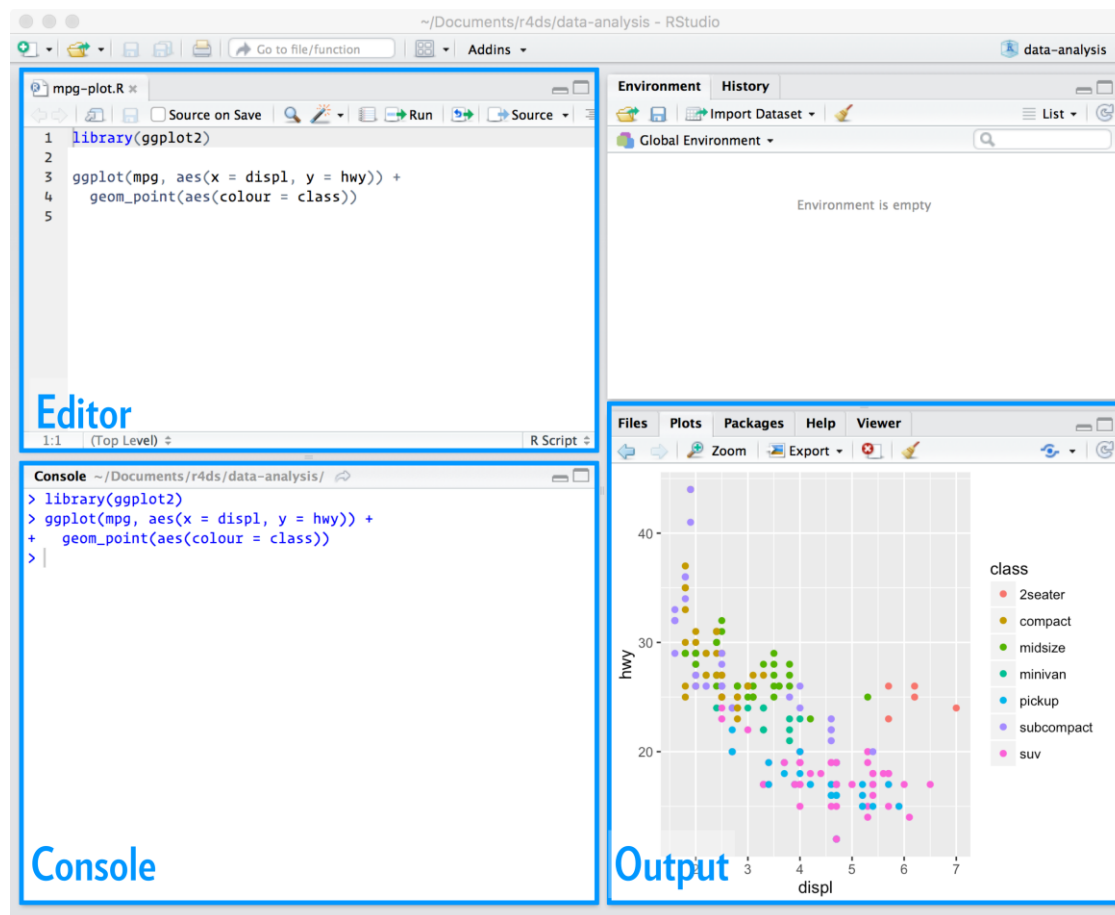
1. Prova hur beräknings- och filtreringsfunktionerna (`mutate()` och `filter()`) förändras då du använder dem på ogrupperade data jämfört med grupperade.
2. Vilket plan (`tailnum`) hade det sämsta utfallet på att vara on time?
3. Vilken tid på dagen skulle du flyga om du vill undvika förseningar så mycket som möjligt?
4. Beräkna det totala antalet minuter som flighterna är försenade per destination. För varje flight, beräkna andelen av den totala tiden för förseningar per destination.

Arbetsflöde: scripts

Hittills har vi använt fr.a. konsolen för att skriva och köra kod. Detta kan fungera hyggligt med relativt enkla script men mer komplexa script skriver man effektivare i Editorn, uppe till vänster.

Öppna upp den genom att klicka på **File/New file/R script**, eller använd kortkommandot **Ctrl+Shift+N**.

Då bör skärmen se ut ungefär så här:



Editorn är ett utmärkt ställe att placera kod som man vill behålla. Experimentera i konsolen och när du skrivit ett kodavsnitt, flytta upp det till konsolen. Rstudio sparar innehållet i Editorn då du avslutar Rstudio och öppnar det automatiskt nästa gång du startar upp.

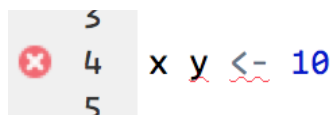
Att köra kod

Editorn är en utmärkt plats att bygga upp komplexa ggplot2-diagram eller längre sekvenser av dplyr-kod. Kom ihåg snabbkommandot **Ctrl+Enter** vilket exekverar den kod som markören finns på. Det innebär att även en längre sekvens som sträcker sig över flera rader körs så länge sekvensen är sammanhängande, t.ex. med hjälp av "the pipe" **%>%** eller i en ggplot med **+**.

Istället för att köra uttryck för uttryck kan du även exekvera hela scriptet i ett steg - kortkommandot `Ctrl+Shift+S`. Genom att använda det regelbundet förvissas du dig om att scriptet fungerar som tänkt. Det är en god vana att börja scriptet med att ladda in samtliga de moduler/packages du behöver. Det är en god hjälp för minnet då man återvänder till scriptet efter ett tag och om du delar ditt script med andra är det lätt för dem att se vilka moduler som behövs för att köra scriptet.

Diagnostics

Editorn har ett antal inbyggda verktyg för att identifiera syntaxfel. Skriv `x y <- 10`:

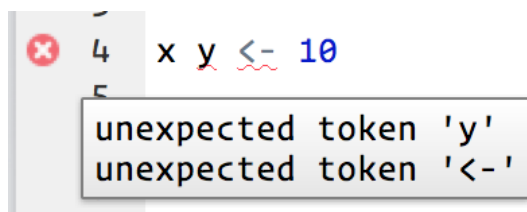


```

5
4 x y <- 10
5

```

och hovra över det röda krysset, notera popup-skylden



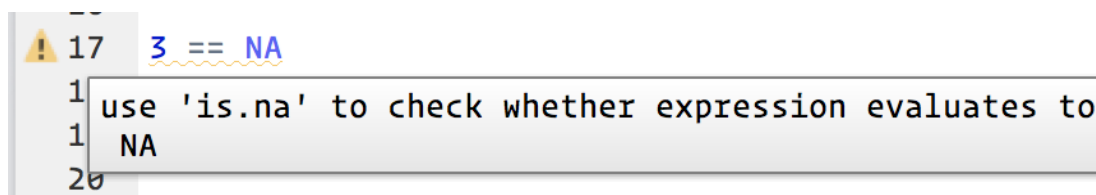
```

4 x y <- 10
5

```

unexpected token 'y'
unexpected token '<-'

Rstudio upmärksammar dig också på potentiella problem. Skriv `3 == NA`. Notera vad som händer då du hovrar över utropstecknet i marginalen.



```

17 3 == NA
1
1
20

```

use 'is.na' to check whether expression evaluates to NA

Praktik

1. Det finns en mängd tips och goda råd out there. Ett sådant ställe är Rstudio tips Twitter-konto, <https://twitter.com/rstudiotips>. Prova med att gå dit och leta upp något du finner intressant.
2. Vilka andra misstag kan Rstudio markera? Gå till Rstudios support-sida och kolla in <https://support.rstudio.com/hc/en-us/articles/205753617-Code-Diagnostics>

Explorativ analys av data

“Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question, which can always be made precise.” — John Tukey

Introduktion

Detta kapitel handlar om hur man kan använda visualisering och transformering för att undersöka data på ett systematiskt sätt, en process som kallas *explorativ dataanalys* (EDA). EDA är en iterativ process som innebär att

- Generera frågeställningar om data
- Söka svar genom att visualisera, transformera och modellera data
- Använd svaren till att förfinas frågeställningarna och/eller generera nya frågeställningar

EDA är inte en formell process med en uppsättning regler utan snarare ett förhållningssätt där du inledningsvis kan pröva varje idé som kommer upp hos dig, för att skärpa frågorna allteftersom din kunskap om datamängden blir bättre.

EDA är en viktig del av varje analys, även om frågeställningarna är givna från början, eftersom man alltid behöver undersöka kvaliteten i data. Datarensning är bara en del av EDA - du ställer frågor huruvida data möter dina förväntningar eller inte.

I det här avsnittet ska vi gå igenom ett sätt att rensa data genom att använda `ggplot2` och `dplyr`. Vi örjar med att ladda in

```
library(tidyverse)
```

Frågor

Målet är att utveckla en förståelse av datamängden. Hadley/Grolemund förespråkar frågor som det mest effektiva verktyget för att vägleda EDA. Genom att formulera en fråga fokuseras vår uppmärksamhet på en specifik del av datamängden och underlättar valet av grafer, transformeringar och modelleringar.

“EDA is fundamentally a creative process. And like most creative processes, the key to asking quality questions is to generate a large quantity of questions.”

Det finns förstås inga regler om vilka frågor som ska ställas när. Men det finns två slag av frågor som ofta är användbara för att förstå data bättre. De kan löst formuleras:

1. Vilken slags variation förekommer i variablerna?
2. Vilken slags samvariation förekommer mellan variablerna?

Resten av detta kapitel vrider och vänder i dessa två frågor. Låt oss börja med att definiera några begrepp: - En *variabel* är en kvantitet, kvalitet eller egenskap som är mätbar - Ett *värde* uttrycker nivån på en variabel när man mäter den och kan förändras från ett tillfälle till ett annat - En *observation* är en

uppsättning mätningar gjorda under liknande förhållanden, vanligtvis vid ett och samma tillfälle och på samma objekt. En observation innehåller en rad värden, vart och ett knutet till en specifik variabel. - *Tabulerade data* är en uppsättning värden vilka är knutna till en variabel och en observation. Tabulerade data är städade (*tidy*) om varje värde är placerade i sin egen cell, varje variabel i en egen kolumn och varje observation i en egen rad.

Vi ska lite senare ägna tid åt att städa data.

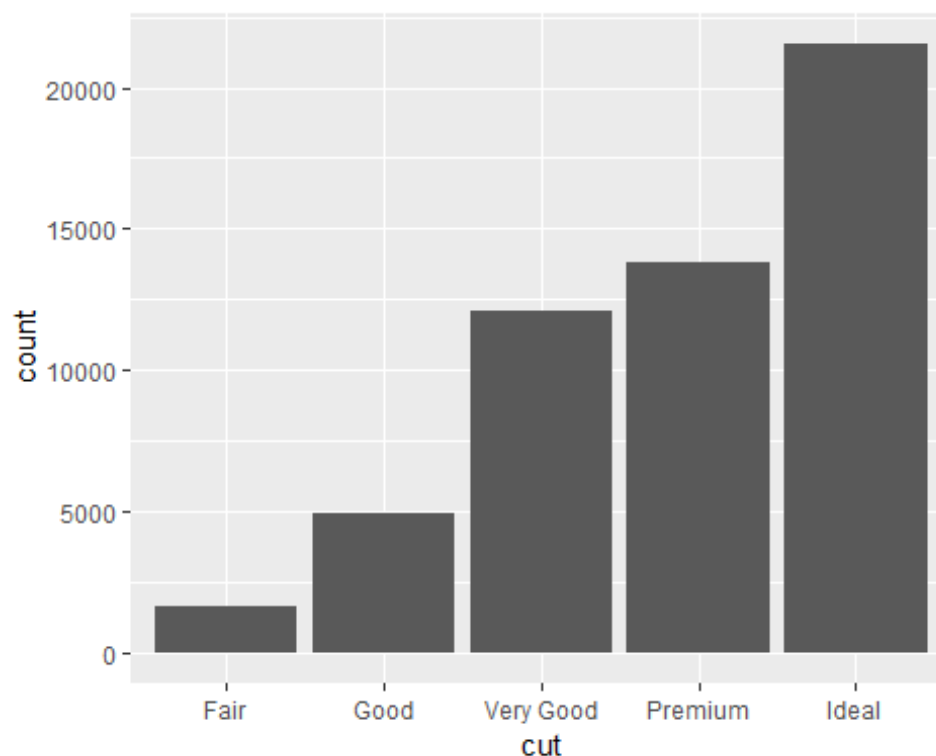
Variation

Variation är den tendens som en variabels värden har att variera från ett mättillfälle till ett annat. Varje variabel har sitt eget variationsmönster vilket kan ge värdefull information om variabeln. Det kanske bästa sättet att förstå ett variationsmönster, en fördelning, är att visualisera det.

Visualisera fördelningar

I R behandlas kategoriska variabler vanligtvis som faktorer (*factor*) eller alfanumeriska vektorer (*character*). För att undersöka fördelningen av en kategorisk variabel används ofta ett stapeldiagram, ett *bar chart*:

```
ggplot(data = diamonds) +  
  geom_bar(mapping = aes(x = cut))
```



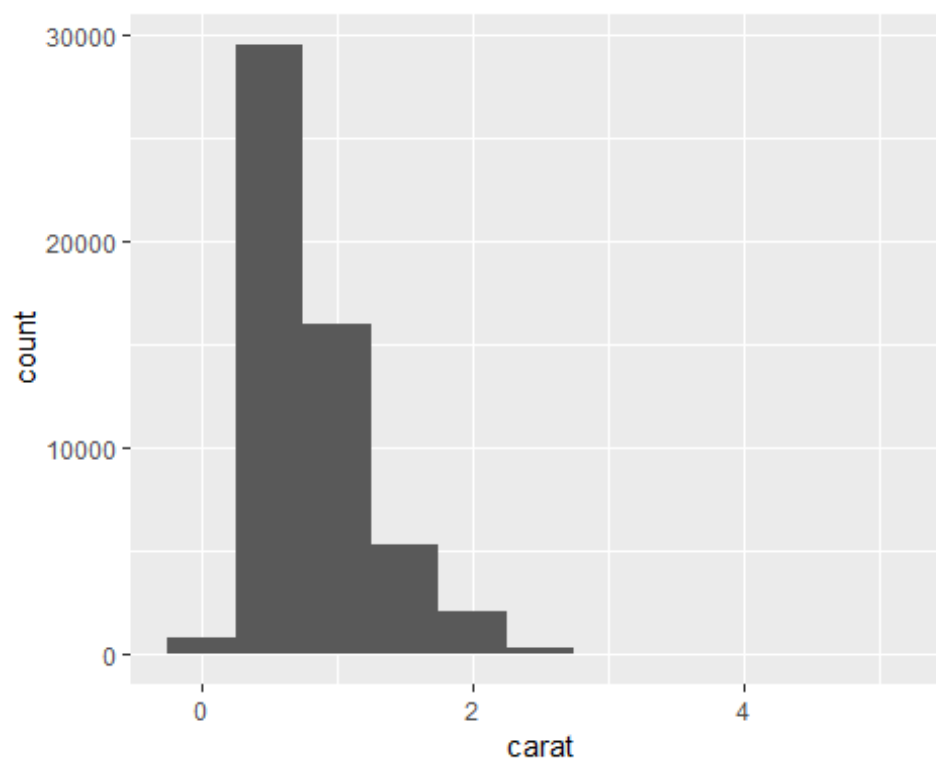
Höjden på staplarna visar hur många observationer som fanns för varje värde på `x`. Det kan man förstås också räkna ut manuellt med `dplyr::count()`:

```
diamonds %>%
  count(cut)

## # A tibble: 5 x 2
##   cut      n
##   <ord>  <int>
## 1 Fair   1610
## 2 Good   4906
## 3 Very Good 12082
## 4 Premium 13791
## 5 Ideal  21551
```

För att undersöka kontinuerliga variabler används ofta histogram:

```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.5)
```



Du kan beräkna detta manuellt genom att kombinera `dplyr::count()` och `ggplot2::cut_width()`:

```
diamonds %>%
  count(cut_width(carat, 0.5))
```

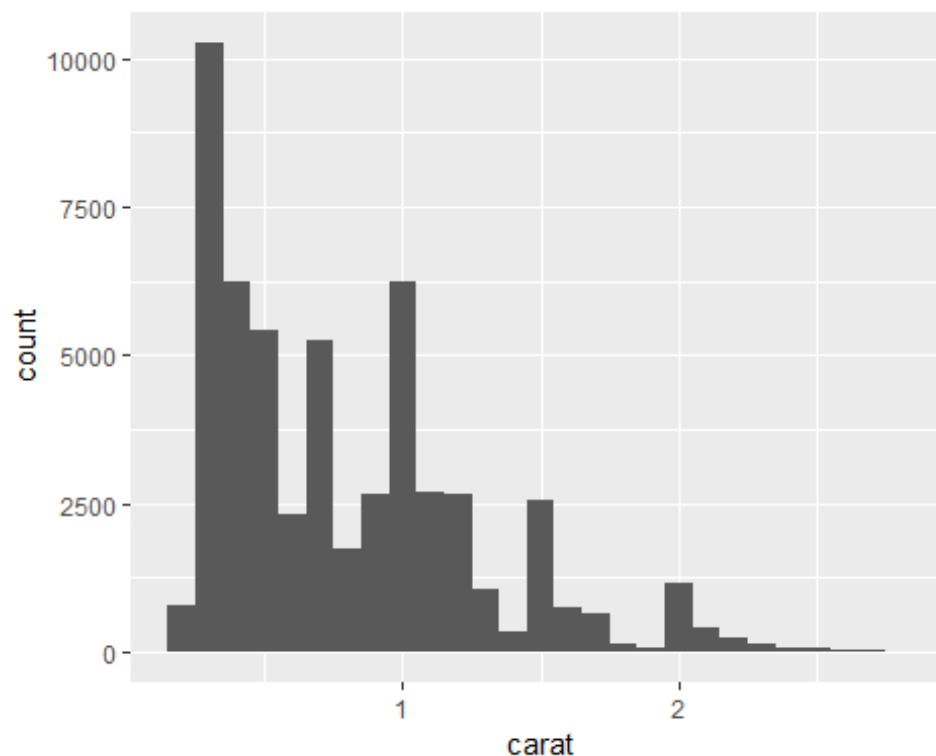
```
## # A tibble: 11 x 2
##   `cut_width(carat, 0.5)`  n
##   <fct>                  <int>
## 1 [-0.25,0.25]           785
## 2 (0.25,0.75]          29498
## 3 (0.75,1.25]          15977
## 4 (1.25,1.75]           5313
## 5 (1.75,2.25]           2002
## 6 (2.25,2.75]           322
## 7 (2.75,3.25]            32
## 8 (3.25,3.75]            5
## 9 (3.75,4.25]            4
## 10 (4.25,4.75]           1
## 11 (4.75,5.25]           1
```

Ett histogram delar x-axeln i likformigt breda *bins* och använder sedan höjden på y-axeln för att visa antalet observationer i varje bin. I grafen ovan innehåller den längsta stapeln nästan 30 000 observationer med ett värde på carat mellan 0.25 and 0.75, vilket motsvarar den vänstra resp högra kanten av stapeln.

Du kan bestämma bredden på intervallen i histogrammet själv genom `binwidth`, som anges i x-axelns enheter. Laborera gärna med flera olika bredder eftersom det kan påvisa olika mönster. Låt oss titta på ett histogram som zoomar in på diamanterna mindre än 3 karat och välj en mindre bredd:

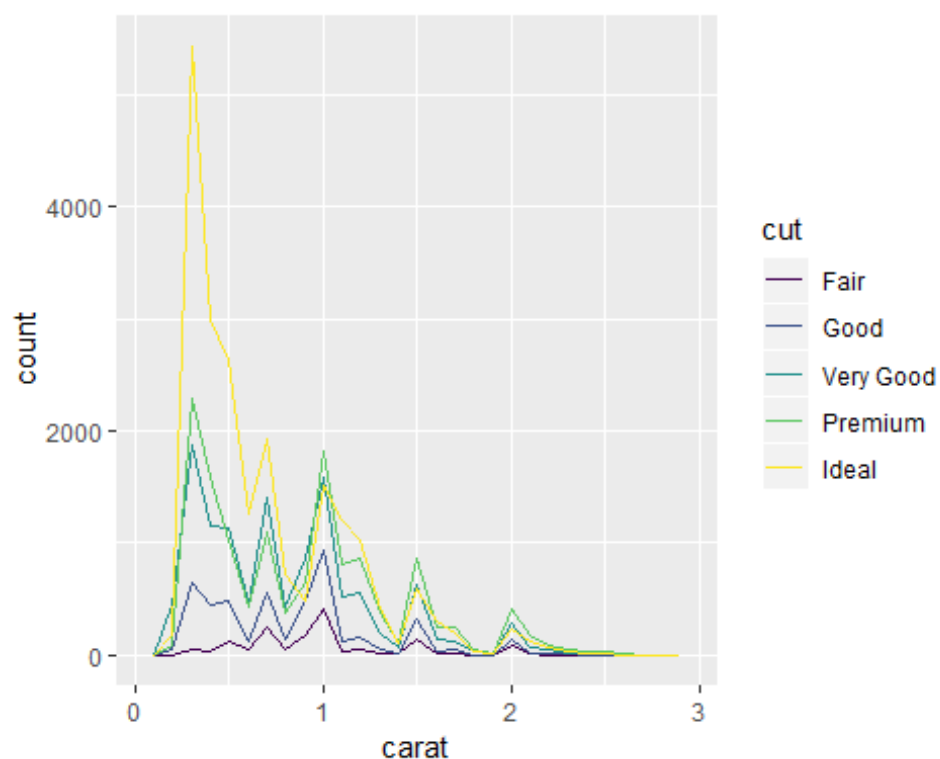
```
smaller <- diamonds %>%
  filter(carat < 3)

ggplot(data = smaller, mapping = aes(x = carat)) +
  geom_histogram(binwidth = 0.1)
```



Om du vill visualisera flera histogram i samma graf är det bättre att använda `geom_freqpoly()` istället för `geom_histogram()`. `geom_freqpoly()` gör samma beräkningar som `geom_histogram()`, men istället för att använda staplar används linjer. Det är lättare att förstå linjer som överlappar varandra än staplar. Prova:

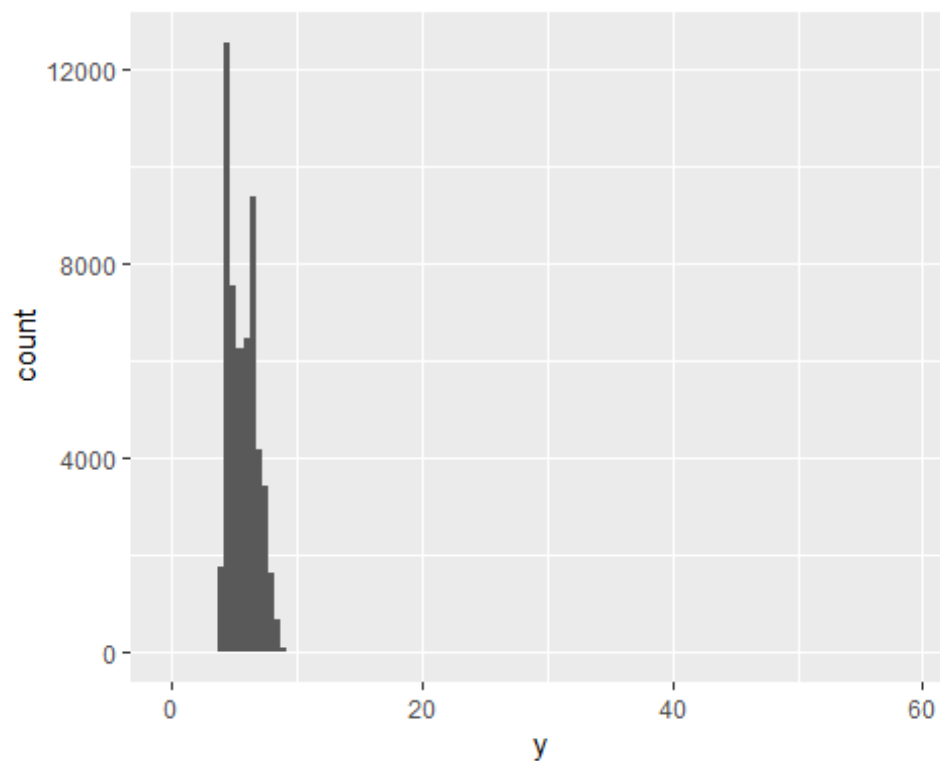
```
ggplot(data = smaller, mapping = aes(x = carat, colour = cut)) +  
  geom_freqpoly(binwidth = 0.1)
```



Outliers

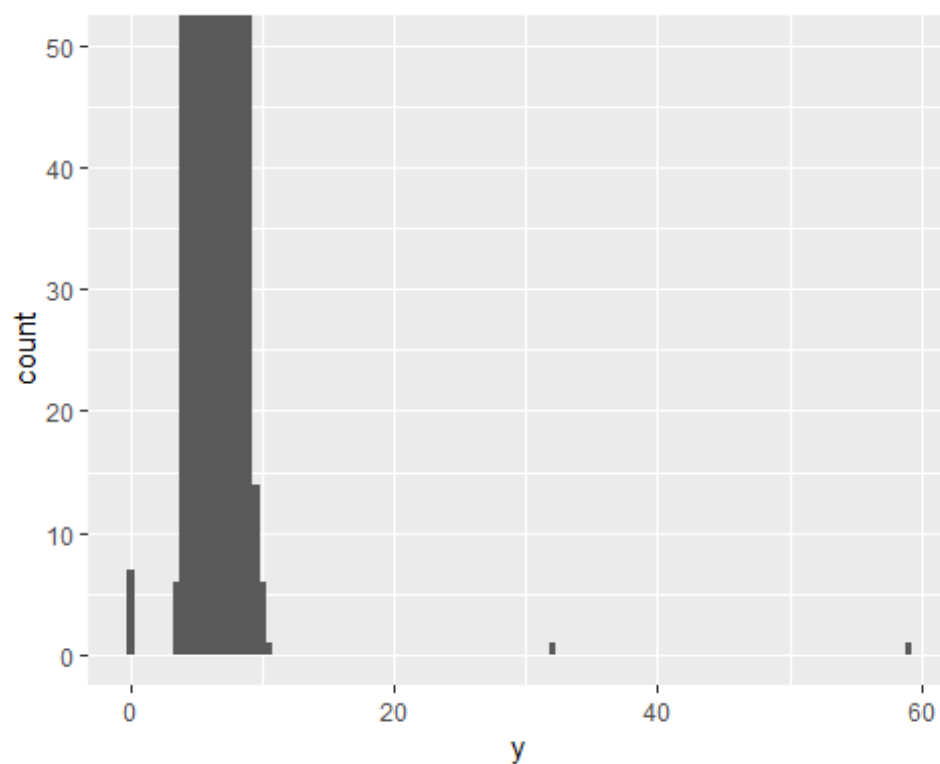
Outliers är observationer som sticker ut, datapunkter som inte verkar passa in i mönstret. Ibland handlar det om fel i indata, men någon gång om viktig ny information. Om datamängden är stor kan det vara svårt att urskilja outliers i ett histogram. Se t.ex. på fördelningen av variabeln `y` i datasetet `diamonds`. Den enda signalen om att det finns outliers är den oväntat omfångsrika x-axeln:

```
ggplot(diamonds) +  
  geom_histogram(mapping = aes(x = y), binwidth = 0.5)
```



Det finns så många observationer i de mer frekventa bins att de kortare bins:en är så korta att de inte syns. Vi behöver zooma in till de små värdena på y-axeln och för det kan vi använda `coord_cartesian()`:

```
ggplot(diamonds) +  
  geom_histogram(mapping = aes(x = y), binwidth = 0.5) +  
  coord_cartesian(ylim = c(0, 50))
```



(`coord_cartesian()` har också ett `xlim()` argument då du behöver zooma in på delar av x-axeln. `ggplot2` har också `xlim()` och `ylim()` funktioner vilka fungerar lite annorlunda: de kastar bort data som ligger utanför dessa gränser.)

Här kan vi se att det finns tre outliers: 0, ~30, and ~60. Vi tar bort dem genom att filtrera:

```
unusual <- diamonds %>%
  filter(y < 3 | y > 20) %>%
  select(price, x, y, z) %>%
  arrange(y)
unusual
```

```
## # A tibble: 9 x 4
##   price    x    y    z
##   <int> <dbl> <dbl> <dbl>
## 1  5139    0    0    0
## 2  6381    0    0    0
## 3 12800    0    0    0
## 4 15686    0    0    0
## 5 18034    0    0    0
## 6  2130    0    0    0
## 7  2130    0    0    0
## 8  2075  5.15 31.8  5.12
## 9 12210  8.09 58.9  8.06
```

Variabeln `y` mäter storleken av dessa diamanter i mm. Vi inser att diamanter inte kan ha en storlek av 0 mm så den observationen mste vara ett felvärde. Vi kan också misstänka att att diamanterna med 32 resp 59mm storlek är felvärden eftersom så stora diamanter måste vara värda miljoner.

Det är god praxis att köra analysen med och utan outliers. Om de har minimal effekt på resultaten är det rimligt att ersätta dem med missing values (NA). Men om de påverkar resultaten behöver man förstås fundera på vad som orsakade dessa värden och redovisa det på lämpligt sätt.

Övningar

1. Undersök fördelningen av `x`, `y` och `z`-värden i `diamonds`. Samband?
2. Undersök fördelningen av priset. Något som förvånar/överraskar? (använd flera värden på `binwidth`)
3. Hur många diamanter är 0.99 carat? Hur många är 1 carat? Vad orsakar skillnaden?
4. Jämför `coord_cartesian()` och `xlim()` eller `ylim()` när du zoomar in ett histogram. Vad händer om du inte ändrar `binwidth` ? Vad händer om du försöker zooma in så att bara halva stapeln syns?

Missing values

Ett sätt att hantera outliers som du bedömer vara orimliga/felaktiga är att ersätta dem med *missing values*, NA. Det enklaste sättet att göra det på är att använda `mutate()` för att ersätta outliern med NA eller ett annat lämpligt värde. Du kan använda `ifelse()`:

```
diamonds2 <- diamonds %>%
  mutate(y = ifelse(y < 3 | y > 20, NA, y))
```

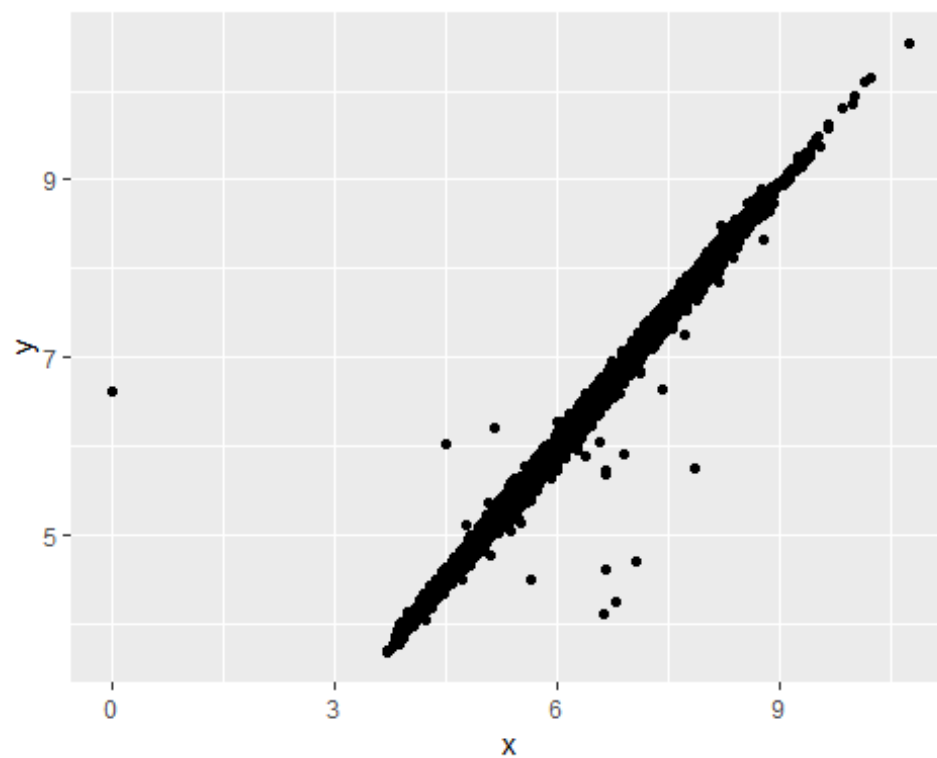
`ifelse()` har tre argument. Det första, `test`, måste vara en logisk vektor. Resultatet kommer att innehålla värdet på det andra argumentet, `yes`, om `test` är sant eller värdet av det tredje argumentet ifall `test` är falskt.

Ett alternativ till `ifelse()` är `dplyr::case_when()` som är speciellt användbar tillsammans med `mutate()` när du vill skapa en ny variabel som bygger på en mer komplex kombination av existerande variabler.

Liksom R i övrigt hyllar `ggplot2` filosofin att *missing values* ska aldrig bli missing i tysthet. det är långt ifrån klart hur man skulle plotta *missing values* så `ggplo2` exkluderar dessa värden från grafen men du får alltid en varning om att d har tagits bort:

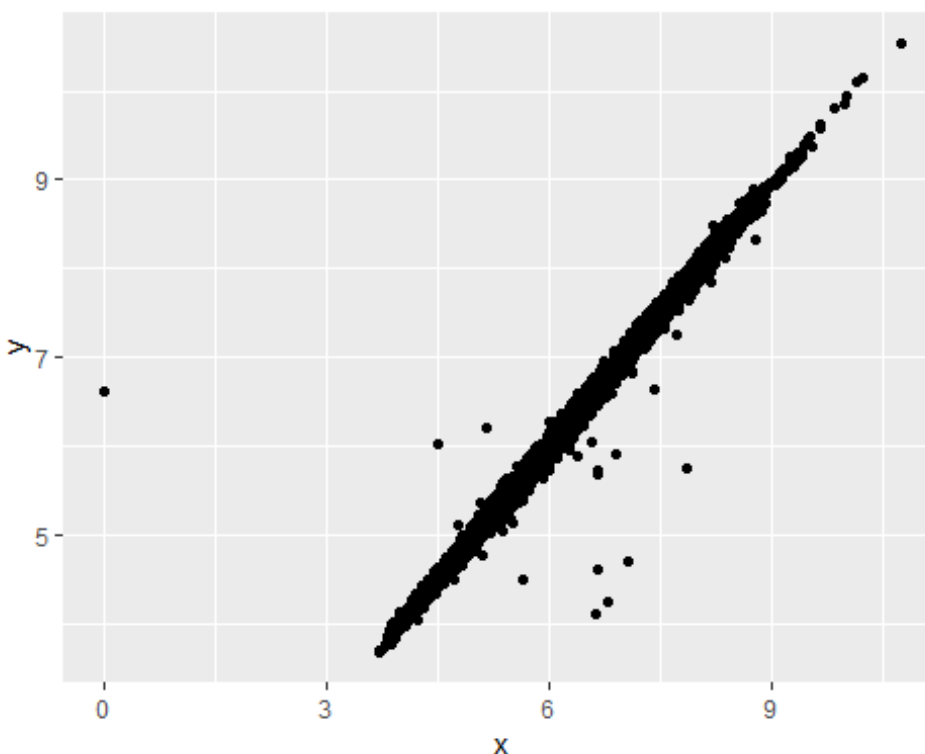
```
ggplot(data = diamonds2, mapping = aes(x = x, y = y)) +
  geom_point()

## Warning: Removed 9 rows containing missing values (geom_point).
```



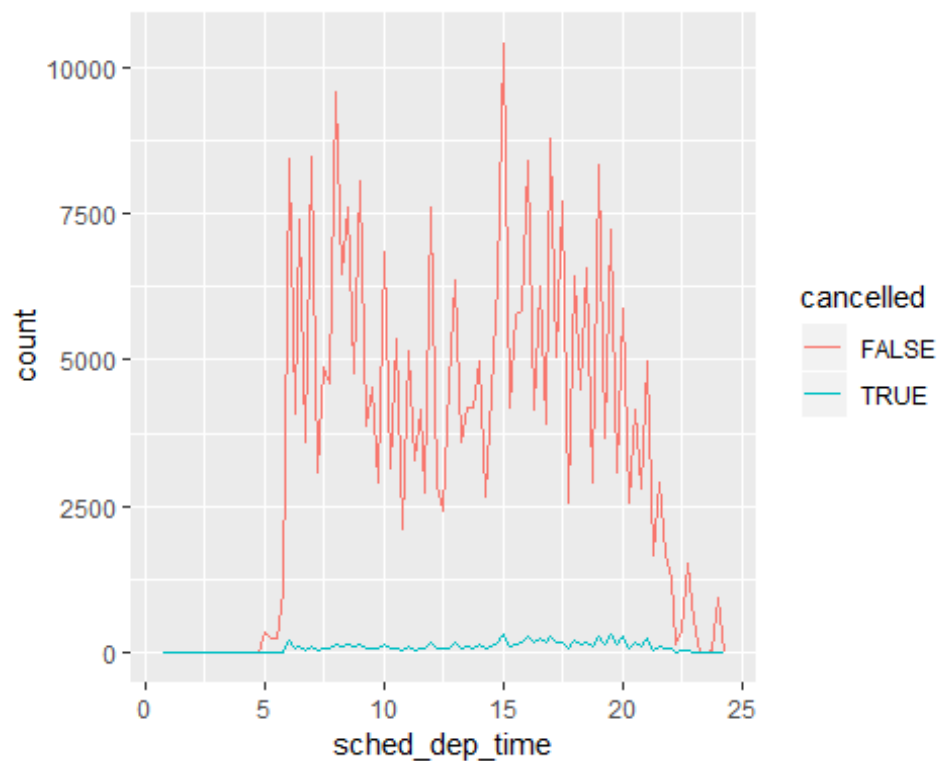
För att undertrycka varninen kan du använda `na.rm = TRUE`.

```
ggplot(data = diamonds2, mapping = aes(x = x, y = y)) +  
  geom_point(na.rm = TRUE)
```

Vid andra tillfällen kanske du vill visualisera vada det är som gör att *missing values* skiljer sig från andra värden. Till exempel, i `nycflights13::flights` indikerar *missing values* inställda flighter. Om du då vill jämföra de planerade avgångstiderna för inställda vs. icke-inställda flighter kan du göra det genom att skapa en ny variabel med hjälp av `is.na()`:

```
nycflights13::flights %>%
  mutate(
    cancelled = is.na(dep_time),
    sched_hour = sched_dep_time %/% 100,
    sched_min = sched_dep_time %% 100,
    sched_dep_time = sched_hour + sched_min / 60
  ) %>%
  ggplot(mapping = aes(sched_dep_time)) +
  geom_freqpoly(mapping = aes(colour = cancelled), binwidth = 1/4)
```



Men detta är inte en optimal graf eftersom det finns så många fler icke-inställda fligheter än inställda. Vi ska i nästa avsnitt kika på några sätt för att underlätta jämförelsen.

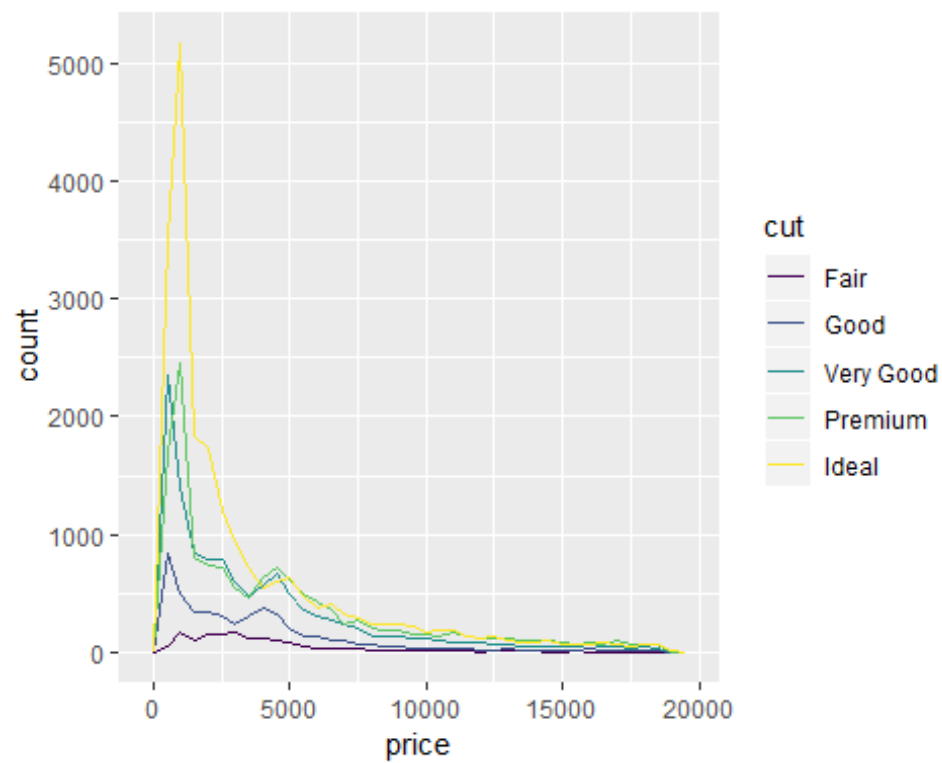
Samvarians - covariance

Om variation beskriver vad som händer *inom* en variabel beskriver samvariation (covariation) vad som sker *mellan* variabler. Visualisering är ett effektivt sätt för att upptäcka samvariation. Hur du ska åstadkomma visualiseringen beror på mellan vilka variabler du vill undersöka samvariationen.

En kategorisk och en kontinuerlig variabel

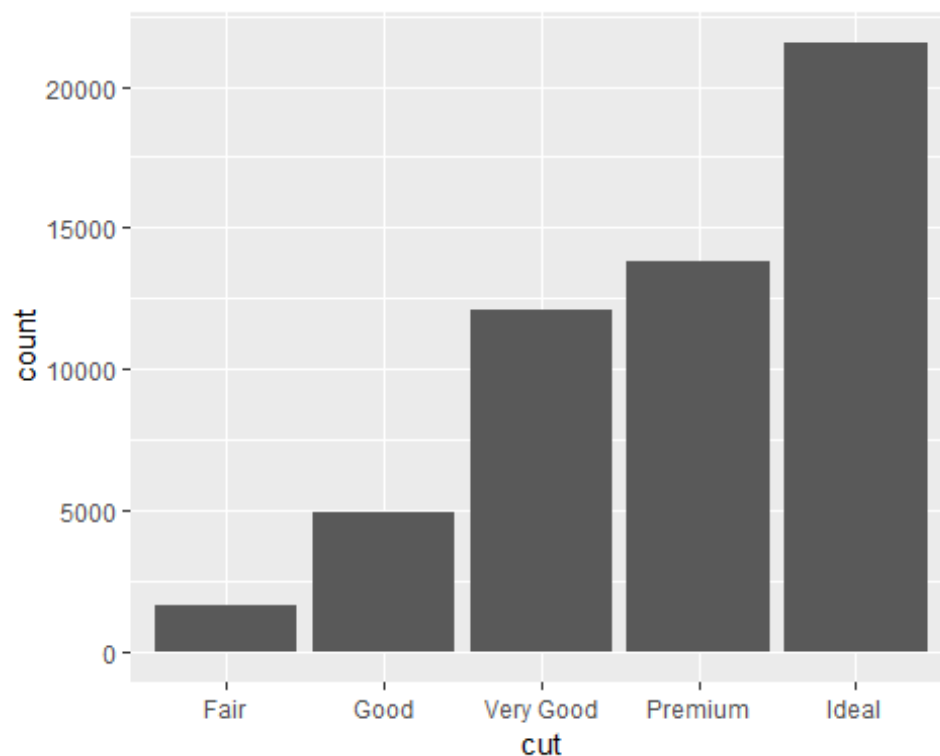
Standard-sättet för `geom_freqpoly()` är inte särskilt bra för den sortens jämförelser eftersom höjden beror på antalet observationer. Om en av grupperna innehåller få observationer blir det svårt att se skillnader mellan grupperna:

```
ggplot(data = diamonds, mapping = aes(x = price)) +  
  geom_freqpoly(mapping = aes(colour = cut), binwidth = 500)
```



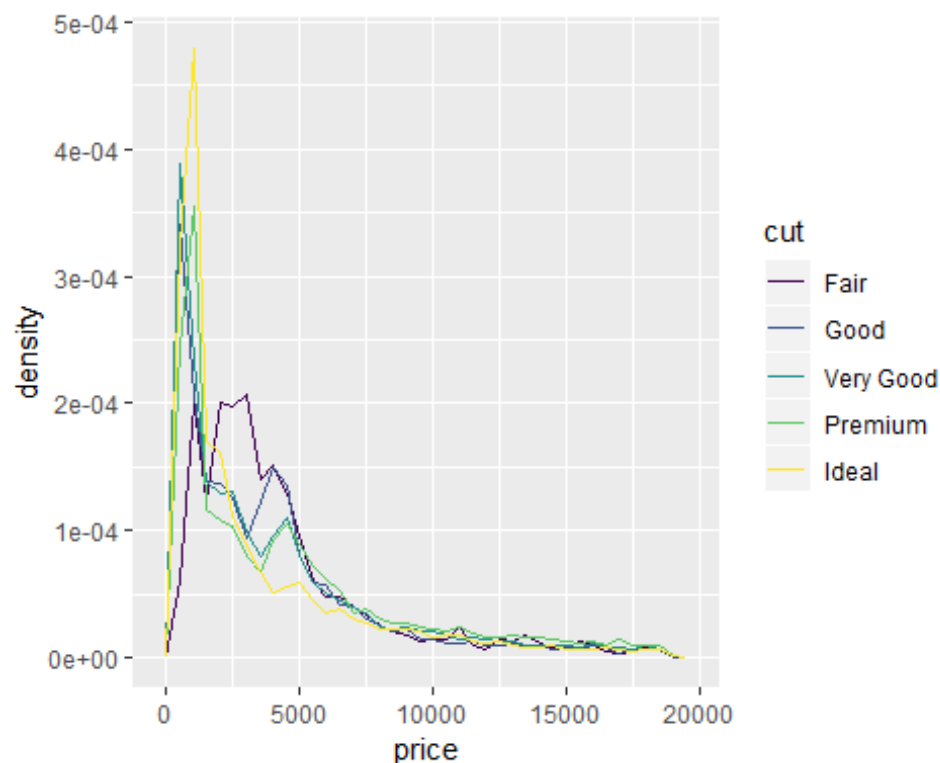
Det är svårt att se skillnader mellan grupperna eftersom det skiljer stort i gruppstorlek:

```
ggplot(diamonds) +  
  geom_bar(mapping = aes(x = cut))
```



För att underlätta jämförelsen behöver vi byta ut det som visas på y-axeln. Istället för att visa antal (count), visar vi täthet (density) vilket är antalet standardiserat så att ytan under varje frekvens-polygon summerar till 1.

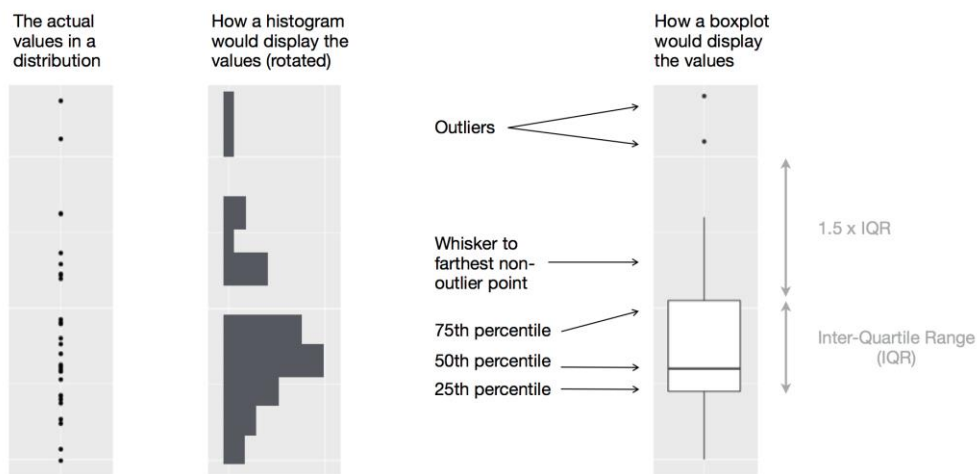
```
ggplot(data = diamonds, mapping = aes(x = price, y = ..density..)) +  
  geom_freqpoly(mapping = aes(colour = cut), binwidth = 500)
```



Det är något knepigt med denna graf - det verkar som om diamanterna med lägst kvalitet (fair) är dyrast i genomsnitt.

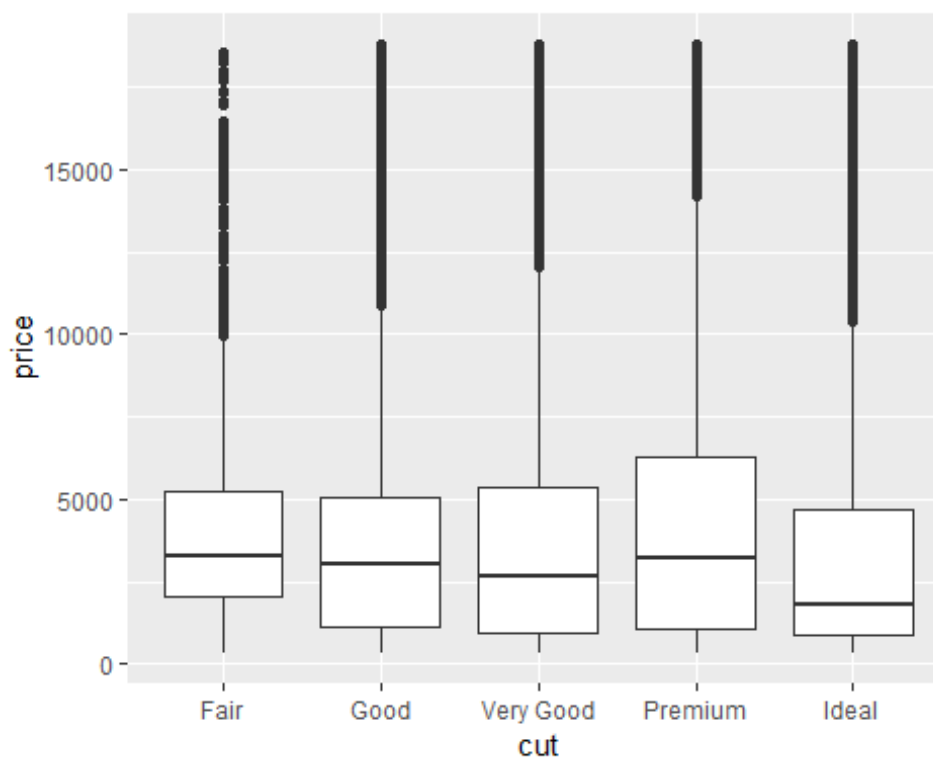
Ett alternativt sätt att visa g fördelningen av en kontinuerlig variabel är att använda en boxplot. Den består av

1. En box med utsträckningen 25 - 75 percentilerna (= the interquartile range (IQR)). I mitten av boxen en markering för medianen.
2. Punkter som visar observationer som faller utanför 1,5 ggr IQR från vardera kanterna av boxen
3. En linje som sträcker sig från vardera kanten på boxen till den observation längst bort men som inte är en outlier.



Låt oss se hur prisdistributionen ser ut med boxplots:

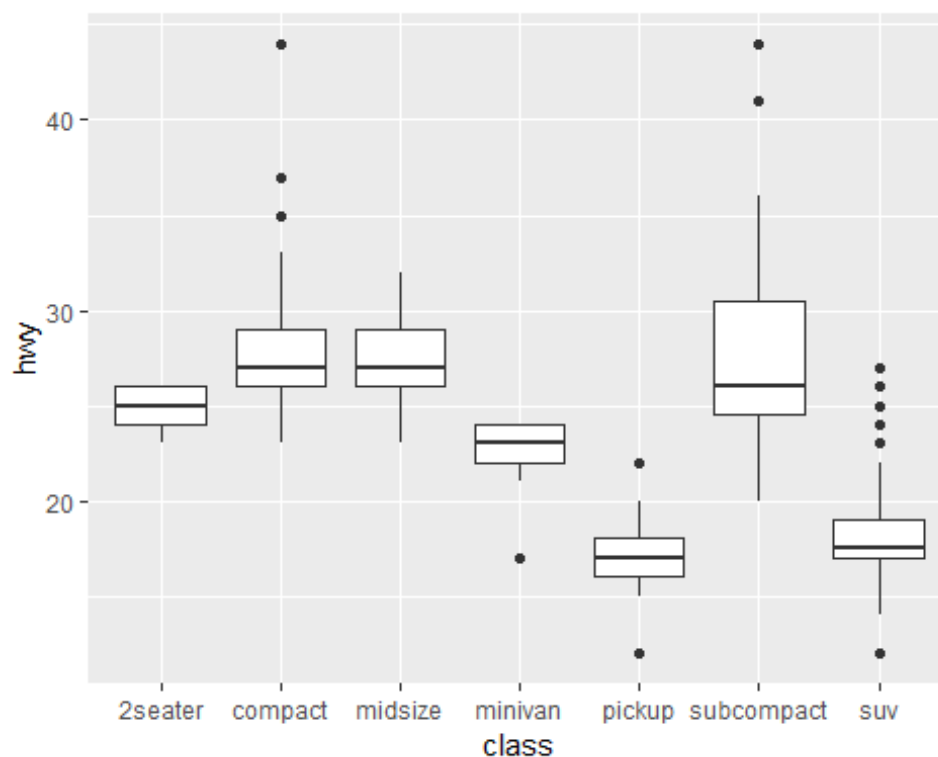
```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) +  
  geom_boxplot()
```



Vi får mindre information om fördelningen men boxplots är kompaktare vilket underlättar jämförelsen och fortfarande får vi intryck av att diamanter med lägre kvalitet är dyrare.

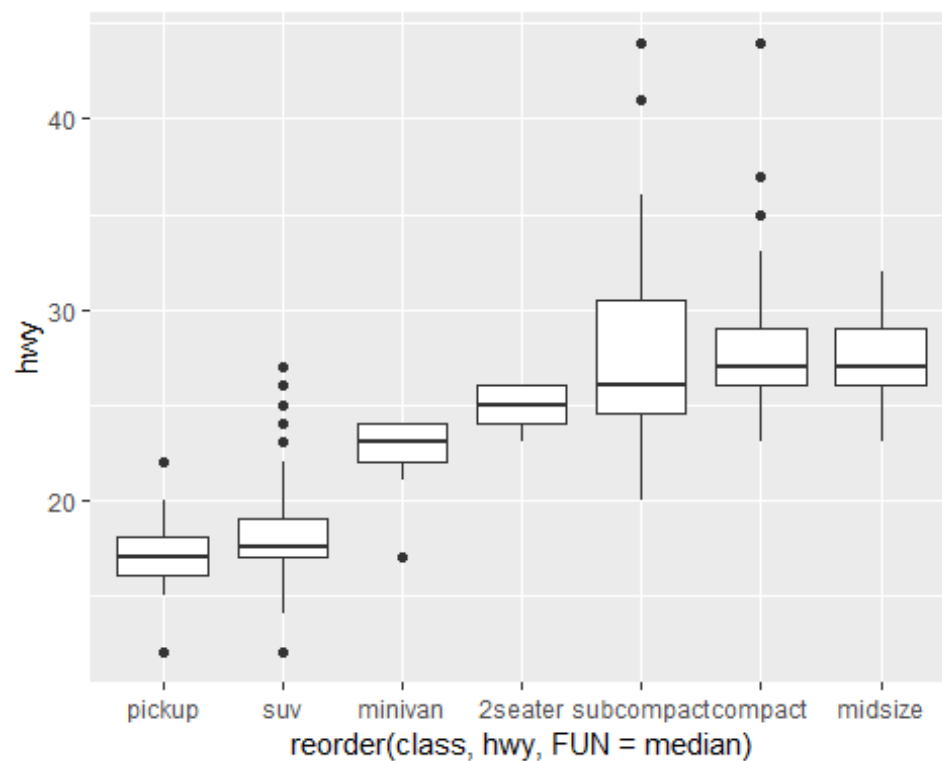
Ibland kan du behöva ändra ordningen av kategorierna för att göra grafen mer lättläst. Det kan du göra med hjälp av `reorder()`. Till exempel, låt oss kika på variabeln `class` i datasetet `mpg`. Du kanske vill se hur bränsleeffektiviteten (`hwy`) varierar mellan biltyperna (`class`):

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +  
  geom_boxplot()
```



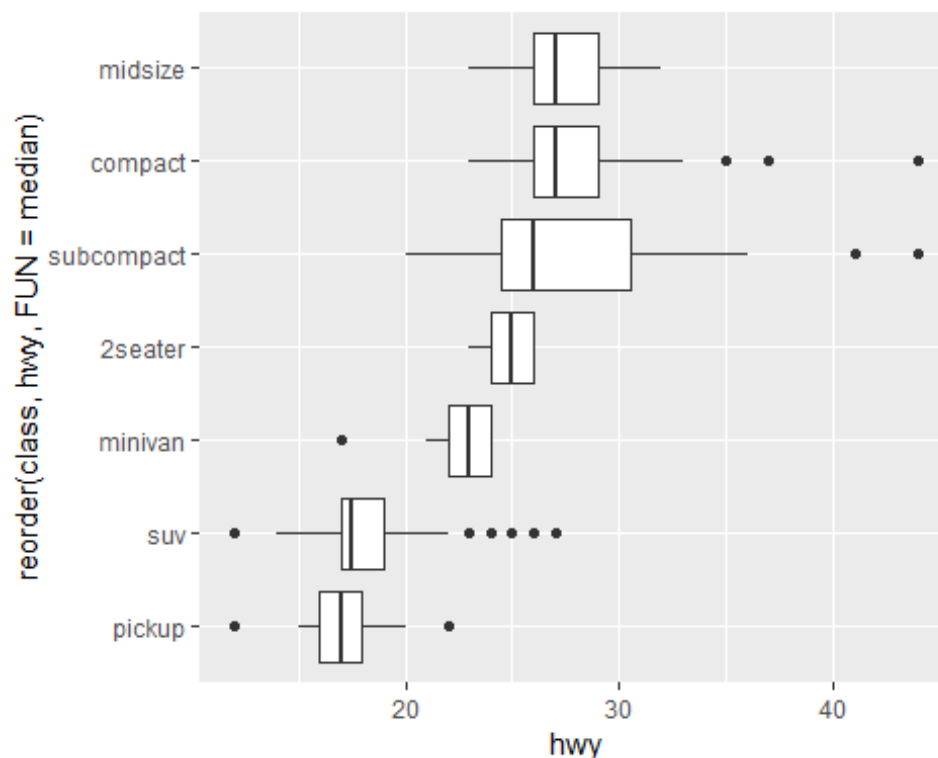
För att se trenden tydligare kan du ordna om classbaserat på median-värdet av `hwy`:

```
ggplot(data = mpg) +  
  geom_boxplot(mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy))
```



Om variabelnamnen är långa fungerar boxplot bättre om du roterar grafen 90 grader. Det kan du göra med hjälp av `coord_flip()`:

```
ggplot(data = mpg) +  
  geom_boxplot(mapping = aes(x = reorder(class, hwy, FUN = median), y = hwy)) +  
  coord_flip()
```

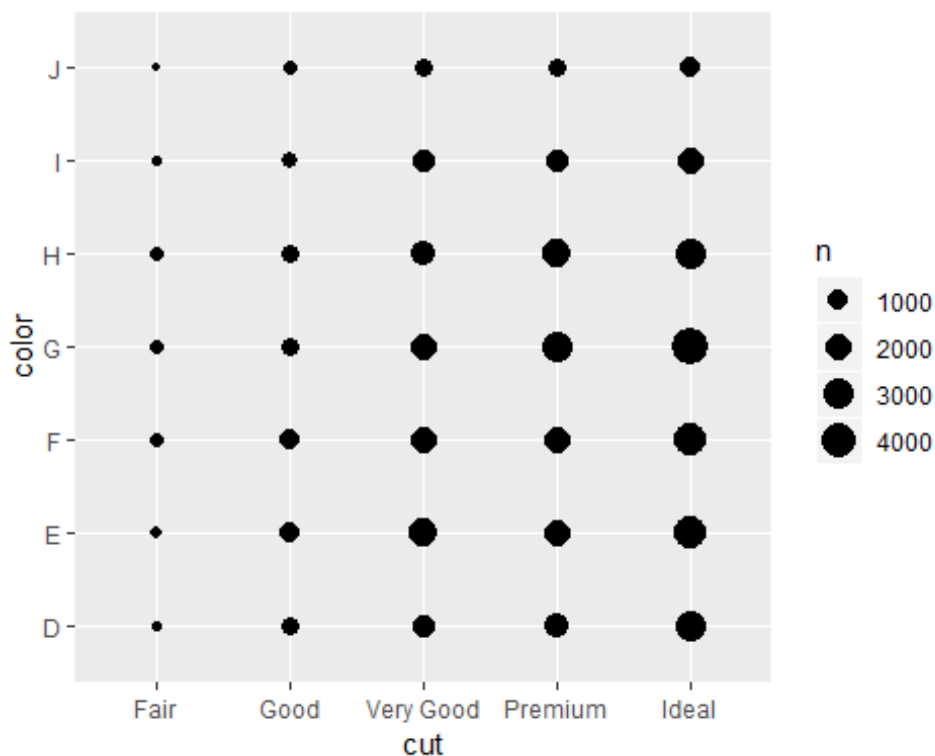
Övningar

1. Vilken variabel i datasetet diamonds är viktigast för att förklara priset på en diamant? Hur korrelerar den variabeln med kvaliteten (cut)? Hur kan kombinationen av dessa två relationer förklara att diamanter med lägre kvalitet förefaller vara dyrare?

Två kategoriska variabler

För att visualisera samvariationen mellan två kategoriska variabler behöver du beräkna antalet observationer för varje unik kombination av de två variablerna. Ett sätt att göra det är att använda den inbyggda funktionen `geom_count()`:

```
ggplot(data = diamonds) +  
  geom_count(mapping = aes(x = cut, y = color))
```



Storleken på varje cirkel markerar hur många observationer som finns under varje kombination.

Ett alternativt sätt är att beräkna antalet observationer med `dplyr`:

```
diamonds %>%
  count(color, cut)

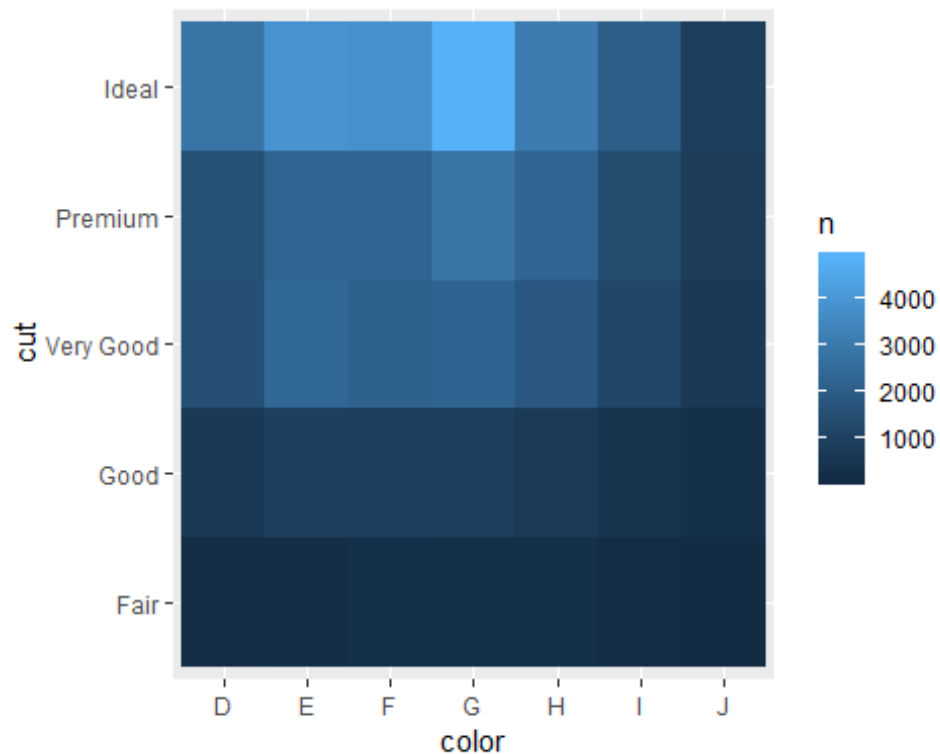
## # A tibble: 35 x 3
##   color cut      n
##   <ord> <ord> <int>
## 1 D    Fair    163
## 2 D    Good    662
## 3 D    Very Good 1513
## 4 D    Premium 1603
## 5 D    Ideal   2834
## 6 E    Fair    224
## 7 E    Good    933
## 8 E    Very Good 2400
## 9 E    Premium 2337
## 10 E   Ideal   3903
## # ... with 25 more rows
```

Sedan kan du visualisera detta med `geom_tile()` och argumentet `fill`:

```

diamonds %>%
  count(color, cut) %>%
  ggplot(mapping = aes(x = color, y = cut)) +
  geom_tile(mapping = aes(fill = n))

```



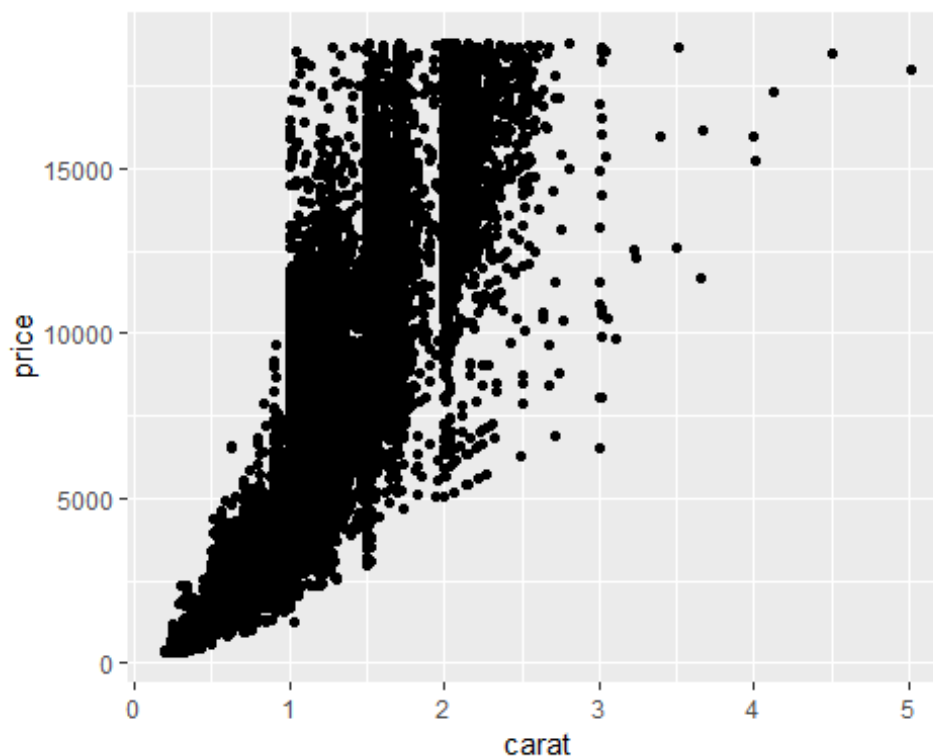
Två kontinuerliga variabler

Ett uppenbart sätt att visualisera sambvariation mellan två kontinuerliga variabler är förstås en scatterplot:

```

ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price))

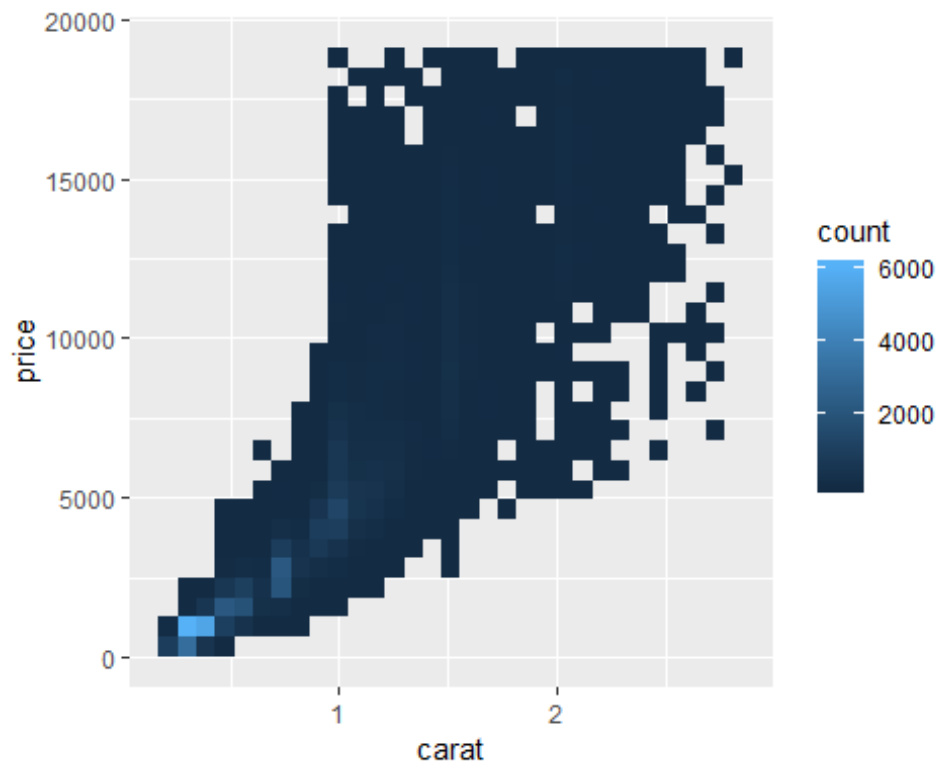
```



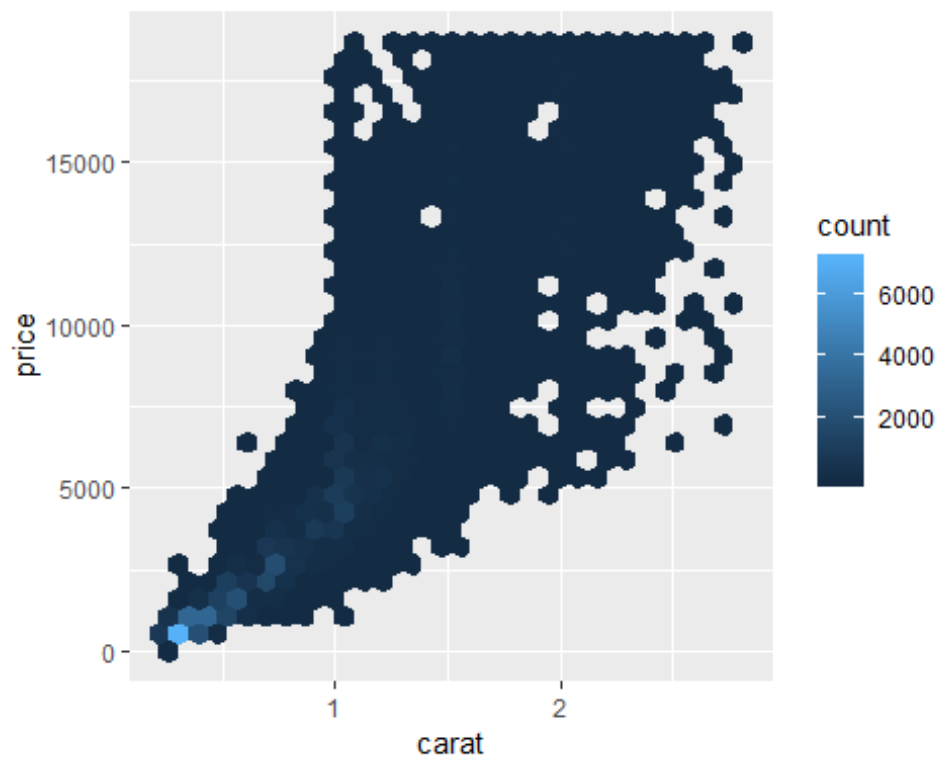
Men dessa grafer blir svårare att läsa om antalet observationer är stort. Vi har tidigare använt argumentet `alpha` som gör punkterna mer eller mindre genomskinliga och på så sätt rundar problemet med överlappning. Ett annat sätt att visualisera samvariation då man har många observationer är att använda `geom_bin2d()` eller `geom_hex()`.

`geom_bin2d()` och `geom_hex()` delar in koordinatsystemet i 2-dimensionella *bins* och använder sedan en färg (`fill`) för att markera antalet observationer inom varje *bin*. `geom_bin2d()` använder rektangulära *bins* och `geom_hex()` använder hexagonala. Du behöver installera modulen `hexbin` för att använda `hex_bin()`

```
ggplot(data = smaller) +  
  geom_bin2d(mapping = aes(x = carat, y = price))
```

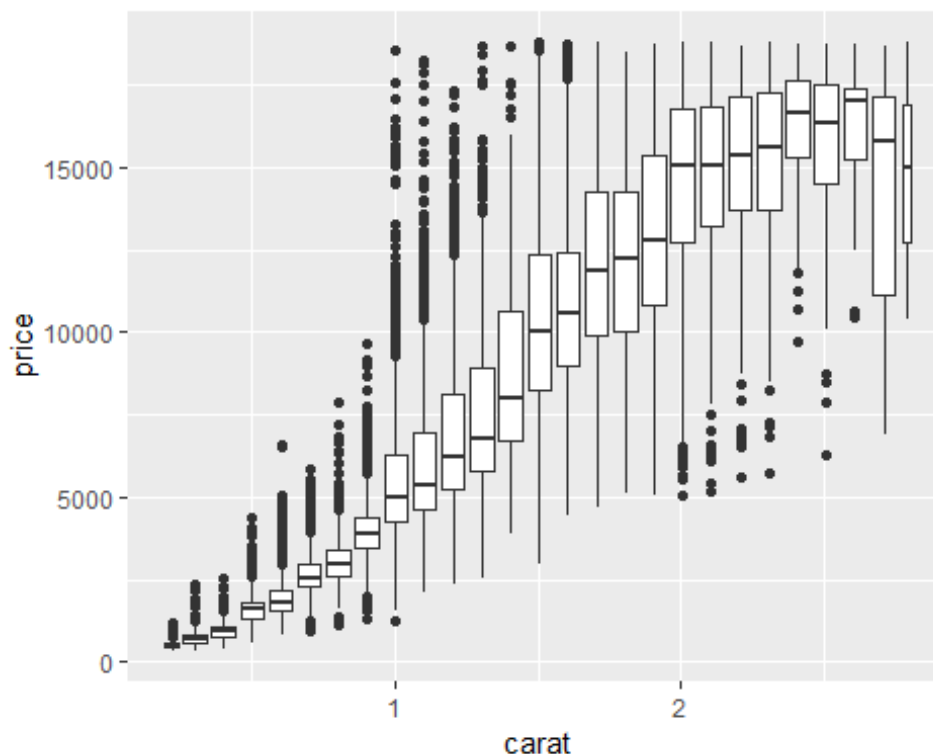


```
# install.packages("hexbin")  
ggplot(data = smaller) +  
  geom_hex(mapping = aes(x = carat, y = price))
```



Ett ytterligare sätt är att dela upp en av de kontinuerliga variablerna så att den fungerar som en kategorisk. Till exempel, kan du dela upp `carat` och sedan göra en boxplot för varje grupp:

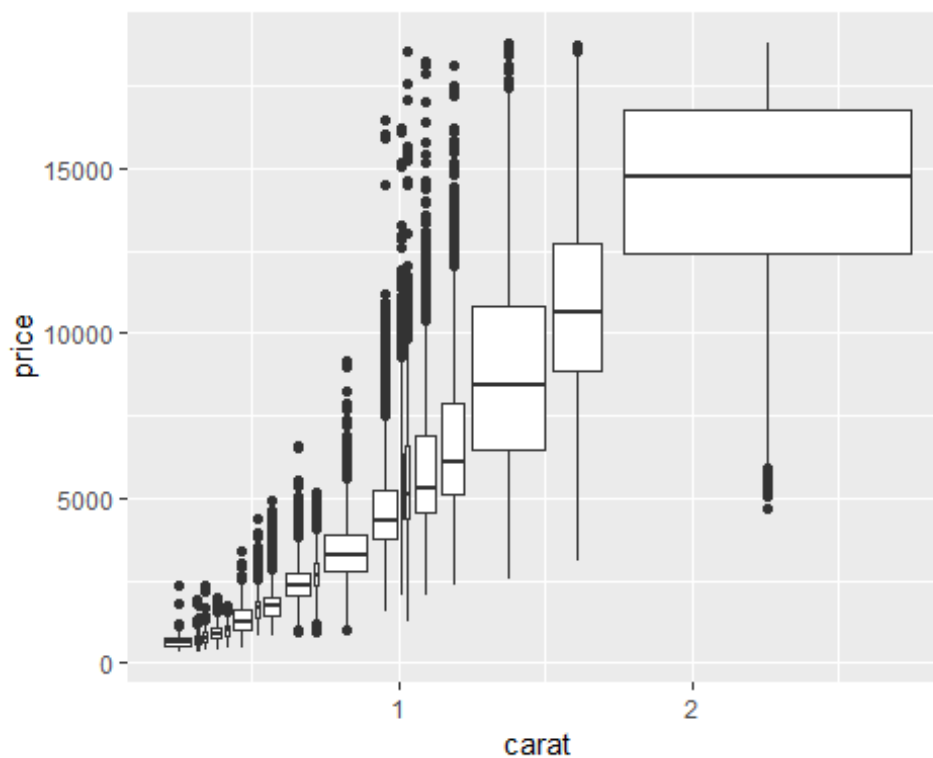
```
ggplot(data = smaller, mapping = aes(x = carat, y = price)) +  
  geom_boxplot(mapping = aes(group = cut_width(carat, 0.1)))
```



`cut_width(x, width)` delar upp `x` i delar med bredden `width`. Som standard ser boxplots ungefär lika smala ut oberoende av antalet observationer så det är svårt att se hur många observationer som varje boxplot rymmer. Men du kan variera bredden på boxplot:en genom att göra den proportionell mot antalet observationer med hjälp av `varwidth = TRUE`.

Ett alternativt sätt är att fördela ungefär lika många observationer i varje *bin*. För detta använder du `cut_number()`:

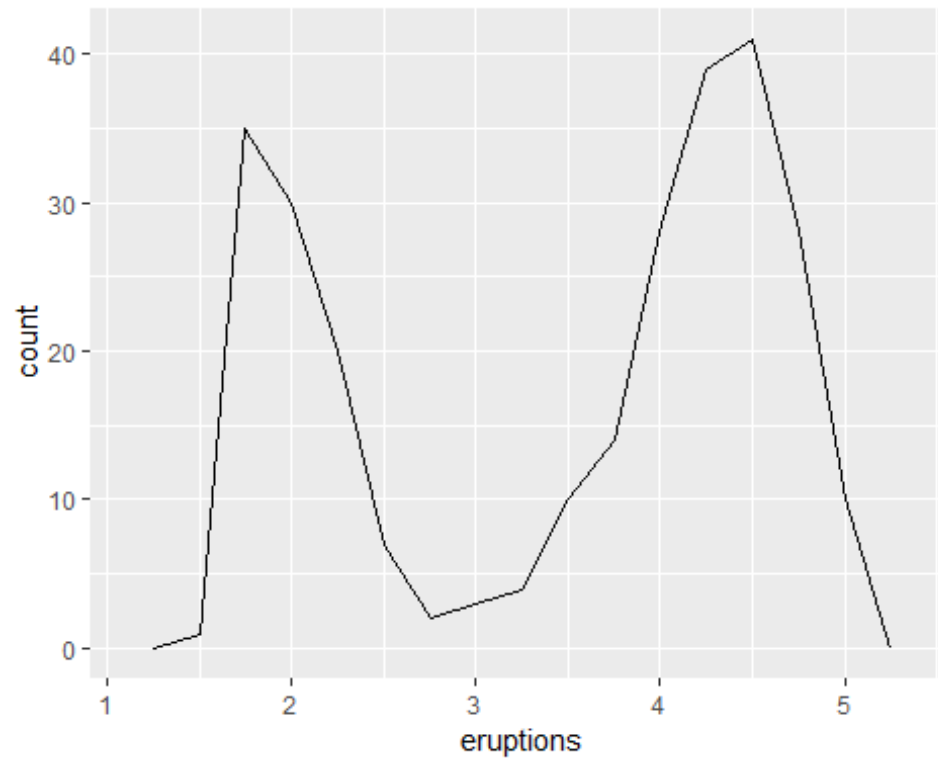
```
ggplot(data = smaller, mapping = aes(x = carat, y = price)) +  
  geom_boxplot(mapping = aes(group = cut_number(carat, 20)))
```



ggplot2 calls

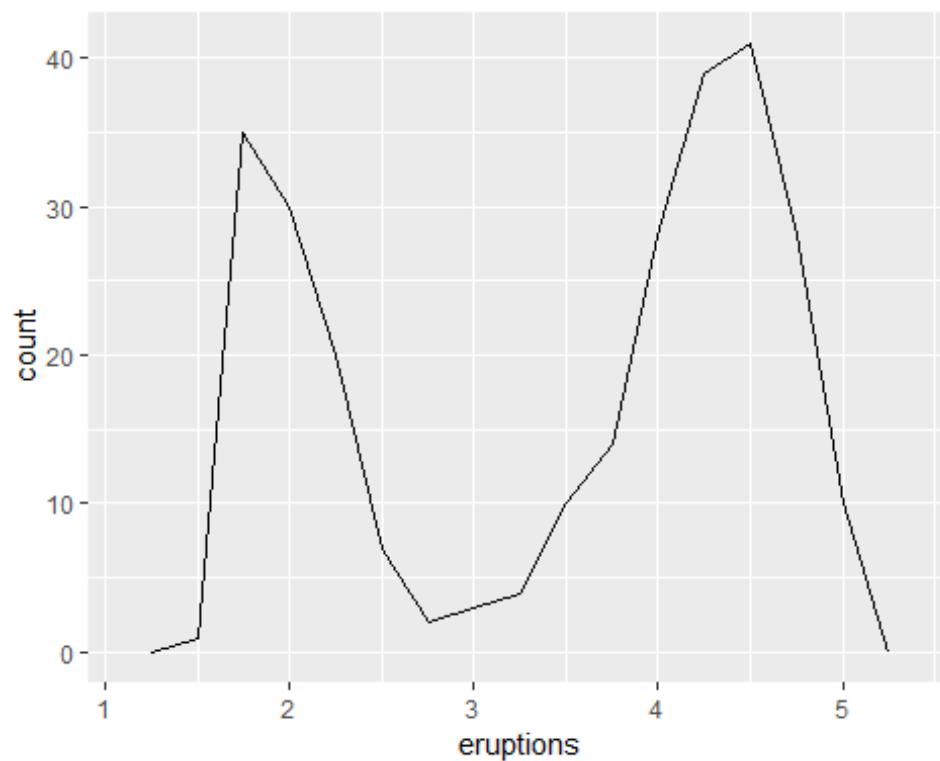
När vi nu börjar komma till slutet av början är det dags att kika på ett mer koncist sätt att skriva kod i ggplot2. Hittills har vi skrivit koden så att den blir riktigt tydlig vilket förstås är bra för att lära in språket:

```
ggplot(data = faithful, mapping = aes(x = eruptions)) +  
  geom_freqpoly(binwidth = 0.25)
```



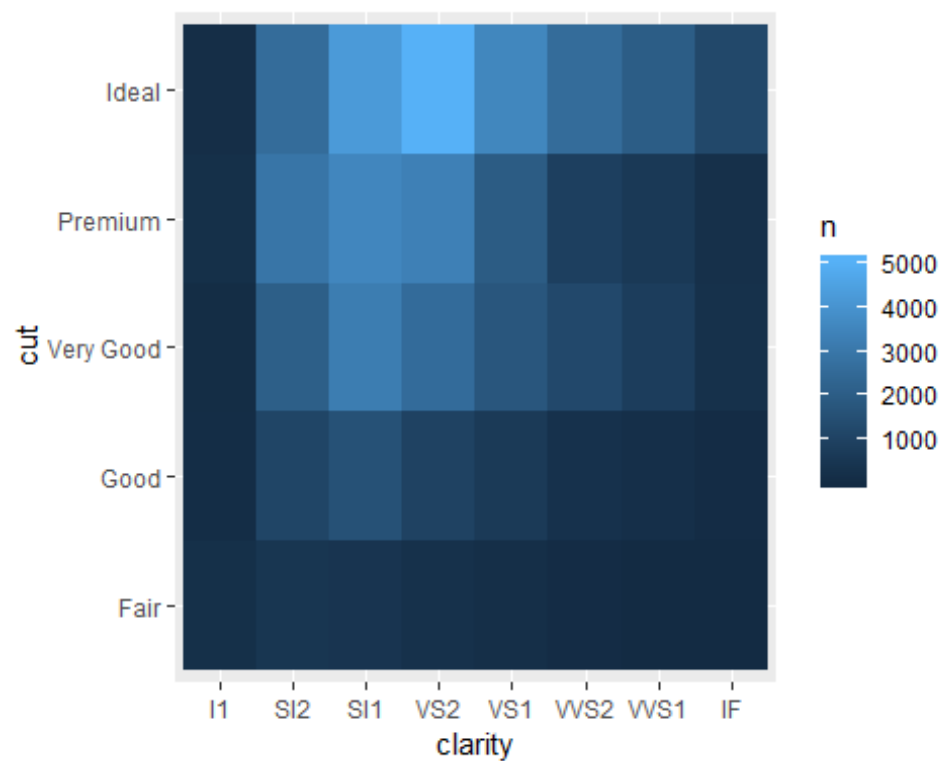
men man kan förenkla koden genom att utelämna argumentens namn:

```
ggplot(faithful, aes(eruptions)) +  
  geom_freqpoly(binwidth = 0.25)
```

Ibland vill du kombinera en “pipeline” med en graf. se upp med övergången från `%>%` till `+`:

```
diamonds %>%
  count(cut, clarity) %>%
  ggplot(aes(clarity, cut, fill = n)) +
  geom_tile()
```



Att lära mer

Om du vill veta mer om `ggplot2` rekommenderas den dedikerade läroboken <https://amzn.com/331924275X>. Tyvärr inte tillgänglig fritt å nätet. Det är däremot *R Graphics Cookbook* av Winston Chang, åtminstone till större delen. Se <http://www.cookbook-r.com/Graphs/>.

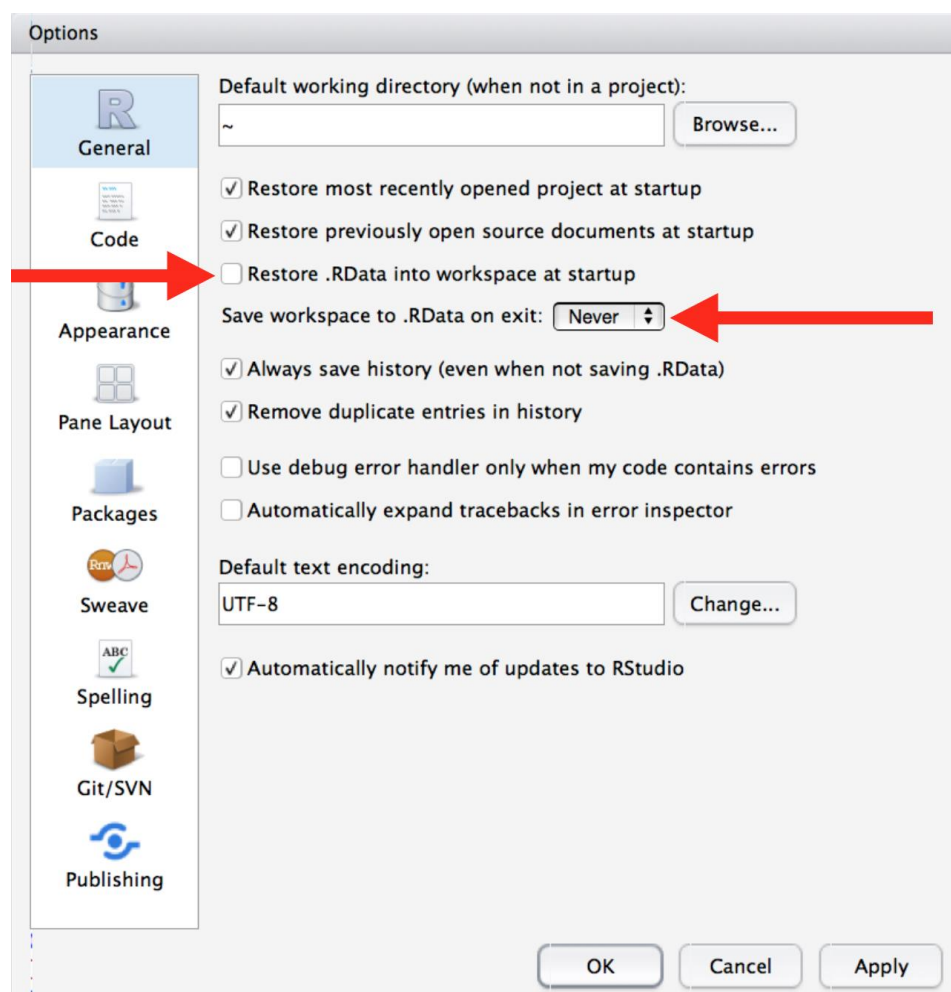
Arbetsflöde: Projekt

Få av oss har förmånen att jobba med en analys i taget, göra färdigt och sedan gå över till nästa arbetsuppgift - oftast håller man flera bollar i luften samtidigt. Då uppstår behovet att hålla isär olika analyser eller att kunna sortera bort script som man inte längre behöver och spara dem man vill använda igen. I RStudio kan sådant hanteras med hjälp av projekt, *Projects*. Det handlar ofta om svar på två frågor:

- Vad är viktigt att spara av det jag arbetar med?
- Vart "lever" analysen, dvs var nånstans kan jag plocka upp det aktuella projektet igen?

Vad är viktigt?

Istället för att betrakta resultaten av exekveringarna som analysens råmaterial *bör man betrakta scripten som ett sådant råmaterial*. Med hjälp av scripten kan man snabbt återskapa resultaten vilket är betydligt lättare än att återskapa scripten. Det är t.o.m. en poäng att aldrig spara resultaten mellan sessionerna! Det kan man undvika genom att kryssa i nedanstående rutor i "Options".

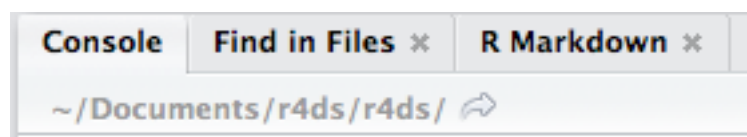


Det är vidare klokt att flitigt använda nedanstående kortkommandon för att förvissa dig om att du fångat de viktiga delarna av din kod i editorn:

1. Ctrl + Shift + F10 för att starta om RStudio.
2. Ctrl + Shift + S för att köra det aktuella scriptet igen.

Vart lever analysen? Arbetsbiblioteket

Arbetsbiblioteket (the working directory) är en viktig del av *Project* i Rstudio. Det är den plats där R letar efter filer som du vill ladda in och där R sparar de filer du vill spara. Rstudio visar den aktuella sökvägen i toppen av konsolen.



Du kan även få sökvägen via funktionen `getwd()`. Prova:

```
getwd()
```

```
## [1] "C:/Users/Goran/Documents/R/KurskompendiumRintrduktion180228"
```

Sökvägar och bibliotek/mappar

R hanterar båda sätten att definiera sökvägar: Mac/Linux och Windows, dvs / alternativt \. Backslash har i R en speciell betydelse och därför behöver man använda dubbla backslash \\.

Så varför inte använda Mac/Linux-varianten?

Använd aldrig absoluta sökvägar, dvs som börjar med C: eller \\, eftersom det gör det krångligare att låta andra dela script eller köra script på en annan dator.

Ett bekvämt kortkommando till Home directory är `~`, men eftersom Windows inte har ett egentligt hembibliotek pekar `~` ut Dokument-mappen.

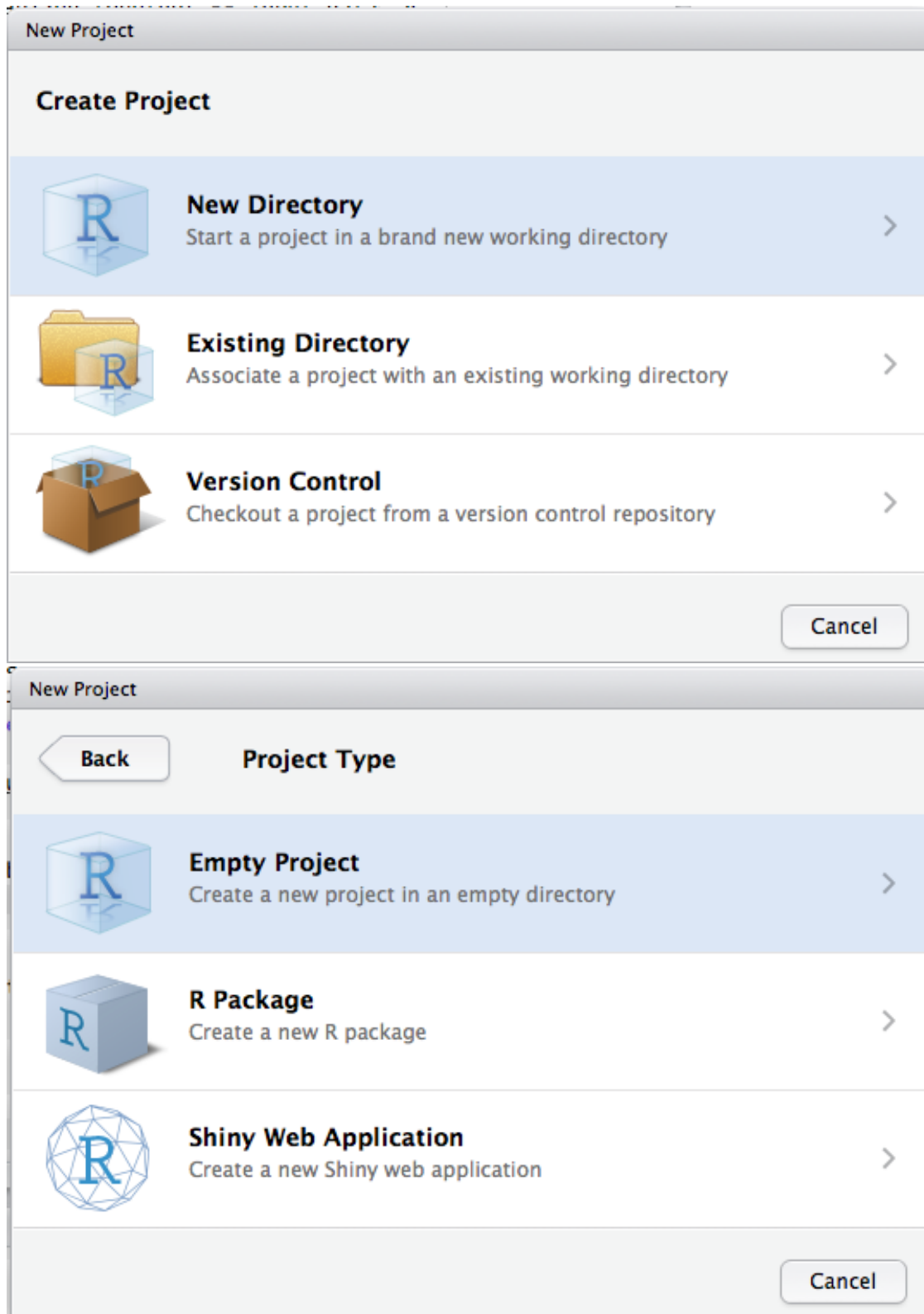
Du bör organisera dina R-projekt i specifika bibliotek, associerade med andra filer för projektet. Man kan definiera sökvägen till arbetsbiblioteket genom :

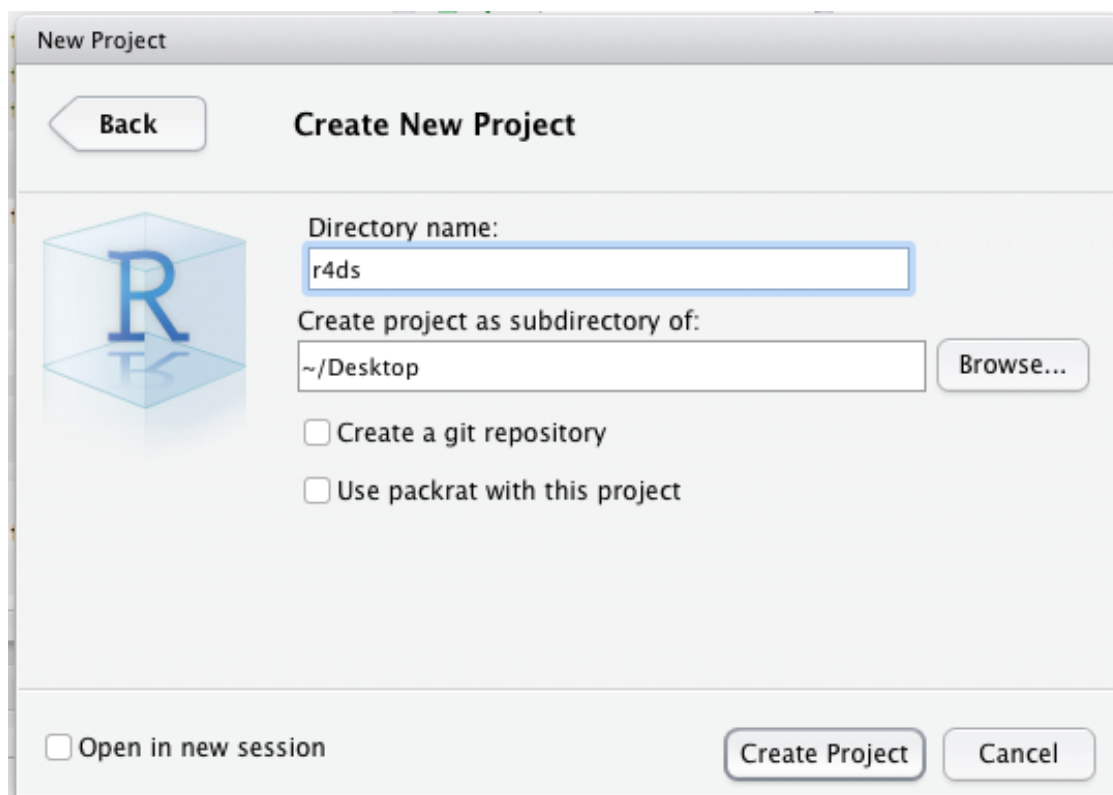
```
setwd("/path/to/my/CoolProject")
```

Men det är en bättre praxis att göra det genom att utnyttja Rstudios inbyggda projekt-hantering.

RStudio projects VIKTIGT!

Spara alla filer som associeras till ett specifikt projekt tillsammans - data, R script, resultat, grafer. I Rstudio görs detta bäst genom att utnyttja programmets projekthantering. Låt oss skapa ett nytt projekt för resten av denna kurs. Klicka File > New Project, sedan:





Du kan kalla mappen för t.ex. Rintro. Tänk igenom var du lägger mappen så att du lätt hittar den igen. När du gjort det skapas ett R-projekt för just denna kurs. Kolla så att "home"-biblioteket/mappen också är din arbetsmapp:

```
getwd()
```

Närhelst du refererar till en fil med en relativ sökväg kommer R att leta i denna mapp. Öppna en ny script-fil och spara ned den som `diamonds.R`. Sedan, skriv in nedanstående kod i scriptet och spara det. Bekymra dig inte om detaljerna för nu.

```
library(tidyverse)
```

```
ggplot(diamonds, aes(carat, price)) +  
  geom_hex()
```

```
ggsave("diamonds.pdf")
```

```
write_csv(diamonds, "diamonds.csv")
```

Stäng Rstudio. Gå till hemmamappen för projektet och notera `.Rproj`-filen. Dubbelklicka på den för att på nytt öppna projektet. Notera att du kommer tillbaka där du lämnade Rstudio.

Eftersom du inte sparade arbetsmiljön (the environment uppe till höger) är nu miljön helt ren.

Kolla in hemma-mappen och att filen `diamonds.pdf` finns med men även att scriptet som du gjorde grafen med (`diamonds.R`). Det här är inte så dumt när man en dag vill göra om grafen eller bara förstå hur det var den gjordes. Om du är nogga med att alltid spara figurer genom att använda kod och aldrig via musen eller clipboard, så kan man alltid återskapa eller modifiera gamla grafer på nytt.

Sammanfattningsvis

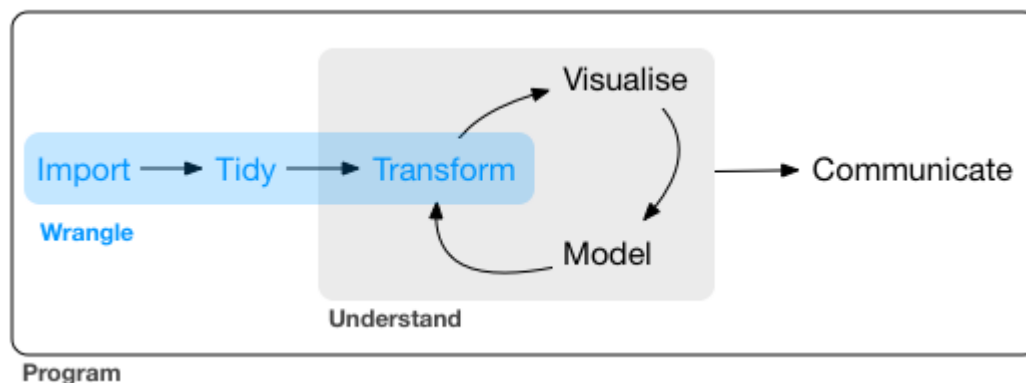
RStudio projects gör det möjligt att:

- Skapa ett Rstudio project för varje databearbetnings projekt.
- Hålla alla projektets datafiler samlade i projektmappen.
- Hålla alla script samlade; redigera dem, exekvera dem helt eller delvis.
- Spara output - grafer och data i projektmappen.

Wrangle - att brottas med data

Introduktion

Nu ska det handla om att få rådata att bli användbara för analyser och visualisering, det som Wickham kallar *data wrangling* och som är en förutsättning för att kunna arbeta med sina data. Man kan urskilja tre delar i *data wrangling*:



Vi ska gå igenom följande:

- Först *tibbles*, den variant av dataram (data frame) som används i denna kurs, om vad som skiljer den från en traditionell data fram och hur de kan skapaas "för hand".
- Sedan *data import*, hur man importerar data från andra källor in till R. Tyngdpunkten ligger på rektangulära text-format men vi ska nämna några andra verktyg för att hantera andra typer av data-format.
- Därefter *tidy data*, som handlar om att städa, rensa rådata, att göra dem "tidy". Mer specifikt om ett konsistent sätt att lagra eller forma data för att underlätta transformering, visualisering och modellering.

Till sist ska vi gå igenom några modernare sätt att hantera olika data-format i R:

- Avsnittet om *Relational data* handlar om verktyg för att hantera relationsdata-mängder.
- I *Strings* introduceras regular expressions, ett kraftfullt verktyg för att manipulera text.
- *Factors* handlar om hur R hanterar kategoriska data.
- *Dates and times* handlar om de viktigaste verktygen för att hantera datum och tid.

Tibbles

I kursen arbetar vi med *tibbles* istället för R:s traditionella `data.frame`. *Tibbles* är data frames men lite modifierade för att underlätta hanteringen av dem. Du kan läsa mer om tibbles om du skriver `vignette("tibble")`.

Börja med att ladda in `tidyverse` i vilket modulen `tibble` är en del:

```
library(tidyverse)
```

Skapa tibbles

Nästan samtliga funktioner som vi använder under kursen skapar *tibbles* eftersom dessa är en central del i *tidyverse*. I många andra moduler används traditionella *data frames* så därför finns en funktion för att konvertera data frames till en tibble, `as_tibble()`. Prova:

```
as_tibble(iris)

## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <dbl>      <dbl>      <dbl>      <dbl> <fct>
## 1      5.1      3.5        1.4        0.2 setosa
## 2      4.9       3         1.4        0.2 setosa
## 3      4.7      3.2        1.3        0.2 setosa
## 4      4.6      3.1        1.5        0.2 setosa
## 5       5       3.6        1.4        0.2 setosa
## 6      5.4      3.9        1.7        0.4 setosa
## 7      4.6      3.4        1.4        0.3 setosa
## 8       5       3.4        1.5        0.2 setosa
## 9      4.4      2.9        1.4        0.2 setosa
## 10     4.9      3.1        1.5        0.1 setosa
## # ... with 140 more rows
```

Du kan skapa en ny tibble från separata vektorer med hjälp av funktionen `tibble()`. `tibble()` återanvänder vektorer av längden 1 och medger att du refererar till variabler du just skapat. Prova:

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)

## # A tibble: 5 x 3
##   x     y     z
```

```
## <int> <dbl> <dbl>
## 1    1    1    2
## 2    2    1    5
## 3    3    1   10
## 4    4    1   17
## 5    5    1   26
```

`tibble()` ändrar aldrig data-typ, t.ex. konverterar text-strängar (*strings*) till kategoriska data (*factors*), den ändrar aldrig variabelnamn och skapar inte radnamn.

Ett annat sätt att skapa tibbles är med `tribble()`, en förkortning av *transposed tibble*. `tribble()` är till för att skriva in data som kod: kolumnrubriker/variabelnamn definieras med formler, dvs de börjar med tildetecken, `~` och data separeras med kommatecken. Detta gör det möjligt att skapa små datamängder på ett sätt som är lätt att läsa. Prova:

```
tribble(
  ~x, ~y, ~z,
  #--/--/----
  "a", 2, 3.6,
  "b", 1, 8.5
)

## # A tibble: 2 x 3
##   x     y     z
##   <chr> <dbl> <dbl>
## 1 a       2   3.6
## 2 b       1   8.5
```

Tibbles vs data frames

Det finns två huvudsakliga skillnader i användningen av en tibble jämfört med en traditionell data frame: utskrift (printing) och urval (subsetting).

Utskrift (Printing)

En utskrift av tibbles visar endast de 10 första raderna i tabellen och alla de kolumner som får plats på skärmen, vilket är en fördel då man arbetar med stora datatabeller. Utöver variabelnamnen visas även variabel-typ (numerisk, factor, etc).

Ibland kan man behöva se ett större utsnitt av tabellen. Det finns ett par sätt att göra detta på. Du kan ange explicit antalet rader (`n`) och kolumner (`width`) som ska visas. `width = Inf` visar samtliga kolumner:

```
nycflights13::flights %>%
  print(n = 10, width = Inf)
```

```
## # A tibble: 336,776 x 19
##   year month  day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>    <dbl>   <int>
## 1 2013     1     1    517           515         2     830
## 2 2013     1     1    533           529         4     850
## 3 2013     1     1    542           540         2     923
## 4 2013     1     1    544           545        -1    1004
## 5 2013     1     1    554           600        -6     812
## 6 2013     1     1    554           558        -4     740
## 7 2013     1     1    555           600        -5     913
## 8 2013     1     1    557           600        -3     709
## 9 2013     1     1    557           600        -3     838
## 10 2013     1     1    558           600        -2     753
##   sched_arr_time arr_delay carrier flight tailnum origin dest air_time
##           <int>   <dbl> <chr>   <int> <chr>  <chr> <chr>   <dbl>
## 1           819     11 UA      1545 N14228 EWR   IAH     227
## 2           830     20 UA      1714 N24211 LGA   IAH     227
## 3           850     33 AA      1141 N619AA JFK   MIA     160
## 4          1022    -18 B6       725 N804JB JFK   BQN     183
## 5           837    -25 DL       461 N668DN LGA   ATL     116
## 6           728     12 UA      1696 N39463 EWR   ORD     150
## 7           854     19 B6       507 N516JB EWR   FLL     158
## 8           723    -14 EV      5708 N829AS LGA   IAD      53
## 9           846     -8 B6       79 N593JB JFK   MCO     140
## 10          745      8 AA       301 N3ALAA LGA   ORD     138
##   distance hour minute time_hour
##   <dbl> <dbl> <dbl> <dtm>
## 1   1400     5    15 2013-01-01 05:00:00
## 2   1416     5    29 2013-01-01 05:00:00
## 3   1089     5    40 2013-01-01 05:00:00
## 4   1576     5    45 2013-01-01 05:00:00
## 5    762     6     0 2013-01-01 06:00:00
## 6    719     5    58 2013-01-01 05:00:00
## 7   1065     6     0 2013-01-01 06:00:00
## 8    229     6     0 2013-01-01 06:00:00
## 9    944     6     0 2013-01-01 06:00:00
## 10    733     6     0 2013-01-01 06:00:00
## # ... with 3.368e+05 more rows
```

Du kan även ange hur en tibble ska skrivas ut genom att ställa in options. Du hittar en komplett lista på options med hjälp av `?tibble`.

En sista möjlighet är att använda Rstudios inbyggda viewer för en skrollningsbar vy på hela datasetet, vilket är användbart t.ex. efter att ha gjort en längre kedja av manipulationer av data:

```
nycflights13::flights %>%  
  View()
```

Urval

Så här långt har alla verktyg/funktioner du använt arbetat med kompletta dataramar. Om du vill arbeta med en enskild variabel är det bra att känna till några ytterligare funktioner, `$` och `[[`. `[[` arbetar med både namn och position, `$` använder endast namn:

```
df <- tibble(  
  x = runif(5),  
  y = rnorm(5)  
)  
  
# Extrahera med namn  
df$x  
## [1] 0.4691702 0.7364010 0.4477658 0.9303001 0.5710481  
  
df[["x"]]  
## [1] 0.4691702 0.7364010 0.4477658 0.9303001 0.5710481  
  
# Extrahera med hjälp av position  
df[[1]]  
## [1] 0.4691702 0.7364010 0.4477658 0.9303001 0.5710481
```

Du kan även använda dessa i en "pipe" och behöver då en "platshållare", `.`:

```
df %>% .$x  
## [1] 0.4691702 0.7364010 0.4477658 0.9303001 0.5710481  
  
df %>% .[[ "x"]]  
## [1] 0.4691702 0.7364010 0.4477658 0.9303001 0.5710481
```

Använda äldre kod

Vissa äldre funktioner fungerar inte med tibbles. Om du stöter på ett sådant problem kan du transformera en tibble tillbaka till en traditionell data frame med hjälp av funktionen `as.data.frame()`:

```
class(as.data.frame(df))
```

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

Det största anledningen till att vissa äldre funktioner inte fungerar med tibbles är att funktionen `[` är involverad. I denna kurs använder vi sällan den funktionen eftersom `dplyr::filter()` och `dplyr::select()` löser samma problem men med tydligare kod (vi återkommer strax till sådana urval (*subsetting*)). Base R funktionen `[` returnerar ibland en data frame, ibland en vektor. Tibbles returnerar alltid en tibble.

Importera data

I det här avsnittet kollar vi på hur man importerar (rektangulära) text-filer till R. Detta är ett omfattande område i R och här blir det ett skrap på ytan. Men många av principerna är gemensamma för andra data-format. Avslutningsvis några tips på andra moduler/packages som är bra på att hantera andra format också.

Vi laddar in

```
library(tidyverse)
```

Och använder modulen `readr` som är en del av `tidyverse`.

De flesta av `readr`:s funktioner handlar om att omvandla textfiler till data frames:

- `read_csv()` läser in komma-avgränsade filer, `read_csv2()` läser in semikolon-separerade filer, `read_tsv()` läser in tabb-avgränsade filer och `read_delim()` läser in filer med valfri avgränsare.
- `read_fwf()` läser in filer med fast kolumnformat. Du kan specificera fälten antingen med dess längd `fwf_widths()` eller position `fwf_positions()`. `read_table()` läser in fast kolumn-format där kolumnerna separeras med blanksteg.

Dessa funktioner har en liknande syntax: när man lärt sig en funktion kan man i princip använda dessa argument i de övriga importfunktionerna. I fortsättningen fokuserar vi `read_csv()`. csv-filer är vanligt förekommande och det är lätt att applicera hanteringen av andra format efter att ha lärt dig `read_csv()`.

Det första argumentet är viktigast - sökvägen och namnet på filen. Prova:

```
# heights <- read_csv("data/heights.csv")
```

När du kör `read_csv()` skrivs en kolumnspecifikation ut som anger namn och typ av data för varje kolumn, vi ska återkomma till detta. Du kan även skapa en csv-fil "in line" vilket kan vara bra för att experimentera med `readr` eller för att dela reproducerbara exempel med andra. T.ex.:

```
read_csv("a,b,c
1,2,3
4,5,6")

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

I båda fallen använder `read_csv()` den första raden som kolumnrubriker. Du kan dock påverka det på två sätt:

1. Ibland finns metadata på de första raderna vilka man inte vill importera. Då kan man använda `skip = n` för att ta bort de n första raderna; eller använda `comment = "#"` för att ta bort alla rader som börjar med (t.ex.) `#`. Prova:

```
read_csv("The first line of metadata
The second line of metadata
x,y,z
1,2,3", skip = 2)
```

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

```
read_csv("# A comment I want to skip
x,y,z
1,2,3", comment = "#")
```

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

2. Det är inte alltid som data har kolumnnamn. Du kan använda `col_names = FALSE` för att inte `read_csv()` ska uppfatta den första raden som kolumnnamn utan istället namnge dem sekventiellt från X1 till Xn:

```
read_csv("1,2,3\n4,5,6", col_names = FALSE)
```

```
## # A tibble: 2 x 3
##   X1    X2    X3
##   <int> <int> <int>
## 1     1     2     3
## 2     4     5     6
```

("n" är ett smart sätt att lägga till en ytterligare rad. Vi återkommer till textsträngar...). Alternativt kan man ange kolumnnamn som ett argument med hjälp av `col_names`. Du tilldelar då en text-vektor till `col_names` som kommer att användas som namn. Prova:

```
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
```

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

Ett ytterligare argument som ofta behöver hanteras är missing values (*na*): här specificeras det värde som representerar missing values i data. Prova:

```
read_csv("a,b,c\n1,2,.", na = ".")
```

```
## # A tibble: 1 x 3
##   a     b c
##   <int> <int> <chr>
## 1     1     2 <NA>
```

Med detta kan du hantera de flesta csv-filer eller enkelt anpassa argument för att hantera andra filformat. Det kan dock vara ide att lära dig mer om hur `readr` hanterar varje kolumn för att få dem till R-vektorer, om du vill kunna hantera mer komplexa filformat.

Jämförelse med base R

Om du tidigare använt R kanske du funderar på varför vi inte använder den traditionella funktionen `read.csv()`. Det finns flera skäl: `read_csv()`

- är typiskt betydligt snabbare (10x) än `read.csv()`.
- skapar tibbles, konverterar inte text-data till kategoriska data, använder inte radnamn,
- **OBS!!! munge the column names.** Vilket är vanliga källor till frustration med den traditionella funktionen.
- är mer reproducerbara - base R-funktioner ärver några egenskaper från ditt operativsystem vilket kan leda till att det som fungerar bra på din dator kanske inte gör det på en annan.

Övningar

1. Vilken funktion skulle du använda för att läsa in en fil med separatoren "|"?
2. Bortsett från `file`, `skip`, och `comment`, vilka andra argument har `read_csv()` och `read_tsv()` gemensamt?
3. Vilka är de viktigaste argumenten i funktionen `read_fwf()`?
4. Identifiera felen i de nedanstående kod-raderna och kolla vad som sker då du kör koderna.

```
read_csv("a,b\n1,2,3\n4,5,6")
read_csv("a,b,c\n1,2\n1,2,3,4")
read_csv("a,b\n"1")
read_csv("a,b\n1,2\na,b")
read_csv("a;b\n1;3")
```

Omvandla vektorer

Innan vi går vidare med att kika på hur `readr` läser in filer från disken ska vi ta lite tid med `parse_*()`-funktionerna. Dessa funktioner tar en text-vektor (*character*) och omvandlar den till en mer specialiserad vektor, t.ex. logisk, kategorisk eller datum. Prova:


```
str(parse_logical(c("TRUE", "FALSE", "NA")))
## logi [1:3] TRUE FALSE NA

str(parse_integer(c("1", "2", "3")))
## int [1:3] 1 2 3

str(parse_date(c("2010-01-01", "1979-10-14")))
## Date[1:2], format: "2010-01-01" "1979-10-14"
```

Dessa funktioner är användbara i sin egen rätt, men är också viktiga "bygg-block" för `readr`. Vi ska gå igenom var och en av dessa *parsers* och därefter se hur de fungerar ihop för att sätta samman en komplett datafil.

Det första argumentet till `parse_*`-funktionerna är en textsträng (character) som ska omvandlas och *na*-argumentet specificerar vilka strängar som ska uppfattas som missing values (NA):

```
parse_integer(c("1", "231", ".", "456"), na = ".")
## [1] 1 231 NA 456
```

Om omvandlingen inte fungerar kommer en *Warning*:

```
x <- parse_integer(c("123", "345", "abc", "123.45"))
## Warning in rbind(names(probs), probs_f): number of columns of result is not
## a multiple of vector length (arg 1)
## Warning: 2 parsing failures.
## row # A tibble: 2 x 4 col   row col expected      actual expected <int> <int> <chr>      <chr>
> actual 1     3  NA an integer      abc row 2     4  NA no trailing characters .45
```

Och de fallerande strängarna kommer att bli missing values i output:

```
x
## [1] 123 345 NA NA
## attr("problems")
## # A tibble: 2 x 4
##   row col expected      actual
##   <int> <int> <chr>      <chr>
## 1     3  NA an integer      abc
## 2     4  NA no trailing characters .45
```

Om det blir många fel kan du använda `problems()` för att få en komplett lista på felen. Denna funktion skapar en tibble som du sedan kan bearbeta med `dplyr`. Prova:

```
problems(x)
```

```
## # A tibble: 2 x 4
##   row col expected      actual
##   <int> <int> <chr>      <chr>
## 1   3   NA an integer      abc
## 2   4   NA no trailing characters .45
```

Att använda parsing-funktionerna är för det mesta en fråga om att vara medveten om vad som finns tillgängligt och hur de hanterar olika typer av input. Det finns åtta särskilt viktiga parser-funktioner:

- `parse_logical()` och `parse_integer()` bearbetar logiska respektive diskreta värden. De fungerar nästan alltid som förväntat.
- `parse_double()` är en strikt numerisk parser och `parse_number()` är en flexibel numerisk parser. Dessa är lite mer komplicerade vilket sammanhänger med att numeriska värden skrivs olika i olika delar av världen.
- `parse_character()` är ganska straightforward men man får se upp med text-kodningen (*character encodings*).
- `parse_factor()` skapar kategoriska variabler.
- `parse_datetime()`, `parse_date()`, and `parse_time()` medger omvandling till olika typer av datum-format.

Numeriska variabler

Det är fr.a. tre problem som kan vara besvärliga att hantera beträffande numeriska variabler:

1. Numeriska värden anges på olika sätt i olika delar av världen, t.ex. avgränsar man decimaler med kommatecken alternativt punkt.
2. Numeriska värden omges inte sällan av andra tecken vilka preciserar vad det är för typ av värden, t.ex. *\$5000* eller *10%*.
3. Numeriska värden innehåller inte sällan "grupperings-tecken" för att göra dem lättare att läsa, t.ex. *1,000,000* och sådana tecken varierar i världen.

För att hantera det första problemet innehåller `readr` ett objekt som kallas *locale* vilket specificerar parser-alternativ. Det kanske viktigaste beträffande numeriska värden är decimaltecknet. Du kan ändra default (".") genom att skapa ett nytt locale och definiera ett nytt decimaltecken. Prova:

```
parse_double("1.23")
## [1] 1.23

parse_double("1,23", locale = locale(decimal_mark = ","))
## [1] 1.23
```

`parse_number()` hanterar det andra problemet: det ignorerar helt enkelt icke-numeriska tecken vilket är särskilt användbart för valuta-tecken och procent-tecken men kan även användas för att extrahera värden ur text. Prova:

```
parse_number("$100")
```

```
## [1] 100

parse_number("20%")

## [1] 20

parse_number("Det kostar SEK123.45")

## [1] 123.45
```

Det tredje problemet hanteras genom att kombinera `parse_number()` och `locale` eftersom `parse_number()` ignorerar grupperings-tecken". Prova:

```
parse_number("$123,456,789")

## [1] 123456789

parse_number("123.456.789", locale = locale(grouping_mark = "."))

## [1] 123456789

parse_number("123'456'789", locale = locale(grouping_mark = "'"))

## [1] 123456789
```

Textsträngar

Man kan tycka att `parse_character()` skulle vara okomplicerat eftersom den funktionen bara ska returnera en input. Men varför göra livet enkelt? Det finns många sätt att representera en viss text på. För att förstå vad som händer behöver vi kika på hur datorer behandlar text. I R kan vi se hur en textsträng är representerad genom att använda `charToRaw()`:

```
charToRaw("Henriksson")

## [1] 48 65 6e 72 69 6b 73 73 6f 6e
```

Varje hexadecimalt nummer representerar en byte information: 48 är H, 65 är e, osv. Mappningen från hexadecimalt format till text kallas för *encoding*, i detta fall kallas kodningen för ASCII. ASCII (akronym för *American Standard Code for Information Interchange*) fungerar utmärkt för att representera engelska/amerikanska texter. Men det blir mer komplicerat för andra språk. Det finns en mängd olika *encodings* (t.ex. Latin1 (aka ISO-8859-1 för västeuropeiska språk); Latin2 (aka ISO-8859-2 för östeuropeiska språk) men det idag förmodligen dominerande kodningen är UTF-8. UTF-8 klarar att koda/representera i stort sett alla tecken som finns, t.o.m. emoji.

`readr` använder UTF-8 överallt: det antar att dina data är UTF-8 kodade när du läser in dem, och alltid använder UTF-8 då du skriver kod. Det är en bra default men kan inte hantera data skapade i andra system som inte förstår UTF-8. I så fall kommer din text att se knepig ut i utskrifter. Prova t.ex. :

```
x1 <- "El Ni\u00f1o was particularly bad this year"
x2 <- "\u0082\u00b1\u0082\u00f1\u0082\u00c9\u0082\u00bf\u0082\u00cd"
```

```
x1
## [1] "El Niño was particularly bad this year"

x2
## [1] ",±,ñ,É,¿,Í"
```

För att hantera detta behöver du specificera kodningen i `parse_character()`:

```
parse_character(x1, locale = locale(encoding = "Latin1"))
## [1] "El Niño was particularly bad this year"

parse_character(x2, locale = locale(encoding = "Shift-JIS"))
## [1] "<U+3053><U+3093><U+306B><U+3061><U+306F>"
```

Men hur hittar man rätt *encoding*? Med lite tur finns det dokumenterat någonstans i dina datafiler/metadata. Men det finns också en `guess_encoding()`-funktion. Den är inte ofelbar och fungerar bättre ju mer text som finns till hands, men det är en bra start för att hitta rätt encoding. Annars fungerar alltid trial and error. Prova:

```
guess_encoding(charToRaw(x1))

## # A tibble: 2 x 2
##   encoding confidence
##   <chr>         <dbl>
## 1 ISO-8859-1    0.46
## 2 ISO-8859-9    0.23

guess_encoding(charToRaw(x2))

## # A tibble: 1 x 2
##   encoding confidence
##   <chr>         <dbl>
## 1 KOI8-R        0.42
```

Det första argumentet kan antingen vara en sökväg till en fil eller en hexadecimal vektor, vilket är smidigt om data redan finns i R.

Om du vill veta mer om encodings rekommenderas <http://kunststube.net/encoding/>.

Kategoriska data

I R används *factors* för att representera kategoriska variabler som har ett givet antal möjliga värden.

Datum och tid

Det finns tre parsers för att hantera datum och tidsformat: datum `parse_date()` (bygger på antalet dagar sedan 1970-01-01); datum-tid `parse_datetime()` (antalet sekunder sedan midnatt 1970-01-01); tid `parse_time()` (antalet sekunder sedan midnatt). Pröva:

```
parse_datetime("2010-10-01T2010")
```

```
## [1] "2010-10-01 20:10:00 UTC"
```

```
parse_datetime("20101010")
```

```
## [1] "2010-10-10 UTC"
```

```
parse_date("2010-10-01")
```

```
## [1] "2010-10-01"
```

```
library(hms)
```

```
parse_time("01:10 am")
```

```
## 01:10:00
```

```
parse_time("20:10:01")
```

```
## 20:10:01
```

Base R har inte särskilt många klasser för tid-data så det kan vara bra att ladda in modulen `hms` (`library(hms)`).

Om inte dessa default-värden fungerar för dina data så kan man bygga egna datum-tid-format med hjälp av följande komponenter:

- År
- `%Y` (4 siffror).
- `%y` (2 siffror); 00-69 -> 2000-2069, 70-99 -> 1970-1999.
- Månad
- `%m` (2 siffror).
- `%b` (förkortat, typ "Jan").
- `%B` (fullständig namn, "January").
- Day
- `%d` (2 siffror).
- `%e` (valfri inledning).
- Time
- `%H` 0-23 timmar.
- `%p` AM/PM indikator.
- `%M` minuter.

- %S heltal sekunder.
- %OS numeriska sekunder.
- %Z Tids zon (t.ex. America/Chicago).
- %z (som offset från UTC, e.g. +0800).

Det bästa sättet att fundera ut det korrekta formatet, om du är osäker, är att skapa ett par alternativa exempel i en character vector och testa med någon av tids-parsers. Prova:

```
parse_date("01/02/15", "%m/%d/%y")
## [1] "2015-01-02"

parse_date("01/02/15", "%d/%m/%y")
## [1] "2015-02-01"

parse_date("01/02/15", "%y/%m/%d")
## [1] "2001-02-15"
```

Om du använder %b eller %B tillsammans med icke-engelska månadsnamn behöver du lägga till argumentet lang till locale(). Se listan över inbyggda språk i date_names_langs(), eller om det inte finns med, skapa ett eget med hjälp av date_names().

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
## [1] "2015-01-01"
```

Övningar

1. Vilken är skillnaden mellan read_csv() och read_csv2()?
2. Generera det korrekta formatet till följande datum och tider:

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```

Hur readr “plockar isär” (parse) en fil

Vi ska kika på hur readr plockar isär eller “förstår” en fil, dvs hur readr 1) automatiskt gissar vilken typ av variabler som finns i kolumnerna, 2) åsidosätter default-specifikationen.

Modulen `readr` läser in upp till de första 1000 raderna och använder ett antal “regler” för att gissa klassen på varje kolumn. Man kan emulera denna process med hjälp av `guess_parser()`, som returnerar `readr`:s bästa gissning, och `parse_guess()`, som använder den gissningen för att formatera kolumnen:

```
guess_parser("2010-10-01")
## [1] "date"

guess_parser("15:01")
## [1] "time"

guess_parser(c("TRUE", "FALSE"))
## [1] "logical"

guess_parser(c("1", "5", "9"))
## [1] "integer"

guess_parser(c("12,352,561"))
## [1] "number"

str(parse_guess("2010-10-10"))
## Date[1:1], format: "2010-10-10"
```

Problem

Det kan uppstå problem särskilt med stora filer. De första 1000 raderna kan t.ex. vara selekterade vilket kan medföra att `readr` vilseleds i sin gissning. En kolumn kan innehålla många missing values (NA) vilket `readr` kommer att tolka som en character-kolumn. I Hadley's bok finns ett avsnitt som beskriver en strategi för att lösa sådana problem, se <http://r4ds.had.co.nz/data-import.html#problems>.

Skriva till en fil

`Readr` har två användbara funktioner för att skriva data till filer: `write_csv()` och `write_tsv()`. Båda dessa funktioner ökar chansen att output-filen blir korrekt inläst genom att funktionerna:

- Alltid kodar strings i UTF-8.
- Sparar datum och tids-format enligt ISO8601 format (https://en.wikipedia.org/wiki/ISO_8601).

Om du vill spara till excel-format använd `write_excel_csv()` — denna funktion skriver en speciell bokstav (“byte order mark”) till början av filen för att deklarera att det är UTF-8 som används.

Andra datatyper

För att importera andra datatyper till R finns ett antal andra moduler i `tidyverse`:

- `haven` läser SPSS, Stata, and SAS filer.
- `readxl` läser excel files (.xls and .xlsx).
- `DBI`, tillsammans med en databas-specifik backend, ex `RMySQL`, `RSQLite` eller `RPostgreSQL`) möjliggör att köra SQL queries mot en databas och returnera en data-frame.
- `jsonlite` för json,
- `xml2` för XML.

Fler exempel finns på <https://jennybc.github.io/purrr-tutorial/>. För andra filtyper se R data import/export manual, modulen `rio`.

Städa data (tidy data)

Introduktion

Detta avsnitt handlar om att tillämpa ett konsistent sätt att organisera dina data i R som Wickham kallar *tidy data*. Det kräver lite handpåläggning innan du kan börja analysera data men är väl värt mödan eftersom du kommer att behöva lägga mindre tid på att organisera om dina data för specifika analyser senare.

Avsnittet fokuserar på modulen `tidyr`, även den en del av `tidyverse`. Wickham har skrivit en artikel i *Journal of Statistical Software*, om teorin bakom *tidy data*: <http://www.jstatsoft.org/v59/i10/paper>.

```
library(tidyverse)
```

Samma underliggande data kan representeras på olika sätt. Nedanstående tibbles visar samma data representerade på fyra olika sätt. Data består av fyra variabler, `country`, `year`, `population`, and `cases`, men varje tabell organiserar värdena på fyra olika sätt: TB-datasetet:

<https://extranet.who.int/tme/generateCSV.asp?ds=estimates>

table1

```
## # A tibble: 6 x 4
##   country   year cases population
##   <chr>    <int> <int>    <int>
## 1 Afghanistan 1999   745  19987071
## 2 Afghanistan 2000  2666  20595360
## 3 Brazil     1999 37737  172006362
## 4 Brazil     2000 80488  174504898
## 5 China      1999 212258 1272915272
## 6 China      2000 213766 1280428583
```

table2

```
## # A tibble: 12 x 4
##   country   year type      count
##   <chr>    <int> <chr>    <int>
## 1 Afghanistan 1999 cases      745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases      2666
## 4 Afghanistan 2000 population 20595360
## 5 Brazil     1999 cases      37737
## 6 Brazil     1999 population 172006362
## 7 Brazil     2000 cases      80488
## 8 Brazil     2000 population 174504898
```

```
## 9 China    1999 cases    212258
## 10 China   1999 population 1272915272
## 11 China   2000 cases    213766
## 12 China   2000 population 1280428583
```

```
table3
```

```
## # A tibble: 6 x 3
##   country    year rate
## * <chr>    <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
```

```
table4a # cases
```

```
## # A tibble: 3 x 3
##   country    `1999` `2000`
## * <chr>    <int> <int>
## 1 Afghanistan 745 2666
## 2 Brazil      37737 80488
## 3 China       212258 213766
```

```
table4b # population
```

```
## # A tibble: 3 x 3
##   country    `1999` `2000`
## * <chr>    <int> <int>
## 1 Afghanistan 19987071 20595360
## 2 Brazil      172006362 174504898
## 3 China       1272915272 1280428583
```

Även om data är desamma är tabellerna inte lika användbara. Om data vore organiserade utifrån principerna i “the tidy dataset”, kommer datasetet att vara betydligt enklare att arbeta med. Det finns tre sammanhängande regler vilka gör ett dataset till ett “tidy” dataset:

1. Varje variabel måste ha sin egen kolumn
2. Varje observation måste ha sin egen rad
3. Varje värde måste ha sin egen cell

Detta illustreras i nedanstående figur:

OBS FIGYR SAKNAS

Dessa tre regler hänger samman genom att det är omöjligt att tillgodose endast två av de tre. Varför bör man göra detta då? Det finns två fördelar:

1. Det finns en generell poäng med att välja ett konsistent sätt att lagra data. Med ett dataset som är *tidy* är det lättare att lära sig verktygen för att bearbeta dessa data genom att verktygen har en underliggande gemensam struktur. `dplyr`, `ggplot2` och övriga packages i `tidyverse` är utformade för att arbeta med *tidy data*.
2. Det finns en speciell fördel med att placera variabler i kolumner genom att det utnyttjar R:s vektor-baserade arbetssätt maximalt. Som vi såg i `mutate()` och `summarise()`-funktionerna arbetar dessa liksom de flesta funktioner med vektorer. Det gör *tidy data* till ett naturligt sätt att organisera data i R.

Här är några exempel på hur du kan arbeta med TB-data från tabell 1 ovan:

Beräkna antal per 10,000

```
table1 %>%
  mutate(rate = cases / population * 10000)

## # A tibble: 6 x 5
##   country   year cases population rate
##   <chr>    <int> <int>    <int> <dbl>
## 1 Afghanistan 1999   745  19987071 0.373
## 2 Afghanistan 2000  2666  20595360 1.29
## 3 Brazil      1999  37737  172006362 2.19
## 4 Brazil      2000  80488  174504898 4.61
## 5 China       1999 212258 1272915272 1.67
## 6 China       2000 213766 1280428583 1.67
```

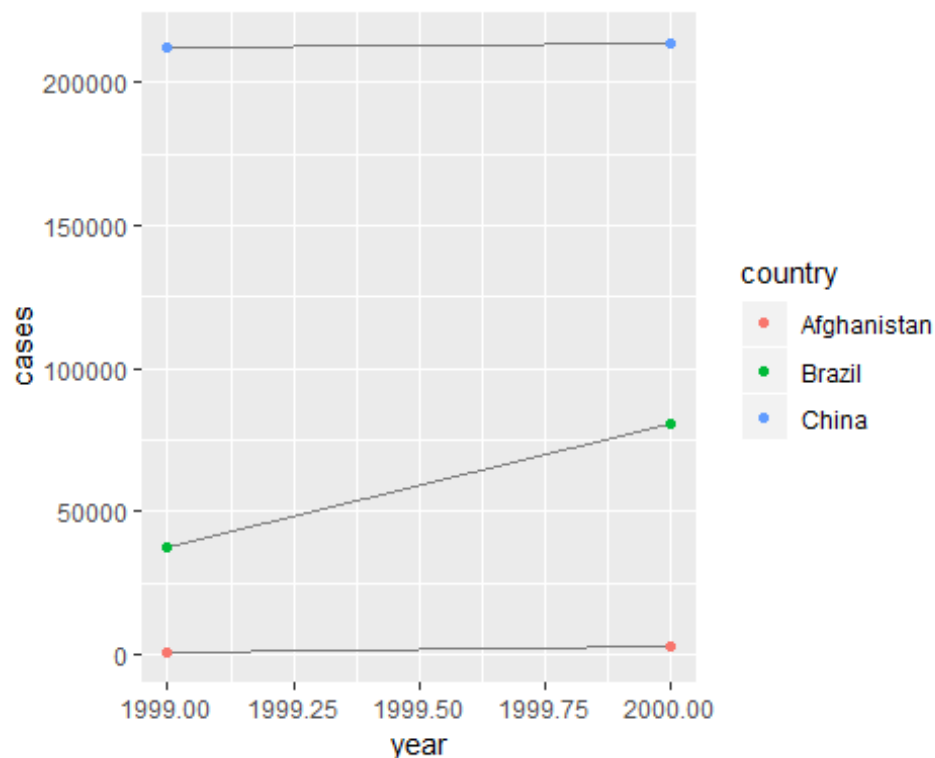
Beräkna antal per år

```
table1 %>%
  count(year, wt = cases)

## # A tibble: 2 x 2
##   year     n
##   <int> <int>
## 1 1999 250740
## 2 2000 296920
```

Visualisera förändring över tid

```
library(ggplot2)
ggplot(table1, aes(year, cases)) +
  geom_line(aes(group = country), colour = "grey50") +
  geom_point(aes(colour = country))
```



Spreading and gathering

Om data inte är *tidy* behöver man rensa datasetet. Vi ska gå igenom ett antal funktioner för detta. Det första steget är att identifiera vad som är variabler och vad som är observationer, vilket ibland är uppenbart, ibland svårare. Det andra steget är att lösa ettdera av två vanliga problem:

1. En variabel kan vara spridd över flera kolumner
2. En observation kan vara spridd över flera kolumner

För att fixa detta finns två funktioner tillgängliga i *tidyr*: `gather()` and `spread()`.

Gathering

Ett vanligt problem är när ett kolumn-namn inte är variabelnamn utan ett värde på en variabel. Se tabell 4a: kolumnerna "199" och "2000" representerar värden av variabeln "year" vilket innebär att varje rad representerar två observationer, inte en:

```
table4a
## # A tibble: 3 x 3
##   country `1999` `2000`
## * <chr>   <int> <int>
## 1 Afghanistan 745 2666
```

```
## 2 Brazil    37737 80488
## 3 China     212258 213766
```

För att göra detta till ett *tidy* dataset behöver vi samla ihop (*gather*) dessa två kolumner till en. Vi behöver tre parametrar för detta:

1. De kolumner som representerar värden, t.ex. kolumnerna 1999 och 2000.
2. Namnet på variabeln vars värden bildar kolumnerna, dvs en "nyckel". I detta fall kallar vi benämner vi nyckeln "year".
3. Namnet på variabeln som ska representera värden på nyckeln, i detta fall "cases". Dessa tre parametrar är de argument som behövs för *gather()*:

```
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")

## # A tibble: 6 x 3
##   country year cases
##   <chr>   <chr> <int>
## 1 Afghanistan 1999    745
## 2 Brazil      1999   37737
## 3 China       1999  212258
## 4 Afghanistan 2000   2666
## 5 Brazil      2000   80488
## 6 China       2000  213766
```

Kolumnerna specificeras som då man använder *select()*. Här är det bara två kolumner så då spec:ar vi dem individuellt. Observera att "1999" och "2000" är "non-syntactic names", eftersom de inte börjar med en bokstav så vi behöver omge dem med "backticks", se *?select*.

De ursprungliga kolumnerna 1999 och 2000 har ersatts med "year" och "cases". Relationerna mellan de ursprungliga variablerna är dock oförändrade.

På samma sätt kan man göra med tabell 4b:

```
table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")

## # A tibble: 6 x 3
##   country year population
##   <chr>   <chr>   <int>
## 1 Afghanistan 1999 19987071
## 2 Brazil      1999 172006362
## 3 China       1999 1272915272
## 4 Afghanistan 2000 20595360
## 5 Brazil      2000 174504898
## 6 China       2000 1280428583
```

För att kombinera de rensade tabellerna till en gemensam behöver vi använda `left_join()`:

```
tidy4a <- table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
tidy4b <- table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
left_join(tidy4a, tidy4b)

## Joining, by = c("country", "year")

## # A tibble: 6 x 4
##   country   year cases population
##   <chr>    <chr> <int>    <int>
## 1 Afghanistan 1999    745  19987071
## 2 Brazil      1999  37737 172006362
## 3 China       1999 212258 1272915272
## 4 Afghanistan 2000   2666  20595360
## 5 Brazil      2000  80488 174504898
## 6 China       2000 213766 1280428583
```

Spreading

Spreading är motsatsen till *gathering*. Man använder det då en observation är spridd över flera rader. T.ex. tabell 2: en observation är ett land i ett visst år, men varje observation är fördelad på två rader:

```
table2

## # A tibble: 12 x 4
##   country   year type      count
##   <chr>    <int> <chr>    <int>
## 1 Afghanistan 1999 cases      745
## 2 Afghanistan 1999 population 19987071
## 3 Afghanistan 2000 cases      2666
## 4 Afghanistan 2000 population 20595360
## 5 Brazil      1999 cases      37737
## 6 Brazil      1999 population 172006362
## 7 Brazil      2000 cases      80488
## 8 Brazil      2000 population 174504898
## 9 China       1999 cases      212258
## 10 China      1999 population 1272915272
## 11 China      2000 cases      213766
## 12 China      2000 population 1280428583
```

För att rensa dessa data behövs endast två parametrar:

1. Kolumnen som innehåller variabelnamnen, i detta fall *type*

2. Kolumnen som innehåller motsvarande värden, i detta fall *count*

Vi använder dessa parametrar som argument i `spread()`:

```
table2 %>%
  spread(key = type, value = count)

## # A tibble: 6 x 4
##   country   year cases population
##   <chr>    <int> <int>    <int>
## 1 Afghanistan 1999   745  19987071
## 2 Afghanistan 2000  2666 20595360
## 3 Brazil      1999 37737 172006362
## 4 Brazil      2000 80488 174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

Övningar

1. Varför fungerar inte nedanstående kod?

```
table4a %>%
  gather(1999, 2000, key = "year", value = "cases")
```

2. Varför fungerar inte `spread` på nedanstående tibble? Kan man t.ex. addera en ny kolumn för att lösa problemet?

```
people <- tribble(
  ~name,      ~key, ~value,
  #-----/-----/-----
  "Phillip Woods", "age", 45,
  "Phillip Woods", "height", 186,
  "Phillip Woods", "age", 50,
  "Jessica Cordero", "age", 37,
  "Jessica Cordero", "height", 156
)
```

Separera och förena

I tabell 3 finns ett annat problem, nämligen att en kolumn (*rate*) innehåller två variabler (*cases* och *population*). För att lösa detta används `separate()`. Komplementet till `separate()` är `unite()`, vilket används då ett värde är spritt över flera kolumner.

Separate

`separate()` drar isär en kolumn till fler kolumner med hjälp av ett icke-alfanumeriskt tecken som förekommer i kolumnen (= default) eller ett annat tecken som spec:as med argumentet `sep =`. Se tabell 3:

```
table3

## # A tibble: 6 x 3
##   country    year rate
## * <chr>    <int> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
```

Kolumnen `rate` innehåller både `cases` och `population`. Vi behöver dela upp den på två variabler. Argumenten till `separate()` är namnet på kolumnen som ska delas upp, och namnen på kolumnerna som är resultatet av uppdelningen. Se exempel och figur nedan:

```
table3 %>%
  separate(rate, into = c("cases", "population"))

## # A tibble: 6 x 4
##   country    year cases population
##   <chr>    <int> <chr>  <chr>
## 1 Afghanistan 1999 745   19987071
## 2 Afghanistan 2000 2666   20595360
## 3 Brazil      1999 37737  172006362
## 4 Brazil      2000 80488  174504898
## 5 China       1999 212258 1272915272
## 6 China       2000 213766 1280428583
```

Som default kommer `separate()` att göra delningen där det finns ett icke-alfanumeriskt tecken, t.ex. `"/"` i exemplet ovan. Man kan dock specificera valfritt tecken i ett argument som heter `sep`, se nedan:

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")

## # A tibble: 6 x 4
##   country    year cases population
##   <chr>    <int> <chr>  <chr>
## 1 Afghanistan 1999 745   19987071
## 2 Afghanistan 2000 2666   20595360
## 3 Brazil      1999 37737  172006362
```



```
## 4 Brazil    2000 80488 174504898
## 5 China     1999 212258 1272915272
## 6 China     2000 213766 1280428583
```

Notera att de nya kolumnerna (liksom den ursprungliga) är chr vilket inte är särskilt användbart. Vi kan emellertid använda argumentet `convert = TRUE` för att `separate()` ska konvertera de nya kolumnerna till numeriska:

```
table3 %>%
  separate(rate, into = c("cases", "population"), convert = TRUE)

## # A tibble: 6 x 4
##   country   year cases population
##   <chr>    <int> <int>    <int>
## 1 Afghanistan 1999   745  19987071
## 2 Afghanistan 2000  2666 20595360
## 3 Brazil     1999 37737 172006362
## 4 Brazil     2000 80488 174504898
## 5 China      1999 212258 1272915272
## 6 China      2000 213766 1280428583
```

Vi kan även ange en numerisk vektor till `sep`. `separate()` kommer då att tolka siffrorna som positioner där delningen ska ske. Positiva värden börjar vid 1 till vänster av textsträngen, negativa värden börjar vid -1 längst till höger:

```
table3 %>%
  separate(year, into = c("century", "year"), sep = 2)

## # A tibble: 6 x 4
##   country century year rate
##   <chr>    <chr> <chr> <chr>
## 1 Afghanistan 19   99 745/19987071
## 2 Afghanistan 20   00 2666/20595360
## 3 Brazil     19   99 37737/172006362
## 4 Brazil     20   00 80488/174504898
## 5 China      19   99 212258/1272915272
## 6 China      20   00 213766/1280428583
```

Unite

`unite()` är komplementet till `separate()`: det kombinerar flera kolumner till en.

Vi kan använda `unite()` för att återförena `century` och `year` som skapades i förra exemplet. `unite()` använder en *data frame*, namnet på den nya variabeln och ett set av kolumner som ska kombineras:

```
table5 %>%
  unite(new, century, year)

## # A tibble: 6 x 3
##   country    new rate
##   <chr>    <chr> <chr>
## 1 Afghanistan 19_99 745/19987071
## 2 Afghanistan 20_00 2666/20595360
## 3 Brazil      19_99 37737/172006362
## 4 Brazil      20_00 80488/174504898
## 5 China       19_99 212258/1272915272
## 6 China       20_00 213766/1280428583
```

I det här fallet bör vi också använda `sep`. Som default använder `separate` underscore (`_`) mellan värden från de olika kolumnerna. Det vill vi inte ha här så vi använder separatoren `""`:

```
table5 %>%
  unite(new, century, year, sep = "")

## # A tibble: 6 x 3
##   country    new rate
##   <chr>    <chr> <chr>
## 1 Afghanistan 1999 745/19987071
## 2 Afghanistan 2000 2666/20595360
## 3 Brazil      1999 37737/172006362
## 4 Brazil      2000 80488/174504898
## 5 China       1999 212258/1272915272
## 6 China       2000 213766/1280428583
```

Övningar

1. Vad gör argumenten `extra` och `fill`? Experimentera med de olika alternativen med följande datasets:

```
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"))

tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"))
```

Missing values

När vi förändrar ett dataset uppstår ofta ett problem med *missing values*. Ett värde kan vara *missing* på två sätt:

1. Explicit, dvs flaggad med NA.

2. Implicit, dvs. finns helt enkelt inte med i data.

Vi illustrerar detta med nedanstående tibble:

```
stocks <- tibble(
  year = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr = c( 1,  2,  3,  4,  2,  3,  4),
  return = c(1.88, 0.59, 0.35, NA, 0.92, 0.17, 2.66)
)
```

Här finns två *missing values*:

- *return* för fjärde kvartalet 2015 är explicit *missing*, dvs innehåller NA.
- *return* för första kvartalet 2016 är implicit *missing* eftersom den inte finns med i data.

Sättet på vilket data representeras kan göra implicit *missing values* explicit. Till exempel kan vi göra detta genom att sätta `year` i kolumner:

```
stocks %>%
  spread(year, return)

## # A tibble: 4 x 3
##   qtr `2015` `2016`
##   <dbl> <dbl> <dbl>
## 1     1  1.88    NA
## 2     2  0.59    0.92
## 3     3  0.35    0.17
## 4     4    NA    2.66
```

Om *missing values* i en viss situation inte behöver ha någon betydelse kan man använda argumentet `na.rm = TRUE` i `gather()` för att göra ett *missing value* implicit:

```
stocks %>%
  spread(year, return) %>%
  gather(year, return, `2015`:`2016`, na.rm = TRUE)

## # A tibble: 6 x 3
##   qtr year return
##   <dbl> <chr> <dbl>
## 1     1 2015  1.88
## 2     2 2015  0.59
## 3     3 2015  0.35
## 4     2 2016  0.92
## 5     3 2016  0.17
## 6     4 2016  2.66
```

Ett annat viktigt verktyg för att göra *missing values* explicit är `complete()`:

```
stocks %>%
  complete(year, qtr)

## # A tibble: 8 x 3
##   year qtr return
##   <dbl> <dbl> <dbl>
## 1 2015   1  1.88
## 2 2015   2  0.59
## 3 2015   3  0.35
## 4 2015   4  NA
## 5 2016   1  NA
## 6 2016   2  0.92
## 7 2016   3  0.17
## 8 2016   4  2.66
```

`complete()` använder en uppsättning kolumner och identifierar alla unika kombinationer. Därigenom försäkras man sig om att data innehåller samtliga värden (med explicita NAs där det behövs).

Det finns ett ytterligare bra verktyg för att arbeta med *missing values*. Man stöter ibland på data där *missing values* indikerar att ett tidigare värde gäller, t.ex. i en tabell som nedan:

```
treatment <- tribble(
  ~ person,      ~ treatment, ~response,
  "Derrick Whitmore", 1,      7,
  NA,             2,      10,
  NA,             3,      9,
  "Katherine Burke", 1,      4
)
```

Du kan fylla i sådana *missing values* med `fill()`. Det använder ett set av kolumner där du vill ersätta *missing values* med det senaste förekommande värdet som inte är NA ("last observation carried forward"):

```
treatment %>%
  fill(person)

## # A tibble: 4 x 3
##   person      treatment response
##   <chr>      <dbl> <dbl>
## 1 Derrick Whitmore     1     7
## 2 Derrick Whitmore     2    10
## 3 Derrick Whitmore     3     9
## 4 Katherine Burke      1     4
```

Övningar

1. Jämför argumentet `fill` med `spread()` och `complete()`. Skillnader?
2. Vad gör riktningsargumentet `.direction` till `fill()` ?

En case study

Som en avslutning på detta avsnitt ska vi använda funktionerna för att rensa ett ganska komplext dataset. Det är ett dataset hämtat från WHO och innehåller data om tuberkulos nedbrutet på år, nation, ålder, gender och diagnostisk metod. Källan är 2014 World Health Organization Global Tuberculosis Report, och finns på <http://www.who.int/tb/country/data/download/en/>. Det finns också inbyggt i `tidyr` och hämtas med hjälp av

```
tidyr::who
```

```
## # A tibble: 7,240 x 60
##   country iso2 iso3 year new_sp_m014 new_sp_m1524 new_sp_m2534
##   <chr>   <chr> <chr> <int>   <int>   <int>   <int>
## 1 Afghan~ AF   AFG  1980     NA     NA     NA
## 2 Afghan~ AF   AFG  1981     NA     NA     NA
## 3 Afghan~ AF   AFG  1982     NA     NA     NA
## 4 Afghan~ AF   AFG  1983     NA     NA     NA
## 5 Afghan~ AF   AFG  1984     NA     NA     NA
## 6 Afghan~ AF   AFG  1985     NA     NA     NA
## 7 Afghan~ AF   AFG  1986     NA     NA     NA
## 8 Afghan~ AF   AFG  1987     NA     NA     NA
## 9 Afghan~ AF   AFG  1988     NA     NA     NA
## 10 Afghan~ AF   AFG  1989     NA     NA     NA
## # ... with 7,230 more rows, and 53 more variables: new_sp_m3544 <int>,
## #   new_sp_m4554 <int>, new_sp_m5564 <int>, new_sp_m65 <int>,
## #   new_sp_f014 <int>, new_sp_f1524 <int>, new_sp_f2534 <int>,
## #   new_sp_f3544 <int>, new_sp_f4554 <int>, new_sp_f5564 <int>,
## #   new_sp_f65 <int>, new_sn_m014 <int>, new_sn_m1524 <int>,
## #   new_sn_m2534 <int>, new_sn_m3544 <int>, new_sn_m4554 <int>,
## #   new_sn_m5564 <int>, new_sn_m65 <int>, new_sn_f014 <int>,
## #   new_sn_f1524 <int>, new_sn_f2534 <int>, new_sn_f3544 <int>,
## #   new_sn_f4554 <int>, new_sn_f5564 <int>, new_sn_f65 <int>,
## #   new_ep_m014 <int>, new_ep_m1524 <int>, new_ep_m2534 <int>,
## #   new_ep_m3544 <int>, new_ep_m4554 <int>, new_ep_m5564 <int>,
## #   new_ep_m65 <int>, new_ep_f014 <int>, new_ep_f1524 <int>,
## #   new_ep_f2534 <int>, new_ep_f3544 <int>, new_ep_f4554 <int>,
## #   new_ep_f5564 <int>, new_ep_f65 <int>, newrel_m014 <int>,
## #   newrel_m1524 <int>, newrel_m2534 <int>, newrel_m3544 <int>,
```

```
## # newrel_m4554 <int>, newrel_m5564 <int>, newrel_m65 <int>,
## # newrel_f014 <int>, newrel_f1524 <int>, newrel_f2534 <int>,
## # newrel_f3544 <int>, newrel_f4554 <int>, newrel_f5564 <int>,
## # newrel_f65 <int>
```

Det finns även en nyckel till detta dataset vilken hämtas med

```
?who
```

Detta är ett exempel på ett *messy* dataset, med mängder av redundanta kolumner, udda variabelnamn, massor av NA:s, som gjort för en övning i att rensa ett data set.

Ett rimligt sätt att börja är att samla ihop alla kolumner som inte är variabler: `country`, `iso2` och `iso3` är tre variabler som alla anger namnet på nationen. `year` är uppenbart också en variabel.

Om vi kikar på de övriga kolumnnamnen (t.ex. `new_sp_m014`, `new_ep_m014`, `new_ep_f014`, osv) och deras innehåll verkar dessa vara värden, inte variabler.

Så låt oss börja med att samla ihop alla dessa kolumner. Eftersom vi inte riktigt vet vad kolumnerna representerar så anger vi den nya nyckel-kolumnen med det generiska `key` och eftersom vi antar att cellerna representerar *antal* TB-fall anger vi den nya värde-kolumnen som `cases`. Här finns massor av *missing values* så tills vidare använder vi `na.rm = TRUE` för att kunna fokusera på de befintliga värdena:

```
who1 <- who %>%
  gather(new_sp_m014:newrel_f65, key = "key", value = "cases", na.rm = TRUE)
```

```
who1
```

```
## # A tibble: 76,046 x 6
##   country iso2 iso3 year key      cases
##   <chr>    <chr> <chr> <int> <chr>    <int>
## 1 Afghanistan AF   AFG  1997 new_sp_m014    0
## 2 Afghanistan AF   AFG  1998 new_sp_m014   30
## 3 Afghanistan AF   AFG  1999 new_sp_m014    8
## 4 Afghanistan AF   AFG  2000 new_sp_m014   52
## 5 Afghanistan AF   AFG  2001 new_sp_m014  129
## 6 Afghanistan AF   AFG  2002 new_sp_m014   90
## 7 Afghanistan AF   AFG  2003 new_sp_m014  127
## 8 Afghanistan AF   AFG  2004 new_sp_m014  139
## 9 Afghanistan AF   AFG  2005 new_sp_m014  151
## 10 Afghanistan AF   AFG  2006 new_sp_m014  193
## # ... with 76,036 more rows
```

Vi kan få en aning om strukturen bland värdena i den nya nyckel-kolumnen genom att räkna antalet kategorier:

```
who1 %>%
  count(key)

## # A tibble: 56 x 2
##   key      n
##   <chr>   <int>
## 1 new_ep_f014 1032
## 2 new_ep_f1524 1021
## 3 new_ep_f2534 1021
## 4 new_ep_f3544 1021
## 5 new_ep_f4554 1017
## 6 new_ep_f5564 1017
## 7 new_ep_f65 1014
## 8 new_ep_m014 1038
## 9 new_ep_m1524 1026
## 10 new_ep_m2534 1020
## # ... with 46 more rows
```

Enligt kodnyckeln (?who) tolkas kategorierna på följande sätt:

- De första tre bokstäverna anger om TB-fallet är nytt eller gammalt.
- Nästa två bokstäver beskriver typen av TB:
- *rel* står för återfall (relapse)
- *ep* står för TB utanför lungvävnaden (extrapulmonary TB)
- *sn* står för lung-TBC so inte kunnat diagnosticeras med hjälp av prov på upphostning (smear negative)
- *sp* står för lung-TBC som kunnat diagnosticeras med hjälp av prov på upphostning (smear positive)
- Den sjätte bokstaven står för kön hos TB-patienten (males (m) och females (f)).
- Övriga siffror anger åldersgrupp (7 st):
 - 014 = 0 – 14 years old
 - 1524 = 15 – 24 years old
 - 2534 = 25 – 34 years old
 - 3544 = 35 – 44 years old
 - 4554 = 45 – 54 years old
 - 5564 = 55 – 64 years old
 - 65 = 65 or older

Det finns en liten inkonsistens i nyckel-kolumnen: istället för `new_rel` används `newrel` (svårt att se vid i tibble-översikten men ses med hjälp av `names(who)`). Det behöver korrigeras för att få `separate()` att fungera som det ska. Vi fixar det genom att ersätta `newrel` med `new_rel`. För detta använder vi `str_replace()` (återkommer till denna funktion i senare avsnitt).

```
who2 <- who1 %>%
  mutate(key = stringr::str_replace(key, "newrel", "new_rel"))
```

```
who2
```

```
## # A tibble: 76,046 x 6
##   country iso2 iso3 year key    cases
##   <chr>    <chr> <chr> <int> <chr>    <int>
## 1 Afghanistan AF   AFG  1997 new_sp_m014    0
## 2 Afghanistan AF   AFG  1998 new_sp_m014   30
## 3 Afghanistan AF   AFG  1999 new_sp_m014    8
## 4 Afghanistan AF   AFG  2000 new_sp_m014   52
## 5 Afghanistan AF   AFG  2001 new_sp_m014  129
## 6 Afghanistan AF   AFG  2002 new_sp_m014   90
## 7 Afghanistan AF   AFG  2003 new_sp_m014  127
## 8 Afghanistan AF   AFG  2004 new_sp_m014  139
## 9 Afghanistan AF   AFG  2005 new_sp_m014  151
## 10 Afghanistan AF   AFG  2006 new_sp_m014  193
## # ... with 76,036 more rows
```

Vi kollar resultatet med

```
names(who2)
```

Vi kan separera värdena under varje kategori med två omgångar med `separate()`. Först delar vi upp kategorierna vid varje underscore:

```
who3 <- who2 %>%
  separate(key, c("new", "type", "sexage"), sep = "_")
```

```
who3
```

```
## # A tibble: 76,046 x 8
##   country iso2 iso3 year new type sexage cases
##   <chr>    <chr> <chr> <int> <chr> <chr> <chr>    <int>
## 1 Afghanistan AF   AFG  1997 new  sp  m014    0
## 2 Afghanistan AF   AFG  1998 new  sp  m014   30
## 3 Afghanistan AF   AFG  1999 new  sp  m014    8
## 4 Afghanistan AF   AFG  2000 new  sp  m014   52
## 5 Afghanistan AF   AFG  2001 new  sp  m014  129
## 6 Afghanistan AF   AFG  2002 new  sp  m014   90
## 7 Afghanistan AF   AFG  2003 new  sp  m014  127
## 8 Afghanistan AF   AFG  2004 new  sp  m014  139
## 9 Afghanistan AF   AFG  2005 new  sp  m014  151
## 10 Afghanistan AF   AFG  2006 new  sp  m014  193
## # ... with 76,036 more rows
```


Vi kan göra oss av med kolumnerna `new` (den är konstant), `iso2` och `iso3` (redundanta):

```
who4 <- who3 %>%
  select(-new, -iso2, -iso3)
```

Sedan separerar vi `sexage` till `sex` och `age` genom att dela upp värdena vid den första bokstaven:

```
who5 <- who4 %>%
  separate(sexage, c("sex", "age"), sep = 1)

who5

## # A tibble: 76,046 x 6
##   country   year type sex  age  cases
##   <chr>     <int> <chr> <chr> <chr> <int>
## 1 Afghanistan 1997 sp   m   014    0
## 2 Afghanistan 1998 sp   m   014   30
## 3 Afghanistan 1999 sp   m   014    8
## 4 Afghanistan 2000 sp   m   014   52
## 5 Afghanistan 2001 sp   m   014  129
## 6 Afghanistan 2002 sp   m   014   90
## 7 Afghanistan 2003 sp   m   014  127
## 8 Afghanistan 2004 sp   m   014  139
## 9 Afghanistan 2005 sp   m   014  151
## 10 Afghanistan 2006 sp   m   014  193
## # ... with 76,036 more rows
```

Därmed har vi rensat data till att bli ett *tidy* dataset.

Istället för att göra det ett steg i sänder kan du göra samma sak genom att bygga upp en "pipe":

```
who %>%
  gather(key, value, new_sp_m014:newrel_f65, na.rm = TRUE) %>%
  mutate(key = stringr::str_replace(key, "newrel", "new_rel")) %>%
  separate(key, c("new", "var", "sexage")) %>%
  select(-new, -iso2, -iso3) %>%
  separate(sexage, c("sex", "age"), sep = 1)

## # A tibble: 76,046 x 6
##   country   year var sex  age  value
##   <chr>     <int> <chr> <chr> <chr> <int>
## 1 Afghanistan 1997 sp   m   014    0
## 2 Afghanistan 1998 sp   m   014   30
## 3 Afghanistan 1999 sp   m   014    8
## 4 Afghanistan 2000 sp   m   014   52
```

```
## 5 Afghanistan 2001 sp m 014 129
## 6 Afghanistan 2002 sp m 014 90
## 7 Afghanistan 2003 sp m 014 127
## 8 Afghanistan 2004 sp m 014 139
## 9 Afghanistan 2005 sp m 014 151
## 10 Afghanistan 2006 sp m 014 193
## # ... with 76,036 more rows
```

Övningar

1. Här satte vi `na.rm = TRUE` bara för att göra det enklare att kolla att vi hade korrekta värden. Men är det rimligt att göra så? Finns implicita/explicita *missing values*?
2. Vad händer om man inte gör beräkningen? (`mutate(key = stringr::str_replace(key, "newrel", "new_rel"))`)
3. Vi antog att `iso2` och `iso3` var redundanta med `country`. Bekräfta detta.

Non-tidy data

Wickham använder ordet “messy” om data som inte är “tidy”. Men han är också noga med att framhålla att det är en grov förenkling - det finns massor av användbara och välorganiserade data som inte är “tidy” i Wickhams betydelse och det kan finnas goda skäl att använda andra sätt att representera data än *tidy data*. Han refererar till ett tankvärt blogginlägg för att lära mer om *non-tidy data*.

<http://simplystatistics.org/2016/02/17/non-tidy-data/>

Relationer mellan datamängder

Introduktion

Oftast omfattar en analys av data att kombinera flera tabeller, att relatera dem till varandra. Relationer definieras alltid mellan två tabeller - relationerna mellan tre eller flera tabeller bygger alltid på att det finns en relation mellan ett par tabeller. För att arbeta med relationer behövs alltså *ett antal verb* för att arbeta med tabell-par. Det finns tre typer av sådana verb:

1. *Mutating joins* - lägger till nya variabler till en tabell från matchande observationer i en annan tabell.
2. *Filtering joins* - filtrerar observationer från en tabell baserat på om de matchar observationer i en annan tabell eller inte.
3. *Set operations* - behandlar observationer som om de vore en uppsättning element.

Vi ska använda data från modulen `nycflights13`:

```
library(tidyverse)
library(nycflights13)
```

`nycflights13`

`nycflights13` innehåller fyra tibbles relaterade till tabellen `flights` som vi använde i data transformation:

1. `airlines` innehåller det fullständiga namnet på fraktlinjebolaget och deras förkortningar:

`airlines`

```
## # A tibble: 16 x 2
##   carrier name
##   <chr> <chr>
## 1 9E    Endeavor Air Inc.
## 2 AA    American Airlines Inc.
## 3 AS    Alaska Airlines Inc.
## 4 B6    JetBlue Airways
## 5 DL    Delta Air Lines Inc.
## 6 EV    ExpressJet Airlines Inc.
## 7 F9    Frontier Airlines Inc.
## 8 FL    AirTran Airways Corporation
## 9 HA    Hawaiian Airlines Inc.
## 10 MQ   Envoy Air
## 11 OO   SkyWest Airlines Inc.
## 12 UA   United Air Lines Inc.
## 13 US   US Airways Inc.
```

```
## 14 VX    Virgin America
## 15 WN    Southwest Airlines Co.
## 16 YV    Mesa Airlines Inc.
```

2. airports innehåller information om varje flygplats, identifierad via dess flygplatskod:

```
airports
```

```
## # A tibble: 1,458 x 8
##   faa name          lat lon alt tz dst tzone
##   <chr> <chr>      <dbl> <dbl> <int> <dbl> <chr> <chr>
## 1 04G Lansdowne Airport  41.1 -80.6 1044 -5 A America/New_~
## 2 06A Moton Field Municipa~ 32.5 -85.7 264 -6 A America/Chic~
## 3 06C Schaumburg Regional 42.0 -88.1 801 -6 A America/Chic~
## 4 06N Randall Airport   41.4 -74.4 523 -5 A America/New_~
## 5 09J Jekyll Island Airport 31.1 -81.4 11 -5 A America/New_~
## 6 0A9 Elizabethton Municip~ 36.4 -82.2 1593 -5 A America/New_~
## 7 0G6 Williams County Airp~ 41.5 -84.5 730 -5 A America/New_~
## 8 0G7 Finger Lakes Regiona~ 42.9 -76.8 492 -5 A America/New_~
## 9 0P2 Shoestring Aviation ~ 39.8 -76.6 1000 -5 U America/New_~
## 10 0S9 Jefferson County Intl 48.1 -123. 108 -8 A America/Los_~
## # ... with 1,448 more rows
```

3. planes innehåller information om varje plan identifierad via dess tailnum:

```
planes
```

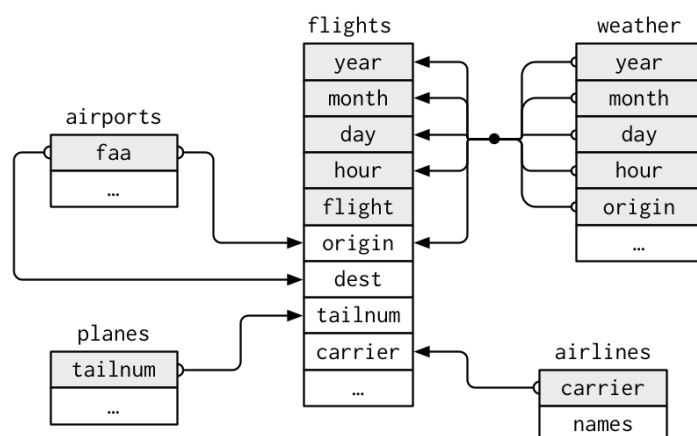
```
## # A tibble: 3,322 x 9
##   tailnum year type    manufacturer model engines seats speed engine
##   <chr> <int> <chr>    <chr>      <chr> <int> <int> <int> <chr>
## 1 N10156 2004 Fixed win~ EMBRAER    EMB-1~    2 55 NA Turbo~
## 2 N102UW 1998 Fixed win~ AIRBUS INDUS~ A320-~    2 182 NA Turbo~
## 3 N103US 1999 Fixed win~ AIRBUS INDUS~ A320-~    2 182 NA Turbo~
## 4 N104UW 1999 Fixed win~ AIRBUS INDUS~ A320-~    2 182 NA Turbo~
## 5 N10575 2002 Fixed win~ EMBRAER    EMB-1~    2 55 NA Turbo~
## 6 N105UW 1999 Fixed win~ AIRBUS INDUS~ A320-~    2 182 NA Turbo~
## 7 N107US 1999 Fixed win~ AIRBUS INDUS~ A320-~    2 182 NA Turbo~
## 8 N108UW 1999 Fixed win~ AIRBUS INDUS~ A320-~    2 182 NA Turbo~
## 9 N109UW 1999 Fixed win~ AIRBUS INDUS~ A320-~    2 182 NA Turbo~
## 10 N110UW 1999 Fixed win~ AIRBUS INDUS~ A320-~    2 182 NA Turbo~
## # ... with 3,312 more rows
```

4. weather innehåller information om vädret vid varje NYC airport per timma:

```
weather
```

```
## # A tibble: 26,115 x 15
##   origin year month day hour temp dewp humid wind_dir wind_speed
##   <chr>   <dbl> <dbl> <int> <int> <dbl> <dbl> <dbl>   <dbl>   <dbl>
## 1 EWR    2013     1     1     1 39.0 26.1 59.4    270    10.4
## 2 EWR    2013     1     1     2 39.0 27.0 61.6    250     8.06
## 3 EWR    2013     1     1     3 39.0 28.0 64.4    240    11.5
## 4 EWR    2013     1     1     4 39.9 28.0 62.2    250    12.7
## 5 EWR    2013     1     1     5 39.0 28.0 64.4    260    12.7
## 6 EWR    2013     1     1     6 37.9 28.0 67.2    240    11.5
## 7 EWR    2013     1     1     7 39.0 28.0 64.4    240    15.0
## 8 EWR    2013     1     1     8 39.9 28.0 62.2    250    10.4
## 9 EWR    2013     1     1     9 39.9 28.0 62.2    260    15.0
## 10 EWR   2013     1     1    10 41   28.0 59.6    260    13.8
## # ... with 26,105 more rows, and 5 more variables: wind_gust <dbl>,
## #   precip <dbl>, pressure <dbl>, visib <dbl>, time_hour <dtm>
```

Ett sätt att visa relationerna mellan dessa tabeller är med ett diagram:



Notera att relationerna alltid bygger på relationer mellan två tabeller:

- **flights** relaterar till **planes** via **tailnum**.
- **flights** relaterar till **airlines** via **carrier**.
- **flights** relaterar till **airports** via **origin** och **dest**.
- **flights** relaterar till **weather** via **origin** (geografi), och **year**, **month**, **day** och **hour** (tidpunkt).

Keys

Variablerna som används för att koppla ihop tabeller kallas *keys*. En key är en variabel (eller variabler) som **entydigt** identifierar en observation. Det finns två typer av *keys*:

1. En *primary key* identifierar entydigt en observation i samma tabell. T.ex. **planes\$tailnum** är en primär nyckel eftersom den entydigt identifierar varje plan i tabellen **planes**.

2. En *foreign key* identifierar entydigt en observation i en annan T.ex. `flights$tailnum` är en foreign key eftersom den i tabellen `flights` matchar varje flight till ett specifikt plan.

När du identifierat de primära nycklarna i tabellerna är det klokt att verifiera att de verkligen entydigt identifierar varje observation. Ett effektivt sätt att göra det är att räkna (med hjälp av `count()`) primärnycklarna och filtrera ut dem som är fler än ett. Prova:

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)

## # A tibble: 0 x 2
## # ... with 2 variables: tailnum <chr>, n <int>

weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)

## # A tibble: 3 x 6
##   year month   day hour origin    n
##   <dbl> <dbl> <int> <int> <chr> <int>
## 1 2013    11     3     1 EWR     2
## 2 2013    11     3     1 JFK     2
## 3 2013    11     3     1 LGA     2
```

Ibland saknas en tabell en explicit *primary key*: varje rad är en observation men ingen kombination av variabler identifierar den. Till exempel, vilken är primärnyckel i `flight`-tabellen? Du kanske tänker att det är datum (`date`) + flight-numret eller `tailnum`, men ingen av dessa kombinationer är unik:

```
flights %>%
  count(year, month, day, flight) %>%
  filter(n > 1)

## # A tibble: 29,768 x 5
##   year month   day flight    n
##   <int> <int> <int> <int> <int>
## 1 2013     1     1     1     2
## 2 2013     1     1     3     2
## 3 2013     1     1     4     2
## 4 2013     1     1    11     3
## 5 2013     1     1    15     2
## 6 2013     1     1    21     2
## 7 2013     1     1    27     4
## 8 2013     1     1    31     2
## 9 2013     1     1    32     2
```

```
## 10 2013 1 1 35 2
## # ... with 29,758 more rows

flights %>%
  count(year, month, day, tailnum) %>%
  filter(n > 1)

## # A tibble: 64,928 x 5
##   year month  day tailnum    n
##   <int> <int> <int> <chr>  <int>
## 1 2013    1    1 N0EGMQ    2
## 2 2013    1    1 N11189    2
## 3 2013    1    1 N11536    2
## 4 2013    1    1 N11544    3
## 5 2013    1    1 N11551    2
## 6 2013    1    1 N12540    2
## 7 2013    1    1 N12567    2
## 8 2013    1    1 N13123    2
## 9 2013    1    1 N13538    3
## 10 2013    1    1 N13566    3
## # ... with 64,918 more rows
```

Om en tabell saknar en primärnyckel kan det ibland vara vettigt att skapa en med hjälp av `mutate()` och `row_number()`, en *surrogat-nyckel*.

En primärnyckel och en korresponderande *foreign key* i en annan tabell utgör en relation. Relationer är vanligen en-till-flera. Till exempel, varje flight har ett plan men varje plan har flera flights. Det finns naturligtvis alla varianter på relationer, en-till-en och flera-till-flera.

Mutating joins

Först ska vi kika på ett verktyg för att kombinera två tabeller: *the mutating join*. Det matchar först poster med hjälp av dess nycklar, sedan kopieras valda variabler från den ena tabellen till den andra.

Liksom `mutate()` adderar *join*-funktionerna variabler till höger så om du har många variabler blir det svårt att se de nya på skärmen. Vi ska därför, för nedanstående exempel, skapa ett *smalare* dataset:

```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
flights2

## # A tibble: 336,776 x 8
##   year month  day hour origin dest tailnum carrier
##   <int> <int> <int> <dbl> <chr>  <chr> <chr>  <chr>
## 1 2013    1    1    5 EWR   IAH  N14228 UA
```

```
## 2 2013 1 1 5 LGA IAH N24211 UA
## 3 2013 1 1 5 JFK MIA N619AA AA
## 4 2013 1 1 5 JFK BQN N804JB B6
## 5 2013 1 1 6 LGA ATL N668DN DL
## 6 2013 1 1 5 EWR ORD N39463 UA
## 7 2013 1 1 6 EWR FLL N516JB B6
## 8 2013 1 1 6 LGA IAD N829AS EV
## 9 2013 1 1 6 JFK MCO N593JB B6
## 10 2013 1 1 6 LGA ORD N3ALAA AA
## # ... with 336,766 more rows
```

(Kom ihåg att du i RStudio alltid kan använda `view()` för att runda detta problem.)

Låt oss säga att du nu vill komplettera med hela namnet på flygbolaget i `flights2`. Det kan du göra med `left_join()`:

```
flights2 %>%
  select(-origin, -dest) %>%
  left_join(airlines, by = "carrier")

## # A tibble: 336,776 x 7
##   year month   day hour tailnum carrier name
##   <int> <int> <int> <dbl> <chr>   <chr>   <chr>
## 1 2013     1     1     5 N14228 UA    United Air Lines Inc.
## 2 2013     1     1     5 N24211 UA    United Air Lines Inc.
## 3 2013     1     1     5 N619AA AA    American Airlines Inc.
## 4 2013     1     1     5 N804JB B6    JetBlue Airways
## 5 2013     1     1     6 N668DN DL    Delta Air Lines Inc.
## 6 2013     1     1     5 N39463 UA    United Air Lines Inc.
## 7 2013     1     1     6 N516JB B6    JetBlue Airways
## 8 2013     1     1     6 N829AS EV    ExpressJet Airlines Inc.
## 9 2013     1     1     6 N593JB B6    JetBlue Airways
## 10 2013     1     1     6 N3ALAA AA    American Airlines Inc.
## # ... with 336,766 more rows
```

Resultatet av sammanslagningen är en ytterligare variabel: `name`. Du kan få samma resultat med hjälp av `mutate()`:

```
flights2 %>%
  select(-origin, -dest) %>%
  mutate(name = airlines$name[match(carrier, airlines$carrier)])

## # A tibble: 336,776 x 7
##   year month   day hour tailnum carrier name
##   <int> <int> <int> <dbl> <chr>   <chr>   <chr>
```



```
## 1 2013 1 1 5 N14228 UA United Air Lines Inc.
## 2 2013 1 1 5 N24211 UA United Air Lines Inc.
## 3 2013 1 1 5 N619AA AA American Airlines Inc.
## 4 2013 1 1 5 N804JB B6 JetBlue Airways
## 5 2013 1 1 6 N668DN DL Delta Air Lines Inc.
## 6 2013 1 1 5 N39463 UA United Air Lines Inc.
## 7 2013 1 1 6 N516JB B6 JetBlue Airways
## 8 2013 1 1 6 N829AS EV ExpressJet Airlines Inc.
## 9 2013 1 1 6 N593JB B6 JetBlue Airways
## 10 2013 1 1 6 N3ALAA AA American Airlines Inc.
## # ... with 336,766 more rows
```

Men detta uttryck är svårt att generalisera när du behöver matcha flera variabler och kräver i så fall en hel del eftertanke.

De följande sektionerna ska diskutera i detalj hur dessa sammanslagningar, *mutating joins*, fungerar. Vi ska först gå igenom hur *joins* kan visualiseras, sedan använda dessa visualiseringar för att förklara de fyra *join*-funktionerna, *the inner join* och tre *outer joins*. Vi ska vidare kika på vad som händer om matchningarna inte är unika och hur man hanterar det. Slutligen ska vi se hur man definierar *key*-variabler för en viss *join*-funktion.

Att förstå *joins*

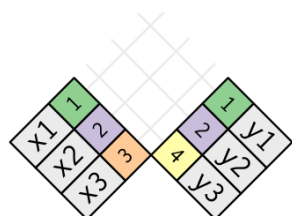
Låt oss kika på en visualisering:

x		y	
1	x1	1	y1
2	x2	2	y2
3	x3	4	y3

```
x <- tribble(
  ~key, ~val_x,
  1, "x1",
  2, "x2",
  3, "x3"
)
y <- tribble(
  ~key, ~val_y,
  1, "y1",
  2, "y2",
  4, "y3"
)
```

De färgade kolumnerna representerar nycklarna, the keys - dessa används för att matcha raderna i tabellerna. Den gråa kolumnen representerar *värde*-kolumnen. I exempen används en nyckel men det är ganska straightforward att använda funktionerna för flera nycklar och värden. En *join* är ett sätt att

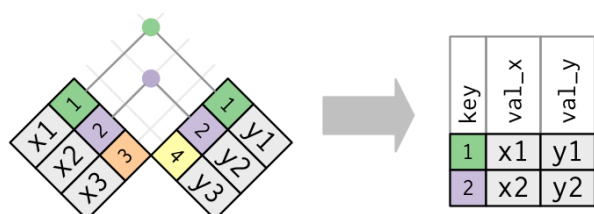
binda samman en rad i tabell x med 0, 1 eller flera poster i tabell y. Nedanstående diagram visar potentiella matchningar som skärningen mellan ett linje-par:



I figurerna kommer matchningarna att representeras av punkter: antalet punkter = antalet matchningar = antalet rader i output.

Inner join

Den enklaste typen av *joins* är *inner join*. En *inner join* matchar par av observationer när nycklarna är lika:



Output från en *inner join* är en ny data-frame som innehåller nyckeln, x-värdena och y-värdena. Vi använder `by` för att tala om för dplyr vilken variabel som är nyckel:

```
x %>%
  inner_join(y, by = "key")

## # A tibble: 2 x 3
##   key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
```

Den viktigaste egenskapen hos en *inner join* är att icke-matchade poster inte inkluderas i output. Generellt bör man därför vara försiktig med *inner joins* eftersom det är lätt att tappa data.

Outer joins

En *inner join* behåller poster som finns i båda tabellerna. En *outer join* behåller poster som finns åtminstone i en av tabellerna. Det finns tre typer av *outer joins*:

- en *left join* behåller alla poster i x
- en *right join* behåller alla poster i y
- en *full join* behåller alla poster i x och y

Dessa *joins* fungerar genom att lägga till en “virtuell” post till vardera tabellerna. Denna post har en *key* som alltid matchar (om ingen annan *key* matchar), och ett värde = NA. Grafiskt kan det illustreras så här:

[(join-outer.png)]

Den vanligast förekommande *join* är *left joins*. Du använder denna närhelst du vill lägga till data från en annan tabell eftersom den bevarar samtliga poster i original-tabellen, även om de inte har någon matchning.

Duplicate keys

Så här långt har vi antagit att *keys* är unika. Men det är inte alltid fallet. Och vafd händer då? Det finns två möjligheter:

1. En av tabellerna har dubbla *keys*. Detta är användbart då du vill lägga till ytterligare information eftersom detta vanligen är en en-till-flera-relation.

[(join-one-to-many.png)]

```
x <- tribble(
  ~key, ~val_x,
  1, "x1",
  2, "x2",
  2, "x3",
  1, "x4"
)
y <- tribble(
  ~key, ~val_y,
  1, "y1",
  2, "y2"
)
left_join(x, y, by = "key")

## # A tibble: 4 x 3
##   key val_x val_y
##   <dbl> <chr> <chr>
## 1     1  x1   y1
## 2     2  x2   y2
## 3     2  x3   y2
## 4     1  x4   y1
```

2. Båda tabellerna har dubbla *keys*. Detta är vanligtvis ett *error* eftersom ingen av nycklarna identifierar en unik post. När du slår samman dubbla nycklar kommer du att få samtliga möjliga kombinationer av poster, den Cartesianska produkten:

[(join-many-to-many.png)]

```
x <- tribble(
  ~key, ~val_x,
  1, "x1",
  2, "x2",
  2, "x3",
  3, "x4"
)
y <- tribble(
  ~key, ~val_y,
  1, "y1",
  2, "y2",
  2, "y3",
  3, "y4"
)
left_join(x, y, by = "key")

## # A tibble: 6 x 3
##   key val_x val_y
##   <dbl> <chr> <chr>
## 1     1  x1   y1
## 2     2  x2   y2
## 3     2  x2   y3
## 4     2  x3   y2
## 5     2  x3   y3
## 6     3  x4   y4
```

Definiera nyckel-kolumner (*key columns*)

Hittills har tabellerna alltid slagits samman med hjälp av en enskild variabel (definierad med `by`) vilken varit densamma i båda tabellerna. Men du kan använda andra värden till `by` för att binda samman tabellerna:

- default är `by = NULL`, som använde samtliga variabler som är lika i båda tabellerna. Till exempel matchar `flights` och `weather`-tabellerna via de gemensamma variablerna: `year`, `month`, `day`, `hour` och `origin`.

```
flights2 %>%
  left_join(weather)

## Joining, by = c("year", "month", "day", "hour", "origin")

## # A tibble: 336,776 x 18
##   year month  day hour origin dest tailnum carrier temp dewp humid
##   <dbl> <dbl> <int> <dbl> <chr>  <chr> <chr>  <chr>  <dbl> <dbl> <dbl>
## 1  2013     1     1     5 EWR   IAH  N14228 UA    39.0  28.0  64.4
```

```
## 2 2013 1 1 5 LGA IAH N24211 UA 39.9 25.0 54.8
## 3 2013 1 1 5 JFK MIA N619AA AA 39.0 27.0 61.6
## 4 2013 1 1 5 JFK BQN N804JB B6 39.0 27.0 61.6
## 5 2013 1 1 6 LGA ATL N668DN DL 39.9 25.0 54.8
## 6 2013 1 1 5 EWR ORD N39463 UA 39.0 28.0 64.4
## 7 2013 1 1 6 EWR FLL N516JB B6 37.9 28.0 67.2
## 8 2013 1 1 6 LGA IAD N829AS EV 39.9 25.0 54.8
## 9 2013 1 1 6 JFK MCO N593JB B6 37.9 27.0 64.3
## 10 2013 1 1 6 LGA ORD N3ALAA AA 39.9 25.0 54.8
## # ... with 336,766 more rows, and 7 more variables: wind_dir <dbl>,
## # wind_speed <dbl>, wind_gust <dbl>, precip <dbl>, pressure <dbl>,
## # visib <dbl>, time_hour <dtm>
```

- en *character*-vektor, by = "x". Detta liknar ovanstående men använder endast en eller några av de gemensamma variablerna. Till exempel har både `flight` och `planes` `year` gemensamt men de betyder olika saker så vi vill bara använda `tailnum` för att slå samman tabellerna:

```
flights2 %>%
```

```
left_join(planes, by = "tailnum")
```

```
## # A tibble: 336,776 x 16
```

```
##   year.x month   day hour origin dest tailnum carrier year.y type
```

```
##   <int> <int> <int> <dbl> <chr> <chr> <chr> <chr> <int> <chr>
```

```
## 1 2013 1 1 5 EWR IAH N14228 UA 1999 Fixe~
```

```
## 2 2013 1 1 5 LGA IAH N24211 UA 1998 Fixe~
```

```
## 3 2013 1 1 5 JFK MIA N619AA AA 1990 Fixe~
```

```
## 4 2013 1 1 5 JFK BQN N804JB B6 2012 Fixe~
```

```
## 5 2013 1 1 6 LGA ATL N668DN DL 1991 Fixe~
```

```
## 6 2013 1 1 5 EWR ORD N39463 UA 2012 Fixe~
```

```
## 7 2013 1 1 6 EWR FLL N516JB B6 2000 Fixe~
```

```
## 8 2013 1 1 6 LGA IAD N829AS EV 1998 Fixe~
```

```
## 9 2013 1 1 6 JFK MCO N593JB B6 2004 Fixe~
```

```
## 10 2013 1 1 6 LGA ORD N3ALAA AA NA <NA>
```

```
## # ... with 336,766 more rows, and 6 more variables: manufacturer <chr>,
```

```
## # model <chr>, engines <int>, seats <int>, speed <int>, engine <chr>
```

Notera att `year`-variablerna fått ett suffix vilket för att understryka att det är två olika variabler.

- en namngiven *character*-vektor, by = c("a" = "b"). Detta kommer att matcha variabler i tabell x med variabler i tabell y. Variablerna i x kommer att användas i output. Om vi till exempel vill göra en karta behöver vi kombinera `flights` med `airports` vilken innehåller koordinaterna för flygplatserna. Eftersom varje `flight` har en avgångs-flygplats och en destination behöver vi specificera vilken av flygplatserna som vi vill använda för kartan:

```
flights2 %>%
```

```
  left_join(airports, c("dest" = "faa"))
```

```
## # A tibble: 336,776 x 15
```

```
##   year month   day hour origin dest tailnum carrier name   lat lon
##   <int> <int> <int> <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl>
## 1 2013     1     1     5 EWR  IAH  N14228 UA   Geor~ 30.0 -95.3
## 2 2013     1     1     5 LGA  IAH  N24211 UA   Geor~ 30.0 -95.3
## 3 2013     1     1     5 JFK  MIA  N619AA AA   Miam~ 25.8 -80.3
## 4 2013     1     1     5 JFK  BQN  N804JB B6   <NA>  NA   NA
## 5 2013     1     1     6 LGA  ATL  N668DN DL   Hart~ 33.6 -84.4
## 6 2013     1     1     5 EWR  ORD  N39463 UA   Chic~ 42.0 -87.9
## 7 2013     1     1     6 EWR  FLL  N516JB B6   Fort~ 26.1 -80.2
## 8 2013     1     1     6 LGA  IAD  N829AS EV   Wash~ 38.9 -77.5
## 9 2013     1     1     6 JFK  MCO  N593JB B6   Orla~ 28.4 -81.3
## 10 2013     1     1     6 LGA  ORD  N3ALAA AA   Chic~ 42.0 -87.9
## # ... with 336,766 more rows, and 4 more variables: alt <int>, tz <dbl>,
## #   dst <chr>, tzone <chr>
```

```
flights2 %>%
```

```
  left_join(airports, c("origin" = "faa"))
```

```
## # A tibble: 336,776 x 15
```

```
##   year month   day hour origin dest tailnum carrier name   lat lon
##   <int> <int> <int> <dbl> <chr> <chr> <chr> <chr> <chr> <dbl> <dbl>
## 1 2013     1     1     5 EWR  IAH  N14228 UA   Newa~ 40.7 -74.2
## 2 2013     1     1     5 LGA  IAH  N24211 UA   La G~ 40.8 -73.9
## 3 2013     1     1     5 JFK  MIA  N619AA AA   John~ 40.6 -73.8
## 4 2013     1     1     5 JFK  BQN  N804JB B6   John~ 40.6 -73.8
## 5 2013     1     1     6 LGA  ATL  N668DN DL   La G~ 40.8 -73.9
## 6 2013     1     1     5 EWR  ORD  N39463 UA   Newa~ 40.7 -74.2
## 7 2013     1     1     6 EWR  FLL  N516JB B6   Newa~ 40.7 -74.2
## 8 2013     1     1     6 LGA  IAD  N829AS EV   La G~ 40.8 -73.9
## 9 2013     1     1     6 JFK  MCO  N593JB B6   John~ 40.6 -73.8
## 10 2013     1     1     6 LGA  ORD  N3ALAA AA   La G~ 40.8 -73.9
## # ... with 336,766 more rows, and 4 more variables: alt <int>, tz <dbl>,
## #   dst <chr>, tzone <chr>
```

Övningar

1. Beräkna den genomsnittliga förseningen per destination, sedan slå samman tabellen med airportsså att du kan visa den geografiska fördelningen av förseningar. Här är ett enkelt sätt att rita en karta över USA:

```
airports %>%
  semi_join(flights, c("faa" = "dest")) %>%
  ggplot(aes(lon, lat)) +
  borders("state") +
  geom_point() +
  coord_quickmap()
```

Bekymra dig inte om du inte förstår vad `semi_join()` gör - vi återkommer strax till denna funktion.

2. Lägg till koordinaterna för avgångs- och destinations-flygplatserna till `flights`.
3. Vid vilka vädertyper är en försening mer sannolik?

Filtering joins

Filtering joins matchar poster på samma sätt som *mutating joins* men påverkar posterna snarare än variablerna. Det finns två typer:

- `semi_join(x, y)` behåller samtliga poster i `x` som har en matchning i `y`.
- `anti_join(x, y)` droppar samtliga poster i `x` som har en matchning i `y`.

Semi-joins är användbara för att matcha filtrerade summerade variabler tillbaka till de ursprungliga posterna. Till exempel, tänk dig att du har identifierat de tio mest populära destinationerna:

```
top_dest <- flights %>%
  count(dest, sort = TRUE) %>%
  head(10)
top_dest

## # A tibble: 10 x 2
##   dest      n
##   <chr> <int>
## 1 ORD  17283
## 2 ATL  17215
## 3 LAX  16174
## 4 BOS  15508
## 5 MCO  14082
## 6 CLT  14064
## 7 SFO  13331
## 8 FLL  12055
## 9 MIA  11728
## 10 DCA   9705
```

Du vill nu identifiera samtliga flighter som gick till någon av dessa destinationer. Du kan själv definiera ett filter:

```

flights %>%
  filter(dest %in% top_dest$dest)

## # A tibble: 141,145 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>      <dbl>   <int>
## 1 2013     1     1    542           540         2     923
## 2 2013     1     1    554           600        -6     812
## 3 2013     1     1    554           558        -4     740
## 4 2013     1     1    555           600        -5     913
## 5 2013     1     1    557           600        -3     838
## 6 2013     1     1    558           600        -2     753
## 7 2013     1     1    558           600        -2     924
## 8 2013     1     1    558           600        -2     923
## 9 2013     1     1    559           559         0     702
##10 2013     1     1    600           600         0     851
## # ... with 141,135 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>

```

Men det är svårt att använda denna ansats för multipla nycklar. Om du till exempel identifierat de 10 dagar som haft de största genomsnittliga förseningarna. Hur skulle du konstruera ett filter som använde `year`, `month` och `day` för att matcha tillbaka till `flights`? Istället kan du använda en *semi-join* som knyter ihop de två tabellerna på samma sätt som en *mutating join* men istället för att lägga till nya kolumner använder endast de poster i `x` som har en matchning i `y`:

```

flights %>%
  semi_join(top_dest)

## Joining, by = "dest"

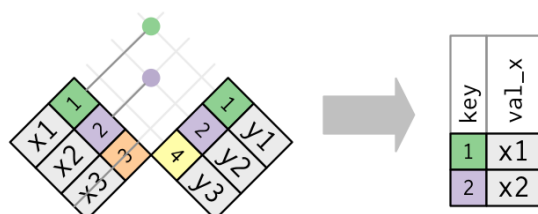
## # A tibble: 141,145 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>      <dbl>   <int>
## 1 2013     1     1    542           540         2     923
## 2 2013     1     1    554           600        -6     812
## 3 2013     1     1    554           558        -4     740
## 4 2013     1     1    555           600        -5     913
## 5 2013     1     1    557           600        -3     838
## 6 2013     1     1    558           600        -2     753
## 7 2013     1     1    558           600        -2     924
## 8 2013     1     1    558           600        -2     923
## 9 2013     1     1    559           559         0     702

```

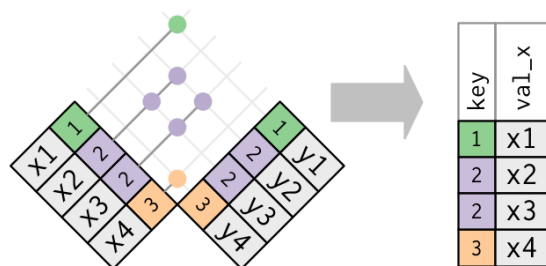


```
## 10 2013 1 1 600 600 0 851
## # ... with 141,135 more rows, and 12 more variables: sched_arr_time <int>,
## # arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## # origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## # minute <dbl>, time_hour <dtm>
```

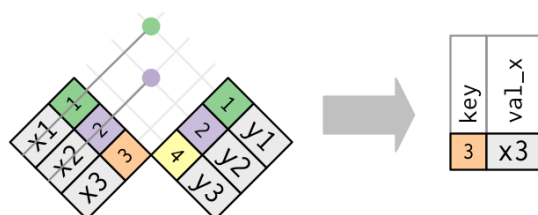
Grafiskt ser en *semi-join* ut så här:



Det är bara själva förekomsten av en matchning som är intressant; det spelar ingen roll vilken post som matchar. Detta innebär att *filtering joins* aldrig duplikerar poster vilket *mutating joins* kan göra:



Inversen av en *semi-join* är en *anti-join*. En *anti-join* behåller poster som *inte* har en matchning:



Anti-joins är användbara för att felsöka sammanslagningar som misslyckats. Till exempel, om man binder samman `flights` och `planes` kan det vara bra att veta hur många flighter som inte har en matchning i `planes`:

```
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)

## # A tibble: 722 x 2
##   tailnum    n
##   <chr> <int>
## 1 <NA>   2512
```

```
## 2 N725MQ 575
## 3 N722MQ 513
## 4 N723MQ 507
## 5 N713MQ 483
## 6 N735MQ 396
## 7 N0EGMQ 371
## 8 N534MQ 364
## 9 N542MQ 363
## 10 N531MQ 349
## # ... with 712 more rows
```

I Wickham's bok finns några nyttiga övningar. Kolla in <https://r4ds.had.co.nz/relational-data.html#exercises-29>.

Problem med *joins*

Några saker att tänka på:

1. Börja med att identifiera *key*-variabeln eller en kombination av variabler i vardera tabellen som identifierar varje post unikt.
2. Se till så att inga av variablerna i den primära nyckeln är *missing* - annars går det inte att identifiera observationen!
3. Kolla så att *foreign keys* matchar primära nycklar i andra tabeller. Ett smart sätt att göra det är att använda `anti_join()`.

Om du upptäcker att nycklar saknas behöver du vara noggrann med hur du användare *inner joins* och *outer joins*, och fundera över om du vill skippa observationer/poster som inte har en matchning.

Set operations

Den sista typen av begrepp är *set operations*. De kan vara bra då du vill bryta upp ett enstaka komplext filter i enklare delar. Samtliga *operations* arbetar på en hel post/rad och jämför värdena för varje variabel. De utgår från att x- och y-tabellerna har samma variabler och behandlar posterna som *set*:

- `intersect(x, y)` returnerar enbart observationer i både x och y.
- `union(x, y)` returnerar unika observationer i både x och y.
- `setdiff(x, y)` returnerar observationer i x som inte finns i y.

t.ex:

```
df1 <- tribble(
  ~x, ~y,
  1, 1,
  2, 1
)
df2 <- tribble(
```

```

~x, ~y,
1, 1,
1, 2
)

```

Det finns fyra möjligheter:

```
intersect(df1, df2)
```

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

```
##   y     x
```

```
## <dbl> <dbl>
```

```
## 1     1     1
```

```
## 2     2     1
```

```
union(df1, df2)
```

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

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

```
##
```

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

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

```
setdiff(df1, df2)
```

```
## Warning: Length of logical index must be 1 or 2, not 0
```

```
## # A tibble: 0 x 0
```

```
setdiff(df2, df1)
```

```
## Warning: Length of logical index must be 1 or 2, not 0
```

```
## # A tibble: 0 x 0
```

Textsträngar (strings)

Introduktion

Här ska det handla om text-manipulation i R. Det ska handla om hur textsträngar fungerar och hur man skapar dem, men fokus är på *regular expressions* eller *regexps*. *Regular expressions* är användbara då textsträngar ofta innehåller mer eller mindre ostrukturerade data och *regexps* är ett språk för att beskriva mönster i textsträngar.

Vi ska använda modulen **stringr** som *inte* finns i tidyverse så vi behöver ladda in det särskilt:

```
library(tidyverse)
library(stringr)
```

Basics

Du kan skapa textsträngar med antingen enkla eller dubbla citat-tecken.

```
string1 <- "This is a string"
string2 <- 'If I want to include a "quote" inside a string, I use single quotes'
```

För att inkludera ett bokstavligt citat-tecken kan du använda en *backslash* \ för att få det på plats:

```
double_quote <- "\" # or \""
single_quote <- "'" # or "'"
```

Om du vill inkludera en *backslash* behöver du alltså dubblera det: "\\".

Det finns en handfull specialtecken. De vanligaste är "" (ny rad) och "", men du finner en komplett lista i hjälpfunktionen för "?:?".

Ibland stöter man på strängar typ "\u00b5" vilket är ett sätt att skriva icke-engelska tecken som fungerar på alla plattformar:

```
x <- "\u00b5"
x
## [1] "μ"
```

Multipla strängar lagras ofta i en *character vector* som du skapar med c():

```
c("one", "two", "three")
## [1] "one" "two" "three"
```

stringr-funktioner

Det finns flera funktioner i base R för att arbeta med textsträngar men de kan vara inkonsistenta, vilket gör de svårare att minnas. Därför ska vi använda funktionerna i `stringr`. Dessa har mer intuitiva namn och samtliga börjar med `str_`. Till exempel visar `str_length` hur många tecken en textsträng innehåller:

```
str_length(c("a", "R for data science", NA))
```

```
## [1] 1 18 NA
```

Prefixet `str_` är särskilt användbart i Rstudio eftersom det triggar autokomplettering vilket gör att du ser samtliga `stringr`-funktioner:

<pre>> str_c > str_conv > str_count > str_detect > str_dup > str_extract > str_extract_all > str_ </pre>	<pre>{stringr} {stringr} {stringr} {stringr} {stringr} {stringr} {stringr} {stringr}</pre>	<pre>str_c(..., sep = "", collapse = NULL) To understand how str_c works, you need to imagine that you are building up a matrix of strings. Each input argument forms a column, and is expanded to the length of the longest argument, using the usual recycling rules. The sep string is inserted between each column. If collapse is NULL each row is collapsed into a single string. If non-NULL that string is inserted at the end of each row, and the entire matrix collapsed to a single string. Press F1 for additional help</pre>
----------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Kombinera strängar

För att kombinera textsträngar kan du använda `str_c()`:

```
str_c("x", "y")
```

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

```
str_c("x", "y", "z")
```

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

Använd argumentet `sep` för att kontrollera hur strängarna är separerade:

```
str_c("x", "y", sep = ", ")
```

```
## [1] "x, y"
```

Missing values är som oftast i R besvärliga. Om du vill ha "NA" explicit i textsträngen kan du använda `str_replace_na()`:

```
x <- c("abc", NA)
```

```
str_c("|-", x, "-|")
```

```
## [1] "|-abc-|" NA
```

```
str_c("|-", str_replace_na(x), "-|")
```

```
## [1] "|-abc-|" "-|NA-|"
```

Som framgår ovan är `str_c()` vektoriserad och recyklar kortare vektorer till samma längd som den längsta vektorn.

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
## [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

Objekt med längden 0 ignoreras. Detta är speciellt användbart tillsammans med `if` :

```
name <- "Göran"
time_of_day <- "morgon"
birthday <- FALSE
str_c(
  "God ", time_of_day, " ", name,
  if (birthday) " och GRATTIS på födelsedagen",
  "."
)
## [1] "God morgon Göran."
```

För att slå samman en vektor av textsträngar kan du använda `collapse`:

```
str_c(c("x", "y", "z"), collapse = ", ")
## [1] "x, y, z"
```

Extrahera delar av textsträngar (subsetting)

Du kan extrahera delar av en textsträng med hjälp av `str_sub()`. Argumenten `startoch` och `end` definierar den del som extraheras:

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
## [1] "App" "Ban" "Pea"
```

Negativa tal räknar från slutet:

```
str_sub(x, -3, -1)
## [1] "ple" "ana" "ear"
```

Du kan också använda `str_sub()` för att modifiera en textsträng:

```
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
## [1] "apple" "banana" "pear"
```

Övningar

1. Vad gör `str_wrap()`? När kan den funktionen vara bra?
2. Vad gör `str_trim()`? Vilken funktion är motsatsen till `str_trim()`?
3. Skriv en kod som amvandlar vektorn `c("a", "b", "c")` till a, b och c.

Matcha textmönster med hjälp av *regular expressions*

Regexps är ett mycket kondenserat språk vilket gör det möjligt att beskriva mönster i textsträngar. Det är inte alldeles enkelt att förstå och använda men när du väl kommit över tröskeln är de mycket användbara.

Vi ska använda funktionerna `str_view()` och `str_view_all()`. Dessa funktioner använder en textsträng och ett *regexp* och visar hur de matchar. Vi ska börja med ett enkelt uttryck och gradvis göra det mer komplicerat.

Basic matches

Den enklaste mönstret matchar ett exakt uttryck:

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
```

```
apple
banana
pear
```

Du kan använda `.` som ersätter valfritt tecken (förutom ny rad = `\n`).

```
str_view(x, ".a.")
```

```
apple
banana
pear
```

Men hur matchas *punkt* explicit? Du behöver använda 'escape-tecknet' (backslash) för att matcha exakt och inte använda tecknets speciella funktion, i detta fall `\.` . Alltså:

```
# För att skapa en regex behövs \\
dot <- "\\."

# ...men själva uttrycket innehåller bara en backslash:
writeLines(dot)

## \.

# ...vilket talar om för R att titta efter en explicit punkt (.)
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

```
abc
a.c
bef
```

I Wickhams bok finns några övningar med regelbundna uttryck (avsnitt 14.3.1.1 (<http://r4ds.had.co.nz/strings.html>)) att tugga på.

Ankare (Anchors)

Som default kommer regexps att matcha vilken del som helst av en textsträng. Det är därför ofta effektivt att förankra uttrycket så att det matchar från början eller slutet av textsträngen. Du kan använda

`^` för att matcha från början av strängen `$` för att matcha från slutet av strängen

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

```
apple
banana
pear
```

```
str_view(x, "a$")
```



```
apple
banana
pear
```

För att tvinga en regexp att matcha enbart en komplett sträng kan du använda båda ankarna:

```
x <- c("apple pie", "apple", "apple cake")
```

```
str_view(x, "apple")
```

```
apple pie
apple
apple cake
```

```
str_view(x, "^apple$")
```

```
apple pie
apple
apple cake
```

Övningar

1. I `stringr` finns en datamängd som kallas `words`. Använd regexps för att hitta alla ord som

- Börjar med "y"
- Slutar med "x"
- Är exakt 3 tecken
- Har 7 eller fler tecken

Tecken-klasser

Det finns ett antal speciella mönster som matchar mer än ett tecken. Ett exempel är `.` som matchar valfritt tecken utom ny rad. Det finns fyra andra användbara verktyg:

`\d` matchar valfri siffra `\s` matchar alla blanksteg (inklusive tabb och ny rad) `[abc]` matchar alla a, b eller c `[^abc]` matchar allt utom a, b eller c

Kom ihåg att använda escape-tecknet (backslash) om du vill inkludera `\d` eller `\s` i en regexp, alltså `\\d` eller `\\s`. Du kan använda `|` för att välja mellan två alternativa mönster. Till exempel kommer `abc|d..f` att matcha antingen "abc" eller "deaf". Notera att prioriteten för `|` är låg så att `abc|xyz` matchar `abc` eller "xyz", inte `abxyz` eller `abxyz`. Använd gärna parenteser för att förtydliga vad du vill matcha:

```
str_view(c("grey", "gray"), "gr(e|a)y")
```

```
grey
gray
```

Övningar

- Skapa regexps för att hitta alla ord som
 - Börjar med vokal
 - Bara innehåller konsonanter
 - Slutar med "ed" men inte "eed"
 - Slutar med "ing" eller "ise"
- Skapa en regexp som matchar telefonnummer i Sverige

Repeterande tecken

Härnäst handlar det om återkommande tecken och hur många gånger de återkommer.

`?` 0 eller 1 gång `+` 1 eller fler gånger `*` 0 eller fler gånger

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, "CC+")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

```
str_view(x, 'C[LX]+')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Notera att prioritet är hög för dessa frekvens-indikatorer, du kan alltså skriva `color` för att matcha både engelskt och amerikanskt språkbruk. För det mesta behöver detta specificeras med parenteser, ex. `bana(na)+`.

Du kan även specificera antalet förekomster exakt:

- `{n}` exakt n
- `{n, }` n eller fler
- `{,m}` max m
- `{n,m}` mellan n och m

```
str_view(x, "C{2}")
```

```
1888 is the longest year in Roman numerals: MDCCCCLXXXVIII
```

```
str_view(x, "C{2,}")
```

```
1888 is the longest year in Roman numerals: MDCCCCLXXXVIII
```

```
str_view(x, "C{2,3}")
```

```
1888 is the longest year in Roman numerals: MDCCCCLXXXVIII
```

Default är att dessa frekvens-matchningar är “giriga”, dvs vill fånga så långa strängar som möjligt. Du kan göra dem mer ignoranta genom att lägga till ? efter dem.

```
str_view(x, 'C{2,3}?')
```

```
1888 is the longest year in Roman numerals: MDCCCCLXXXVIII
```

```
str_view(x, 'C[LX]+?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

I Wickhams bok finns ett avsnitt om att gruppera regexps och “back-referenser” (<http://r4ds.had.co.nz/strings.html#grouping-and-backreferences>) som vi hoppar över här och går vidare till

Verktyg

Här ska vi gå igenom hur man använder regexps på reella problem. Det finns en rad `stringr`-funktioner som låter dig

- Bestämma vilka strängar som matchar ett visst mönster
- Hitta positionen av en matchning
- Extrahera innehållet i en matchning
- Ersätta en matchning med ett annat värde
- Dela upp en sträng baserad på en matchning

Glöm inte att R är ett programmeringsspråk och det finns andra verktyg till förfogande. Ofta kan man bryta ned ett komplext regex till flera enklare och det kan vara värt att fundera på hur man kan dela upp ett problem när man kör fast.

Upptäck matchningar

För att bestämma om en vektor matchar ett mönster använd `str_detect()`. Den returnerar en logisk vektor med samma längd som input.

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
## [1] TRUE FALSE TRUE
```

Kom ihåg att i en numerisk kontext blir `FALSE` lika med 0 och `TRUE` blir lika med 1. Det medför att funktionerna `sum()` och `mean()` blir användbara för att besvara frågor om matchningar i en större vektor:

```
#Hur många ord börjar med t?
sum(str_detect(words, "^t"))
## [1] 65
```

Hur stor andel av orden börjar med en vokal?

```
mean(str_detect(words, "[aeiou]$"))
```

```
## [1] 0.2765306
```

Ett vanligt sätt att använda `str_detect()` är att välja ut de element som matchar ett visst mönster. Du kan göra det genom en logisk *subsetting* eller genom att använda den mer direkta `str_subset()`-funktionen.

```
words[str_detect(words, "x$")]
```

```
## [1] "box" "sex" "six" "tax"
```

```
str_subset(words, "x$")
```

```
## [1] "box" "sex" "six" "tax"
```

I en dataram/tabell använder du istället `filter()`.

```
df <- tibble(
  word = words,
  i = seq_along(word)
)
df %>%
  filter(str_detect(words, "x$"))
```

```
## # A tibble: 4 x 2
```

```
##   word      i
```

```
##   <chr> <int>
```

```
## 1 box   108
```

```
## 2 sex    747
```

```
## 3 six    772
```

```
## 4 tax    841
```

En variant på `str_detect()` är `str_count()`, som räknar antalet matchningar i en vektor:

```
x <- c("apple", "banana", "pear")
```

```
str_count(x, "a")
```

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

Hur många vokaler finns det i genomsnitt i orden?

```
mean(str_count(words, "[aeiou]"))
```

```
## [1] 1.991837
```

Det är smart att använda `str_count()` tillsammans med `mutate()`:

```
df %>%
```

```
  mutate(
```

```
vowels = str_count(word, "[aeiou]"),
consonants = str_count(word, "[^aeiou]")
)
```

```
## # A tibble: 980 x 4
##   word      i vowels consonants
##   <chr>   <int> <int>   <int>
## 1 a       1    1     0
## 2 able    2    2     2
## 3 about   3    3     2
## 4 absolute 4    4     4
## 5 accept  5    2     4
## 6 account  6    3     4
## 7 achieve  7    4     3
## 8 across  8    2     4
## 9 act     9    1     2
## 10 active 10    3     3
## # ... with 970 more rows
```

Notera att matchningar aldrig överlappar. I "abababa", till exempel, hur många gånger förekommer "aba"? Regexp hävdar två gånger, inte tre!

```
str_count("abababa", "aba")
## [1] 2
str_view_all("abababa", "aba")
```

```
abababa
```

Notera även hur `str_view_all()` används. Många `stringr`-funktioner kommer i komplementära par - en funktion som fungerar med en enkel matchning, en annan som fungerar med alla matchningar. Den senare funktionen har suffixet `_all`.

Övningar

1. För var och en av nedanstående problem försök att använda både ett regexp och en kombination av flera `str_detect()`.
 - a. Hitta samtliga ord som börjar eller slutar på x.

- b. Hitta alla ord som börjar på en vokal och slutar med en konsonant
2. Vilket ord har flest vokaler? Vilket ord har störst andel vokaler?

Extrahera matchningar

Använd `str_extract()` för att extrahera matchningar. Vi ska använda *Harvard sentences*

[https://en.wikipedia.org/wiki/Harvard_sentences] De ingår i `stringr::sentences`:

```
length(sentences)
```

```
## [1] 720
```

```
head(sentences)
```

```
## [1] "The birch canoe slid on the smooth planks."
```

```
## [2] "Glue the sheet to the dark blue background."
```

```
## [3] "It's easy to tell the depth of a well."
```

```
## [4] "These days a chicken leg is a rare dish."
```

```
## [5] "Rice is often served in round bowls."
```

```
## [6] "The juice of lemons makes fine punch."
```

Anta att vi vill hitta alla meningar som innehåller en färg. Vi skapar först en vektor av färgnamn och gör en regex av den.

```
colours <- c("red", "orange", "yellow", "green", "blue", "purple")
```

```
colour_match <- str_c(colours, collapse = "|")
```

```
colour_match
```

```
## [1] "red|orange|yellow|green|blue|purple"
```

Nu kan vi välja de meningar som innehåller en färg och sedan extrahera färgerna för att se vilka de var:

```
has_colour <- str_subset(sentences, colour_match)
```

```
matches <- str_extract(has_colour, colour_match)
```

```
head(matches)
```

```
## [1] "blue" "blue" "red" "red" "red" "blue"
```

Notera att `str_extract()` endast extraherar den första matchningen. Det kan vi se om vi väljer ut de meningar som innehåller fler än en färg:

```
more <- sentences[str_count(sentences, colour_match) > 1]
```

```
str_view_all(more, colour_match)
```



```
It is hard to erase blue or red ink.
The green light in the brown box flickered.
The sky in the west is tinged with orange red.
```

```
str_extract(more, colour_match)
```

```
## [1] "blue" "green" "orange"
```

Detta är ett vanligt mönster för stringr-funktioner eftersom enkel matchning genererar enklare datastrukturer. För att få samtliga matchningar används `str_extract_all()` vilken genererar en lista med matchningar:

```
str_extract_all(more, colour_match)
```

```
## [[1]]
## [1] "blue" "red"
##
## [[2]]
## [1] "green" "red"
##
## [[3]]
## [1] "orange" "red"
```

Vi återkommer till *lists* och *iteration*.

Om du använder `simplify = TRUE` kommer `str_extract_all()` att returnera en matris som expanderas till samma längd som den längsta matchningen:

```
str_extract_all(more, colour_match, simplify = TRUE)
```

```
##      [,1] [,2]
## [1,] "blue" "red"
## [2,] "green" "red"
## [3,] "orange" "red"

x <- c("a", "a b", "a b c")
str_extract_all(x, "[a-z]", simplify = TRUE)

##      [,1] [,2] [,3]
## [1,] "a" "" ""
## [2,] "a" "b" ""
## [3,] "a" "b" "c"
```

Gruppereade matchningar

Tidigare talade vi om att använda parenteser för att tydliggöra prioriteringar. Du kan även använda parenteser för att extrahera delar i en komplex matchning. Om vi t.ex. ska extrahera substantiv från meningar kan vi komma en bit på väg genom att välja varje ord som kommer efter "a" eller "the". För att definiera ett "ord" i en regex använder vi en approximering - en sekvens av minst ett tecken som inte är ett mellanslag.

```
noun <- "(a|the) ([^ ]+)"
has_noun <- sentences %>%
  str_subset(noun) %>%
  head(10)
has_noun %>%
  str_extract(noun)

## [1] "the smooth" "the sheet" "the depth" "a chicken" "the parked"
## [6] "the sun" "the huge" "the ball" "the woman" "a helps"
```

`str_extract()` ger oss en komplett matchning, `str_match()` ger oss de enskilda komponenterna i en matchning. Istället för en vektor (*character vector*) returnerar den en matris med en kolumn för den kompletta matchningen och en kolumn för varje komponent:

```
has_noun %>%
  str_match(noun)

##      [,1]      [,2] [,3]
## [1,] "the smooth" "the" "smooth"
## [2,] "the sheet" "the" "sheet"
## [3,] "the depth" "the" "depth"
## [4,] "a chicken" "a" "chicken"
## [5,] "the parked" "the" "parked"
## [6,] "the sun" "the" "sun"
## [7,] "the huge" "the" "huge"
## [8,] "the ball" "the" "ball"
## [9,] "the woman" "the" "woman"
## [10,] "a helps" "a" "helps"
```

Ersätt matchningar

`str_replace()` och `str_replace_all()` låter dig ersätta matchningar med nya strängar. Det enklaste exemplet är att ersätta ett mönster med en definierad sträng:

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")

## [1] "-pple" "p-ar" "b-nana"
```

```
str_replace_all(x, "[aeiou]", "-")
```

```
## [1] "-ppl-" "p--r"  "b-n-n-"
```

Med `str_replace_all()` kan man ersätta flera matchningar på en gång genom att ange en vektor:

```
x <- c("1 house", "2 cars", "3 people")
```

```
str_replace_all(x, c("1" = "one", "2" = "two", "3" = "three"))
```

```
## [1] "one house"  "two cars"   "three people"
```

Istället för att ersätta med en eller flera definierade strängar kan du använda backreferenser för att infoga komponenter. I exemplet byter det andra och tredje ordet plats:

```
sentences %>%
```

```
  str_replace("([^\s]+) ([^\s]+) ([^\s]+)", "\\1 \\3 \\2") %>%
```

```
  head(5)
```

```
## [1] "The canoe birch slid on the smooth planks."
```

```
## [2] "Glue sheet the to the dark blue background."
```

```
## [3] "It's to easy tell the depth of a well."
```

```
## [4] "These a days chicken leg is a rare dish."
```

```
## [5] "Rice often is served in round bowls."
```

Dela upp textsträngar

Använd `str_split()` för att dela upp strängar i delar. Vi kan t.ex. dela upp meningar i ord:

```
sentences %>%
```

```
  head(5) %>%
```

```
  str_split(" ")
```

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

```
## [1] "The"  "birch" "canoe" "slid" "on"   "the"  "smooth"
```

```
## [8] "planks."
```

```
##
```

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

```
## [1] "Glue"  "the"   "sheet" "to"    "the"
```

```
## [6] "dark"  "blue"  "background."
```

```
##
```

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

```
## [1] "It's" "easy" "to"   "tell" "the"  "depth" "of"  "a"  "well."
```

```
##
```

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

```
## [1] "These" "days" "a"     "chicken" "leg"  "is"   "a"
```

```
## [8] "rare"  "dish."
```

```
##
```

```
## [[5]]
## [1] "Rice" "is" "often" "served" "in" "round" "bowls."
```

Eftersom varje mening kan innehålla olika många ord returneras en lista (*a list*). Liksom för de andra `str_`-funktionerna kan man använda argumentet `simplify = TRUE` för att returnera en matris:

```
sentences %>%
  head(5) %>%
  str_split(" ", simplify = TRUE)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,] "The" "birch" "canoe" "slid" "on" "the" "smooth"
## [2,] "Glue" "the" "sheet" "to" "the" "dark" "blue"
## [3,] "It's" "easy" "to" "tell" "the" "depth" "of"
## [4,] "These" "days" "a" "chicken" "leg" "is" "a"
## [5,] "Rice" "is" "often" "served" "in" "round" "bowls."
##      [,8]      [,9]
## [1,] "planks." ""
## [2,] "background." ""
## [3,] "a" "well."
## [4,] "rare" "dish."
## [5,] "" ""
```

Istället för att dela upp strängar enligt ett mönster kan man även dela upp efter tecken, rad, mening och ord genom att använda argumentet `boundary()`:

```
x <- "This is a sentence. This is another sentence."
str_view_all(x, boundary("word"))
```

```
This is a sentence. This is another sentence.
```

```
str_split(x, " ")[[1]]

## [1] "This" "is" "a" "sentence." "" "This"
## [7] "is" "another" "sentence."

str_split(x, boundary("word"))[[1]]
```

```
## [1] "This"  "is"    "a"     "sentence" "This"  "is"  
## [7] "another" "sentence"
```

Hitta matchningar

Med `str_locate()` och `str_locate_all()` får man den första och sista positionen för matchningen. Detta är speciellt användbart då ingen av de andra funktionerna gör exakt det du vill. Du kan använda `str_locate()` för att lokalisera matchningen och sedan `str_sub()` för att extrahera och/eller modifiera dem.

Andra typer av mönster

När du använder ett mönster som är en sträng används automatiskt funktionen `regex()`:

```
# Uttrycket  
str_view(fruit, "nana")
```

apple
apricot
avocado
banana
bell pepper
bilberry
blackberry
blackcurrant
blood orange
blueberry
boysenberry
breadfruit
canary melon
cantaloupe
cherimoya
cherry
chili pepper
clementine
cloudberry
coconut
cranberry
cucumber
currant
damson
date
dragonfruit
durian
eggplant
elderberry
feijoa
fig
goji berry
gooseberry
grape
grapefruit
guava
honeydew
huckleberry
jackfruit
jambul
jujube
kiwi fruit
kumquat
lemon
lime
loquat
lychee
mandarine
mango
mulberry
nectarine
nut
olive
orange
pamelo
papaya
passionfruit
peach
pear
persimmon
physalis
pineapple
plum
pomegranate
pomelo
purple mangosteen
quince
raisin
rambutan
raspberry
redcurrant
rock melon
salal berry
satsuma

```
# ...är en "genväg" för  
str_view(fruit, regex("nana"))
```

apple
apricot
avocado
banana
bell pepper
bilberry
blackberry
blackcurrant
blood orange
blueberry
boysenberry
breadfruit
canary melon
cantaloupe
cherimoya
cherry
chili pepper
clementine
cloudberry
coconut
cranberry
cucumber
currant
damson
date
dragonfruit
durian
eggplant
elderberry
feijoa
fig
goji berry
gooseberry
grape
grapefruit
guava
honeydew
huckleberry
jackfruit
jambul
jujube
kiwi fruit
kumquat
lemon
lime
loquat
lychee
mandarine
mango
mulberry
nectarine
nut
olive
orange
pamelo
papaya
passionfruit
peach
pear
persimmon
physalis
pineapple
plum
pomegranate
pomelo
purple mangosteen
quince
raisin
rambutan
raspberry
redcurrant
rock melon
salal berry
satsuma

Du kan använda andra argument till `regex()` för att kontrollera detaljer i matchningen:

`ignore_case = TRUE` tillåter matchning på versaler eller gemener.

```
bananas <- c("banana", "Banana", "BANANA")
str_view(bananas, "banana")
```

```
banana
Banana
BANANA
```

```
str_view(bananas, regex("banana", ignore_case = TRUE))
```

```
banana
Banana
BANANA
```

- `multiline = TRUE` tillåter `^` och `$` att matcha från början resp slut på varje rad snarare än på hela strängen:

```
x <- "Line 1\nLine 2\nLine 3"
str_extract_all(x, "^Line")[[1]]
## [1] "Line"

str_extract_all(x, regex("^Line", multiline = TRUE))[[1]]
## [1] "Line" "Line" "Line"
```

- `comments = TRUE` tillåter dig att använda kommentarer och mellansteg för att göra regexp tydligare och begripligare. Mellansteg ignoreras liksom allt efter `#`. För att matcha ett bokstavligt mellansteg behöver du använda en escape `"\"`.

```
phone <- regex("
  \\(?  # optional opening parens
  (\\d{3}) # area code
  [-]?  # optional closing parens, space, or dash
  (\\d{3}) # another three numbers
")
```

```
[ -]? # optional space or dash
(\\d{3}) # three more numbers
", comments = TRUE)
str_match("514-791-8141", phone)

##      [,1]      [,2] [,3] [,4]
## [1,] "514-791-814" "514" "791" "814"
```

Det finns några ytterligare funktioner för att matcha andra typer av mönster i Wickham's bok som vi lämnar därhän för nu.

Andra typer av regexps

Det finns två andra funktioner som kan vara väldigt användbara i base R och som använder regexps.

`apropos()` söker igenom alla tillgängliga objekt från den globala miljön. Det är användbart om du inte kommer ihåg namnet på en funktion.

```
apropos("replace")

## [1] "%+replace%"      ".rs.registerReplaceHook"
## [3] ".rs.replaceBinding" "replace"
## [5] "replace_na"       "setReplaceMethod"
## [7] "str_replace"      "str_replace_all"
## [9] "str_replace_na"   "theme_replace"
```

`dir()` listar alla filer i en mapp. Argumentet `pattern` använder en regexp och returnerar filnamn som matchar denna regexp. Du kan t.ex. hitta samtliga R-datafiler i mappen genom:

```
head(dir(pattern = "\\R\\.Rmd$"))

## [1] "_07_workflow_projects.Rmd" "_08_wrangle.Rmd"
## [3] "_09_tibbles.Rmd"         "_10_data_import.Rmd"
## [5] "_11_tidy_data.Rmd"       "_12_relational_data.Rmd"
```

stringi

Några ord om modulen `stringi`: vi har använt `stringr` som innehåller ett *minimum* av funktioner (46 st) vilka täcker det mesta för att kunna hantera sträng-manipulationer. `stringi` innehåller betydligt fler (234 st!) och det kan därför vara värt att kika igenom denna modul om man kör fast i `stringr`. Syntaxen är i princip densamma som i `stringr` men istället för `str_*` skriver man `stri_*`.

Kategoriska variabler (Factors)

Introduktion

Kategoriska variabler brukar i R hanteras som *factors*, vilket betecknar variabler som har en fixerad och känd uppsättning av möjliga värden. Factors kan också användas för att visa text-vektorer i en icke-alfabetisk ordning.

För att arbeta med factors ska vi använda modulen `forcats` (notera anagrammet):

```
library(tidyverse)
library(forcats)
```

Wickham rekommenderar Amelia McNamara and Nicholas Horton's artikel, *Wrangling categorical data in R* (<https://peerj.com/preprints/3163/>) för den som vill veta mer om factors.

Skapa factors

Antag att vi har en variabel som betecknar månader:

```
x1 <- c("Dec", "Apr", "Jan", "Mar")
```

Att använda en textsträng som denna har två problem:

1. Det finns endast 12 möjliga månader och det finns inget som förhindrar typos:

```
x2 <- c("Dec", "Apr", "Jam", "Mar")
```

2. De går inte att sortera på något meningsfullt sätt:

```
sort(x1)
```

```
## [1] "Apr" "Dec" "Jan" "Mar"
```

Man kan lösa båda dessa problem med hjälp av en *factor*. Vi börjar med att skapa en lista på de möjliga utfallen (*levels*):

```
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
```

Och:

```
y1 <- factor(x1, levels = month_levels)
y1
```

```
## [1] Dec Apr Jan Mar
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

```
sort(y1)
```

```
## [1] Jan Mar Apr Dec
## Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Notera att varje värde/utfall som inte finns representerat i variabeln betecknas som missing (NA).

Om du vill få en varning om något går fel använder du `parse_factor()`:

```
y2 <- parse_factor(x2, levels = month_levels)
```

```
## Warning: 1 parsing failure.
```

```
## row # A tibble: 1 x 4 col   row   col expected      actual expected <int> <int> <chr>      <chr> ac
tial 1     3   NA value in level set Jam
```

Om man struntar i att definiera levels kommer `factor()` att använda befintliga data i alfabetisk ordning:

```
factor(x1)
```

```
## [1] Dec Apr Jan Mar
## Levels: Apr Dec Jan Mar
```

Om du vill att levels ska matcha ordningen i vilken data uppträder kan du använda `unique()` för att definiera *levels* eller med `fct_inorder()`:

```
f1 <- factor(x1, levels = unique(x1))
```

```
f1
```

```
## [1] Dec Apr Jan Mar
## Levels: Dec Apr Jan Mar
```

```
f2 <- x1 %>% factor() %>% fct_inorder()
```

```
f2
```

```
## [1] Dec Apr Jan Mar
## Levels: Dec Apr Jan Mar
```

Om du behöver komma åt uppsättningen giltiga levels använder du `levels()`:

```
levels(f2)
```

```
## [1] "Dec" "Apr" "Jan" "Mar"
```

General Social Survey

I resten av detta avsnitt ska vi använda `forcats::gss_cat` som är ett urval av data från General Social Survey, en amerikansk survey som sedan lång tid genomförs av en oberoende forskningsorganisation via universitetet i Chicago. Undersökningen innehåller tusentals frågor och i `gss_cat` finns en handfull av dessa, speciellt utvalda för att de illustrerar några utmaningar när man arbetar med factors.

```
gss_cat
```

```
## # A tibble: 21,483 x 9
##   year marital   age race rincome partyid relig denom tvhours
##   <int> <fct>   <int> <fct> <fct>   <fct>   <fct>   <fct>   <int>
## 1 2000 Never ma~ 26 White $8000 to~ Ind,near ~ Protes~ Southe~ 12
## 2 2000 Divorced 48 White $8000 to~ Not str r~ Protes~ Baptis~ NA
## 3 2000 Widowed 67 White Not appl~ Independe~ Protes~ No den~ 2
## 4 2000 Never ma~ 39 White Not appl~ Ind,near ~ Orthod~ Not ap~ 4
## 5 2000 Divorced 25 White Not appl~ Not str d~ None Not ap~ 1
## 6 2000 Married 25 White $20000 -~ Strong de~ Protes~ Southe~ NA
## 7 2000 Never ma~ 36 White $25000 o~ Not str r~ Christ~ Not ap~ 3
## 8 2000 Divorced 44 White $7000 to~ Ind,near ~ Protes~ Luther~ NA
## 9 2000 Married 44 White $25000 o~ Not str d~ Protes~ Other 0
## 10 2000 Married 47 White $25000 o~ Strong re~ Protes~ Southe~ 3
## # ... with 21,473 more rows
```

Du får som vanligt mer information om innehållet i data genom `?gss_cat`.

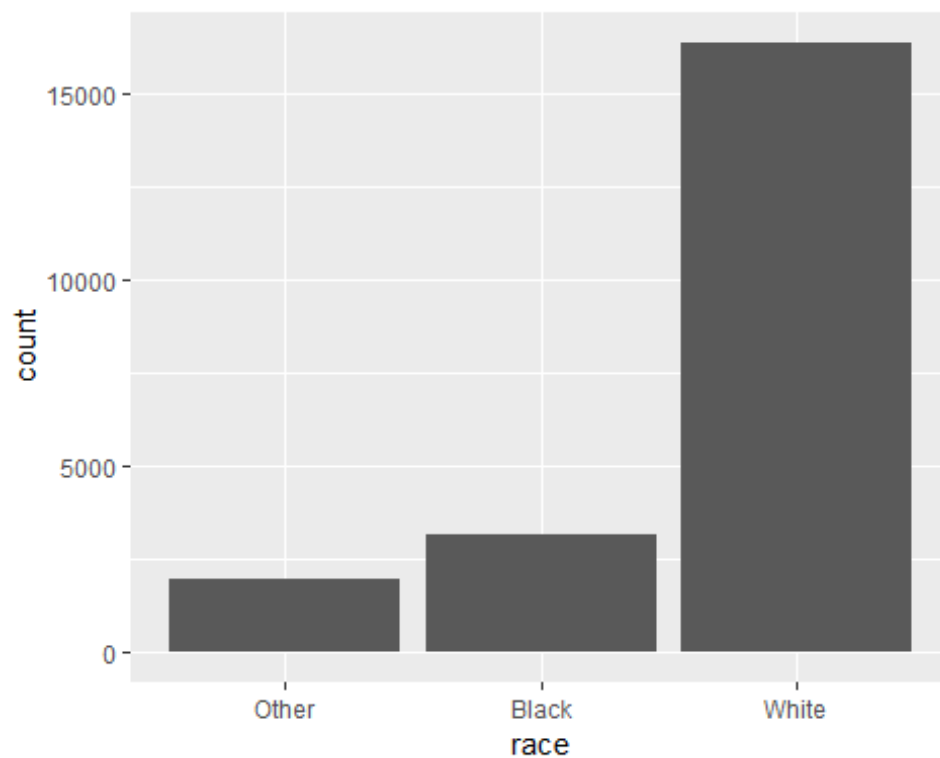
I en tibble går det inte att enkelt se levels. Ett sätt att göra det är att använda `count()`:

```
gss_cat %>%
  count(race)

## # A tibble: 3 x 2
##   race    n
##   <fct> <int>
## 1 Other 1959
## 2 Black 3129
## 3 White 16395
```

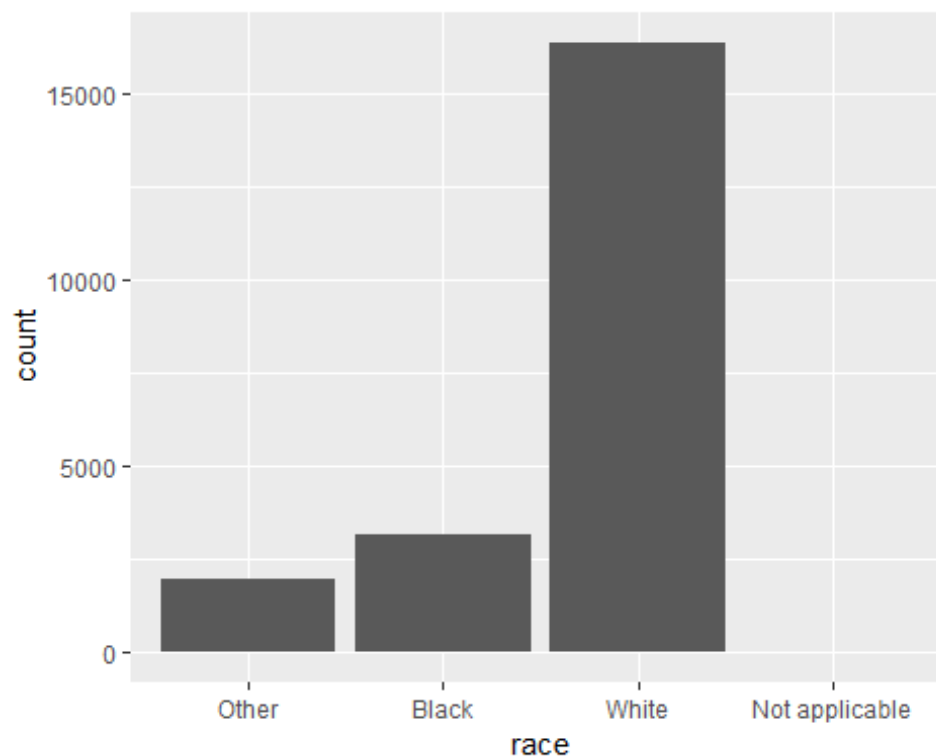
Eller med ett stapeldiagram:

```
ggplot(gss_cat, aes(race)) +
  geom_bar()
```



Man kan tvinga `ggplot2` att visa även levels som inte innehåller data genom argumentet `drop = FALSE`:

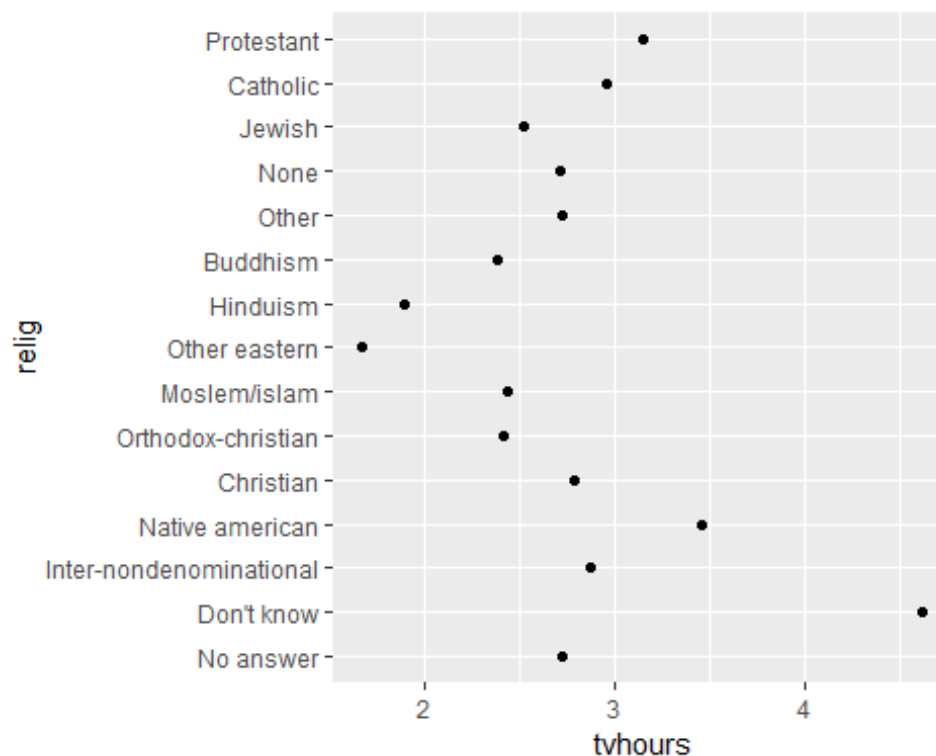
```
ggplot(gss_cat, aes(race)) +  
  geom_bar() +  
  scale_x_discrete(drop = FALSE)
```



Modifiera ordningen av factors

Det är ofta önskvärt att ändra ordningen av levels i en visualisering. Till exempel för att undersöka det genomsnittliga antalet timmar som spenderas framför TV över religiös tillhörighet:

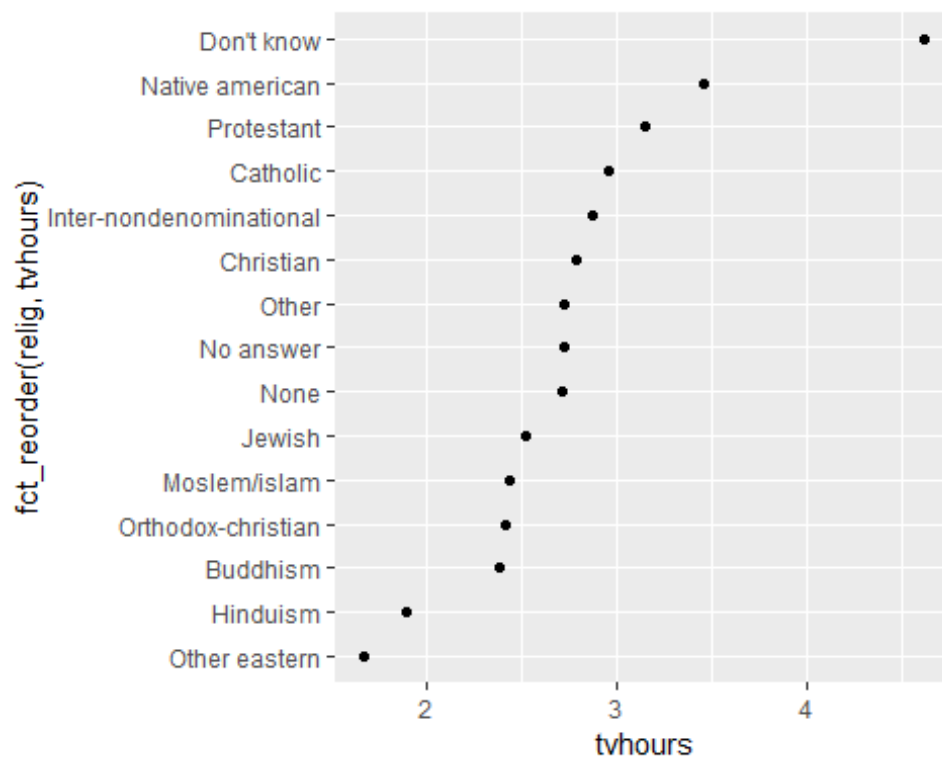
```
relig_summary <- gss_cat %>%
  group_by(relig) %>%
  summarise(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )
ggplot(relig_summary, aes(tvhours, relig)) + geom_point()
```



Det kan vara svårt att tolka denna *dotplot*. Vi kan förbättra grafen genom att ändra ordningen på levels i variabeln `relig` genom att använda `fct_reorder()`. Funktionen tar tre argument:

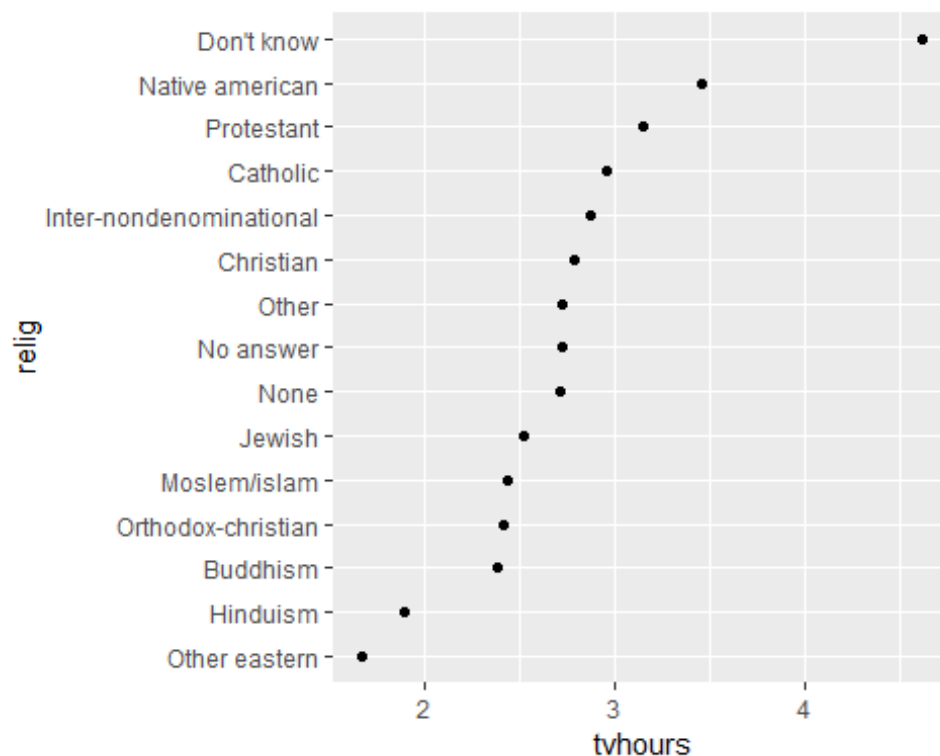
1. `f`, factor:n som man vill ändra ordningen på
2. `x`, en numerisk vektor man vill använda för att ändra ordningen
3. `fun`, (optional) en funktion som används då det finns flera värden av `x` för varje värde av `f`. Default är median.

```
ggplot(relig_summary, aes(tvhours, fct_reorder(relig, tvhours))) +  
  geom_point()
```

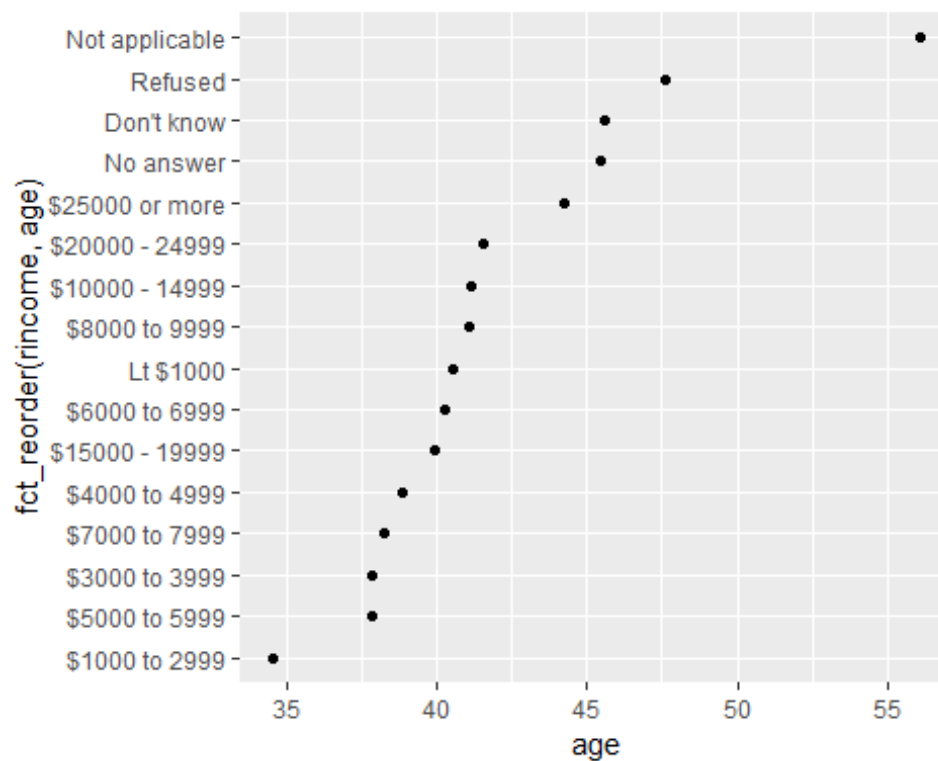
Då du börjar göra mer komplexa transformationer rekommenderar Wickham att flytta ut dem från `aes()` och till ett särskilt `mutate()` steg. T.ex. kan man skriva om grafen ovan som:

```
relig_summary %>%
  mutate(relig = fct_reorder(relig, tvhours)) %>%
  ggplot(aes(tvhours, relig)) +
  geom_point()
```



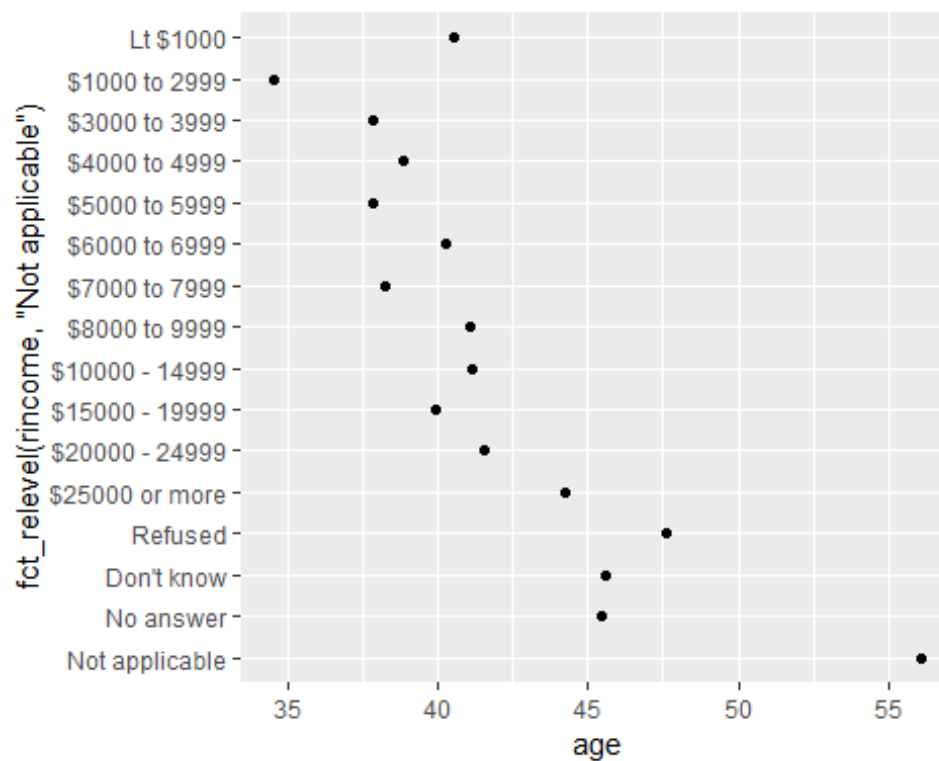
Om vi skapar en liknande graf för hur genomsnittlig ålder varierar med inkomstnivå:

```
rincome_summary <- gss_cat %>%
  group_by(rincome) %>%
  summarise(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )
ggplot(rincome_summary, aes(age, fct_reorder(rincome, age))) + geom_point()
```



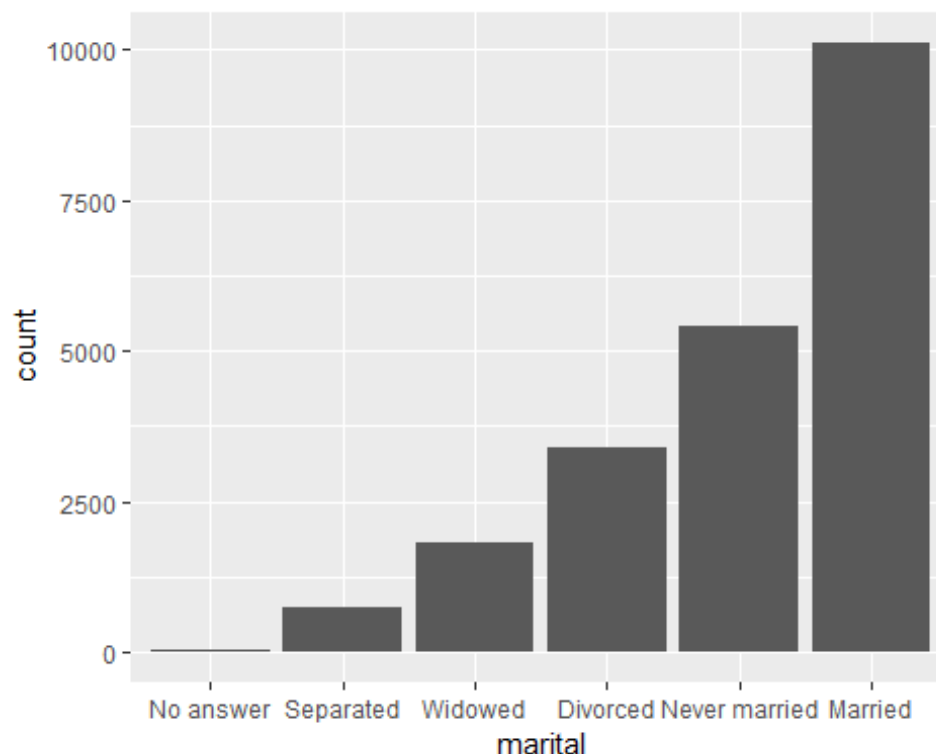
Här är en godtycklig ändring av levels ordning inte någon bra idé eftersom `rincome` redan har en implicit ordning som vi inte bör rådda med. Använd bara `fct_reorder()` för factors som är godtyckligt ordnade. Men det verkar klokt att ändå ha kategorin "Not applicable" vid sidan av övriga levels. För det kan man använda `fct_relevel()`. Funktionen tar en factor `f` och sedan godtyckligt antal levels som man vill ha "vid sidan av":

```
ggplot(rincome_summary, aes(age, fct_relevel(rincome, "Not applicable")) +  
  geom_point())
```



För stapeldiagram kan man använda `fct_infreq()` för att ordna levels efter ökande frekvens. Man kan kombinera med `fct_rev()`.

```
gss_cat %>%
  mutate(marital = marital %>% fct_infreq() %>% fct_rev()) %>%
  ggplot(aes(marital)) +
  geom_bar()
```



Modifiera factor levels

Inte sällan vill man ändra värdet på levels, t.ex. för att göra grafer tydligare eller i publikationer. Det kan man göra med `fct_recode()`. Till exempel, ta variabeln `gss_cat$partyid`:

```
gss_cat %>% count(partyid)

## # A tibble: 10 x 2
##   partyid      n
##   <fct>      <int>
## 1 No answer    154
## 2 Don't know     1
## 3 Other party   393
## 4 Strong republican 2314
## 5 Not str republican 3032
## 6 Ind,near rep   1791
## 7 Independent   4119
## 8 Ind,near dem   2499
## 9 Not str democrat 3690
## 10 Strong democrat 3490
```

Dessa levels är komprimerade och inkonsistenta. Låt oss göra om dem:

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong" = "Strong republican",
    "Republican, weak"   = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"      = "Not str democrat",
    "Democrat, strong"    = "Strong democrat"
  )) %>%
  count(partyid)

## # A tibble: 10 x 2
##   partyid      n
##   <fct>      <int>
## 1 No answer    154
## 2 Don't know     1
## 3 Other party   393
## 4 Republican, strong 2314
## 5 Republican, weak  3032
## 6 Independent, near rep 1791
## 7 Independent     4119
## 8 Independent, near dem 2499
## 9 Democrat, weak   3690
## 10 Democrat, strong 3490
```

`fct_recode()` ignorerar levels som inte nämns explicit och varnar om man refererar till en level som inte existerar.

För att kombinera grupper kan man använda fler gamla levels till samma nya level:

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong" = "Strong republican",
    "Republican, weak"   = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"      = "Not str democrat",
    "Democrat, strong"    = "Strong democrat",
    "Other"               = "No answer",
    "Other"               = "Don't know",
    "Other"               = "Other party"
  )) %>%
  count(partyid)
```

```
## # A tibble: 8 x 2
##   partyid      n
##   <fct>      <int>
## 1 Other      548
## 2 Republican, strong  2314
## 3 Republican, weak   3032
## 4 Independent, near rep 1791
## 5 Independent      4119
## 6 Independent, near dem 2499
## 7 Democrat, weak     3690
## 8 Democrat, strong   3490
```

Om man vill slå ihop flera levels används `fct_collapse()` vilken är en användbar variant av `fct_recode()`. För varje ny level kan man koppla en vektor av gamla levels:

```
gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
    dem = c("Not str democrat", "Strong democrat")
  )) %>%
  count(partyid)

## # A tibble: 4 x 2
##   partyid      n
##   <fct>      <int>
## 1 other      548
## 2 rep      5346
## 3 ind      8409
## 4 dem      7180
```

Man kan även slå samman levels med de minsta frekvenserna till en "restpost". Då använder man `fct_lump()` och anger parametern `n` för att ange antalet grupper man vill redovisa. Funktionen `fct_lump()` slår då samman de minsta grupperna till en restpost så att det slutliga antalet grupper blir `n`:

```
gss_cat %>%
  mutate(relig = fct_lump(relig, n = 10)) %>%
  count(relig, sort = TRUE) %>%
  print(n = Inf)

## # A tibble: 10 x 2
##   relig      n
##   <fct>      <int>
## 1 Protestant 10846
```

## 2 Catholic	5124
## 3 None	3523
## 4 Christian	689
## 5 Other	458
## 6 Jewish	388
## 7 Buddhism	147
## 8 Inter-nondenominational	109
## 9 Moslem/islam	104
## 10 Orthodox-christian	95

Datum och tidsformat

Introduktion

Detta avsnitt handlar om att hantera datum- och tidsformat. Det är en översiktlig genomgång för att få de nödvändigaste verktygen. Det finns betydligt mer detaljer i Wickhams bok (<http://r4ds.had.co.nz/dates-and-times.html>) vilken behandlar mycket av den komplexitet som ryms i datum- och tidsvariabler särskilt i en global kontext.

Vi ska arbeta med modulen `lubridate` som innehåller verktyg för att hantera datum och tid. Data till exempen finns i `nycflights13`. Vi laddar in

```
library(tidyverse)
library(lubridate)
library(nycflights13)
```

Skapa datum och tid

Det finns tre typer av datum/tid-data som refererar till en specifik tidpunkt:

- Ett *datum*. Tibbles betecknar dessa som `.`
- Ett *klockslag*. Tibbles betecknar dessa som `.`
- En *datumtid* (date-time) är ett klockslag plus ett datum. Det identifierar en tidpunkt (preciserad vanligen till närmaste sekund). Tibbles betecknar dessa som `.`. I *base R* kallas dessa för POSIXct, men Wickham har valt för större klarhet.

Om du kan använda ett datum istället för datumtid bör du göra det eftersom datumtid är betydligt mer komplicerat (behöver hantera tidszoner).

För att få aktuellt datum eller datumtid kan du använda `today()` resp `now()`:

```
today()
## [1] "2019-03-23"

now()
## [1] "2019-03-23 10:56:48 CET"
```

Det finns tre andra sätt att skapa datum/tid-data:

- Från en textsträng
- Från enskilda datum/tid-komponenter
- Från ett existerande datum/tid-objekt

Från textsträngar

Ett sätt att hantera datumformat från textsträngar har vi sett tidigare (date-times). Ett annat sätt är att använda verktygen i lubridate. De tolkar automatiskt formatet när du väl definierat hur år (y), månad (m) och dag (d) är ordnade. Den ordningen anger även namnet på funktionen i lubridate som används för att generera datum-objektet:

```
ymd("2017-01-31")
## [1] "2017-01-31"

mdy("January 31st, 2017")
## [1] "2017-01-31"

dmy("31-Jan-2017")
## [1] "2017-01-31"
```

Dessa funktioner accepterar även siffror utan citationstecken:

```
ymd(20170131)
## [1] "2017-01-31"
```

För att skapa en datumtid kan du lägga till en underscore och ett eller fler av "h", "m", and "s" för att få namnet från till rätt funktion:

```
ymd_hms("2017-01-31 20:11:59")
## [1] "2017-01-31 20:11:59 UTC"

mdy_hm("01/31/2017 08:01")
## [1] "2017-01-31 08:01:00 UTC"
```

Från enskilda komponenter

Istället för en textsträng kan du ha datum och tid som enskilda komponenter spridda över flera kolumner, som i nycflights13:

```
flights %>%
  select(year, month, day, hour, minute)

## # A tibble: 336,776 x 5
##   year month   day hour minute
##   <int> <int> <int> <dbl> <dbl>
## 1 2013     1     1     5     15
## 2 2013     1     1     5     29
## 3 2013     1     1     5     40
```

```
## 4 2013 1 1 5 45
## 5 2013 1 1 6 0
## 6 2013 1 1 5 58
## 7 2013 1 1 6 0
## 8 2013 1 1 6 0
## 9 2013 1 1 6 0
## 10 2013 1 1 6 0
## # ... with 336,766 more rows
```

För att skapa datum/tid från sådan input kan du använda `make_date()` för datum eller `make_datetime()` för datumtid:

```
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(departure = make_datetime(year, month, day, hour, minute))

## # A tibble: 336,776 x 6
##   year month  day hour minute departure
##   <int> <int> <int> <dbl> <dbl> <dtm>
## 1 2013     1     1     5    15 2013-01-01 05:15:00
## 2 2013     1     1     5    29 2013-01-01 05:29:00
## 3 2013     1     1     5    40 2013-01-01 05:40:00
## 4 2013     1     1     5    45 2013-01-01 05:45:00
## 5 2013     1     1     6     0 2013-01-01 06:00:00
## 6 2013     1     1     5    58 2013-01-01 05:58:00
## 7 2013     1     1     6     0 2013-01-01 06:00:00
## 8 2013     1     1     6     0 2013-01-01 06:00:00
## 9 2013     1     1     6     0 2013-01-01 06:00:00
## 10 2013     1     1     6     0 2013-01-01 06:00:00
## # ... with 336,766 more rows
```

Vi gör samma sak för var och ett av de fyra kolumnerna i `flights`. (Eftersom tidsdata i dessa kolumner är lite udda använder vi modulatoräkning för att få ut timmmar och minuter):

```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(year, month, day, sched_dep_time),
    sched_arr_time = make_datetime_100(year, month, day, sched_arr_time)
```

```

) %>%
select(origin, dest, ends_with("delay"), ends_with("time"))
flights_dt

## # A tibble: 328,063 x 9
##   origin dest dep_delay arr_delay dep_time      sched_dep_time
##   <chr> <chr>   <dbl>   <dbl> <dtm>         <dtm>
## 1 EWR   IAH     2     11 2013-01-01 05:17:00 2013-01-01 05:15:00
## 2 LGA   IAH     4     20 2013-01-01 05:33:00 2013-01-01 05:29:00
## 3 JFK   MIA     2     33 2013-01-01 05:42:00 2013-01-01 05:40:00
## 4 JFK   BQN    -1    -18 2013-01-01 05:44:00 2013-01-01 05:45:00
## 5 LGA   ATL    -6    -25 2013-01-01 05:54:00 2013-01-01 06:00:00
## 6 EWR   ORD    -4     12 2013-01-01 05:54:00 2013-01-01 05:58:00
## 7 EWR   FLL    -5     19 2013-01-01 05:55:00 2013-01-01 06:00:00
## 8 LGA   IAD    -3    -14 2013-01-01 05:57:00 2013-01-01 06:00:00
## 9 JFK   MCO    -3     -8 2013-01-01 05:57:00 2013-01-01 06:00:00
## 10 LGA  ORD    -2      8 2013-01-01 05:58:00 2013-01-01 06:00:00
## # ... with 328,053 more rows, and 3 more variables: arr_time <dtm>,
## #   sched_arr_time <dtm>, air_time <dbl>

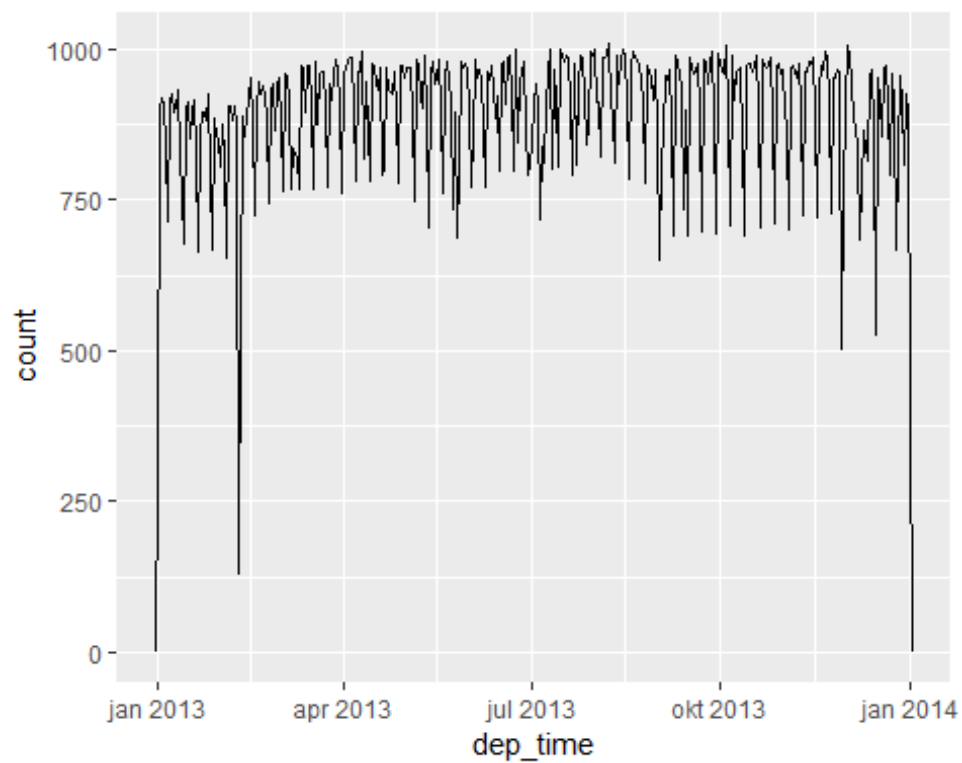
```

Med dessa data kan vi visualisera fördelningen av avgångstider över året:

```

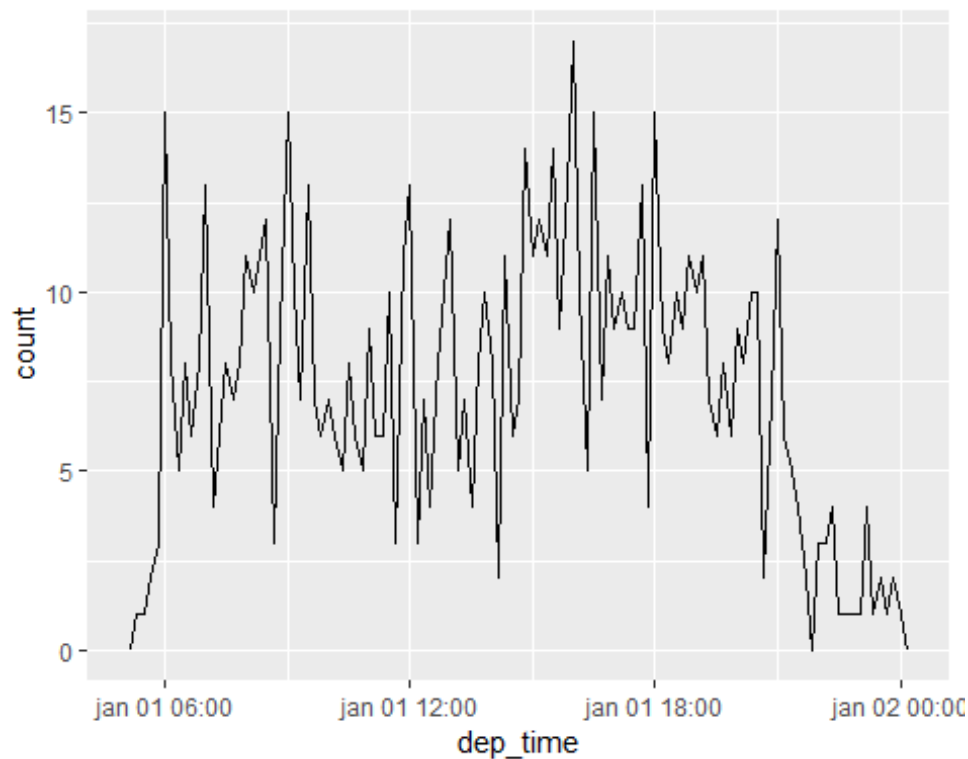
flights_dt %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 86400) # 86400 seconds = 1 day

```



Eller över en enskild dag:

```
flights_dt %>%
  filter(dep_time < ymd(20130102)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 600) # 600 s = 10 minutes
```



Notera att när man använder datumtid-formatet i en numerisk kontext anger siffran 1 en (1) sekund (därav `binwidth = 86400` således 1 dygn). För datum-format gäller 1 för 1 dygn.

Från andra datum/tid-format

Om du vill omvandla datumtid och datum används `as_datetime()` resp `as_date()`:

```
as_datetime(today())
```

```
## [1] "2019-03-23 UTC"
```

```
as_date(now())
```

```
## [1] "2019-03-23"
```

Ibland får man datum/tidsdata som numeriska offsets från ett visst datum, t.ex. "Unix Epoch", 1970-01-01. Om offset är i sekunder använd `as_datetime()`, om det är i dagar använd `as_date()`:

```
as_datetime(60 * 60 * 10)
```

```
## [1] "1970-01-01 10:00:00 UTC"
```

```
as_date(365 * 10 + 2)
```

```
## [1] "1980-01-01"
```

Övningar

1. Vad händer om du försöker processa en textsträng med ogiltiga datum?

```
ymd(c("2010-10-10", "bananas"))
```

```
## Warning: 1 failed to parse.
```

```
## [1] "2010-10-10" NA
```

2. Vad gör argumentet `tzzone` till funktionen `today()`? Varför är det viktigt?
3. Använd en lämplig funktion för att omvandla följande datum-format:

```
d1 <- "January 1, 2010"
```

```
d2 <- "2015-Mar-07"
```

```
d3 <- "06-Jun-2017"
```

```
d4 <- c("August 19 (2015)", "July 1 (2015)")
```

```
d5 <- "12/30/14" # Dec 30, 2014
```

Datum/tid-komponenter

Vad kan man göra med datum/tid-variabler rent praktiskt? Den här sektionen beskriver hur du extraherar och definierar enskilda komponenter och i nästa går vi igenom hur man utför aritmetiska operationer med dem.

Extrahera komponenter

Du kan extrahera enskilda delar i ett datum/tid-objekt med access-funktionerna `year()`, `month()`, `mday()` (dag i månad), `yday()` (dag i året), `wday()` (veckodag), `hour()`, `minute()` och `second()`.

```
datetime <- ymd_hms("2016-07-08 12:34:56")
```

```
year(datetime)
```

```
## [1] 2016
```

```
month(datetime)
```

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

```
mday(datetime)
```

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

```
yday(datetime)
```

```
## [1] 190
```

```
wday(datetime)
```

```
## [1] 6
```

För `month()` och `wday()` kan du sätta argumentet `label = TRUE` för att returnera förkortat månadsnamn resp veckodag. Ange `abbr = FALSE` för att returnera hela namnet.

```
month(datetime, label = TRUE)
```

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

```
## 12 Levels: jan < feb < mar < apr < maj < jun < jul < aug < sep < ... < dec
```

```
wday(datetime, label = TRUE, abbr = FALSE)
```

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

```
## 7 Levels: söndag < måndag < tisdag < onsdag < torsdag < ... < lördag
```

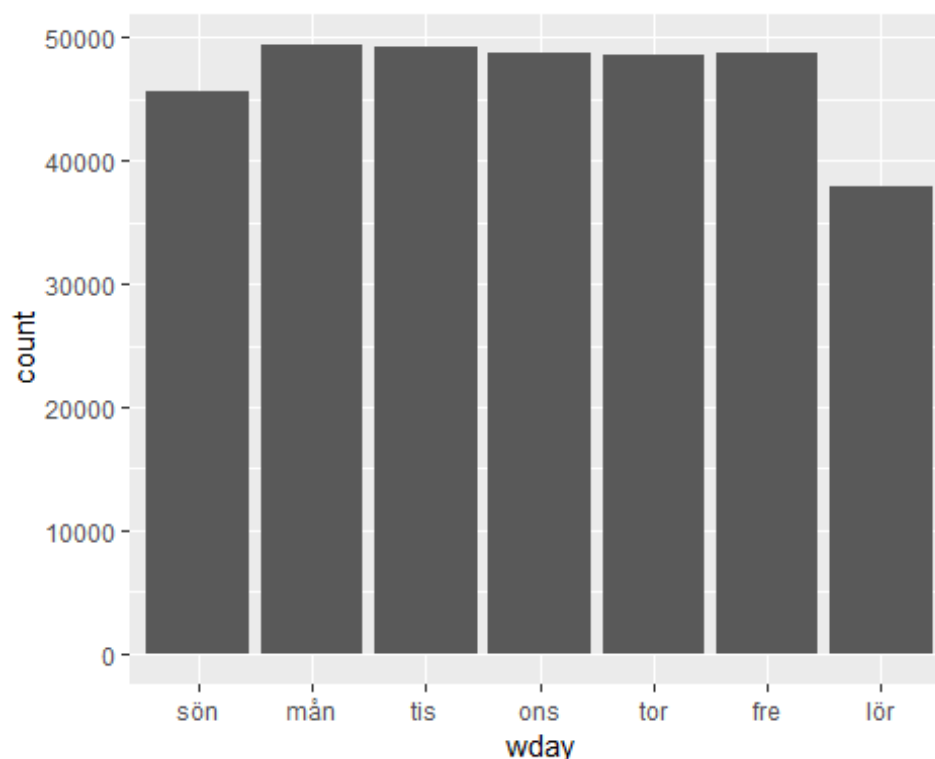
Vi kan använda `wday()` för att se att fler flighter avgår under vardagar än under helger:

```
flights_dt %>%
```

```
  mutate(wday = wday(dep_time, label = TRUE)) %>%
```

```
  ggplot(aes(x = wday)) +
```

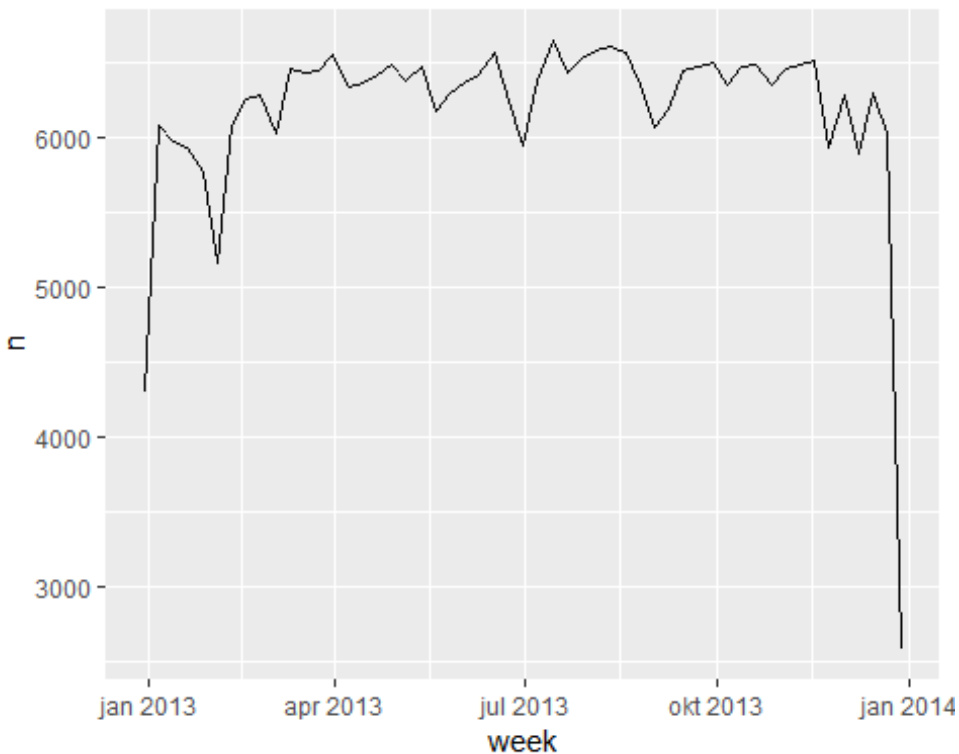
```
  geom_bar()
```



Avrundning

Ett alternativt sätt att plotta enskilda komponenter är att avrunda datum till en närliggande tidpunkt. Det kan man göra via `floor_date()`, `round_date()` och `ceiling_date()`. Varje funktion tar två argument: en datum-vektor som ska avrundas och ett till vad värdena ska avrundas. T.ex. kan vi med hjälp av `floor_date()` plotta antalet flighter per vecka:


```
flights_dt %>%
  count(week = floor_date(dep_time, "week")) %>%
  ggplot(aes(week, n)) +
  geom_line()
```



Ange komponenter

Du kan använda varje accessfunktion för att definiera komponenterna i ett datum/tid-format:

```
(datetime <- ymd_hms("2016-07-08 12:34:56"))
```

```
## [1] "2016-07-08 12:34:56 UTC"
```

```
year(datetime) <- 2020
```

```
datetime
```

```
## [1] "2020-07-08 12:34:56 UTC"
```

```
month(datetime) <- 01
```

```
datetime
```

```
## [1] "2020-01-08 12:34:56 UTC"
```

```
hour(datetime) <- hour(datetime) + 1
```

```
datetime
```

```
## [1] "2020-01-08 13:34:56 UTC"
```

Alternativt kan du skapa ett nytt datum/tid-objekt med hjälp av `update()`. Då kan du ange flera komponenter på samma gång:

```
ymd("2015-02-01") %>%
  update(mday = 30)

## [1] "2015-03-02"

ymd("2015-02-01") %>%
  update(hour = 400)

## [1] "2015-02-17 16:00:00 UTC"
```

Tidsintervall

Hur fungerar aritmetiska operationer med datum/tid-format? Vi går igenom tre viktiga klasser som representerar tidsintervall:

- **varaktighet** (durations), som representerar ett exakt antal sekunder
- **Perioder** (periods), representerar ofta använda tids-enheter, t.ex. veckor och dagar
- **Intervall** (intervals), representerar en start- och slutpunkt

Varaktighet

När man subtraherar två datum i R erhåller man ett *difftime-objekt*.

```
## Hur länge har VGR funnits?
h_age <- today() - ymd(19990101)
h_age

## Time difference of 7386 days
```

Ett *difftime-objekt* innehåller ett tidsspann i sekunder, minuter, timmar dagar eller veckor. Denna tvetydighet kan vara irriterande så `lubridate` erbjuder ett alternativ som alltid använder sekunder (`duration`):

```
as.duration(h_age)

## [1] "638150400s (~20.22 years)"
```

Duration har ett antal smidiga funktioner:

```
dseconds(15)

## [1] "15s"

dminutes(10)

## [1] "600s (~10 minutes)"
```

```
dhours(c(12, 24))
```

```
## [1] "43200s (~12 hours)" "86400s (~1 days)"
```

```
ddays(0:5)
```

```
## [1] "0s" "86400s (~1 days)" "172800s (~2 days)"
```

```
## [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
```

```
dweeks(3)
```

```
## [1] "1814400s (~3 weeks)"
```

```
dyears(1)
```

```
## [1] "31536000s (~52.14 weeks)"
```

Högre tidsenheter än sekunder skapas genom att omvandla minuter, timmar osv till sekunder enligt "standard-mått" (60 sek är en minut, 60 min är en timma, 24 timmar ett dygn, 7 dygn är en vecka, 365 dagar är ett år).

Du kan addera och multiplicera *durations*:

```
2 * dyears(1)
```

```
## [1] "63072000s (~2 years)"
```

```
dyears(1) + dweeks(12) + dhours(15)
```

```
## [1] "38847600s (~1.23 years)"
```

Du kan addera och subtrahera *durations* till och från dagar:

```
tomorrow <- today() + ddays(1)
```

```
last_year <- today() - dyears(1)
```

Perioder

Perioder är tidsspänn utan en fixerad längd i sekunder. Istället räknar den med mer "människo-bekanta" storheter som dagar eller månader. Det gör beräkningar mer intuitiva:

```
one_pm <- ymd_hms("2016-03-12 13:00:00")
```

```
one_pm
```

```
## [1] "2016-03-12 13:00:00 UTC"
```

```
one_pm + ddays(1)
```

```
## [1] "2016-03-13 13:00:00 UTC"
```

Även här finns ett antal smidiga funktioner:

```
seconds(15)
```

```
## [1] "15S"
minutes(10)
## [1] "10M 0S"
hours(c(12, 24))
## [1] "12H 0M 0S" "24H 0M 0S"
days(7)
## [1] "7d 0H 0M 0S"
months(1:6)
## [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
## [5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
weeks(3)
## [1] "21d 0H 0M 0S"
years(1)
## [1] "1y 0m 0d 0H 0M 0S"
```

Du kan addera och multiplicera perioder:

```
10 * (months(6) + days(1))
## [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
## [1] "50d 25H 2M 0S"
```

Och addera dem till datum:

```
## Ett kalenderår
ymd("2016-01-01") + dyears(1)
## [1] "2016-12-31"
## Ett exakt år senare
ymd("2016-01-01") + years(1)
## [1] "2017-01-01"
```

Låt oss använda perioder för att fixa en knepighet i `flights`. Några plan verkar ha ankommit till destinationen innan de startade från NYC:

```

flights_dt %>%
  filter(arr_time < dep_time)

## # A tibble: 10,633 x 9
##   origin dest dep_delay arr_delay dep_time      sched_dep_time
##   <chr> <chr>   <dbl>   <dbl> <dtm>      <dtm>
## 1 EWR  BQN     9     -4 2013-01-01 19:29:00 2013-01-01 19:20:00
## 2 JFK  DFW    59    NA 2013-01-01 19:39:00 2013-01-01 18:40:00
## 3 EWR  TPA    -2     9 2013-01-01 20:58:00 2013-01-01 21:00:00
## 4 EWR  SJU    -6   -12 2013-01-01 21:02:00 2013-01-01 21:08:00
## 5 EWR  SFO    11   -14 2013-01-01 21:08:00 2013-01-01 20:57:00
## 6 LGA  FLL   -10    -2 2013-01-01 21:20:00 2013-01-01 21:30:00
## 7 EWR  MCO    41    43 2013-01-01 21:21:00 2013-01-01 20:40:00
## 8 JFK  LAX    -7   -24 2013-01-01 21:28:00 2013-01-01 21:35:00
## 9 EWR  FLL    49    28 2013-01-01 21:34:00 2013-01-01 20:45:00
## 10 EWR  FLL    -9   -14 2013-01-01 21:36:00 2013-01-01 21:45:00
## # ... with 10,623 more rows, and 3 more variables: arr_time <dtm>,
## #   sched_arr_time <dtm>, air_time <dbl>

```

Dessa är flighter som sträcker sig över midnatt. Vi använder samma datum för för både avgångar och ankomster, men dessa flighter ankom påföljande dygn. Vi kan fixa detta genom att addera `days(1)` till ankomsttiden för varje sådan flight:

```

flights_dt <- flights_dt %>%
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight * 1),
    sched_arr_time = sched_arr_time + days(overnight * 1)
  )

flights_dt %>%
  filter(overnight, arr_time < dep_time)

## # A tibble: 0 x 10
## # ... with 10 variables: origin <chr>, dest <chr>, dep_delay <dbl>,
## #   arr_delay <dbl>, dep_time <dtm>, sched_dep_time <dtm>,
## #   arr_time <dtm>, sched_arr_time <dtm>, air_time <dbl>,
## #   overnight <lgl>

```

Intervall

Det är uppenbart att `dyears(1) / ddays(365)` borde ge 1, eftersom *durations* alltid räknas i sekunder och `ddays(365) = dyear(1)`. Men om vi använder perioder, vad ger `years(1)/days(1)`? Ja, om året var 2015

skulle det ge 365 men om det var 2016 skulle det bli 366. `lubridate` innehåller helt enkelt inte tillräckligt med information för att ge ett entydigt svar. Istället får man en skattning tillsammans med en varning:

```
years(1) / days(1)
```

```
## estimate only: convert to intervals for accuracy
```

```
## [1] 365.25
```

Om du vill ha ett mer exakt besked behöver du använda ett intervall. Ett intervall är en *duration* med en startpunkt:

```
next_year <- today() + years(1)
```

```
(today() %--% next_year) / ddays(1)
```

```
## [1] 366
```

Summering

Vilket av *duration*, *period* och intervall ska man välja? Som vanligt, använd det enklaste alternativet som är tillräckligt för att lösa ditt problem. Om du enbart är intresserad av tid som sådan, använd en *duration*; om du behöver addera "människo-vänliga" tidpunkter, använd en *period*; om du behöver beräkna hur lång ett tidsspänn är använd ett *intervall*.

Figuren nedan summerar tillåtna aritmetiska operationer för de olika alternativen:

	date			date time			duration			period			interval			number		
date	-						-	+		-	+					-	+	
date time				-			-	+		-	+					-	+	
duration	-	+		-	+		-	+	/							-	+	×
period	-	+		-	+					-	+					-	+	×
interval									/			/						
number	-	+		-	+		-	+	×	-	+	×	-	+	×	-	+	×

I Wickhams bok finns en handfull övningar med tidsintervall (<http://r4ds.had.co.nz/dates-and-times.html#exercises-47>) och som vi hoppar över här.

Programmering i R

Introduktion

Nu är det dags för programmering i R/Rstudio. I denna del ska vi gå igenom fyra avsnitt som vart och ett ger lite olika aspekter på att programmera i R och som underlättar för dig att skriva bra kod.

1. Första avsnittet handlar om **pipes**, `%>%`, som ger större möjligheter att skriva kod på ett mer överskådligt sätt och när man bör respektive inte bör använda det
2. Sedan handlar det om att skriva **funktioner** vilket såväl underlättar skrivningen som läsbarheten i koden
3. Därefter blir det **data-strukturer** i R. Det handlar om vektorer, de fyra vanligaste samt tre *S3 klasser* byggda ovanpå dem och dessutom vad listor vs data-frames är och hur de skiljer sig.
4. Till sist några verktyg som underlättar *iterationer* i programkod - loopar och funktionsprogrammering.

Vi kommer att beröra det nödvändigaste för att hantera R/Rstudio på ett effektivt sätt men det finns en närmast överväldigande mängd programmerings-kunskaper att tillgå. Några tips:

- En hel del finns i Wickhams bok, lätt tillgängligt. <http://r4ds.had.co.nz/program-intro.html>
- *Hands on Programming with R*, av Garrett Golemund. En introduktion till R som ett programmeringsspråk. Påminner om innehållet i Wickhams bok men har andra exempel och lite annat angreppssätt.
- *Advanced R* av Hadley Wickham. En fortsättning/fördjupning av Wickhams grundbok, dvs den som ligger till grund för denna kurs. Finns online på <http://adv-r.had.co.nz/>

Pipes

En *pipe* är ett kraftfullt verktyg för att skriva en sekvens av multipla operationer. Pipes (`%>%`) kommer från modulen `magrittr` och laddas automatiskt med `tidyverse`.

Det finns flera fördelar med att använda *pipes* då man skriver kod, fr.a då man skriver kod som innehåller sekventiella operationer men det finns också tillfällen då pipes är mindre användbara. t.ex. om

- Pipes är längre än typ 10 steg. Då bör man istället skapa intermediära objekt med meningsfulla namn vilket ger överskådlighet och bättre underlättar debugging
- Du har multipla inputs eller outputs
- Du arbetar med DAGs och komplexa strukturer/relationer mellan objekt. Pipes är i grunden ett linjärt verktyg med vilket det lätt blir svåröverskådlig kod om man vill uttrycka komplexa relationer.

Det finns mer att lära sig om pipes i Wickhams bok <http://r4ds.had.co.nz/pipes.html> för den som vill veta mer.

Funktioner

Funktioner gör det möjligt att automatisera operationer som är mer frekventa istället för att kopiera och klistra in kod-avsnitt.

1. Du kan ge funktionen ett suggestivt namn så att koden blir lättare att förstå
2. Vid behov räcker det med att ändra koden på ett ställe istället för på flera
3. Du eliminerar risken för att göra fel då du klipper o klistrar, t.ex. uppdaterar ett variabelnamn på ett ställe men inte på ett annat

När bör du skapa en funktion?

Du bör överväga att göra en funktion närhelst du kopierat och klistrat in en kod fler än två gånger. Kolla in nedanstående kod, vad gör den?

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Koden skalar om samtliga värden för a, b, c och d så att de hamnar mellan 0 och 1. Men det finns ett fel i koden som uppstod då operationen klistrades in.

Om man istället skapar en funktion undviker man risken för sådana misstag. Låt oss göra det för ovanstående exempel. Först behöver man analysera koden. Hur många inputs har den?

```
(df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))

## [1] 1.0000000 0.3700012 0.9619970 0.4312099 0.0000000 0.1068024 0.8823159
## [8] 0.3910108 0.1442619 0.8073080
```

Den här koden har endast ett input, df\$a. För att göra inputs tydligare bör man ange input i generella termer. Här handlar det om en enda numerisk vektor så låt oss kalla input för x.

```
x <- df$a
(x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))

## [1] 1.0000000 0.3700012 0.9619970 0.4312099 0.0000000 0.1068024 0.8823159
## [8] 0.3910108 0.1442619 0.8073080
```

Det finns en del upprepningar i denna kod. Vi beräknar *range* tre gånger. Det verkar rimligt att göra det i ett enda steg:

```
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])

## [1] 1.0000000 0.3700012 0.9619970 0.4312099 0.0000000 0.1068024 0.8823159
## [8] 0.3910108 0.1442619 0.8073080
```

Att dra ut intermediära operationer till namngivna variabler gör det lättare att förstå vad koden gör. Nu kan vi göra en funktion av ovanstående kod.

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10))

## [1] 0.0 0.5 1.0
```

En funktion görs i tre steg:

1. Ge funktionen ett (beskrivande) namn, här `rescale01` eftersom den skalar om en numerisk vektor till värden mellan 0 och 1.
2. Lista argumenten för funktionen, inom `function()`, i detta fall endast ett argument `x`. Om det finns fler skriver man dessa typ `function(x, y, z)`.
3. Placera den utvecklade koden i funktionens *body*, ett block inom `{}`, omedelbart efter `function()`.

Det är lättare att börja med att skapa koden och sedan placera den i funktionen än att skapa funktionen och sedan göra koden. När vi kommit så här långt är det klokt att kolla om funktionen fungerar som tänkt genom att pröva den på några värden:

```
rescale01(c(-10, 0, 10))

## [1] 0.0 0.5 1.0

rescale01(c(1, 2, 3, NA, 5))

## [1] 0.00 0.25 0.50 NA 1.00
```

Nu kan vi förenkla det inledande exemplet med hjälp av funktionen:

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

En annan fördel med funktioner är att du behöver bara ändra på ett ställe om förutsättningarna i data förändras. Om vi t.ex. upptäcker att några variabler innehåller oändliga värden (`Inf`) kommer funktionen inte att fungera:

```
x <- c(1:10, Inf)
rescale01(x)

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

Eftersom koden finns i funktionen behöver vi endast ändra på ett ställe:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)

## [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
## [8] 0.7777778 0.8888889 1.0000000      Inf
```

Funktioner är till för människor och datorer

Namnet på funktionen är viktigt. Idealt bör det vara kort men ändå beskriva vad funktionen gör. Men, bättre ett något längre beskrivande namn än att det är kort eftersom Rstudio auto-kompletterar namnet.

Generellt bör namnet vara ett verb och argumenten substantiv. Undantag är om funktionen använder välkända operationer - `mean()` är bättre än `compute_mean()` t.ex.

Var konsekvent om funktionen innehåller flera ord - använd konsekvent ett sätt att binda samman orden, ex `snake_case` eller `snakeCase`.

Använd gärna kommentars-möjligheterna (inled raden med `#`) för att förklara varför du valt just den funktion du valt (finns oftast fler sätt att lösa samma problem). Ett annat sätt att använda kommentarer är att avdela koden i sektioner med hjälp av multippla bindestreck eller lika med-tecken för att göra koden mer lättläst:

```
Load data ----- Plot data -----
```

TIPS: Rstudio har ett kortkommando för att skapa dessa "rubrikrader", `Cmd/Ctrl + Shift + R` och visar dem i navigationsrutan som en drop-down-lista.

Villkorlig exekvering

Ett `if`-statement gör det möjligt att exekvera koden villkorligt:

```
if (condition) {
  # code executed when condition is TRUE
} else {
  # code executed when condition is FALSE
}
```

Villkor

Villkoret måste vara ett logiskt uttryck och kunna utvärderas till `TRUE` eller `FALSE`. Om det är en vektor får du en varning, om det är ett missing value får du ett error:

```
if (c(TRUE, FALSE)) {}
if (NA) {}
```

Du kan använda `||` eller `&&` för att kombinera flera logiska uttryck. Dessa operatorer kortsluter exekveringen - så fort `||` möter det första `TRUE` returneras `TRUE` och avbryter exekveringen; så fort `&&` möter det första `FALSE` returneras `FALSE` och avbryter. Använd aldrig `|` resp `&` i ett `if`-uttryck eftersom dessa enkla operatorer är *vektorisade operationer* och används för multipla värden, t.ex. i `filter()`. Om du har en vektor med logiska värden kan du använda `any()` eller `all()` för att slå samman vektorn till ett logiskt värde.

Var försiktig med att testa för likhet. `==` är vektoriserad och det är därför lätt att få mer än ett output. Kolla att längden på input till det logiska villkoret är 1 eller slå samman det med `all()` eller `any()`, eller använd det icke-vektorisade `identical()` vilket är strikt - det returnerar alltid ett enkelt `TRUE` eller `FALSE` och slår inte samman olika typer, t.ex. om det finns både heltal (integers) och numeriska värden (numeric) i vektorn:

```
identical(0L, 0)

## [1] FALSE
```

Se också upp med flytande decimaltecken:

```
x <- sqrt(2) ^ 2
x

## [1] 2

x == 2

## [1] FALSE

x - 2

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

Använd istället `dplyr::near()` för jämförelser så som beskrivs i jämförelser tidigare. [Länk här!](#)

Multipla villkor

Du kan länka multipla if-uttryck tillsammans:

```
if (this) {
  # do that
} else if (that) {
  # do something else
} else {
  #
}

centre <- function(x, type) {
  switch(type,
    mean = mean(x),
    median = median(x),
    trimmed = mean(x, trim = .1))
}

x <- rnorm(1000)

centre(x, "mean")
## [1] -0.02420306

centre(x, "median")
## [1] -0.01980242

centre(x, "trimmed")
## [1] -0.02071111
```

En annan användbar funktion för att eliminera långa if-uttryck är `cut()` vilken används för att *kategorisera numeriska variabler*.

Funktionsargument

Argumenten till en funktion kan delas in i två grupper: en som förser funktionen med data, och en annan som kontrollerar hur funktionen beräknar data (detail). Till exempel:

- I `log()` är data `x` och detail är logaritm-basen (default = `e`).
- I `mean()` är data `x` och details hur mycket data ska trimmas från dess ändar (`trim`) och hur missing values ska hanteras (`na.rm`)

- I `t.test()` är `data` `x` och `y`, `details` är `alternative`, `mu`, `paired`, `var.equal` och `conf.level`
- I `str_c()` kan du lägga till valfritt antal strängar till `...` och `details` kontrolleras med `sep` och `collapse`

Som princip bör `data`-argumentet komma först. `Details` därefter och har vanligtvis default-värden. Du kan förstås specificera ett eget default-värde:

```
#Compute confidence interval around mean using normal approximation
```

```
mean_ci <- function(x, conf = 0.95) {
  se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
  mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}
```

```
x <- runif(100)
```

```
mean_ci(x)
```

```
## [1] 0.4248719 0.5389206
```

```
mean_ci(x, conf = 0.99)
```

```
## [1] 0.4069535 0.5568390
```

Kolla värden

För att undvika att man förser funktionen med ogiltiga värdemängder bör man ligga till kontroll-villkor i koden. Om vi t.ex. har gjort ett antal funktioner för att beräkna viktade summeringar och lägger till input med olika längd:

```
wt_mean <- function(x, w) {
  sum(x * w) / sum(w)
}
wt_var <- function(x, w) {
  mu <- wt_mean(x, w)
  sum(w * (x - mu) ^ 2) / sum(w)
}
wt_sd <- function(x, w) {
  sqrt(wt_var(x, w))
}
```

```
wt_mean(1:6, 1:3)
```

```
## [1] 7.666667
```

Vi får inget felmeddelande på grund av R:s recycling-regler. Men om vi lägger till ett kodavsnitt för att kolla att förutsättningarna för funktionen gäller kan vi se till att R returnerar ett *error* om dessa inte uppfylls. I detta kan vi göra det med en stopp-funktion, `stop()`:

```
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }
  sum(w * x) / sum(w)
}
```

Eller med `stopifnot()`, vilket ofta är effektivare:

```
wt_mean <- function(x, w, na.rm = FALSE) {
  stopifnot(is.logical(na.rm), length(na.rm) == 1)
  stopifnot(length(x) == length(w))

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(w)
}

wt_mean(1:6, 6:1, na.rm = "foo")

#>Error in wt_mean(1:6, 6:1, na.rm = "foo") : is.logical(na.rm) is not TRUE
```

punkt-punkt-punkt

Många funktioner kan ta ett godtyckligt antal inputs:

```
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

## [1] 55

stringr::str_c("a", "b", "c", "d", "e", "f")

## [1] "abcdef"
```

Det är möjligt tack vare argumentet `...`. Detta är användbart då du kan skicka dessa argument till en annan funktion. Till exempel kan du göra hjälpfunktioner som bygger på `str_c()`:

```
commas <- function(...) stringr::str_c(..., collapse = ", ")

commas(letters[1:10])
```

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

rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
rule("Important output")

## Important output -----
```

Returnera värden

Vad funktionen ska returnera är oftast uppenbart - det är ju därför du gjorde funktionen! Två saker att tänka på då ett värde ska returneras:

1. Blir funktionen lättare att läsa om värdet returneras tidigt?
2. Kan du göra funktionen med hjälp av pipes?

Explicita `return`-uttryck

Värdet som returneras är vanligen från det sista uttrycket som utvärderas, men du kan välja att returnera tidigare med hjälp av `return()`. Ett vanligt skäl är att inputs är tomma:

```
complicated_function <- function(x, y, z) {
  if (length(x) == 0 || length(y) == 0) {
    return(0)
  }

  # Complicated code here
}
```

Ett annat skäl är om du har ett `if`-uttryck med ett block av komplex kod och ett annat med ett enkelt. Till exempel:

```
f <- function() {
  if (x) {
    # Do
    # something
    # that
    # takes
    # many
    # lines
    # to
    # express
```



```

} else {
  # return something short
}
}

```

När man läser denna kod är det lätt att ha glömt av vad villkoret gällde då man kommer fram till `else`. Ett sätt att skriva om denna funktion är att tidigt returnera värdet för det enkla alternativet:

```

f <- function() {
  if (!x) {
    return(something_short)
  }
  Do
    # something
    # that
    # takes
    # many
    # lines
    # to
    # express
}

```

Skriva funktioner med hjälp av pipes

Om du skriver funktioner med hjälp av *pipes* är det viktigt att komma ihåg vad det är för en typ av objekt som returneras. T.ex. med `dplyr` och `tidyr` är objektet alltid en *dataframe*.

Det finns två grundläggande typer av *pipeable functions*: *transformations* och *side-effects*. Med *transformations* är objektet funktionens första argument och den returnerar ett modifierat objekt. Med *side-effects* modifieras inte objektet i sig. Istället gör funktionen något med hjälp av objektet - t.ex. ritar ett diagram eller sparar till en fil. *Side-effects* funktioner ska returnera det första argumentet "osynligt" så att även om det inte printas kan det ändå användas i en pipe. Till exempel printar nedanstående funktion antalet missing values i en dataframe:

```

show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Missing values: ", n, "\n", sep = "")

  invisible(df)
}

```

Om vi anropar funktionen kommer `invisible()` inte att printa input.

```
show_missings(mtcars)
```

```
## Missing values: 0
```

Men den finns fortfarande där:

```
x <- show_missings(mtcars)
```

```
## Missing values: 0
```

```
class(x)
```

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

```
dim(x)
```

```
## [1] 32 11
```

Och den går att använda i en pipe:

```
mtcars %>%
```

```
  show_missings() %>%
```

```
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
```

```
  show_missings()
```

```
## Missing values: 0
```

```
## Missing values: 18
```

Environment (svårt hitta ett svenskt uttryck)

Den sista komponenten i en funktion är dess environment. Environment kontrollerar hur R hittar värdet associerat med ett namn. Om vi tar nedanstående funktion:

```
f <- function(x) {  
  x + y  
}
```

...skulle funktionen i de flesta program ge ett *error* eftersom `y` inte är definierat i funktionen. Men i R är denna kod giltig eftersom R kommer att leta i den Environment där funktionen definierades:

```
y <- 100
```

```
f(10)
```

```
## [1] 110
```

```
y <- 1000
```

```
f(10)
```

```
## [1] 1010
```

Vektorer

Fokus i detta avsnitt är på base R-funktioner så vi behöver egentligen inte ladda in några packages, men eftersom vi ska använda en handfull funktioner från purrr för att hantera några inkonsistenser i base R laddar vi ändå in

```
library(tidyverse)
```

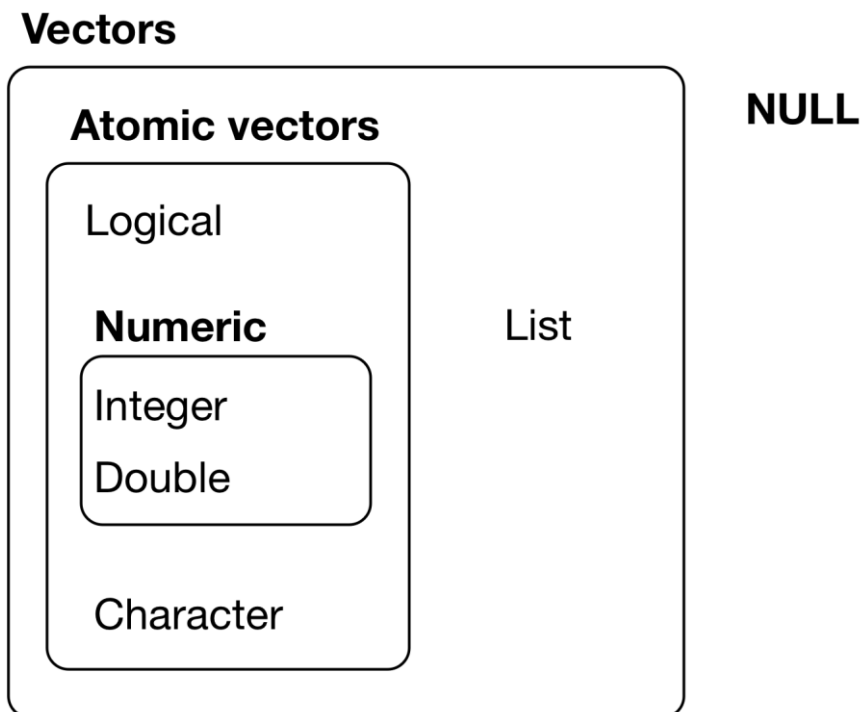
Introduktion

Vektorer är ett centralt begrepp i R och betecknar objekt som är grundläggande för de flesta funktioner i R. Det finns två typer av vektorer i R:

1. Atomic vectors, av vilka det finns sex typer: logical, integer, double, character, complex och raw. Integer och double är tillsammans numeriska vektorer.
2. Lists, kallas ibland rekursiva vektorer eftersom lists kan innehålla andra lists.

Den huvudsakliga skillnaden mellan *atomic* och *list*-vektorer är att *atomic* är homogena medan *lists* kan vara heterogena. Objektet `NULL` används ofta för att markera frånvaron av en vektor (till skillnad från `NA` som markerar frånvaron av ett värde i en vektor!). `NULL` uppför sig som en vektor med längden 0.

Nedanstående figur sammanfattar relationen mellan de olika typerna:



Varje vektor har två nyckel-egenskaper:

1. Dess typ, vilken kan bestämmas med hjälp av `typeof()`.

```
typeof(letters)
## [1] "character"

typeof(1:10)
## [1] "integer"
```

2. Dess längd som bestäms med hjälp av `length()`.

```
x <- list("a", "b", 1:10)
length(x)
## [1] 3
```

Vektorer kan även innehålla andra metadata i form av attribut. Dessa attribut används för att skapa förstärkta vektorer (augmented vectors). Det finns tre typer av förstärkta vektorer:

1. **Factors** bygger på *integer* vektorer
2. **Datum** och datumtid byggs på *numeriska* vektorer
3. **Dataframes** och **tibbles** byggs på *lists*.

Vi ska gå igenom de viktigaste typerna av vektorer: *atomic vectors* (och då endast *logical*, *integer*, *double* och *character*), *lists* och *augmented vectors*.

Atomic vectors

Logical vectors

Logiska vektorer är den enklaste typen av atomic vectors. Den kan anta endast tre olika värden: *TRUE*, *FALSE* eller *NA*. Logiska vektorer skapas oftast med hjälp av *jämförelsevektorer* alternativt med hjälp av `c()`:

```
1:10 %% 3 == 0
## [1] FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE TRUE FALSE

c(TRUE, TRUE, FALSE, NA)
## [1] TRUE TRUE FALSE NA
```

Numeriska vektorer

Heltal (*integers*) och *double*-vektorer är de *numeriska* vektorerna. I R är siffer-data *double* som default. Vill du ha dem som *integer* kan du markera det genom att sätta ett L bakom heltalet:

```
typeof(1)
## [1] "double"

typeof(1L)
## [1] "integer"
```

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

```
1.5L
```

```
## [1] 1.5
```

Distinktionen mellan *integer* och *double* är vanligtvis inte viktig men det finns två väsentliga skillnader som kan vara bra att känna till:

1. *Doubles* är approximationer och representeras av decimaltal vilka inte alltid kan preciseras med en given mängd minne. Ta t.ex. det kvadrerade värdet av roten ur 2:

```
x <- sqrt(2) ^ 2
```

```
x
```

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

```
x - 2
```

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

Räkna med att det alltid finns ett visst approximeringsfel när man hanterar *double* vektorer. Istället för att använda exakt likhet (==) när man jämför *doubles* bör man använda `dplyr::near()` vilken har en viss numerisk tolerans.

2. Integers har ett special-värde `NA` medan doubles har fyra: `NA`, `NaN`, `Inf` och `-Inf`. De tre sistnämnda kan uppstå genom division:

```
c(-1, 0, 1) / 0
```

```
## [1] -Inf NaN Inf
```

Istället för att använda exakt likhet för dessa special-värden bör man, analogt med `is.na()`, använda någon av hjälpfunktionerna `is.finite()`, `is.infinite()` eller `is.nan()`.

Character

Text-strängar är de mest komplexa av atomic vectors eftersom de kan innehålla en godtycklig mängd data. Vi har redan gått igenom hanteringen av textsträngar [[länk till tidigare avsnitt](#)] och kan därför lämna denna typ av atomic vectors.

Använda atomic vectors

Hur använder man då *atomic vectors* och vilka verktyg finns det för att göra det? Vi ska titta närmare på

1. Hur man konverterar en typ till en annan och när det sker automatiskt.
2. Hur man kan vrta att ett objekt är en viss typ av vektor.
3. Vad som händer då man använder vektorer av olika längd
4. Hur man benämner elementen i en vektor
5. Hur man extraherar de element man är intresserad av

Konvertering (Coercion)

Det finns två sätt att konvertera en typ av vektor till en annan.

1. *Explicit konvertering* sker när du använder funktioner som `as.character()`, `as.logical()`, `as.integer()` eller `as.double()`. Det kan vara värt att kolla om du kan åstadkomma detta mer "upstream" så att du inte behöver göra någon konvertering, t.ex. genom att justera `col.types` då du importerar data.
2. *Implicit konvertering* sker när du använder en vektor i en specifik kontext som förväntar en viss typ av vektor. t.ex. då du använder en logisk vektor tillsammans med en summeringsfunktion eller när du använder en *double vector* där en integer förväntas.

Det första sättet är tämligen straightforward men vi ska dröja vid det andra. Den viktigaste typen av implicit konvertering är då man använder en logisk vektor i en numerisk kontext. I det fallet blir TRUE konverterat till 1 och FALSE till 0. Så summan av en logisk vektor är antalet TRUE och medelvärde av en logisk vektor är andelen TRUE i vektorn:

```
x <- sample(20, 100, replace = TRUE)
y <- x > 10
sum(y) # Hur många är större än 10?
## [1] 44
mean(y) # Hur stor är andelen större än 10?
## [1] 0.44
```

Det är också viktigt att förstå vad som händer när man försöker mixa flera typer av vektorer med hjälp av `c()` - den mest komplexa vinner alltid:

```
typeof(c(TRUE, 1L))
## [1] "integer"
typeof(c(1L, 1.5))
## [1] "double"
typeof(c(1.5, "a"))
## [1] "character"
```

En *atomic vector* kan inte ha en mix av typer eftersom typen av vektor är en egenskap hos hela vektorn inte dess enskilda element. Om du behöver mixa olika typer ska du istället använda en lista (*a list*).

Testfunktioner

För att ta reda på vilken typ av vektor du har kan du använda `typeof()`. Ett annat sätt är att använda funktioner som returnerar logiska värden. De tidigare funktionerna i base R (t.ex. `is.atomic()`) kan ibland ge överraskande resultat. De testfunktioner som finns i `purrr`, `if_*`, är säkrare och summeras i tabellen nedan:

	lgl	int	dbl	chr	list
<code>is_logical()</code>	X				
<code>is_integer()</code>		X			
<code>is_double()</code>			X		
<code>is_numeric()</code>		X	X		
<code>is_character()</code>				X	
<code>is_atomic()</code>	X	X	X	X	
<code>is_list()</code>					X
<code>is_vector()</code>	X	X	X	X	X

Scalars och recycling

I R opererar de flesta matematiska funktioner med vektorer. Det innebär att du inte behöver explicit göra en iterering när du använder matematiska funktioner. Därför fungerar t.ex. nedanstående kod:

```
sample(10) + 100
```

```
## [1] 108 107 110 105 103 102 104 106 101 109
```

```
runif(10) > 0.5
```

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

Det är uppenbart vad som händer då man adderar två vektorer av samma längd, men vad händer om man adderar två vektorer av *olika* längd?

```
1:10 + 1:2
```

```
## [1] 2 4 4 6 6 8 8 10 10 12
```

Här kommer R att expandera den kortare vektorn till samma längd som den längre, vilket kallas *recycling*. Om längden på den kortare vektorn inte är en multipel av den längre vektorns längd får du en varning:

```
1:10 + 1:3
```

```
## Warning in 1:10 + 1:3: longer object length is not a multiple of shorter
```

```
## object length
```

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

Viktigt att veta: medan recycling av vektorer kan användas för att skriva effektiv kod kan recycling också i tysthet dölja problem. Därför kommer vektoriserade funktioner i R att ge *error* om du använder något

annat än en skalär (*scalar*) för recycling. Om du vill recyccla behöver du göra det själv genom att använda `replicate rep()`:

```
tibble(x = 1:4, y = 1:2)
```

```
## Error: Tibble columns must have consistent lengths, only values of length one are recycled:
```

```
## * Length 2: Column `y`
```

```
## * Length 4: Column `x`
```

```
tibble(x = 1:4, y = rep(1:2, 2))
```

```
## # A tibble: 4 x 2
```

```
##   x     y
```

```
##   <int> <int>
```

```
## 1     1     1
```

```
## 2     2     2
```

```
## 3     3     1
```

```
## 4     4     2
```

```
tibble(x = 1:4, y = rep(1:2, each = 2))
```

```
## # A tibble: 4 x 2
```

```
##   x     y
```

```
##   <int> <int>
```

```
## 1     1     1
```

```
## 2     2     1
```

```
## 3     3     2
```

```
## 4     4     2
```

Benämna vektorer

Samtliga typer av vektorer kan namnges. Du kan göra det då du skapar vektorn med `c()`:

```
c(x = 1, y = 2, z = 4)
```

```
## x y z
```

```
## 1 2 4
```

Eller efteråt med `purrr::set_names()`:

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

```
## a b c
```

```
## 1 2 3
```

Detta är användbart då du vill extrahera ett urval av datamängden (*subsetting*).

Subsetting

`dplyr::filter()` fungerar endast på *tibbles*. Om vi vill extrahera data ur vektorer behövs ett annat verktyg, nämligen `[]`.

Det finns fyra sätt att extrahera data ur vektorer:

1. En numerisk vektor innehåller endast tal vilka kan vara positiva, negativa eller noll. Extrahering med positiva tal anger elementen med dessa positioner:

```
x <- c("one", "two", "three", "four", "five")
x[c(3, 2, 5)]

## [1] "three" "two" "five"
```

Genom att upprepa positionen kan man göra vektorn längre:

```
x[c(1, 1, 5, 5, 5, 2)]

## [1] "one" "one" "five" "five" "five" "two"
```

Negativa värden droppar elementen vid motsvarande positioner:

```
x[c(-1, -3, -5)]

## [1] "two" "four"
```

Det går inte att blanda positiva och negativa värden:

```
x[c(1, -1)]

## Error in x[c(1, -1)]: only 0's may be mixed with negative subscripts
```

2. Subsetting med logiska värden behåller samtliga vektorelement som motsvarar ett TRUE värde. Detta är särskilt användbart tillsammans med jämförelseoperatorerna:

```
x <- c(10, 3, NA, 5, 8, 1, NA)
# Alla värden i x som inte är NA:
x[!is.na(x)]

## [1] 10 3 5 8 1

# Alla jämna värden av x (inkl NA!):
x[x %% 2 == 0]

## [1] 10 NA 8 NA
```

3. Om du har en namngiven vektor kan du extrahera med en sträng-vektor (*character vector*):

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]

## xyz def
## 5 2
```

4. Du kan extrahera data från matriser eller dataframes genom att välja rad- resp kolumnpositioner. Om `x` är en 2 dimensionell matris anger `x[1,]` den första raden (före kommatecknet) och samtliga kolumner; `x[, -1]` anger samtliga rader (tomt före kommatecknet) och samtliga kolumner utom den första.

Det finns mer att läsa om subsetting i boken Advanced R: <http://adv-r.had.co.nz/Subsetting.html#applications>.

En viktig variation av `[` är `[[` vilken alltid extraherar *ett enkelt värde*. Distinktionen mellan `[` och `[[` är viktigast då du hanterar listor vilket vi återkommer till.

Rekursiva vektorer (lists)

Lists är mer komplexa än atomic vectors eftersom *lists* kan innehålla andra *lists*. Detta kan vara särskilt användbart då man har hierarkiska data. Du skapar en list med hjälp av `list()`:

```
x <- list(1, 2, 3)
x
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

En mycket användbar funktion för att arbeta med lists är `str()`. Denna funktion fokuserar på list-*strukturen* snarare än innehållet:

```
str(x)
## List of 3
## $ : num 1
## $ : num 2
## $ : num 3
x_named <- list(a = 1, b = 2, c = 3)
str(x_named)
## List of 3
## $ a: num 1
## $ b: num 2
## $ c: num 3
```

Till skillnad mot atomiska vektorer kan lists innehålla olika klasser av objekt:

```
y <- list("a", 1L, 1.5, TRUE)
str(y)

## List of 4
## $ : chr "a"
## $ : int 1
## $ : num 1.5
## $ : logi TRUE
```

Listor kan t.o.m. innehålla andra lists:

```
z <- list(list(1, 2), list(3, 4))
str(z)

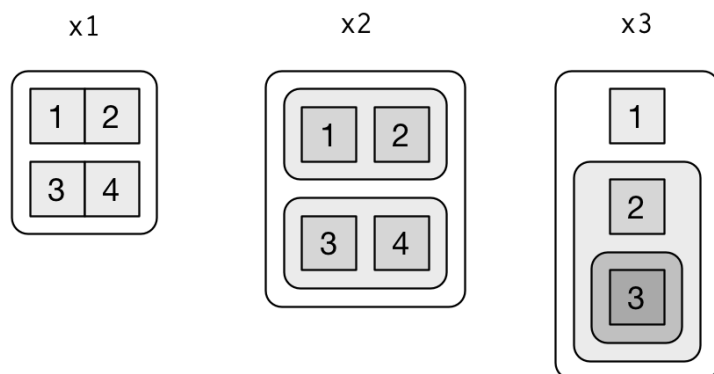
## List of 2
## $ :List of 2
## ..$ : num 1
## ..$ : num 2
## $ :List of 2
## ..$ : num 3
## ..$ : num 4
```

Visualisera lists

Det kan underlätta förståelsen av mer komplexa operationer på lists om man har en visuell bild av lists. Låt oss säga att vi har nedanstående tre lists:

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

I bilden nedan har lists rundade hörn och atomic vectors har skarpa hörn. *Children* återges innanför parentes och har mörkare nyans för att se den hierakiska strukturen lättare.



Subsetting lists

Det finns tre sätt att extrahera objekt ur lists. Vi skapar en lista som vi kallar `a`:

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```

- `[` extraherar en underordnad lista (*sublist*). Resultatet är alltid en *list*:

```
str(a[1:2])

## List of 2
## $ a: int [1:3] 1 2 3
## $ b: chr "a string"

str(a[4])

## List of 1
## $ d:List of 2
## ..$ : num -1
## ..$ : num -5
```

Precis som med andra vektorer kan du extrahera objekt med logiska, integer- och character-vektorer.

- `[[` extraherar en enskild komponent ur en lista. Funktionen tar bort en nivå ur hierarkin:

```
str(a[[1]])

## int [1:3] 1 2 3

str(a[[4]])

## List of 2
## $ : num -1
## $ : num -5
```

- `$` är en genväg för att extrahera namngivna element ur en lista. Den fungerar på samma sätt men du behöver inte ange `[[`:

```
a$a

## [1] 1 2 3

a[["a"]]

## [1] 1 2 3
```

Distinktionen mellan `[` och `[[` är viktig eftersom `[[` borrar sig ned i listans hierarkiska struktur medan `[` returnerar en mindre, avgränsad lista. Jämför koden ovan med nedanstående visualisering.

Attribut

Du kan lägga till godtyckliga metadata via vektorns *attribut*. Attribut kan ses som en namngiven lista av vektorer som kan kopplas till valfritt objekt. Du kan få fram attributen till en enskild vektor genom `attr()` eller se samtliga attribut med hjälp av `attributes()`. Du skapar attributen med hjälp av samma funktioner:

```
x <- 1:10
attr(x, "greeting")

## NULL

attr(x, "greeting") <- "Hi!"
attr(x, "farewell") <- "Bye!"
attributes(x)

## $greeting
## [1] "Hi!"
##
## $farewell
## [1] "Bye!"
```

Det finns tre särskilt viktiga attribut som används för att implementera grundläggande delar av R:

1. *Names* - används till att namnge elementen i en vektor
2. *Dimensions* (förkortat *dims*) gör att en vektor uppträder som en matris eller array
3. *Class* - används till att implementera S3 objektorienterat systemet (styr hur generiska funktioner arbetar, se <http://adv-r.had.co.nz/OO-essentials.html#s3>)

Förstärkta vektorer (Augmented vectors)

Atomic vectors och lists är byggstenarna till andra vektortyper t.ex. *factors* och *dates*. De senare kallas förstärkta (augmented) vektorer eftersom de utöver värde har attribut, inkl *class*. Eftersom dessa vektorer har en class-tillhörighet kommer de att uppträda annorlunda än andra vektorer. Här ska fyra typer av förstärkta vektorer beröras:

- Factors
- Dates
- Date-times
- Tibbles

Factors

Factors representerar kategoriska data och bygger på integers. De har *levels* som sitt attribut (och *class*):

```
x <- factor(c("ab", "cd", "ab"), levels = c("ab", "cd", "ef"))
typeof(x)

## [1] "integer"
```

```
attributes(x)

## $levels
## [1] "ab" "cd" "ef"
##
## $class
## [1] "factor"
```

Dates och date-times

Dates och date-times är numeriska vektorer med klasserna *date* respektive *POSIXct*. Eftersom *POSIXct* är sällsynt förekommande då man arbetar med *tidyverse* och att *lubridate* erbjuder flera verktyg för att hantera dessa klasser, går vi här inte närmare in på dem. För den som är intresserad finns i Wickhams bok ett avsnitt som fördjupar resonemanget, <http://r4ds.had.co.nz/vectors.html#attributes>

Tibbles

Tibbles är förstärkta lists med klassen "tbl_df" + "tbl" + "data.frame" samt attributen `names` och `row.names`:

```
tb <- tibble::tibble(x = 1:5, y = 5:1)
typeof(tb)

## [1] "list"

attributes(tb)

## $names
## [1] "x" "y"
##
## $row.names
## [1] 1 2 3 4 5
##
## $class
## [1] "tbl_df"      "tbl"        "data.frame"
```

En viktig skillnad mellan en list och en tibble är att i en tibble måste datatabellen ha vektorer av samma längd. Traditionella dataramar har en liknande struktur:

```
df <- data.frame(x = 1:5, y = 5:1)
typeof(df)

## [1] "list"

attributes(df)

## $names
## [1] "x" "y"
```

```
##  
## $class  
## [1] "data.frame"  
##  
## $row.names  
## [1] 1 2 3 4 5
```

Skillnaden ligger i att klassen skiljer sig - tibbles class inkluderar data.frame vilket innebär att tibbles "ärver" den traditionella dataramens karaktäristika som default.

Itereringar

Ett verktyg för att slippa upprepa kod är funktioner, som vi sett ovan. Ett annat är *iterering*. Vi ska nosa på två itererings-paradigm: *imperativ programmering* och *funktionell programmering*.

Imperativ programmering (IP) innehåller verktyg som t.ex. *for-loops* och *while-loops*, och det framgår tydligt vad som händer. Nackdelen är att IP är något omständigt, kräver många ord och man behöver hålla reda på intermediära variabler. Funktionell programmering (FP) innehåller verktyg för att extrahera återkommande kodavsnitt så att gemensamma loop-mönster får sin egen funktion. Det innebär att man kan lösa många itereringsproblem med mindre kod och mindre risk för fel.

For-loops

Vi utgår från en enkel tibble:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

Vi vill beräkna medianen av varje kolumn. Du kan göra det med copy-and-paste:

```
median(df$a)
## [1] -0.03035565

median(df$b)
## [1] 0.07643633

median(df$c)
## [1] 0.185524

median(df$d)
## [1] 0.4464841
```

Men det bryter mot tumregeln att aldrig klipp/klistra mer än två gånger. Istället kan du göra en *for-loop*:

```
output <- vector("double", ncol(df)) # 1. output
for (i in seq_along(df)) {           # 2. sequence
  output[[i]] <- median(df[[i]])      # 3. body
}
output
```



```
## [1] -0.03035565 0.07643633 0.18552395 0.44648406
```

Varje loop har tre komponenter:

1. Output - `output <- vector("double", length(x))`

Ett generiskt sätt att skapa en tom vektor med given längd är `vector()`. Funktionen innehåller två argument: *typen* av vektor ("logical", "integer", "double", "character", etc) och *längden* på vektorn.

2. Sekvensen - `i` in `seq_along(df)`. Detta bestämmer vad loopen ska köras över. Varje omgång av loopen lägger `i` till ett värde från `seq_along(df)`. `seq_along()` är en säkrare version av det vanligare `1:length(l)`, med en viktig skillnad: om du har en tom vektor hanterar `seq_along()` det korrekt:

```
y <- vector("double", 0)
seq_along(y)

## integer(0)

1:length(y)

## [1] 1 0
```

3. Body - `output[[i]] <- median(df[[i]])`. Detta avsnitt är det som gör jobbet. Kodavsnittet upprepas, varje gång med ett nytt värde på `i`. Den första iterationen kommer att köra

`output[[1]] <- median(df[[1]])`, den andra `output[[2]] <- median(df[[2]])` och den tredje `output[[3]] <- median(df[[3]])`.

Kolla igenom övningarna i Wickhams bok <http://r4ds.had.co.nz/iteration.html#exercises-55>

Variationer av for-loopar

Det finns fyra variationer av for-loopar som är viktiga att känna till:

1. Modifiera ett existerande objekt istället för att skapa ett nytt objekt
2. Loopa genom namn eller värden istället för index
3. Hantera outputs av okänd längd
4. Hantera sekvenser av okänd längd

Modifiera ett existerande objekt

Ett exempel är det då vi ville transformera varje kolumn i en datara:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

För att lösa detta med en for-loop tänker vi igenom de tre komponenterna: 1. Output - har vi redan; samma som input 2. Sekvens - vi kan betrakta en dataram som en lista av kolumner så vi kan iterera över kolumnerna med hjälp av `seq_along(df)`. 3. Body - vi implementerar funktionen `rescale01()`:

```
for (i in seq_along(df)) {
  df[[i]] <- rescale01(df[[i]])
}
```

När längden på output inte är känd

Ibland vet man inte hur lång output blir. Man kan frestas att lösa detta problem genom att bygga på output-vektorn:

```
means <- c(0, 1, 2)
output <- double()
for (i in seq_along(means)) {
  n <- sample(100, 1)
  output <- c(output, rnorm(n, means[[i]]))
}
str(output)

## num [1:141] 0.482 -2.693 -0.252 0.446 -1.012 ...
```

Men det är inte särskilt effektivt. För varje iterering behöver R kopiera all data från tidigare iterationer och man får ett "kvadratisk" beteende, dvs en loop med tre ggr så många element kommer att behöva 3^2 ggr så lång tid att köra. En bättre lösning är att spara resultaten i en lista och sedan kombinera listan till en gemensam vektor då looperna är kompletta:

```
out <- vector("list", length(means))

for (i in seq_along(means)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, means[[i]])
}
```

```
str(out)

## List of 3
## $ : num [1:77] -0.873 -0.458 0.298 1.135 0.561 ...
## $ : num [1:75] 1.008 -0.722 0.417 2.139 1.168 ...
## $ : num [1:24] 2.18 2.51 1.19 3.13 2.29 ...

str(unlist(out))

## num [1:176] -0.873 -0.458 0.298 1.135 0.561 ...
```

För att kombinera listan till en vektor kan man utöver `unlist()` använda `purrr::flatten_dbl()`. Det är mer stringent eftersom funktionen ger ett *error* om inte input är en lista med numeriska värden (`dbl`).

Detta mönster kan återkomma vid andra tillfällen:

- Du kanske vill generera en lång textsträng. Istället för att använda `paste()` vid varje iterering bör du spara output i en *character vector* och sedan kombinera dessa med hjälp av `paste(output, collapse = "")`.
- Du kanske vill skapa en stor dataram. Istället för att knyta ihop output från varje iterering med tidigare med hjälp av `rbind()` är det bättre att spara output till en lista och sedan kombinera output med hjälp av `dplyr::bind_rows(output)` till en dataram.

När sekvensens längd är okänd

Ibland är inte ens input-sekvensens längd känd. Det är vanligt vid simuleringar, t.ex. om man vill köra looperna till dess man fått klave tre gånger i rad. Det kan man inte göra med en `for`-loop. Istället bör man använda en *while*-loop. Den har bara två komponenter, ett villkor och en body.

```
while (condition) {
  # body
}
```

En *while*-loop är mer generell än en *for*-loop. Här är ett exempel på en *while*-loop för att undersöka hur många försök som behövs för att få tre klave (eller krona) i rad:

```
flip <- function() sample(c("T", "H"), 1)

flips <- 0
nheads <- 0

while (nheads < 3) {
  if (flip() == "H") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
}
```

```

}
flips <- flips + 1
}

flips
## [1] 12

```

I Wickhams bok finns ett antal bra övningar: <http://r4ds.had.co.nz/iteration.html#exercises-56>

for-loops vs. Funktioner

For-loops är mindre viktiga i R än i andra programmeringsspråk eftersom R är ett funktionsorienterat programspråk. Det innebär att man vanligtvis kan omforma for-loops till en funktion och anropa den funktionen istället för att använda loopopen. Varför är detta en poäng? Ja, låt oss titta på följande exempel:

```

df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

```

Vi vill beräkna medelvärde för varje kolumn. Man kan göra det med en for-loop:

```

output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}

output
## [1] 0.13459477 0.22621395 0.09949778 -0.31922665

```

Du inser att denna beräkning kommer du att göra ganska ofta så du gör om den till en funktion, `col_mean()`:

```

col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}

```

Men så inser du att det också skulle vara bra att ersätta medelvärde med median och standardavvikelse. Så du klipper o klistrar:

```
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}

col_sd <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

Här blir det en mängd upprepad kod och det kan vara på sin plats att fundera på hur man kan generalisera koden. Vad skulle du göra med följande uppsättning kod:

```
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

La du märke till att den mesta koden är upprepningsbar (och att du därför kan extrahera koden till en funktion):

```
f <- function(x, i) abs(x - mean(x)) ^ i
```

Vi kan göra precis samma sak med `col_mean()`, `col_median()` och `col_sd()` genom att lägga till ett argument som använder funktionen på varje kolumn:

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}

col_summary(df, median)

## [1] 0.28728858 -0.08682806 -0.03412453 -0.38735340

col_summary(df, mean)
```

```
## [1] 0.13459477 0.22621395 0.09949778 -0.31922665
```

Mappningsfunktioner

Modulen `purrr` innehåller många funktioner som reducerar behovet av att konstruera loopar. Dessa funktioner gör att du lättare kan bryta ned manipulationer av listor till enklare delar.

Det finns en funktion för varje typ av output:

- `map()` gör en lista.
- `map_lgl()` gör en logisk vektor.
- `map_int()` skapar en integer vektor.
- `map_dbl()` skapar en numerisk vektor.
- `map_chr()` skapar en text (character) vektor.

Funktionerna tar en vektor som input, tillämpar en funktion på varje element och returnerar en ny vektor av samma längd som input. Dessa funktioner underlättar väsentligt för att finna lösningar på itereringsproblem när man väl förstått hur man ska använda dem.

Vi kan använda mappningsfunktionerna för att göra samma beräkningar som den senaste for loopen ovan. Dessa summeringsfunktioner returnerade numeriska objekt (doubles) så vi behöver använda `map_dbl()`:

```
map_dbl(df, mean)

##      a      b      c      d
## 0.13459477 0.22621395 0.09949778 -0.31922665

map_dbl(df, median)

##      a      b      c      d
## 0.28728858 -0.08682806 -0.03412453 -0.38735340

map_dbl(df, sd)

##      a      b      c      d
## 0.7244117 0.8634665 0.9263836 0.9791581
```

Jämfört med att använda loopar fokuserar mappningsfunktionerna på den operation som ska göras, ex `mean()` eller `median()`, inte själva bokhållandet över varje element och lagringen av output. Detta blir än mer uppenbart då vi använder *pipes*:

```
df %>% map_dbl(mean)

##      a      b      c      d
## 0.13459477 0.22621395 0.09949778 -0.31922665

df %>% map_dbl(median)
```

```
##      a      b      c      d
## 0.28728858 -0.08682806 -0.03412453 -0.38735340

df %>% map_dbl(sd)

##      a      b      c      d
## 0.7244117 0.8634665 0.9263836 0.9791581
```

Det finns några små skillnader mellan `map_*()` och `col_summary()`:

- Samtliga purrr-funktioner är implementerade i programmeringsspråket C vilket gör att de är något snabbare på bekostnad av läsbarheten.
- Det andra argumentet, `.f`, funktionen som ska användas, kan vara en formel, en *character* vektor eller en numerisk vektor. Vi återkommer strax till dessa.
- `map_*()` använder ... för att göra det möjligt lägga till argument till `.f` varje gång funktionen åberopas:

```
map_dbl(df, mean, trim = 0.5)

##      a      b      c      d
## 0.28728858 -0.08682806 -0.03412453 -0.38735340
```

- Mapningsfunktionerna bevarar namnen:

```
z <- list(x = 1:3, y = 4:5)
```

```
map_int(z, length)

## x y
## 3 2
```

Genvägar

Det finns några genvägar som du kan använda med `.f` för att spara lite kod. Antag att du vill applicera en linjär modell för varje grupp i ett dataset. Följande exempel delar upp `mtcars`-data i tre delar (ett per värde för antalet cylindrar) och anpassar samma modell till varje del-dataset:

```
models <- mtcars %>%
  split(.$cyl) %>%
  purrr::map(function(df) lm(mpg ~ wt, data = df))
```

Syntaxen för att skapa en anonym funktion i R är ganska omständlig så i purrr finns en bekväm genväg: en enkelsidig formel:

```
models <- mtcars %>%
  split(.$cyl) %>%
  purrr::map(~lm(mpg ~ wt, data = .))
```

Här används `.` som ett *pronomen* – det refererar till det aktuella elementet i listan på samma sätt som `i` refererade till det aktuella indexet i loopen.

När du undersöker flera modeller kanske du vill extrahera en summering typ R^2 . För att göra det behöver vi först köra `summary()` och därefter extrahera komponenten som kallas `r.squared`. Vi kan göra det genom att använda kortversionen för anonyma funktioner:

```
models %>%
  purrr::map(summary) %>%
  map_dbl(~.$r.squared)

##      4      6      8
## 0.5086326 0.4645102 0.4229655
```

Men att extrahera namngivna komponenter är en gemensam operation så `purrr` har en ännu kortare genväg - en textsträng:

```
models %>%
  purrr::map(summary) %>%
  map_dbl("r.squared")

##      4      6      8
## 0.5086326 0.4645102 0.4229655
```

Du kan även använda en siffra för att välja ut elementen genom deras position:

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>%
  map_dbl(2)

## [1] 2 5 8
```

Hantera errors

När du använder mappningsfunktioner för att upprepa många operationer ökar chansen att någon av dessa operationer misslyckas och du får ett error och inget resultat. Varför förhindrar en misslyckad operation att du får åtkomst till alla lyckade?

Vi ska kika närmare på en funktion med vars hjälp errors kan hanteras lättare - `safely()`. Det är en funktion som returnerar ett modifierat output från en viss funktion. Den returnerar alltid en lista med två element:

1. result som är originalet och om operationen misslyckas returnerar NULL
2. Error – ett error-objekt som om operationen lyckas returnerar NULL

Vi kan illustrera detta med ett enkelt exempel med funktionen `log()`:


```
safe_log <- safely(log)
str(safe_log(10))

## List of 2
## $ result: num 2.3
## $ error : NULL

str(safe_log("a"))

## List of 2
## $ result: NULL
## $ error :List of 2
## ..$ message: chr "non-numeric argument to mathematical function"
## ..$ call : language .Primitive("log")(x, base)
## ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

safely() är gjort för att användas tillsammans med mappningsfunktionerna:

```
x <- list(1, 10, "a")

y <- x %>%
  purrr::map(safely(log))

str(y)

## List of 3
## $ :List of 2
## ..$ result: num 0
## ..$ error : NULL
## $ :List of 2
## ..$ result: num 2.3
## ..$ error : NULL
## $ :List of 2
## ..$ result: NULL
## ..$ error :List of 2
## ...$ message: chr "non-numeric argument to mathematical function"
## ...$ call : language .Primitive("log")(x, base)
## ...- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

När funktionen är felfri kommer result-elementet att innehålla resultatet och error-elementet är NULL. När funktionen inte fungerar kommer result-elementet att vara NULL och error-elementet ett error-objekt.

Detta kan vara enklare att hantera om vi hade en lista för resultaten och en lista för errors. Det kan vi få med `purrr::transpose()`:

```

y <- y %>% transpose()
str(y)

## List of 2
## $ result:List of 3
## ..$ : num 0
## ..$ : num 2.3
## ..$ : NULL
## $ error :List of 3
## ..$ : NULL
## ..$ : NULL
## ..$ :List of 2
## .. ..$ message: chr "non-numeric argument to mathematical function"
## .. ..$ call : language .Primitive("log")(x, base)
## .. ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"

```

Du bestämmer hur du vill hantera errors men vanligtvis vill du antingen undersöka de `x` för vilka `y` är error *eller* arbeta med de `y` som är OK.

```

is_ok <- y$error %>% map_lgl(is_null)

x[!is_ok]

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

y$result[is_ok] %>% flatten_dbl()

## [1] 0.000000 2.302585

```

`purrr` innehåller två användbara adverb:

- `possibly()` - enklare än `safely()` eftersom man anger ett default värde som returneras om error
- `quietly()` - istället för att fånga errors fångar `quietly()` printat output, messages och warnings:

```

x <- list(1, -1)
x %>% purrr::map(quietly(log)) %>% str()

## List of 2
## $ :List of 4
## ..$ result : num 0
## ..$ output : chr ""
## ..$ warnings: chr(0)
## ..$ messages: chr(0)
## $ :List of 4
## ..$ result : num NaN
## ..$ output : chr ""

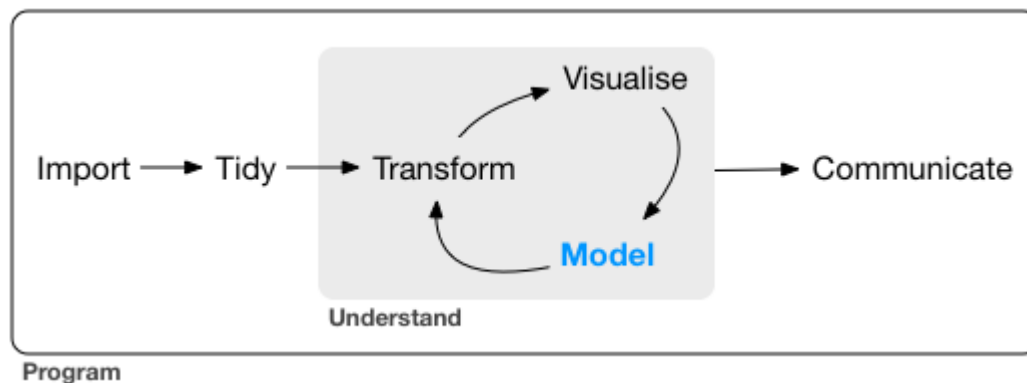
```

```
## ..$ warnings: chr "NaNs produced"  
## ..$ messages: chr(0)
```

Det finns ytterligare finesser med mappningsfunktioner som beskrivs i Wickhams bok. Rekommenderad läsning: <http://r4ds.had.co.nz/iteration.html#mapping-over-multiple-arguments>

Introduction

Detta avsnitt ska handla om modellering av data. Syftet i detta skede är snarare *explorativt* (att lära känna data) än analytiskt. Tanken är att vi ska kika på ett antal verktyg för att bättre förstå *variation* i datamängder. Det kommer inte att behandla statsitsisk teori utan fpokus ligger på hur R kan användas för att undersöka datamängder.



- I nästa del, *Modellering. Basics*, ska vi titta närmare på “mekaniken” bakom fr.a. linjära modeller.
- I *Bygga modeller i R* ska vi göra just det, och
- I *Flera modeller* ska vi nosa på hur man kan använda multipla modeller och därigenom få kombinera modellerings- och programmerings-verktyg.

Modellering: basics

Wickhams bok innehåller en del statistisk teori om modellering vilket vi här kommer att lämna därhän. För den som är intresserad av att ta del av detta hänvisas till <http://r4ds.had.co.nz/model-intro.html>

Vi kommer istället att lägga fokus på hur man skapar statistiska modeller i R/Rstudio hands-on alltså. Vi ska börja med att känna på R:s grundläggande begrepp för att bygga upp modeller med hjälp av simulerade data för att senare arbeta mer praktiskt med verkliga dataset.

Det finns två delar i en *modell* i R:

1. Först behöver du definiera vilken modell-familj (*family of models*) som ska användas. Det handlar om vilken typ av funktion som bäst beskriver dina data, det kan vara en rät linje eller en kvadratisk eller en polynomisk.
2. Du använder denna funktion till att hitta den modell som bäst beskriver dina data.

Vi laddar in modulerna som vi ska använda:

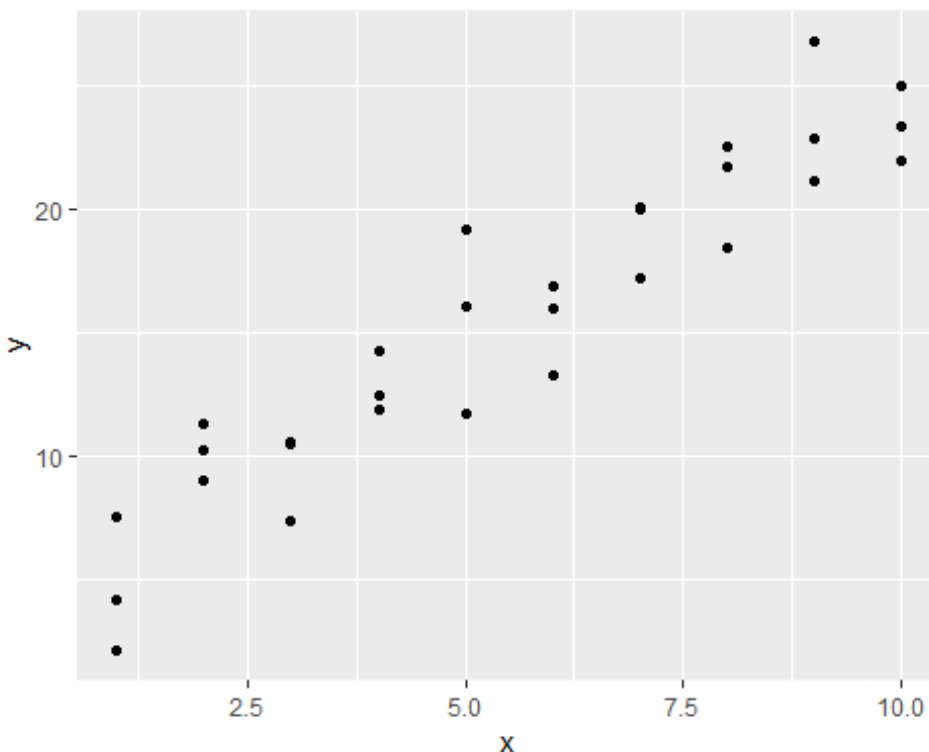
```
library(tidyverse)
library(modelr)
options(na.action = na.warn)
```

Modulen `modelr` innehåller en rad funktioner för att få R:s bas-funktioner för modellering att fungera i pipes.

En enkel modell

Låt oss titta på ett simulerat dataset, `sim1`, som finns i `modelr`. Det innehåller två kontinuerliga variabler, `x` och `y`, vilka vi plottar för att se hur de är relaterade:

```
ggplot(sim1, aes(x, y)) +
  geom_point()
```



Här finns ett tydligt mönster. Vi ska hitta en modell som fångar detta mönster explicit. Relationen mellan x och y förefaller vara linjär vilket kan skrivas $y = a_0 + b_1 \cdot x$.

R har ett specifikt verktyg för linjära modeller `lm()`, *linear model*. Som argument i detta verktyg används *formler* för att specificera modellen. Formler i R har formen $y \sim x$. Prova:

```
sim1_mod <- lm(y ~ x, data = sim1)
summary(sim1_mod)

##
## Call:
## lm(formula = y ~ x, data = sim1)
##
## Residuals:
##   Min     1Q   Median     3Q    Max
## -4.1469 -1.5197  0.1331  1.4670  4.6516
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  4.2208    0.8688   4.858 4.09e-05 ***
## x           2.0515    0.1400  14.651 1.17e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 2.203 on 28 degrees of freedom
## Multiple R-squared:  0.8846, Adjusted R-squared:  0.8805
## F-statistic: 214.7 on 1 and 28 DF,  p-value: 1.173e-14
```

Visualisera modeller

Prediktioner

Vi ska kika lite på prediktioner som ett sätt att förstå en modell. Vi börjar med att generera en “grid” av värden som täcker in det intervall som täcker våra data. Det enklaste sättet att göra det på är att använda `modelr::data_grid()`. Dess första argument är en dataram, och för varje ytterligare argument finner funktionen de unika variablerna och genererar alla kombinationer:

```
grid <- sim1 %>%
  data_grid(x)
grid

## # A tibble: 10 x 1
##   x
##   <int>
## 1     1
## 2     2
## 3     3
## 4     4
## 5     5
## 6     6
## 7     7
## 8     8
## 9     9
## 10    10
```

Därefter adderar vi prediktionerna med hjälp av `modelr::add_predictions()`. Den tar en dataram och en modell och lägger till prediktionerna givet data och modellen:

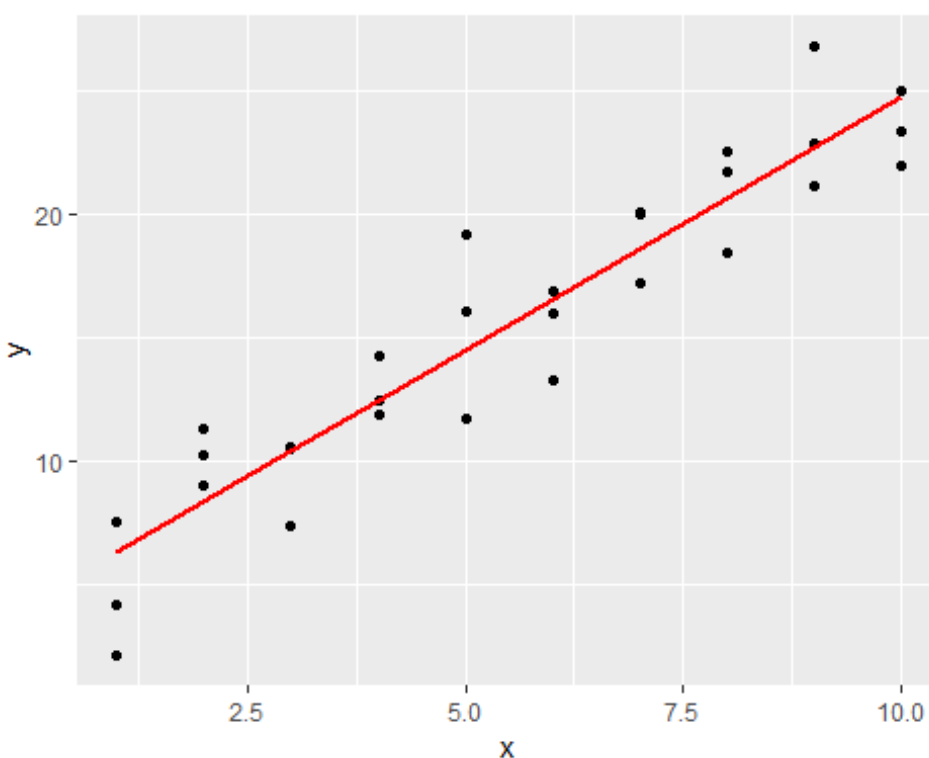
```
grid <- grid %>%
  add_predictions(sim1_mod)
grid

## # A tibble: 10 x 2
##   x pred
##   <int> <dbl>
## 1     1  6.27
## 2     2  8.32
## 3     3 10.4
## 4     4 12.4
```

```
## 5  5 14.5
## 6  6 16.5
## 7  7 18.6
## 8  8 20.6
## 9  9 22.7
## 10 10 24.7
```

Sedan plottar vi prediktionerna. Vi gör det tillsammans med de observerade värdena.

```
ggplot(sim1, aes(x)) +
  geom_point(aes(y = y)) +
  geom_line(aes(y = pred), data = grid, colour = "red", size = 1)
```



Man kan även använda funktionen `geom_abline()` för att visualisera en linjär regressionslinje men fördelen med att använda `add_predictions()` är att denna funktion fungerar för vilken modell som helst, oavsett komplexitet. För fler sätt att visualisera komplexa modeller se gärna

<http://vita.had.co.nz/papers/model-vis.html>.

Residualer

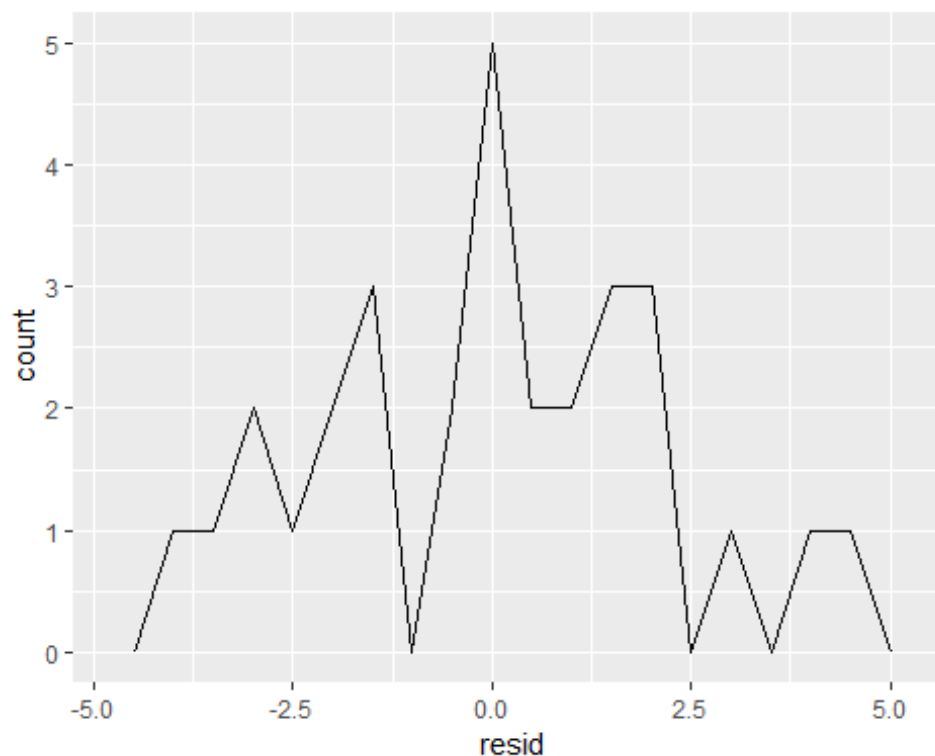
Residualer är komplementet till prediktioner och är differensen mellan de observerade värdena och predicerade värdena. Vi lägger till residualerna med hjälp av `add_residuals()`, som fungerar ungefär som `add_predictions()`. Men här behövs förstås det ursprungliga data där y-värdena finns:


```
sim1 <- sim1 %>%
  add_residuals(sim1_mod)
sim1

## # A tibble: 30 x 3
##   x   y resid
##   <int> <dbl> <dbl>
## 1     1  4.20 -2.07
## 2     1  7.51  1.24
## 3     1  2.13 -4.15
## 4     2  8.99  0.665
## 5     2 10.2   1.92
## 6     2 11.3   2.97
## 7     3  7.36 -3.02
## 8     3 10.5   0.130
## 9     3 10.5   0.136
## 10    4 12.4   0.00763
## # ... with 20 more rows
```

Det finns några olika sätt att förstå vad residualerna ger för information om modellen vi valt. Ett sätt är att göra en *frekvenspolygon* för att se spridningen av residualerna:

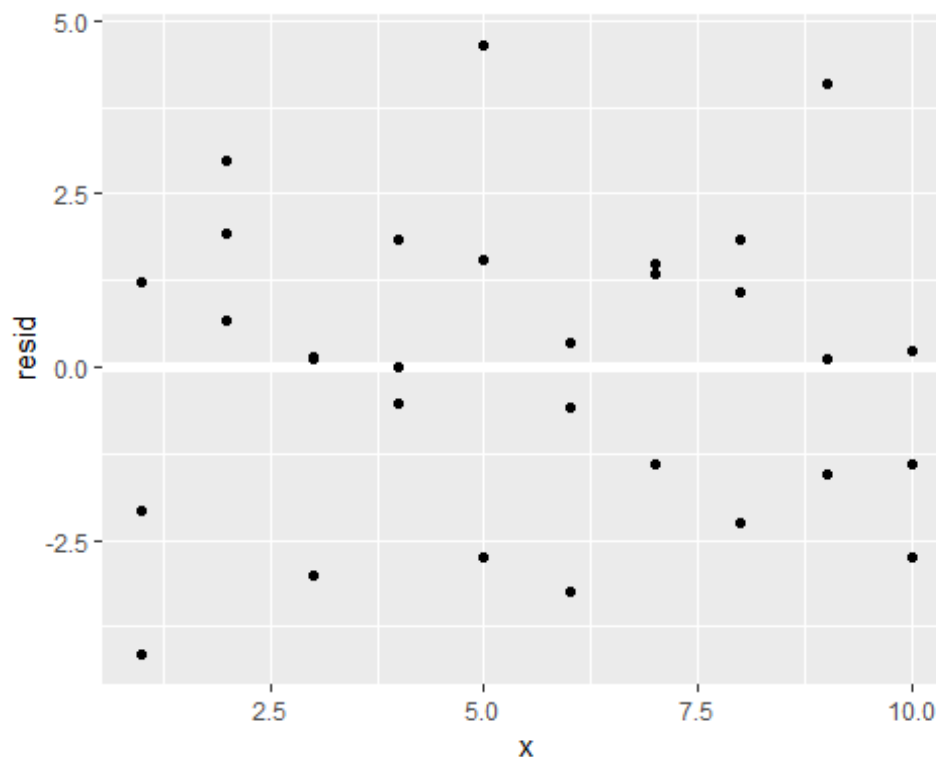
```
ggplot(sim1, aes(resid)) +
  geom_freqpoly(binwidth = 0.5)
```



Här framgår hur långt från regressionslinjen (prediktionerna) finns de observerade värdena.

För att kunna se residualerna hellre än prediktionerna:

```
ggplot(sim1, aes(x, resid)) +  
  geom_ref_line(h = 0) +  
  geom_point()
```



Residualerna förefaller fördelas slumpmässigt vilket talar för att modellen fångar essensen i data ganska väl.

Formler och modell-familjer

Formler i R är av en speciell karaktär. Formler i R är ett generellt sätt att fånga "ett speciellt beteende". Snarare än att utvärdera variabelvärden fångar R-formeln in data så att de kan tolkas av den funktion man vill använda.

De flesta modellfunktioner i R använder ett standardiserat sätt att omvandla formler till funktioner. Omvandlingen vi använde innan, $y \sim x$ omvandlas av funktionen `lm()` till $y = a + b_1 \cdot x$. Det framgår tydligare om vi tar fram modell-matrisen med hjälp av funktionen `model_matrix()`. Den tar en dataram och en formel och genererar en tibble som definierar modellens ekvation: varje kolumn i tibblen är associerad till en koefficient i modellen. Vi kollar i den enklaste modellen, $y \sim x1$:

```
df <- tribble(  
  ~y, ~x1, ~x2,
```

```

4, 2, 5,
5, 1, 6
)

model_matrix(df, y ~ x1)

## # A tibble: 2 x 2
##   `(Intercept)` x1
##   <dbl> <dbl>
## 1      1      2
## 2      1      1

```

Lägger man till fler variabler till modellen utvidgas matrisen:

```

model_matrix(df, y ~ x1 + x2)

## # A tibble: 2 x 3
##   `(Intercept)` x1 x2
##   <dbl> <dbl> <dbl>
## 1      1      2      5
## 2      1      1      6

```

Sättet som R lägger till interceptet till modellen är via en kolumn fylld med värdet 1. För den som är intresserad av algebra kan rekommenderas *Wilkinson & Rogers Symbolic Description of Factorial Models for Analysis of Variance*, <https://www.jstor.org/stable/2346786> som beskriver och motiverar detta sätt att definiera ekvationen och modellmatrisen.

Låt oss kika på hur detta fungerar för kategoriska variabler, interaktioner och transformeringar.

Kategoriska variabler

Låt oss säga att vi har en formel $y \sim \text{sex}$, där sex kan vara antingen man eller kvinna. I detta fall är ju sex en kategorisk variabel vilken i R konverteras till dummyvariabler, i detta fall i modellen $y \sim a + b_1 * \text{sex_man}$ där sex_man är 1 om sex är en man och 0 om sex är en kvinna:

```

df <- tribble(
  ~ sex, ~ response,
  "male", 1,
  "female", 2,
  "male", 1
)

model_matrix(df, response ~ sex)

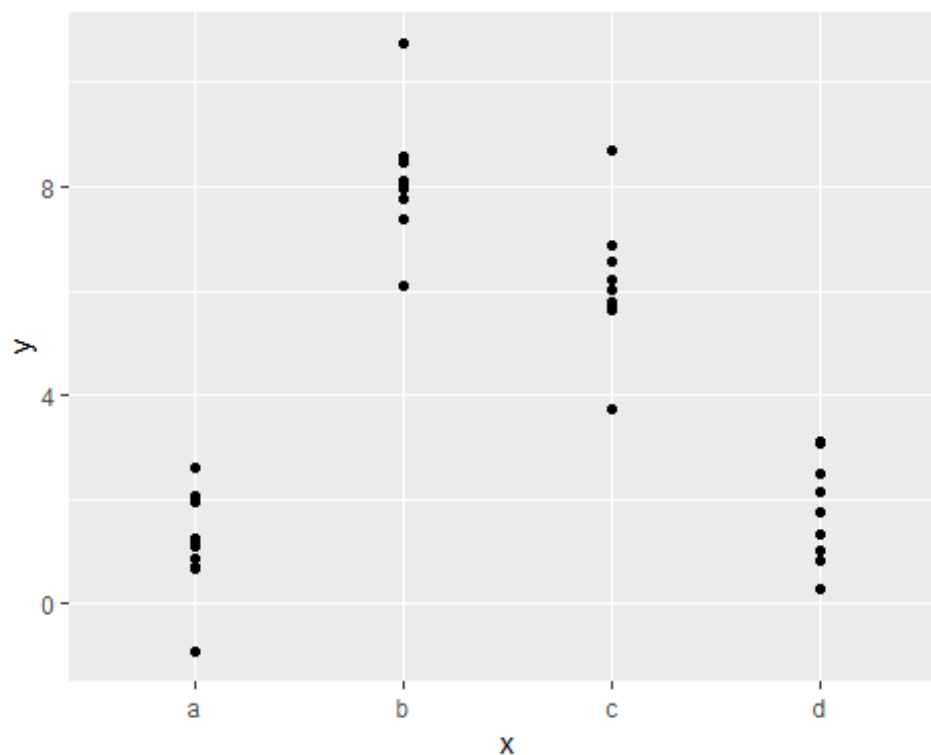
## # A tibble: 3 x 2
##   `(Intercept)` sexmale
##   <dbl> <dbl>

```

```
## 1      1      1
## 2      1      0
## 3      1      1
```

Vi kan utveckla detta resonemang med hjälp av `sim2`:

```
ggplot(sim2) +
  geom_point(aes(x, y))
```



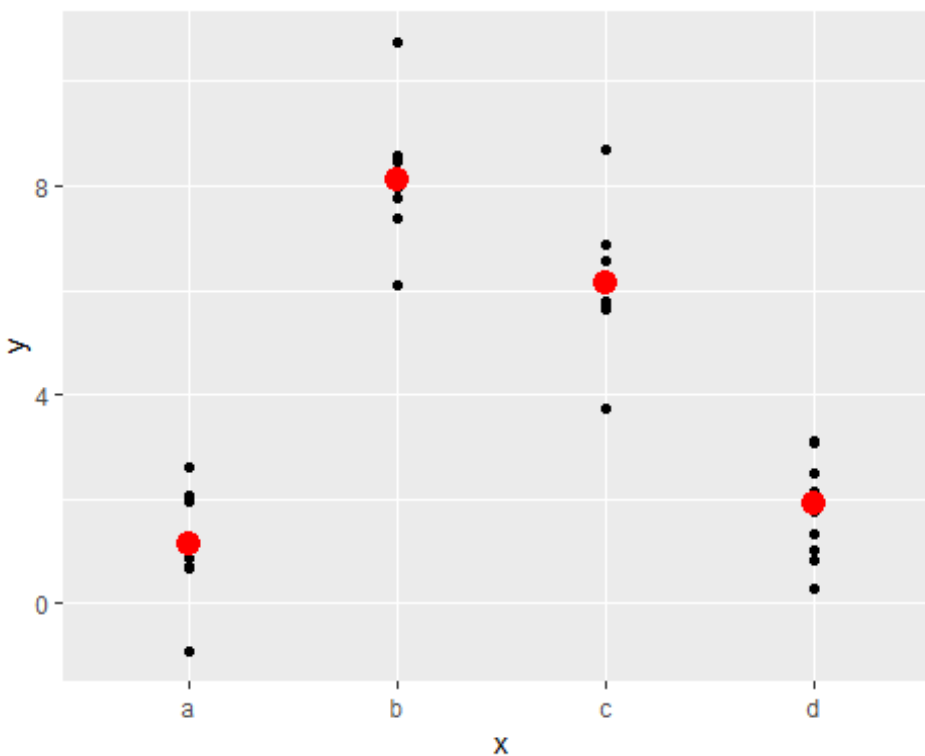
Vi kan anpassa en modell till dessa data och generera prediktioner:

```
mod2 <- lm(y ~ x, data = sim2)
grid <- sim2 %>%
  data_grid(x) %>%
  add_predictions(mod2)
grid

## # A tibble: 4 x 2
##   x     pred
##   <chr> <dbl>
## 1 a     1.15
## 2 b     8.12
## 3 c     6.13
## 4 d     1.91
```

Prediktionerna från en modell med kategoriska data kommer att vara lika med kategoriernas medelvärden eftersom dessa minimerar prediktionernas summerade avvikelse från modellens regressionlinje the root mean squared distance. Det blir uppenbart genom att lägga till medelvärden för kategorierna i grafen:

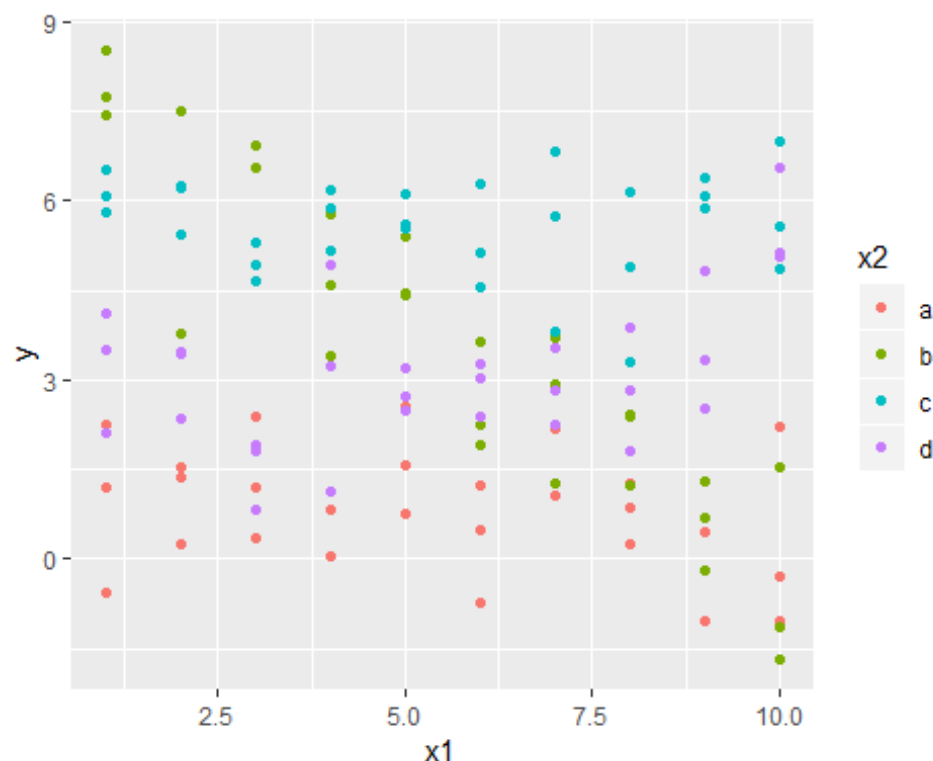
```
ggplot(sim2, aes(x)) +  
  geom_point(aes(y = y)) +  
  geom_point(data = grid, aes(y = pred), colour = "red", size = 4)
```



Interaktioner

Vad händer när du kombinerar en kontinuerlig och en kategorisk variabel? I `sim3` finns en kategorisk variabel och en kontinuerlig:

```
ggplot(sim3, aes(x1, y)) +  
  geom_point(aes(colour = x2))
```



Dessa data kan beskrivas på två sätt:

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

När man adderar två variabler med `+` kommer modellen att utvärdera effekten för varje variabel oberoende av de övriga. När man lägger till variabler med `*` utvärderas interaktionen mellan de två variablerna. Till exempel tolkas formeln $y \sim x1 * x2$ som $y = a + b1*x1 + b2*x2 + b3*x1*x2$. Använder man `*` istället för `+` inkluderas alltså både de enskilda variablerna och interaktionsvariabeln.

För att visualisera dessa modeller behöver vi justera koden från det tidigare exemplet:

1. Vi har två variabler så vi behöver ge `data_grid()` båda. Då kommer funktionen att ange samtliga unika värden på `x1` och `x2` och generera alla kombinationer av dessa.
2. För att generera prediktioner från båda modellerna samtidigt kan vi använda `gather_predictions()` som lägger till varje prediktion som en rad. Komplementet till `gather_predictions()` är `spread_predictions()` som lägger till varje prediktion till en ny kolumn.

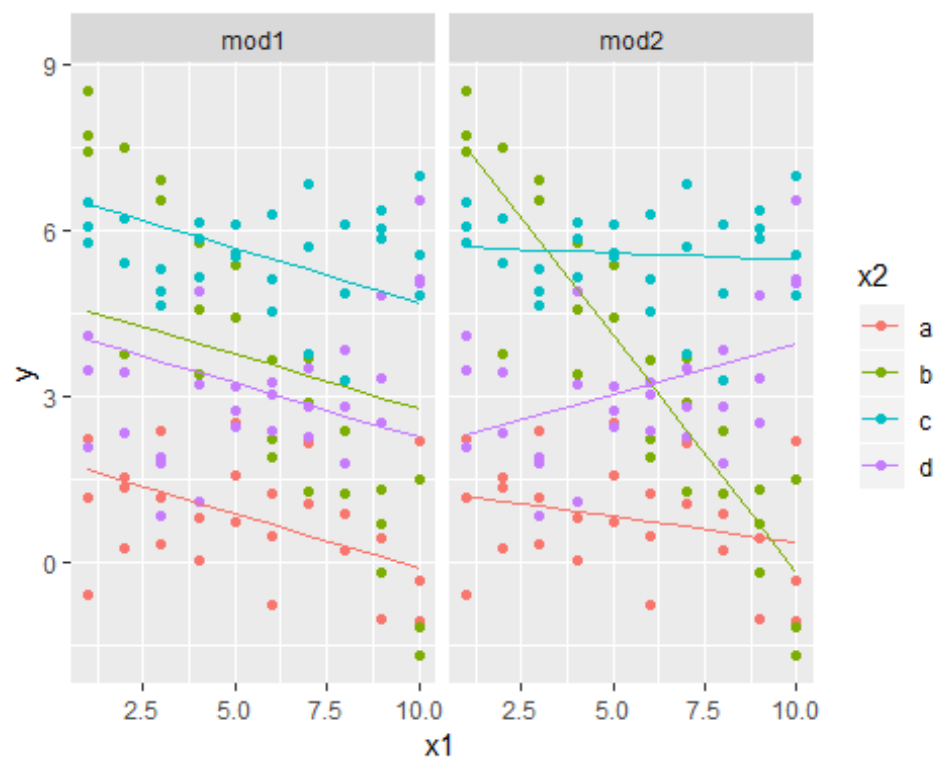
Sammantaget ger oss detta:

```
grid <- sim3 %>%
  data_grid(x1, x2) %>%
  gather_predictions(mod1, mod2)
grid
```

```
## # A tibble: 80 x 4
##   model x1 x2  pred
##   <chr> <int> <fct> <dbl>
## 1 mod1    1 a   1.67
## 2 mod1    1 b   4.56
## 3 mod1    1 c   6.48
## 4 mod1    1 d   4.03
## 5 mod1    2 a   1.48
## 6 mod1    2 b   4.37
## 7 mod1    2 c   6.28
## 8 mod1    2 d   3.84
## 9 mod1    3 a   1.28
## 10 mod1   3 b   4.17
## # ... with 70 more rows
```

Vi kan visualisera resultaten för båda modellerna genom att använda facets:

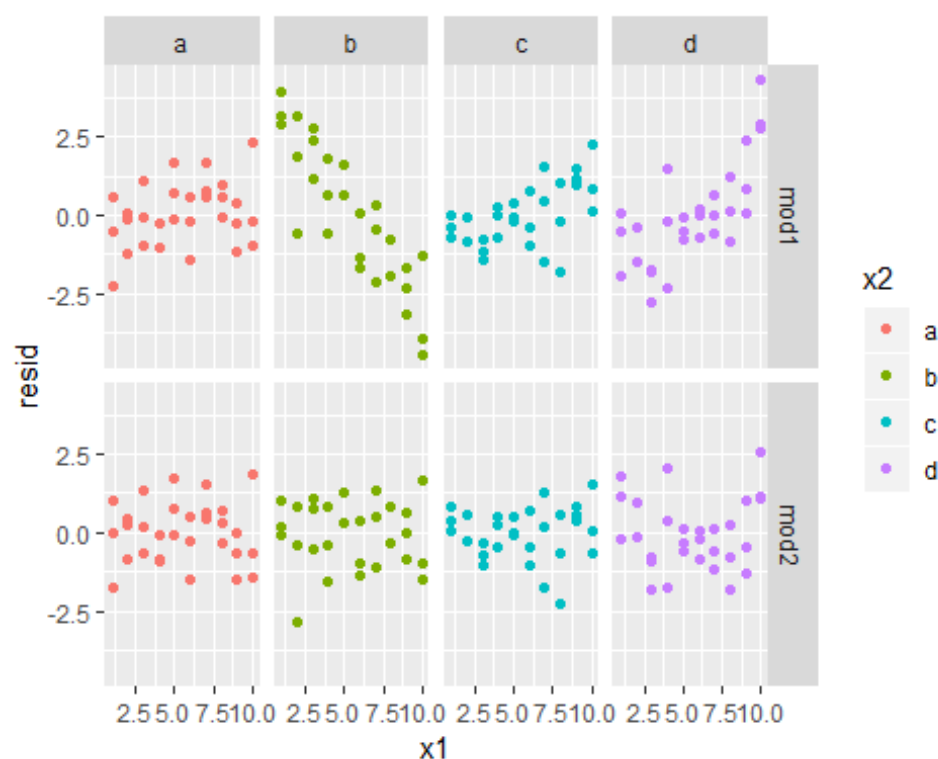
```
ggplot(sim3, aes(x1, y, colour = x2)) +
  geom_point() +
  geom_line(data = grid, aes(y = pred)) +
  facet_wrap(~ model)
```



Notera att modellen som använder + har samma lutning för varje regressionslinje men skilda intercept, medan modellen med interaktionsvariabeln (*) har regressionslinjer med olika lutning.

Vilken modell passar data bäst? Vi kan kika på residualerna. Vi använder facets för modell och x2 för att kunna se mönster inom varje grupp:

```
sim3 <- sim3 %>%
  gather_residuals(mod1, mod2)
ggplot(sim3, aes(x1, resid, colour = x2)) +
  geom_point() +
  facet_grid(model ~ x2)
```



Det finns inte något uppenbart mönster beträffande residualerna för de olika kategorierna a-d i modell 2. Residualerna i modell 1 uppvisar knappast något slumpmässigt mönster; i b är detta uppenbart men även för de övriga kategorierna, vilket kan tolkas som att modell 1 återspeglar data sämre än modell 2.

Interaktioner (två kontinuerliga variabler)

Vi kikar på motsvarande modell för två kontinuerliga variabler. Vi kan i princip upprepa koden från ovanstående exempel till att börja med:

```
mod1 <- lm(y ~ x1 + x2, data = sim4)
mod2 <- lm(y ~ x1 * x2, data = sim4)
grid <- sim4 %>%
  data_grid(
```



```

x1 = seq_range(x1, 5),
x2 = seq_range(x2, 5)
) %>%
gather_predictions(mod1, mod2)
grid

## # A tibble: 50 x 4
##   model x1  x2 pred
##   <chr> <dbl> <dbl> <dbl>
## 1 mod1 -1  -1  0.996
## 2 mod1 -1  -0.5 -0.395
## 3 mod1 -1   0 -1.79
## 4 mod1 -1   0.5 -3.18
## 5 mod1 -1   1 -4.57
## 6 mod1 -0.5 -1  1.91
## 7 mod1 -0.5 -0.5 0.516
## 8 mod1 -0.5  0 -0.875
## 9 mod1 -0.5  0.5 -2.27
## 10 mod1 -0.5  1 -3.66
## # ... with 40 more rows

```

Notera funktionen `seq_range()` som ett argument i `data_grid()`. Istället för att använda varje unikt värde i `x` använder vi en regelbunden *grid* bestående av fem värden inom intervallet minsta och högsta värde på `x`. Det finns två andra användbara argument till `seq_range()`:

1. `pretty = TRUE` genererar en “pretty” sekvens, dvs som ser snyggt ut för ögat. Detta kan vara bra om man vill skapa tabeller från output.

```
seq_range(c(0.0123, 0.923423), n = 5)
```

```
## [1] 0.0123000 0.2400808 0.4678615 0.6956423 0.9234230
```

```
seq_range(c(0.0123, 0.923423), n = 5, pretty = TRUE)
```

```
## [1] 0.0 0.2 0.4 0.6 0.8 1.0
```

2. `trim = 0.1` klipper 10% av svansvärdena vilket kan vara användbart om du är mer intresserad av värden nära centralvärden i en skev fördelning.

```
x1 <- rcauchy(100)
```

```
seq_range(x1, n = 5)
```

```
## [1] -7.466945 1.884934 11.236813 20.588691 29.940570
```

```
seq_range(x1, n = 5, trim = 0.10)
```

```
## [1] -4.3969060 -2.3733371 -0.3497682 1.6738007 3.6973697
```

```
seq_range(x1, n = 5, trim = 0.25)
```

```
## [1] -2.2531003 -1.2959249 -0.3387495 0.6184259 1.5756013
```

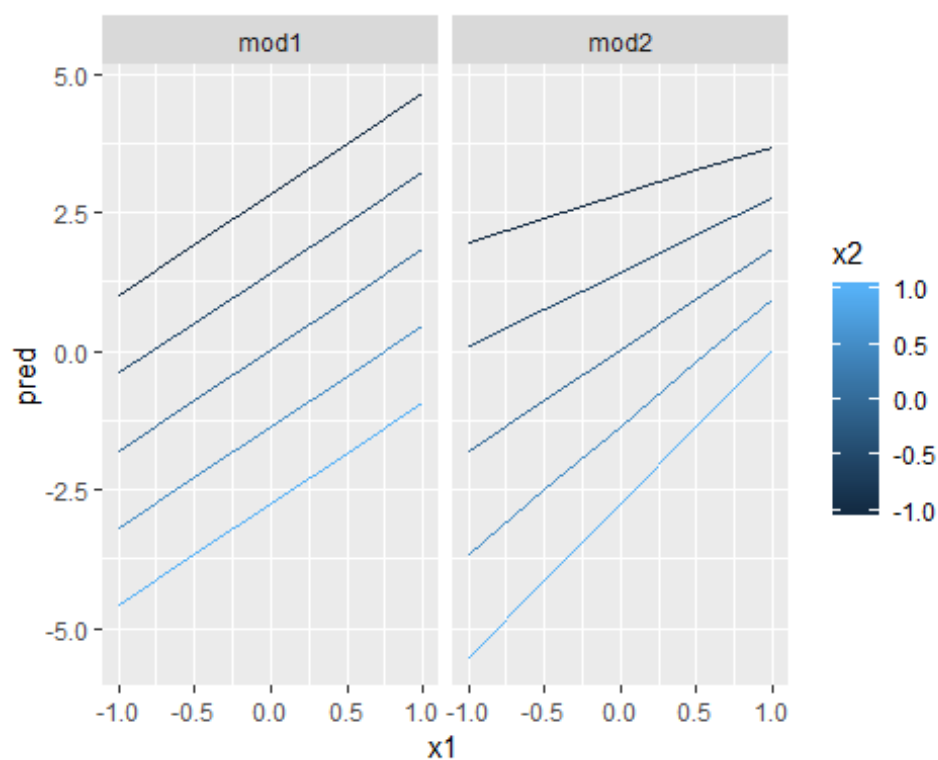
```
seq_range(x1, n = 5, trim = 0.50)
```

```
## [1] -1.0416636 -0.5806571 -0.1196506 0.3413559 0.8023624
```

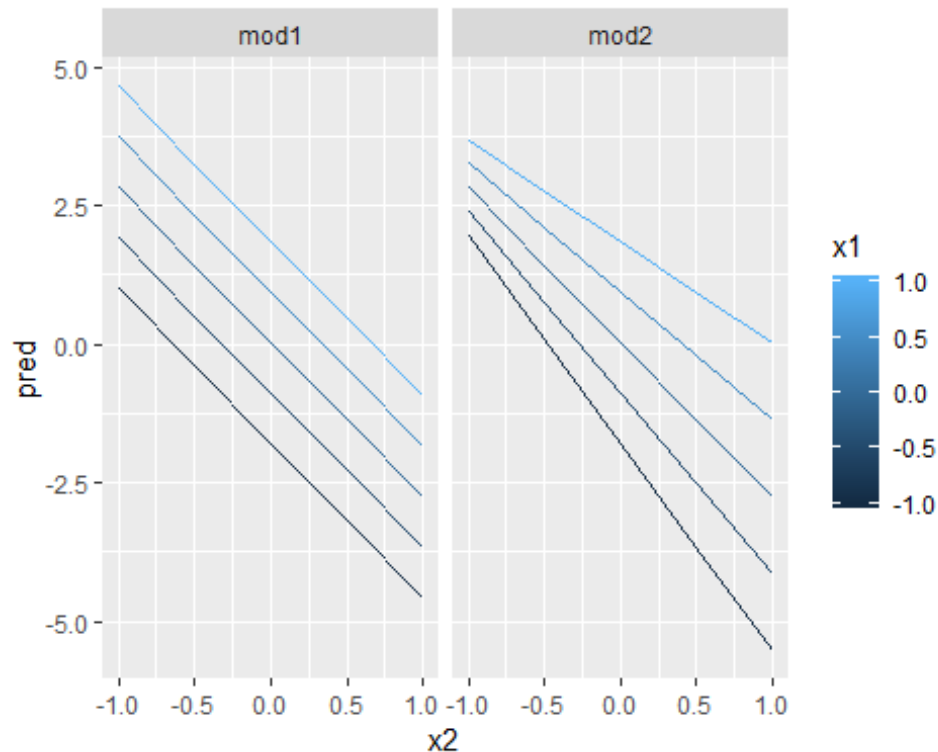
3. `expand = 0.1` är på sätt o vis motsatsen till `trim` - det expanderar omfånget (the range) med 10%

Låt oss visualisera den här modellen. Vi kan plotta prediktionerna för en av variablerna givet ett visst värde för den andra:

```
ggplot(grid, aes(x1, pred, colour = x2, group = x2)) +  
  geom_line() +  
  facet_wrap(~ model)
```



```
ggplot(grid, aes(x2, pred, colour = x1, group = x1)) +  
  geom_line() +  
  facet_wrap(~ model)
```



Detta visar att interaktioner med kontinuerliga variabler fungerar i princip på samma sätt som för kategoriska - för att predicera y behöver man beakta värdena för båda variablerna simultant.

Transformationer

Du kan även utföra transformationer inuti formeln för modellen. Till exempel, $\log(y) \sim \sqrt{x_1} + x_2$ transformeras till $\log(y) = a + b_1 \cdot \sqrt{x_1} + b_2 \cdot x_2$.

Om din transformation innehåller $+$, $-$, $*$ eller $^$ behöver du "paketera" uttrycket i funktionen $I()$ så att R inte behandlar uttrycket som en del av modell-specifikationen. Till exempel:

$y \sim x + I(x^2)$ tolkas som $y = a + b_1 \cdot x + b_2 \cdot x^2$. om du glömmet $I()$ och specificerar modellen till $y \sim x^2 + x$ kommer R att tolka modellen som $y = x * x + x$ där $x * x$ är i R en interaktion av x med sig själv vilket är lika med x och därmed blir modellen enligt R:s sätt att se $y = x + x$. Eftersom R automatiskt droppar redundanta variabler kommer uttrycket $y \sim x^2 + x$ att tolkas som $y = a + b_1 \cdot x$ vilket inte är vad vi ville.

Om du funderar på hur R tolkar ett uttryck kan du använda `model_matrix()` för att se vilken ekvation som `lm()` utvärderar:

```
df <- tribble(
  ~y, ~x,
  1, 1,
  2, 2,
  3, 3
```

```
)
model_matrix(df, y ~ x^2 + x)

## # A tibble: 3 x 2
##   `(Intercept)`  x
##   <dbl> <dbl>
## 1      1      1
## 2      1      2
## 3      1      3

model_matrix(df, y ~ l(x^2) + x)

## # A tibble: 3 x 3
##   `(Intercept)` `l(x^2)`  x
##   <dbl> <dbl> <dbl>
## 1      1      1      1
## 2      1      4      2
## 3      1      9      3
```

Wickham har några fler finesser beträffande transformationer i sin bok, se <http://r4ds.had.co.nz/model-basics.html#transformations>, men vi lämnar dem därhän för nu.

Missing values

Eftersom missing values inte innehåller någon användbar information om relationen mellan variabler droppar R automatiskt rader med sådana värden. Man får dock en varning om att detta har skett, t.ex.:

```
df <- tribble(
  ~x, ~y,
  1, 2.2,
  2, NA,
  3, 3.5,
  4, 8.3,
  NA, 10
)
mod <- lm(y ~ x, data = df)

## Warning: Dropping 2 rows with missing values
```

För att undertrycka varningen kan du ange argumentet `na.action = na.exclude`:

```
mod <- lm(y ~ x, data = df, na.action = na.exclude)
```

Du kan alltid se hur många observationer som använts i `lm()` genom `nobs()`:

```
nobs(mod)
```

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

Andra modell-familjer

R hanterar förstås andra än linjära modeller som vi just diskuterat t.ex.:

- *Generaliserade linjära modeller (GLM)* - t.ex. `stats::glm()`. GLM hanterar icke-kontinuerliga utfall såsom binära eller antal (counts).
- *Generaliserade additiva modeller (GAM)* - t.ex. `mgcv::gam()` vilka utvecklar GLMs till att inkludera godtyckliga smoothing-funktioner. Det innebär att du kan ange en formel $y \sim s(x)$ vilket blir tolkat som en ekvation av typ $y = f(x)$ och låter `gam()` estimerar vad den funktionen är.
- *Penalised linear models, robust linear models, trees, random forests* och ett antal ytterligare. Men det är överkurs här. Det är emellertid på sin plats att påpekat att modellerna i R fungerar på i princip ett likartat sätt vilket innebär att när du väl bemästrar linjära modeller kommer du att känna dig hemma när du ska hantera övriga, mer komplexa modeller.

Bygga modeller i R

Nu ska vi ägna tid åt att arbeta med reella data för att förstå hur R kan användas för att successivt bygga upp modeller för en ökad förståelse av data.

Modelleringen handlar ofta om att förstå dels mönster och samband mellan variabler och utfall samt om hur residualer förhåller sig till sambanden. Vi kan finna mönster och samband med hjälp av visualiseringar och konkretisera dem genom att testa olika modeller. Vi upprepar visualisering och modellbygge men ersätter utfallsvariabeln med residualer från modellen och lär oss successivt mer om data - målet är att gå från en slags implicit kunskap om data till mer precis, exakt kunskap som genereras från allt bättre kvantitativa modeller.

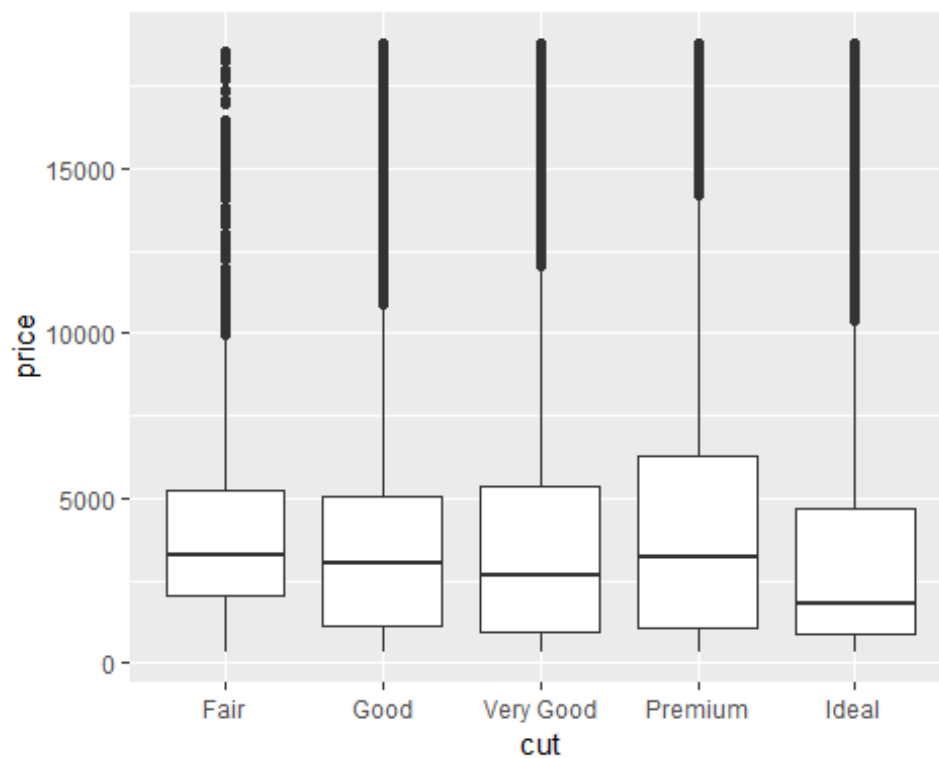
Vi använder samma verktyg som i föregående avsnitt men lägger till ett par verkliga dataset: `diamonds` från `ggplot2` och `flights` från `nycflights13`. Eftersom vi också ska arbeta med datum laddar vi in `lubridate`:

```
library(tidyverse)
library(modelr)
options(na.action = na.warn)
library(nycflights13)
library(lubridate)
```

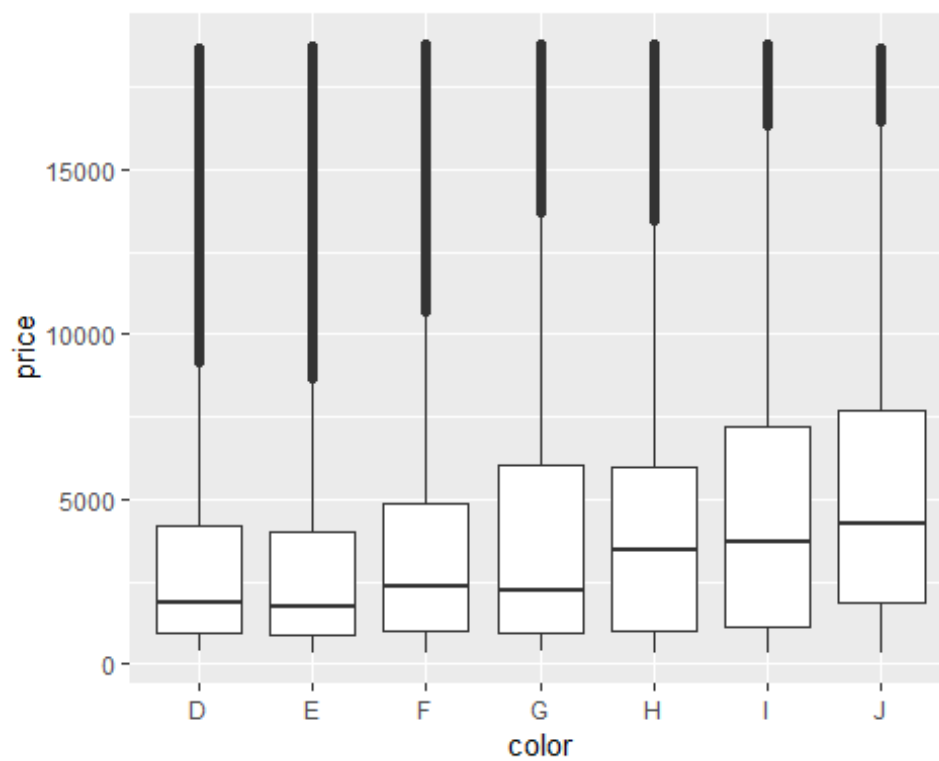
Varför är diamanter av låg kvalitet dyrare?

Data i `diamonds` talar för att diamanter av lägre kvalitet förefaller vara dyrare:

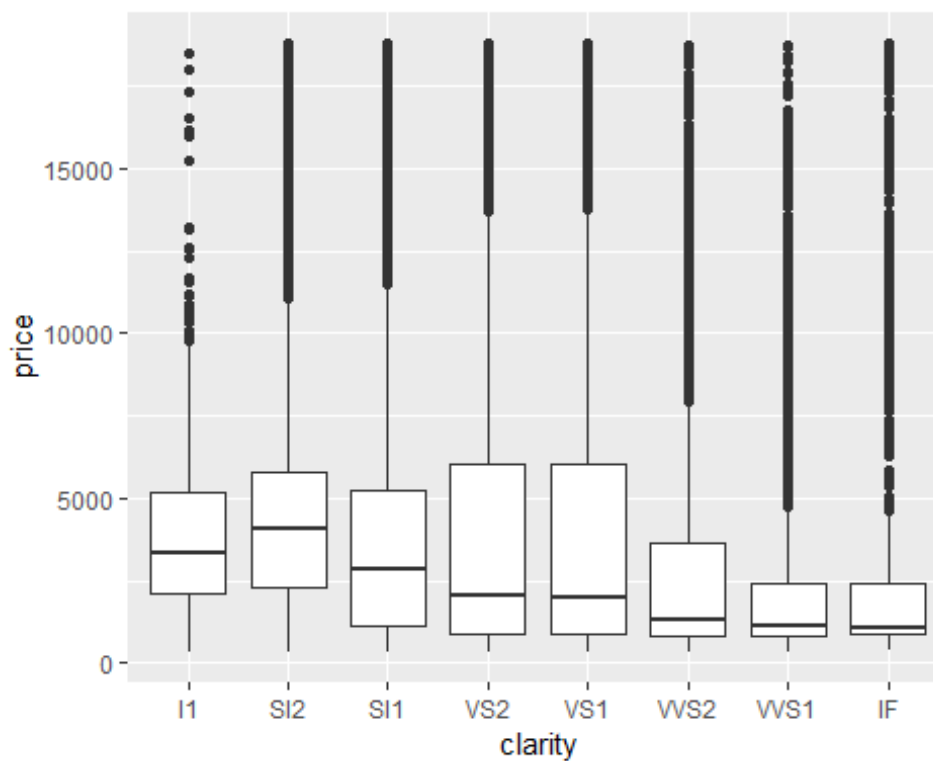
```
ggplot(diamonds, aes(cut, price)) + geom_boxplot()
```



```
ggplot(diamonds, aes(color, price)) + geom_boxplot()
```



```
ggplot(diamonds, aes(clarity, price)) + geom_boxplot()
```

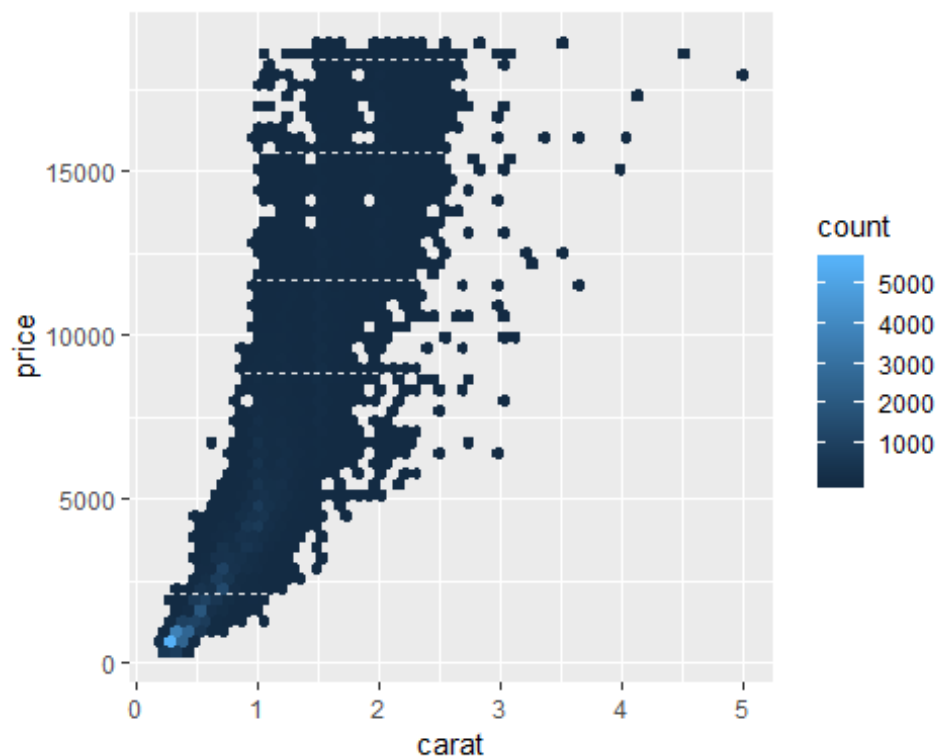


Notera att den sämsta diamant-färgen är svagt gulaktig (J) och den sämsta klarhetsgraden är synliga artefakter (I1).

Sambandet pris och karat

Det ser ut som att diamanter med lägre kvalitet har högre priser på grund av en viktig confounder, nämligen vikten (carat) på diamanten - vikten på diamanten är den enskilt viktigaste faktorn för priset och diamanter med lägre kvalitet tenderar att vara större.

```
ggplot(diamonds, aes(carat, price)) +  
  geom_hex(bins = 50)
```

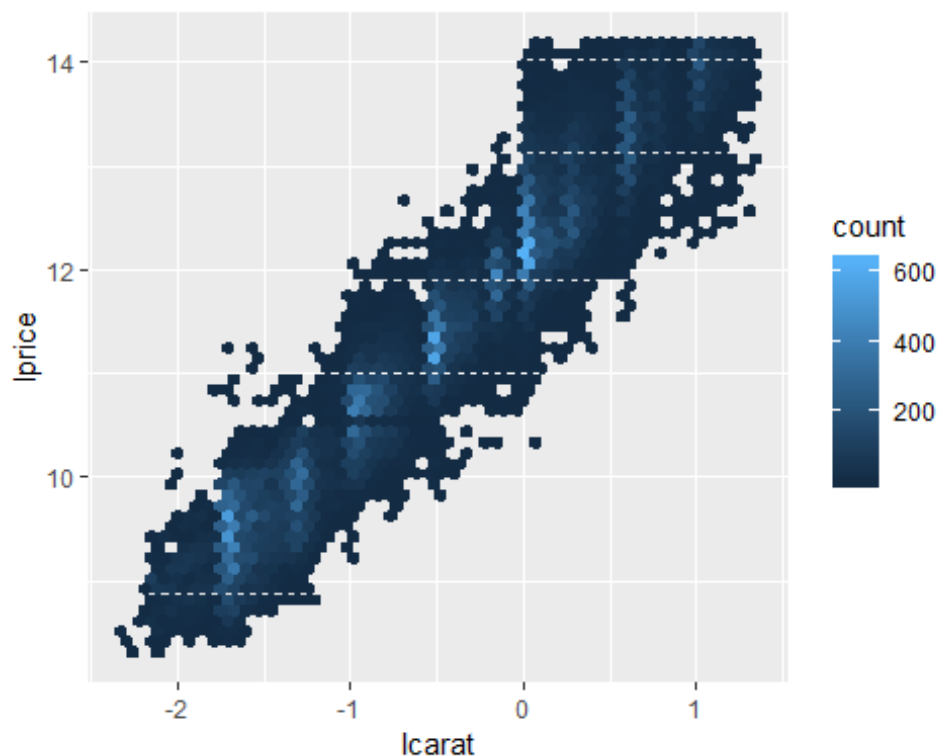
För att se hur övriga kvaliteter påverkar priset kan vi bygga en modell vilken justerar för vikt (carat). Men först ska vi förenkla datasetet för att göra det lättare att arbeta med:

1. Fokusera på diamanter mindre än 2,5 karat (vilka utgör 99,7% av data).
2. Logaritmerar variablerna carat och price.

```
diamonds2 <- diamonds %>%  
  filter(carat <= 2.5) %>%  
  mutate(lprice = log2(price), lcarat = log2(carat))
```

Sammantaget blir det lättare att se sambanden mellan carat och price:

```
ggplot(diamonds2, aes(lcarat, lprice)) +  
  geom_hex(bins = 50)
```



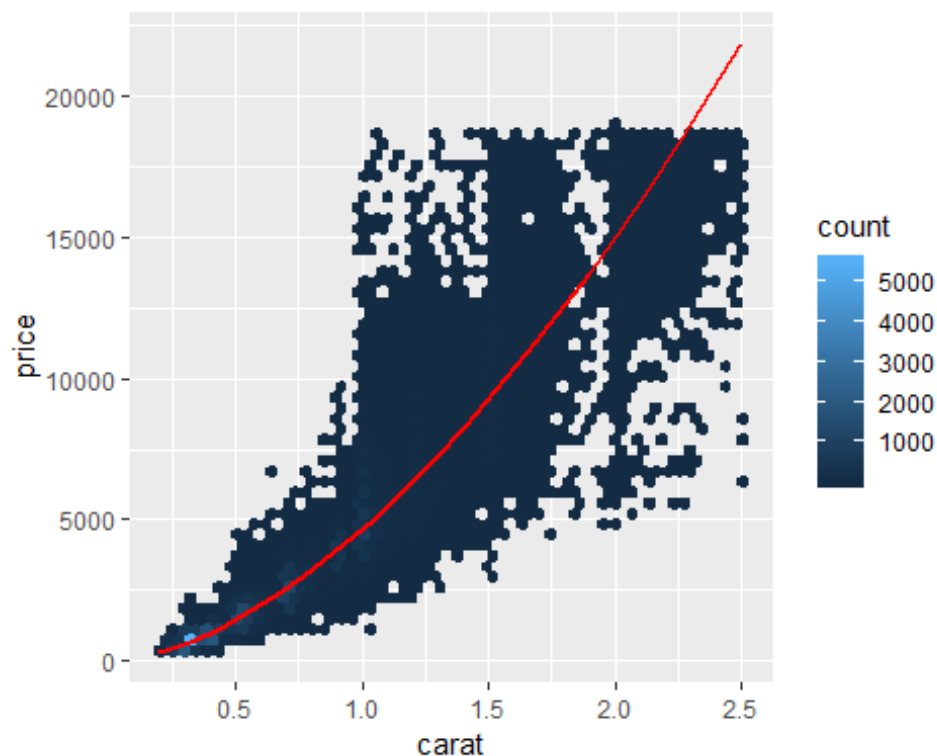
Log-transformeringen medför att mönstret blir linjärt vilket gör den mer lättare att arbeta med. Nu ska vi eliminera det linjära sambandet. Först tydliggör vi det linjära sambandet genom att skapa en modell:

```
mod_diamond <- lm(lprice ~ lcarat, data = diamonds2)
```

Sedan undersöker vi vad modellen säger om data. OBS att vi här använder icke-logaritmerade data för att kunna lägga på prediktionerna på rå-data:

```
grid <- diamonds2 %>%
  data_grid(carat = seq_range(carat, 20)) %>%
  mutate(lcarat = log2(carat)) %>%
  add_predictions(mod_diamond, "lprice") %>%
  mutate(price = 2 ^ lprice)

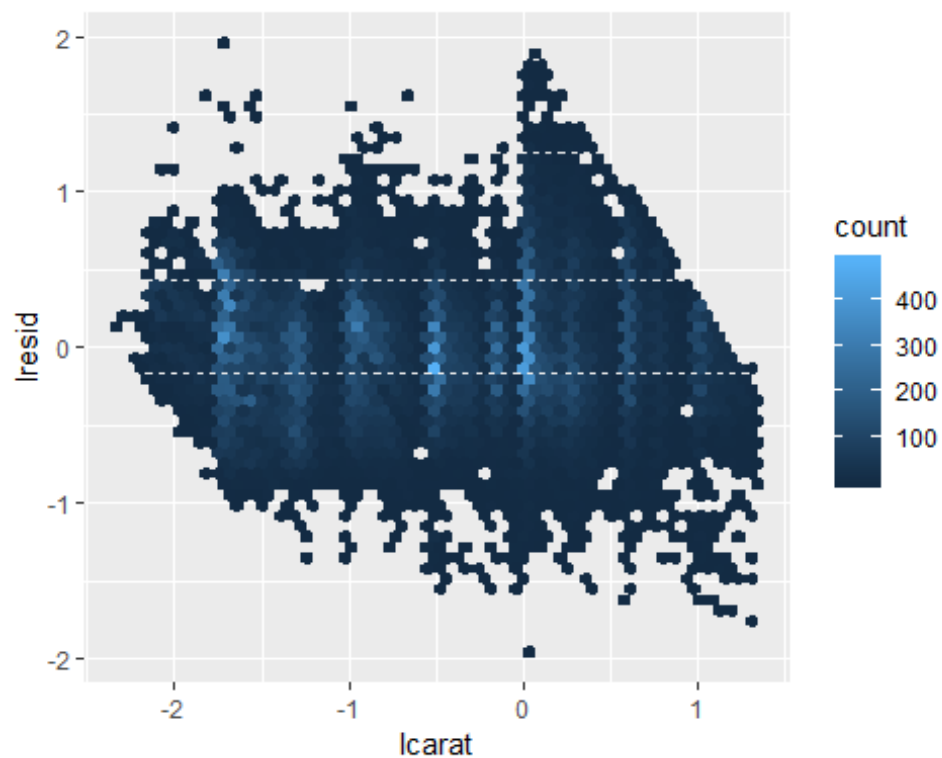
ggplot(diamonds2, aes(carat, price)) +
  geom_hex(bins = 50) +
  geom_line(data = grid, colour = "red", size = 1)
```



Detta säger något intressant om data. Enligt modellen är stora diamanter mycket billigare än förväntat, sannolikt beroende på att ingen diamanter kostat mer än \$19 000.

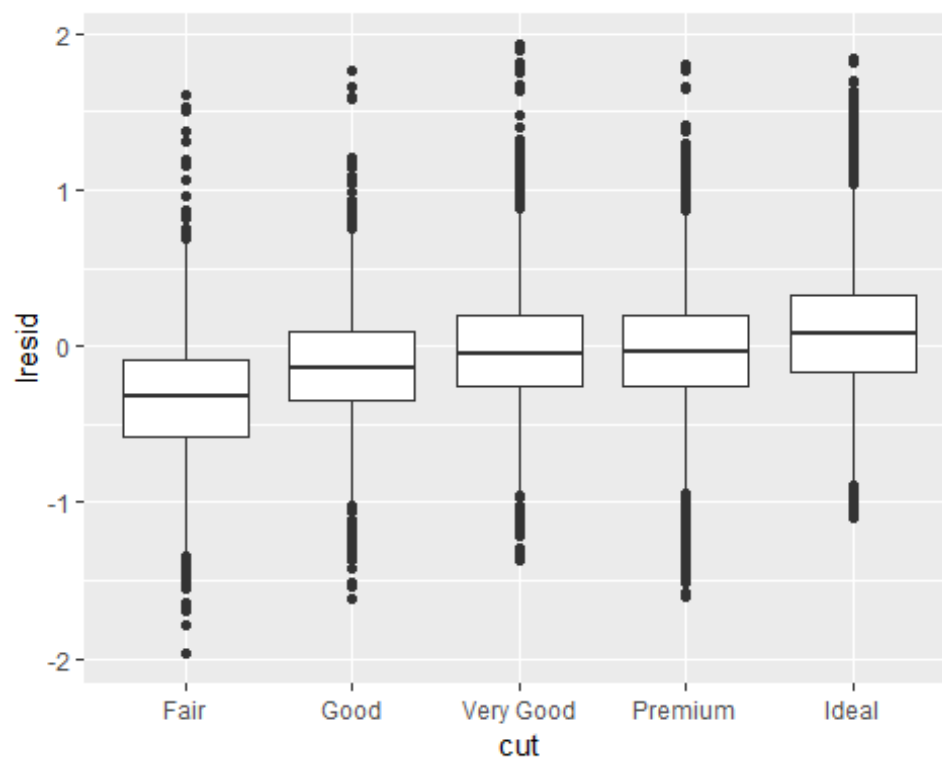
Om vi nu kikar på residualernas fördelning blir det tydligt att vi eliminerat det starka linjära sambandet, dvs vikten är en mycket viktig förklaring till priset:

```
diamonds2 <- diamonds2 %>%  
  add_residuals(mod_diamond, "lresid")  
  
ggplot(diamonds2, aes(lcarat, lresid)) +  
  geom_hex(bins = 50)
```

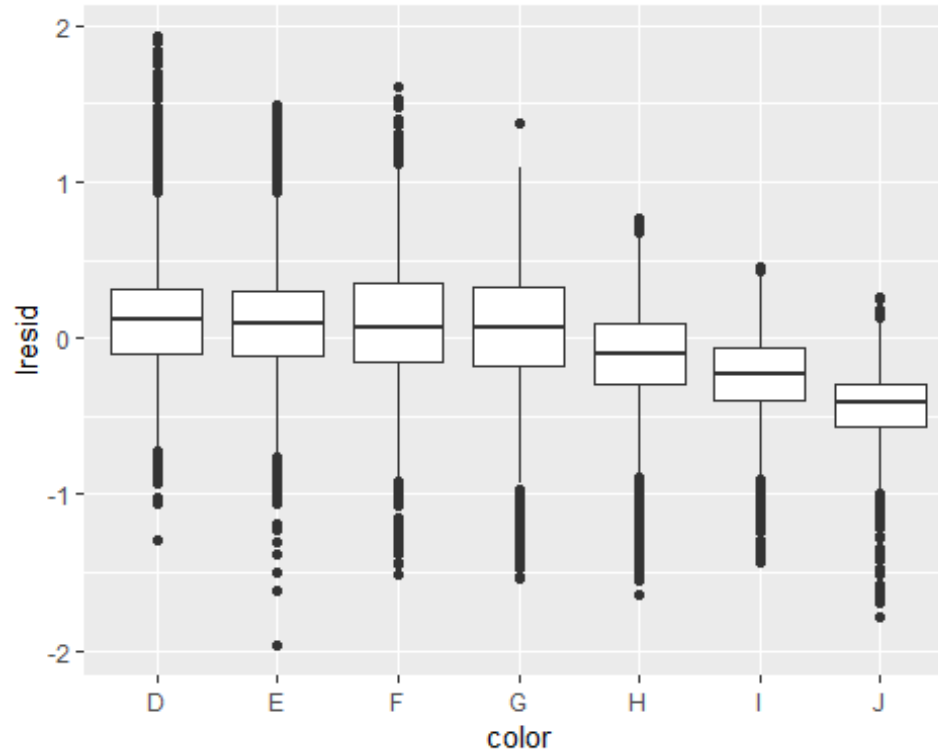


Om vi nu gör om våra initiala grafer oberoende av `carat` får vi följande:

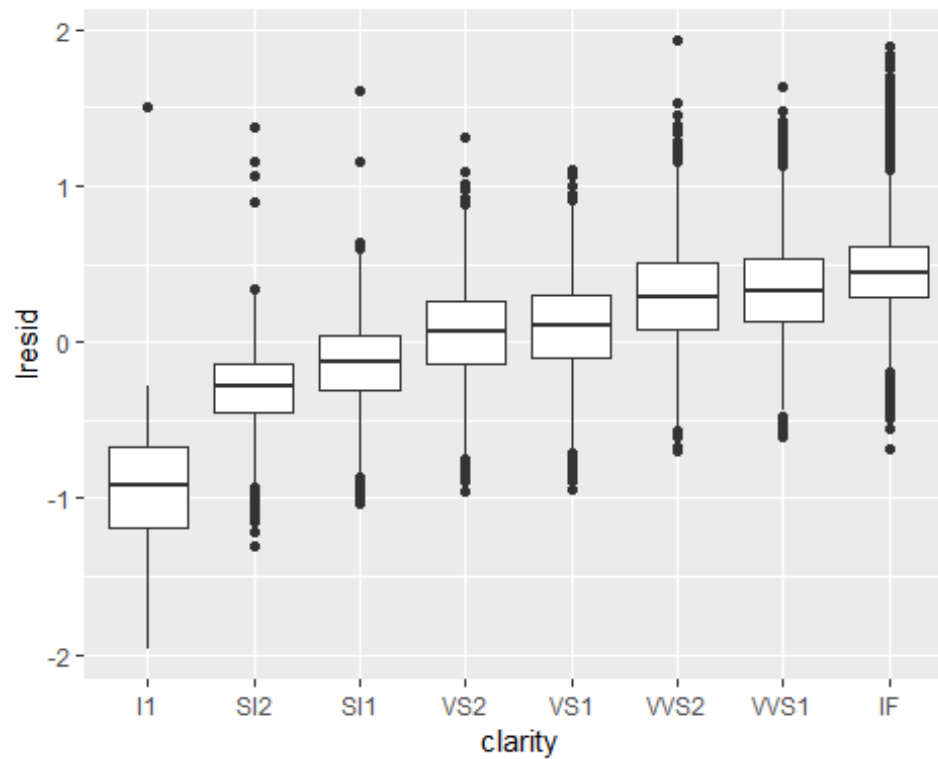
```
ggplot(diamonds2, aes(cut, lresid)) + geom_boxplot()
```



```
ggplot(diamonds2, aes(color, lresid)) + geom_boxplot()
```



```
ggplot(diamonds2, aes(clarity, lresid)) + geom_boxplot()
```



Här ser vi sambanden som vi intuitivt förväntar: diamanter med högre kvalitet kostar mer om vi tar hänsyn till deras storlek.

En mer komplicerad modell

Nu kan vi bygga ut vår modell genom att inkludera kvalitetsvariablerna:

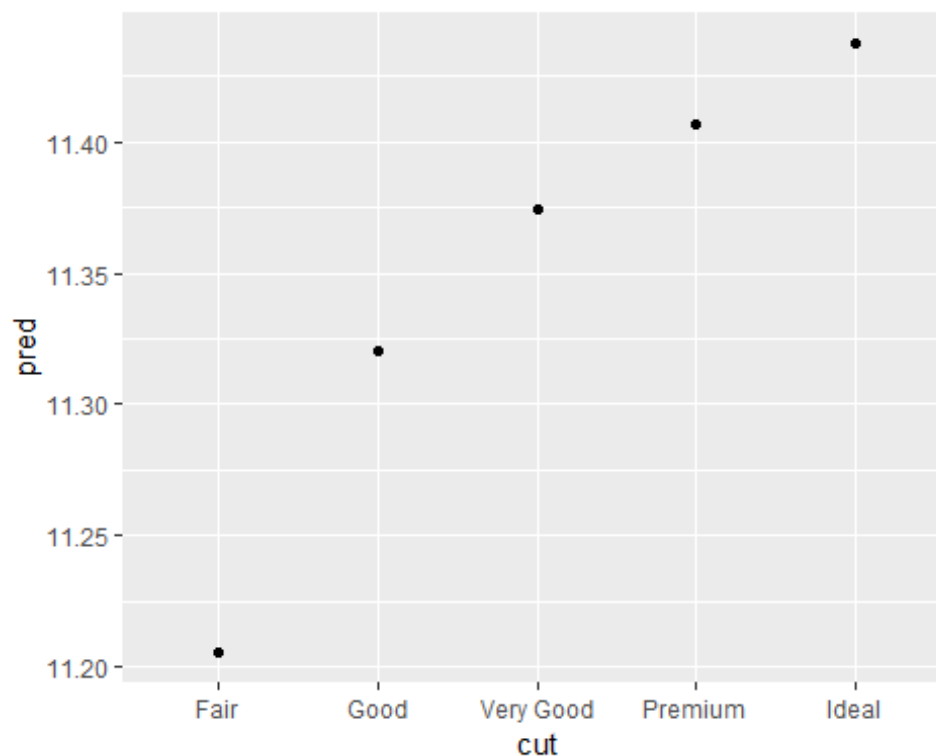
```
mod_diamond2 <- lm(lprice ~ lcarat + color + cut + clarity, data = diamonds2)
```

Eftersom modellen nu innehåller 4 variabler/prediktorer blir det svårare att visualisera utfallet. Vi kan, under antagandet att de är oberoende av varandra, plotta dem individuellt i fyra grafer. För att göra det lite enklare för oss kan vi använda argumentet `.model` i `data_grid()`:

```
grid <- diamonds2 %>%
  data_grid(cut, .model = mod_diamond2) %>%
  add_predictions(mod_diamond2)
grid

## # A tibble: 5 x 5
##   cut    lcarat color clarity pred
##   <ord>   <dbl> <chr> <chr>  <dbl>
## 1 Fair   -0.515 G   VS2    11.2
## 2 Good   -0.515 G   VS2    11.3
## 3 Very Good -0.515 G   VS2    11.4
## 4 Premium -0.515 G   VS2    11.4
## 5 Ideal  -0.515 G   VS2    11.4

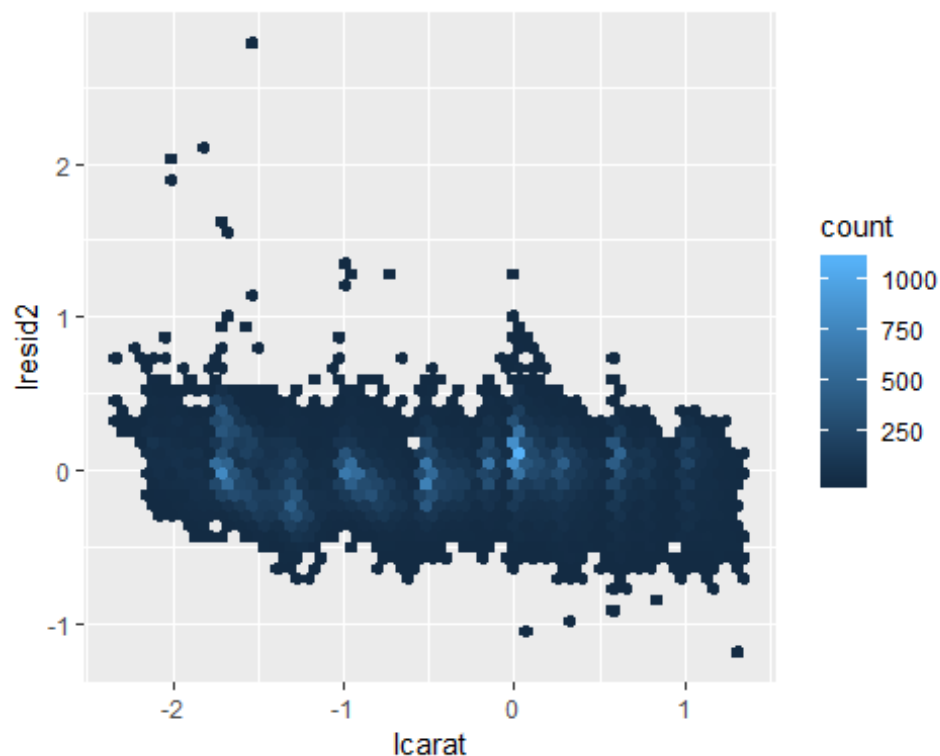
ggplot(grid, aes(cut, pred)) +
  geom_point()
```



Om modellen innehåller variabler du inte specificerat explicit kommer `data_grid()` automatiskt att fylla i med "typiska" värden. För kontinuerliga variabler används medianen och för kategoriska används den vanligast förekommande kategorin.

```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond2, "lresid2")

ggplot(diamonds2, aes(lcarat, lresid2)) +
  geom_hex(bins = 50)
```



Denna graf indikerar att det finns ett antal diamanter med ganska avvikande residualer (en residual = 2 motsvarar ju ett pris på diamanten som är 4 ggr så högt än förväntat). I detta läge är det klokt att kika på sådana outliers individuellt:

```
diamonds2 %>%
  filter(abs(lresid2) > 1) %>%
  add_predictions(mod_diamond2) %>%
  mutate(pred = round(2 ^ pred)) %>%
  select(price, pred, carat:table, x:z) %>%
  arrange(price)
```

```
## # A tibble: 16 x 11
##   price  pred carat cut    color clarity depth table  x    y    z
##   <int> <dbl> <dbl> <ord>  <ord> <ord>  <ord>  <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  1013   264 0.25 Fair  F    SI2    54.4   64  4.3  4.23  2.32
## 2  1186   284 0.25 Premium G    SI2    59    60  5.33  5.28  3.12
## 3  1186   284 0.25 Premium G    SI2    58.8   60  5.33  5.28  3.12
## 4  1262  2644 1.03 Fair  E    I1     78.2   54  5.72  5.59  4.42
## 5  1415   639 0.35 Fair  G    VS2    65.9   54  5.57  5.53  3.66
## 6  1415   639 0.35 Fair  G    VS2    65.9   54  5.57  5.53  3.66
## 7  1715   576 0.32 Fair  F    VS2    59.6   60  4.42  4.34  2.61
## 8  1776   412 0.290 Fair  F    SI1    55.8   60  4.48  4.41  2.48
## 9  2160   314 0.34 Fair  F    I1     55.8   62  4.72  4.6   2.6
```



```
## 10 2366 774 0.3 Very Good D VVS2 60.6 58 4.33 4.35 2.63
## 11 3360 1373 0.51 Premium F SI1 62.7 62 5.09 4.96 3.15
## 12 3807 1540 0.61 Good F SI2 62.5 65 5.36 5.29 3.33
## 13 3920 1705 0.51 Fair F VVS2 65.4 60 4.98 4.9 3.23
## 14 4368 1705 0.51 Fair F VVS2 60.7 66 5.21 5.11 3.13
## 15 10011 4048 1.01 Fair D SI2 64.6 58 6.25 6.2 4.02
## 16 10470 23622 2.46 Premium E SI2 59.7 59 8.82 8.76 5.25
```

Tabellen ger väl knappast några ledtrådar till en rimlig förklaring men generellt bör outlieras av detta slag föranleda oss att fundera över om det finns något problem med vår modell eller om det finns felaktiga data.

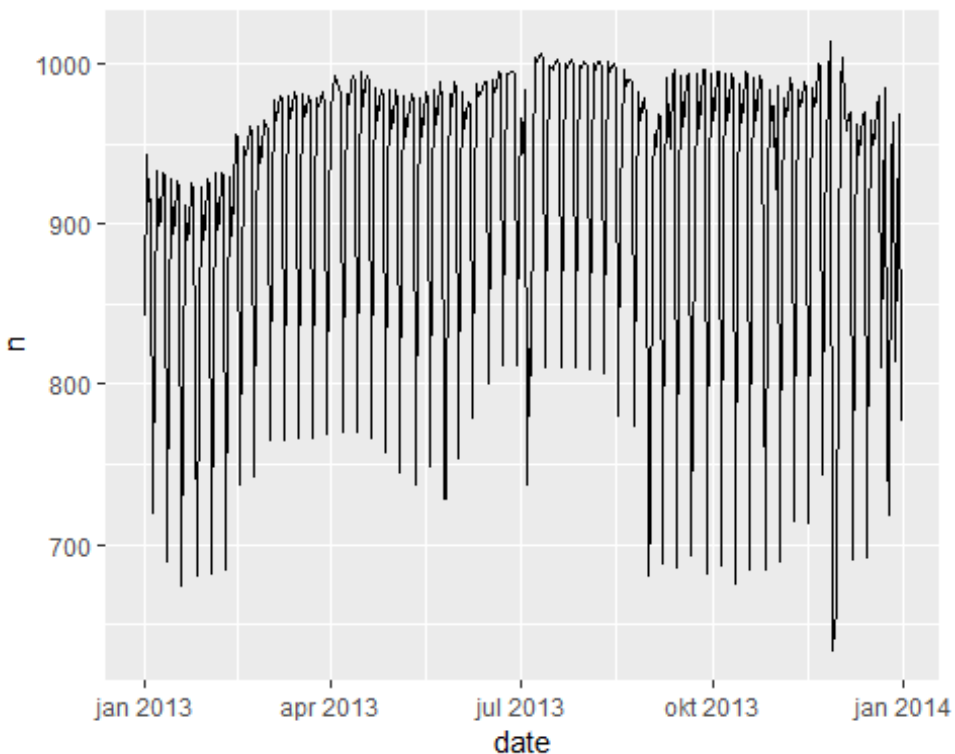
Vad är det som påverkar antalet dagliga fligheter?

Låt oss gå igenom en liknande process med det andra datasetet, flights. Vi börjar med att beräkna antalet fligheter per dag och visualisera dessa med ggplot2:

```
daily <- flights %>%
  mutate(date = make_date(year, month, day)) %>%
  group_by(date) %>%
  summarise(n = n())
daily

## # A tibble: 365 x 2
##   date      n
##   <date>   <int>
## 1 2013-01-01 842
## 2 2013-01-02 943
## 3 2013-01-03 914
## 4 2013-01-04 915
## 5 2013-01-05 720
## 6 2013-01-06 832
## 7 2013-01-07 933
## 8 2013-01-08 899
## 9 2013-01-09 902
## 10 2013-01-10 932
## # ... with 355 more rows

ggplot(daily, aes(date, n)) +
  geom_line()
```

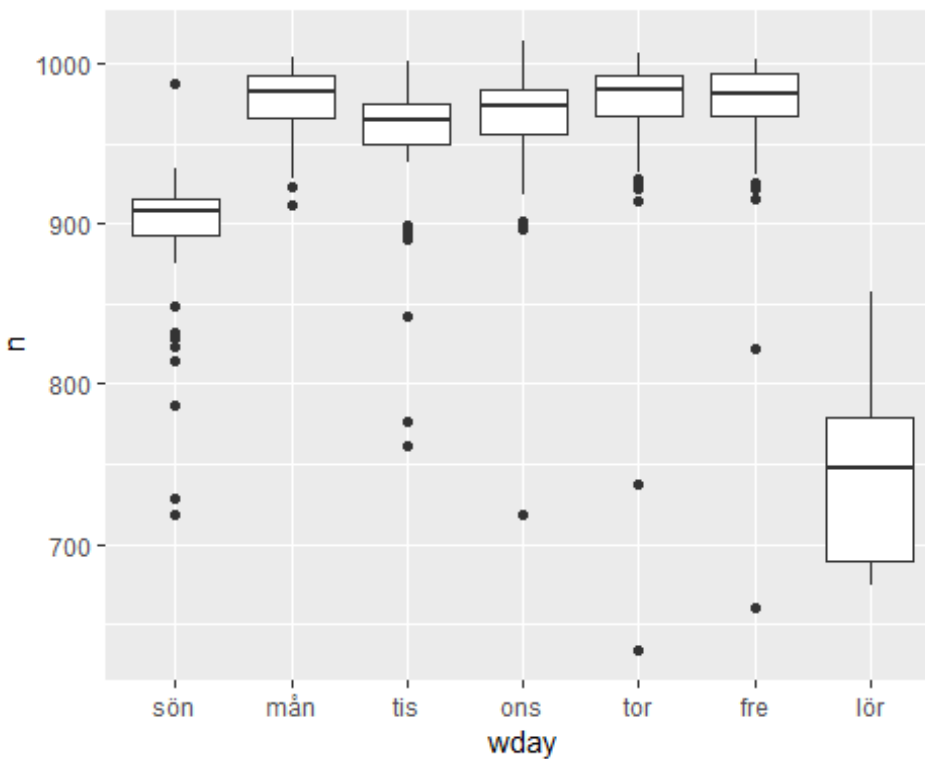


Veckodag

Det är knappast intuitivt att utifrån rådata förstå långtidstrenden eftersom det finns en stark effekt utifrån vilken veckodag som flighterna sker. Vi kollar in fördelningen över veckodagarna:

```
daily <- daily %>%
  mutate(wday = wday(date, label = TRUE))

ggplot(daily, aes(wday, n)) +
  geom_boxplot()
```



Det finns betydligt färre flighter på helgerna eftersom de flesta är affärsresor. Vi kan skapa en modell för att eliminera effekten av veckodagar och lägga på prediktionerna på observationerna:

```
mod <- lm(n ~ wday, data = daily)
```

```
grid <- daily %>%
```

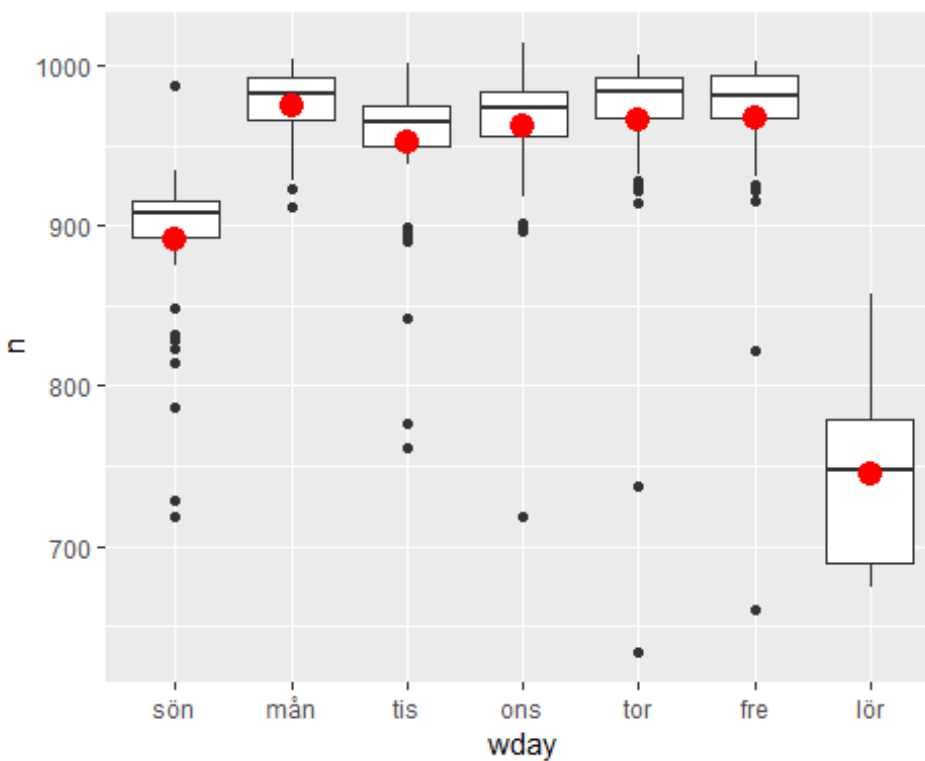
```
  data_grid(wday) %>%
```

```
  add_predictions(mod, "n")
```

```
ggplot(daily, aes(wday, n)) +
```

```
  geom_boxplot() +
```

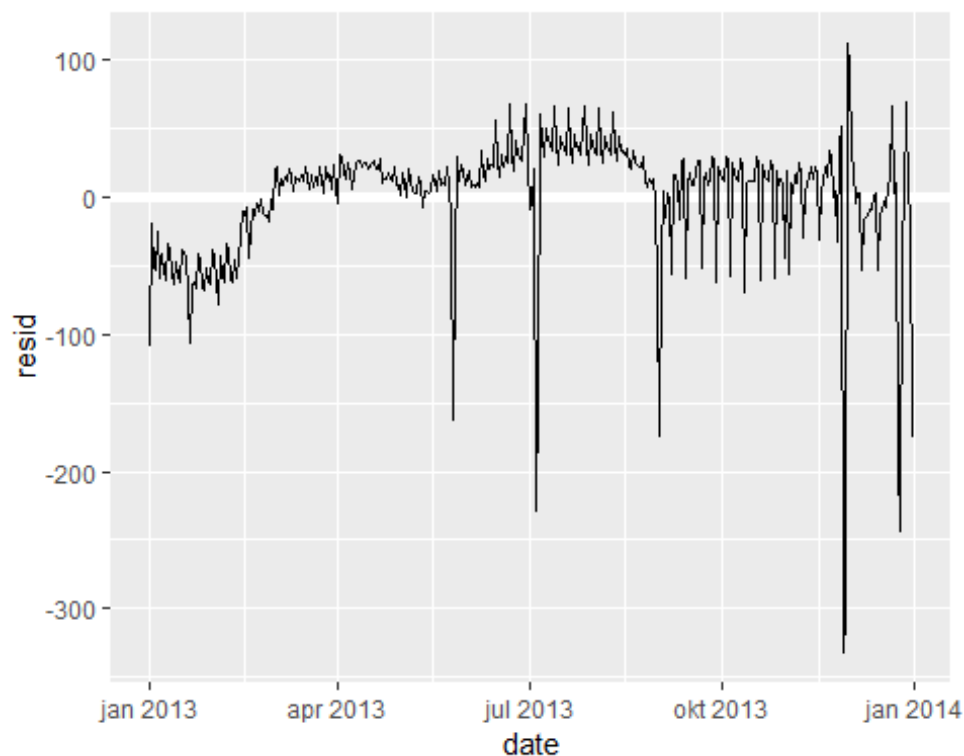
```
  geom_point(data = grid, colour = "red", size = 4)
```



Så kan vi beräkna och visualisera residualerna:

```
daily <- daily %>%
  add_residuals(mod)

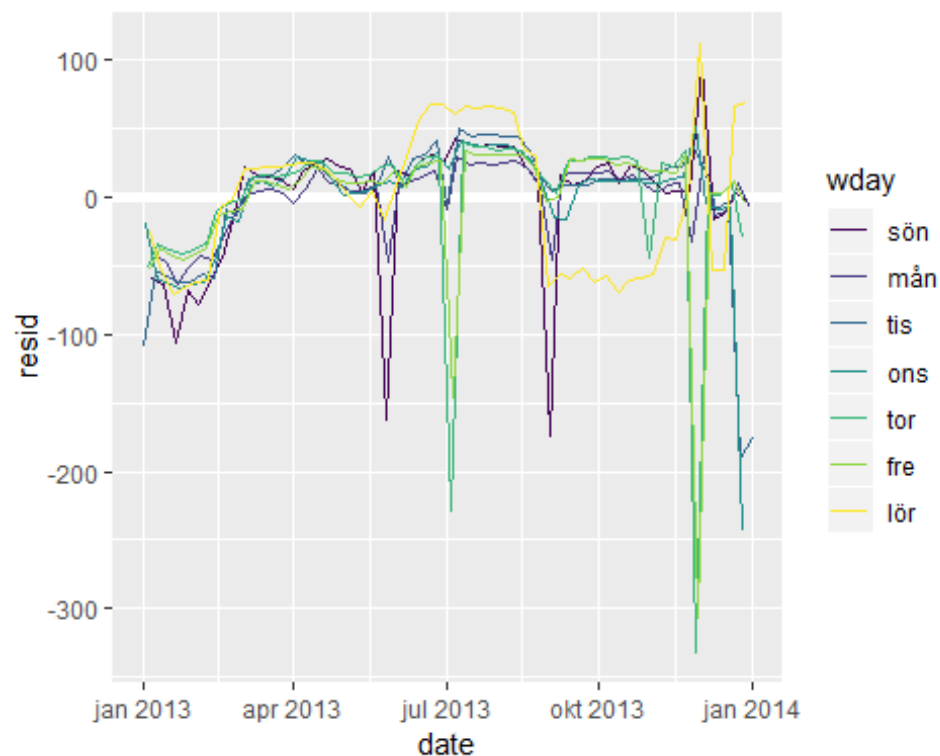
daily %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line()
```



Notera förändringen på y-axeln - Den anger nu avvikelsen från det förväntade antalet fligheter givet veckodag. Detta är bra information eftersom vi nu avlägsnat mycket av den starka effekten av veckodag och vi ser tydligare andra, mer subtila mönster som kvarstår:

1. Modellen verkar fungera sämre på datum i juni och senare. Om vi ritar en plot per veckodag kan vi se en möjlig förklaring:

```
ggplot(daily, aes(date, resid, colour = wday)) +  
  geom_ref_line(h = 0) +  
  geom_line()
```



Vi ser att modellen är sämre på att fånga antalet flighter på lördagar - under sommaren finns det fler flighter än förväntat och under hösten är det färre. Vi ska lite senare se hur vi kan fånga detta mönster lite effektivare.

2. Det finns några dagar med betydligt färre flighter än förväntat. Låt oss se vilka dessa är:

```
daily %>%
```

```
  filter(resid < -100)
```

```
## # A tibble: 11 x 4
```

```
##   date      n wday resid
```

```
##   <date>   <int> <ord> <dbl>
```

```
## 1 2013-01-01 842 tis -109.
```

```
## 2 2013-01-20 786 sön -105.
```

```
## 3 2013-05-26 729 sön -162.
```

```
## 4 2013-07-04 737 tor -229.
```

```
## 5 2013-07-05 822 fre -145.
```

```
## 6 2013-09-01 718 sön -173.
```

```
## 7 2013-11-28 634 tor -332.
```

```
## 8 2013-11-29 661 fre -306.
```

```
## 9 2013-12-24 761 tis -190.
```

```
## 10 2013-12-25 719 ons -244.
```

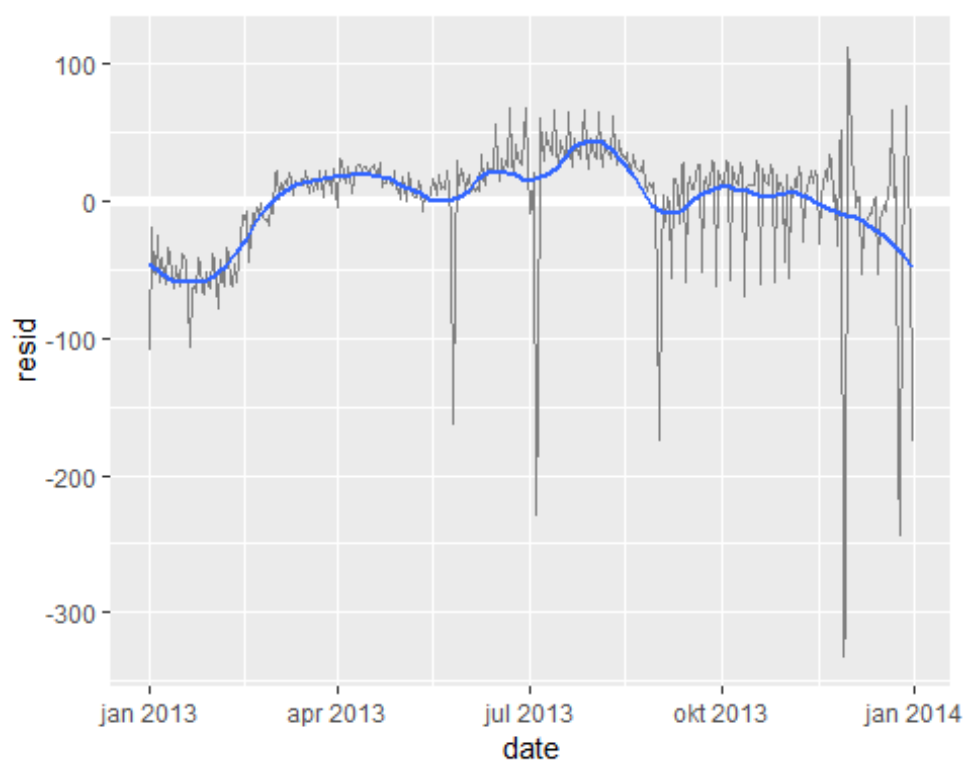
```
## 11 2013-12-31 776 tis -175.
```

För den som är bekant med helgdagar i USA känner igen dessa dagar: Nyårsdagen, 4 juli, thanksgiving och julhelgen + några till.

- Det verkar finnas någon mindre uttalad långvarigare trend över året. Vi kan lyfta fram den med hjälp av `geom_smooth()`:

```
daily %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line(colour = "grey50") +
  geom_smooth(se = FALSE, span = 0.20)

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



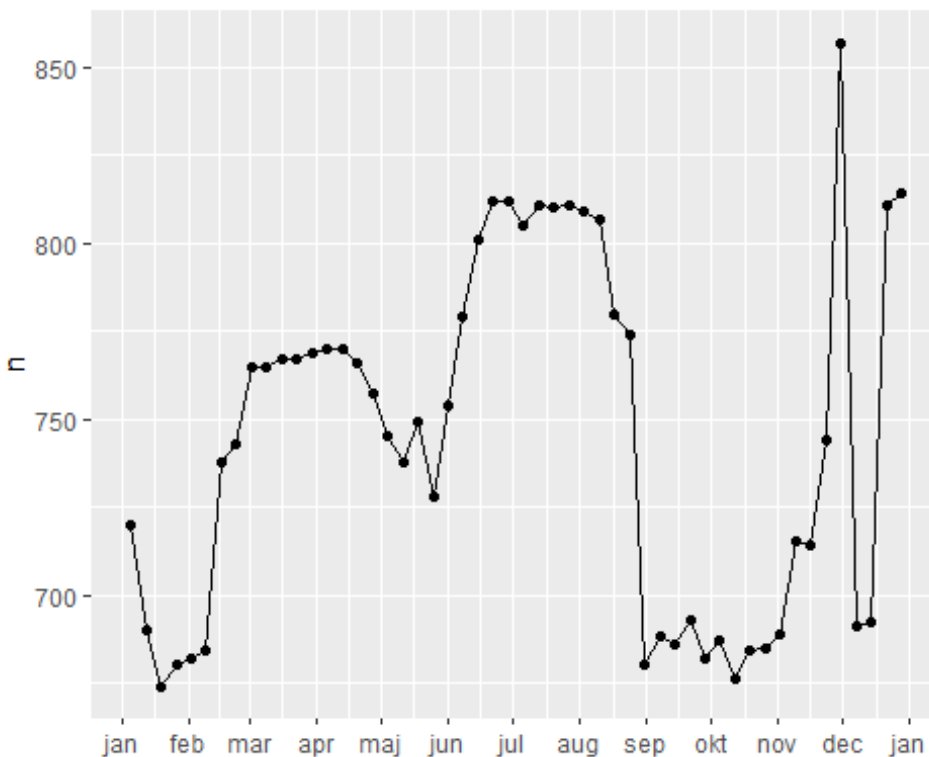
Det är färre flighter i januari och december men fler under sommaren. Vi skulle sannolikt behöva data över fler år för att förklara detta så vi lämnar det därhän för nu.

Lördags-effekten

Hur kan vi förklara avvikelserna på lördagar? En rimlig startpunkt är att gå tillbaka till rådata och välja ut lördagar:

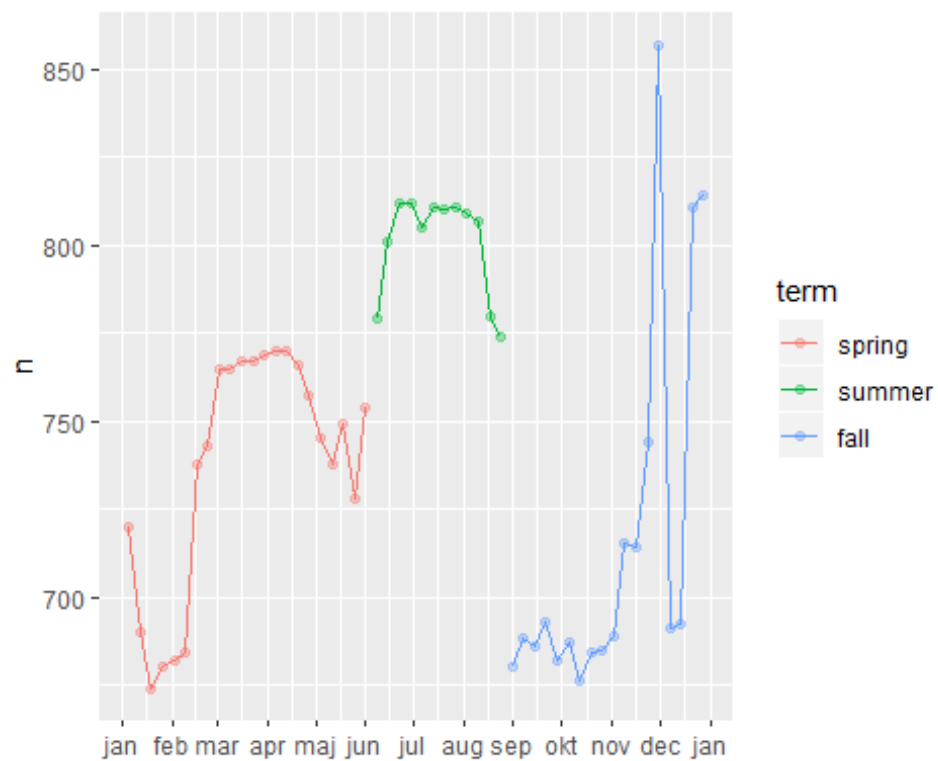
```
daily %>%
  filter(wday == "lör") %>%
  ggplot(aes(date, n)) +
  geom_point() +
```

```
geom_line() +
scale_x_date(NULL, date_breaks = "1 month", date_labels = "%b")
```



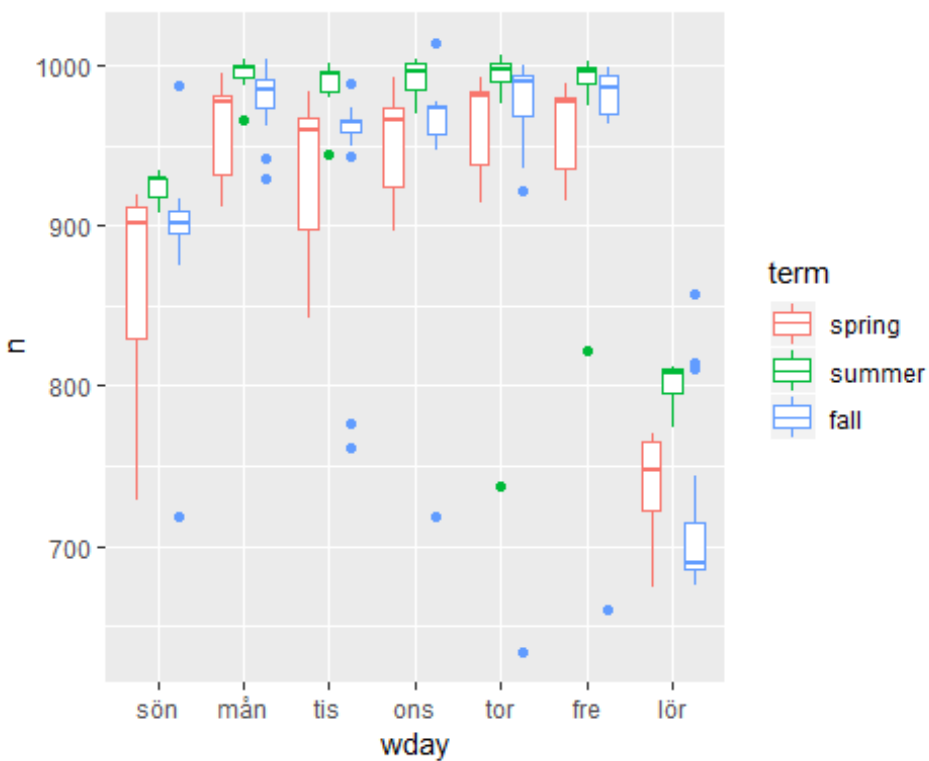
Detta mönster är förenligt med semesterperioden på sommaren, fler kan antas semester-resa på lördagar. Men varför finns fler lördags-fligheter under våren än på hösten? Kan det vara så att det är mindre vanligt att ta semester under hösten eftersom de stora Thanksgiving och Julhelgerna infaller då? Det finns inte data att undersöka denna hypotes men för övningens skull, låt oss skapa en termins-variabel som grovt fångar det tre skol-terminerna:

```
term <- function(date) {
  cut(date,
    breaks = ymd(20130101, 20130605, 20130825, 20140101),
    labels = c("spring", "summer", "fall")
  )
}
daily <- daily %>%
  mutate(term = term(date))
daily %>%
  filter(wday == "lör") %>%
  ggplot(aes(date, n, colour = term)) +
  geom_point(alpha = 1/3) +
  geom_line() +
  scale_x_date(NULL, date_breaks = "1 month", date_labels = "%b")
```

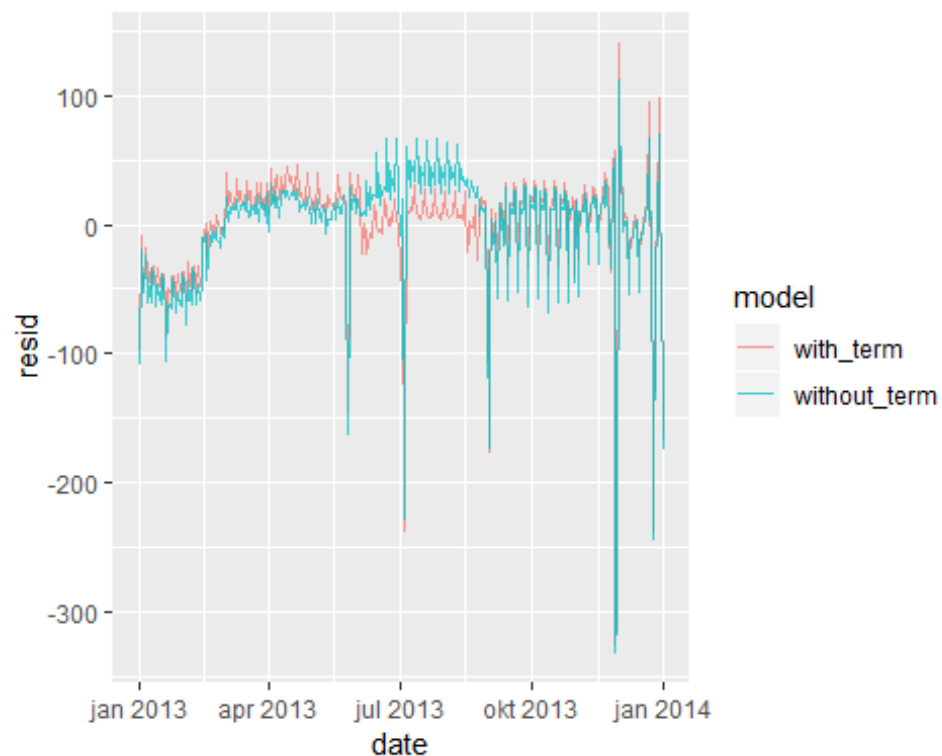
Och vi kollar hur termins-variabeln påverkar de andra dagarna i veckan:

```
daily %>%
  ggplot(aes(wday, n, colour = term)) +
  geom_boxplot()
```



Det ser ut som om det finns en påtaglig variation över terminerna så det verkar rimligt att lägga till veckodagar till modellen:

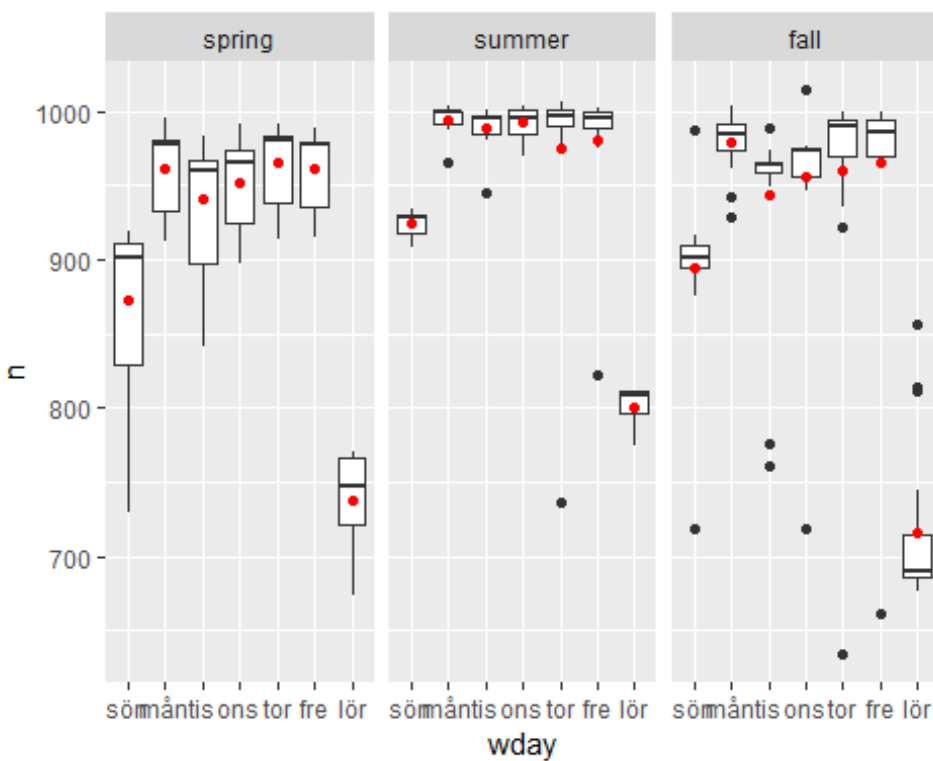
```
mod1 <- lm(n ~ wday, data = daily)
mod2 <- lm(n ~ wday * term, data = daily)
daily %>%
  gather_residuals(without_term = mod1, with_term = mod2) %>%
  ggplot(aes(date, resid, colour = model)) +
  geom_line(alpha = 0.75)
```



Modellen förbättras men inte så mycket som vi önskat kanske. Om vi nu läggaer på prediktionerna från modellen till rådata ser vi problemet tydligare:

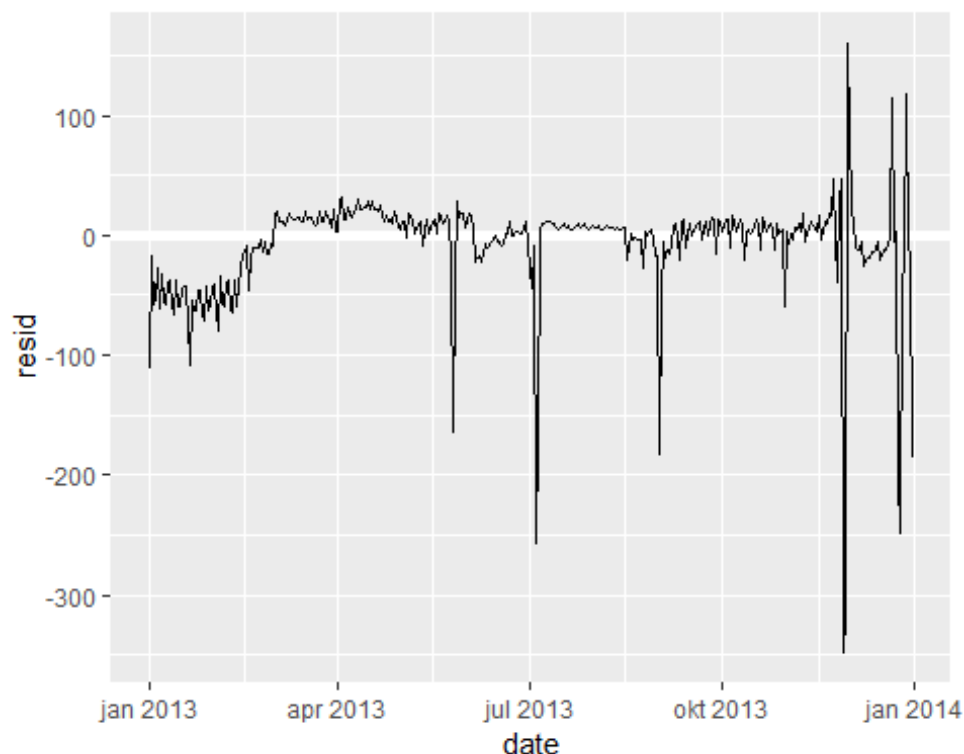
```
grid <- daily %>%
  data_grid(wday, term) %>%
  add_predictions(mod2, "n")

ggplot(daily, aes(wday, n)) +
  geom_boxplot() +
  geom_point(data = grid, colour = "red") +
  facet_wrap(~ term)
```



Modellen ger den genomsnittliga effekten men här finns en rad extrema outliers. Vi kan minska effekten från dessa genom att använda en modell som är mindre känslig för sådana outliers, en *robust linear model*, `MASS::rlm()`:

```
mod3 <- MASS::rlm(n ~ wday * term, data = daily)
daily %>%
  add_residuals(mod3, "resid") %>%
  ggplot(aes(date, resid)) +
  geom_hline(yintercept = 0, size = 2, colour = "white") +
  geom_line()
```



Nu är det betydligt enklare att se den mer långvariga trenden och outliers.

Lära mer om modellering

Det finns oerhört mycket material för att lära sig mer om hur man kan använda R för att bygga modeller. Det faller utanför ramen för denna kurs men för den som vill veta mer finns mer att läsa och pröva i Wickhams bok (<https://r4ds.had.co.nz/model-building.html#time-of-year-an-alternative-approach>) och det efterföljande kapitlet *Many models* (<https://r4ds.had.co.nz/many-models.html>).

Och därutöver kan rekommenderas

- *Statistical Modeling: A Fresh Approach* Danny Kaplan, http://project-mosaic-books.com/?page_id=13
- *An Introduction to Statistical Learning* Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, <http://www-bcf.usc.edu/~gareth/ISL/>

Grafik för att kommunicera

Det här avsnittet handlar om verktyg för att göra grafiken mer lättförståelig för andra, för att kunna kommunicera din förståelse av data. Vi ska använda ggplot2 + några extensions: ggrepel och viridis så dessa behöver installeras.

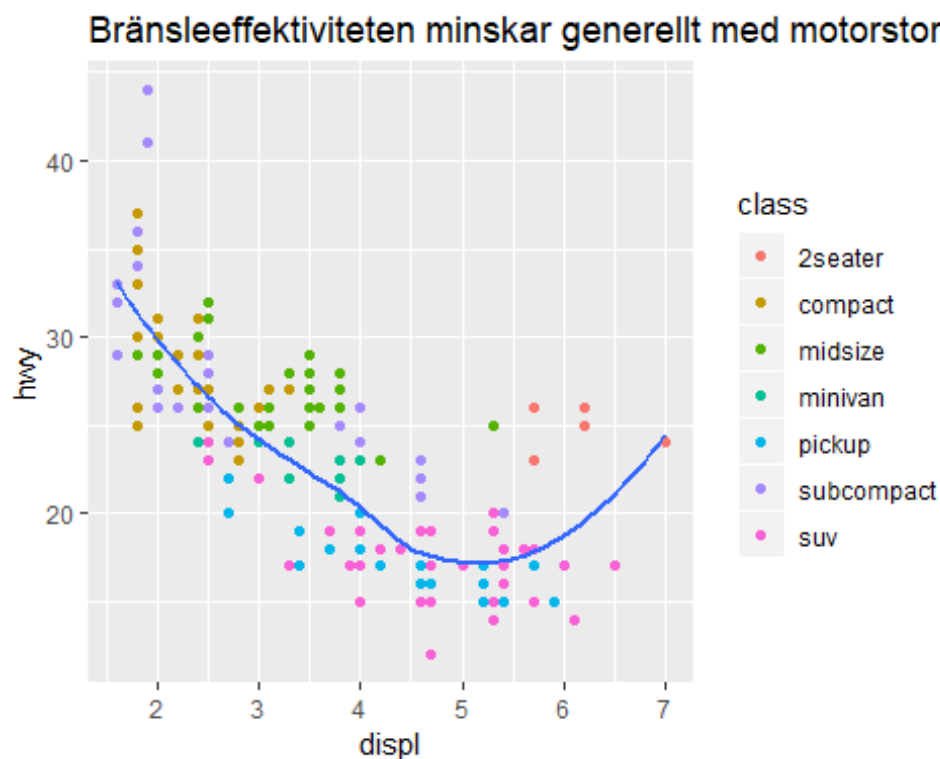
```
library(tidyverse)
library(ggrepel)
library(viridis)
```

Etiketter (labels)

Du behöver definiera tydliga etiketter som förklarar vad det är som man ser i grafen. Du lägger till etiketter med hjälp av labs(). Till exempel, för att lägga till en titel på hela grafen:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(title = "Bränsleeffektiviteten minskar generellt med motorstorlek")
```

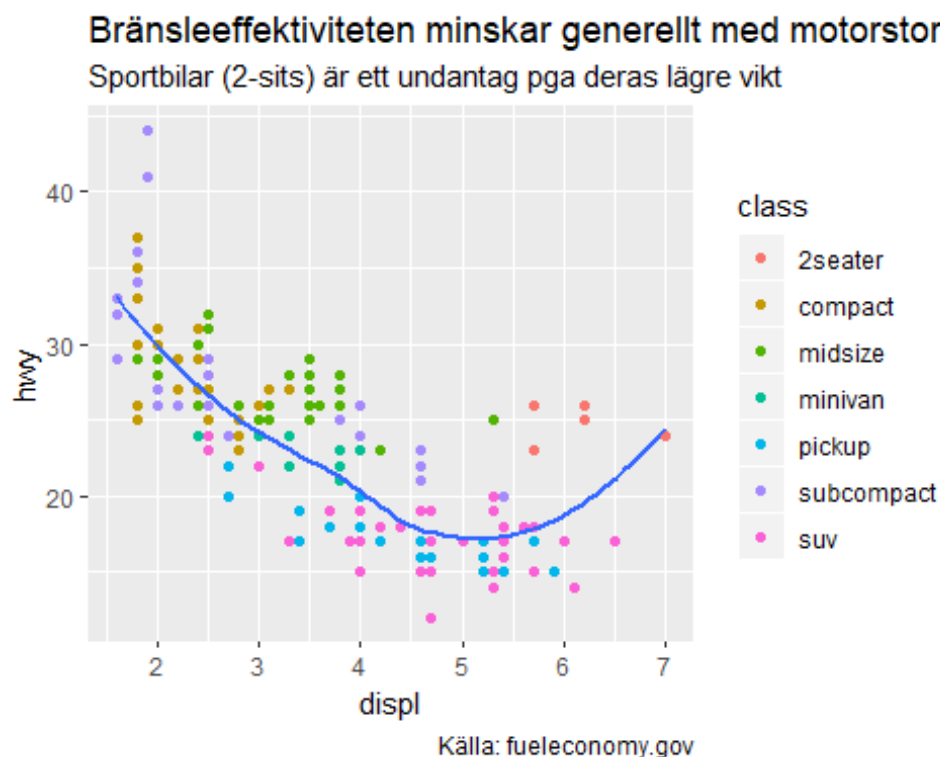
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Här summeras huvudfyndet. Det finns ett par ytterligare som kan hjälpa till att förtydliga vad grafen innehåller:

- *subtitle* - lägger till ytterligare detaljer i en mindre font under huvudtiteln
- *caption* - lägger till text nedtill höger i grafen t.ex. för att beskriva data-källan.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    title = "Bränsleeffektiviteten minskar generellt med motorstorlek",
    subtitle = "Sportbilar (2-sits) är ett undantag pga deras lägre vikt",
    caption = "Källa: fueleconomy.gov"
  )
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



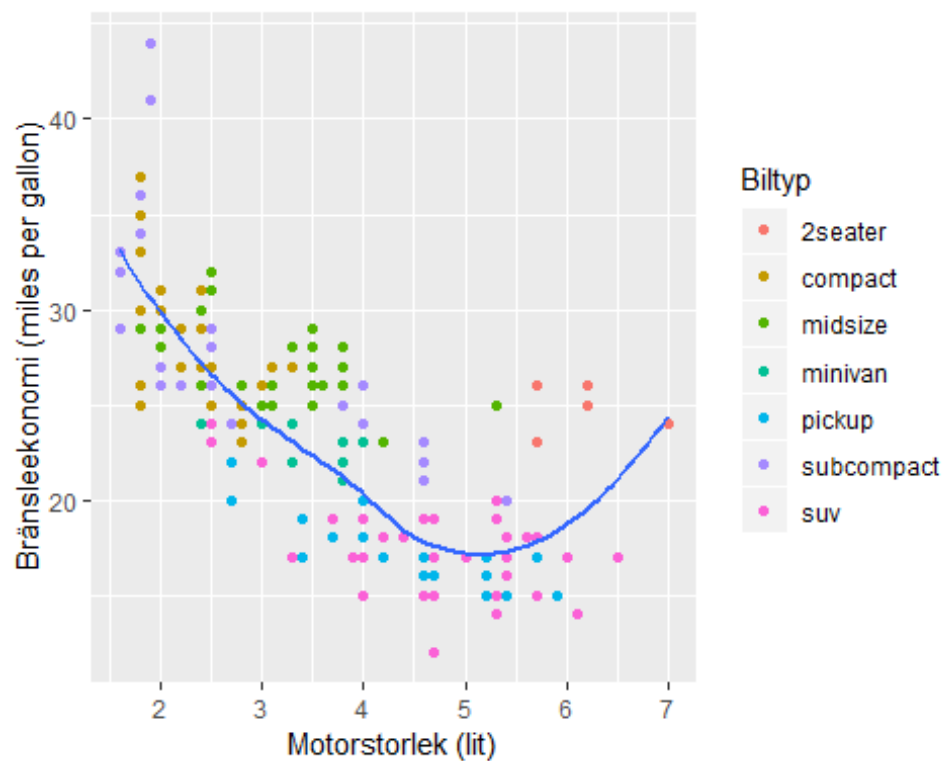
Du kan även använda `labs()` till att byta ut etiketterna för x- och y-axlarna och till ev teckenförklaring:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_smooth(se = FALSE) +
  labs(
    x = "Motorstorlek (lit)",
    y = "Bränsleekonomi (miles per gallon)",
```

```

colour = "Biltyp"
)
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'

```

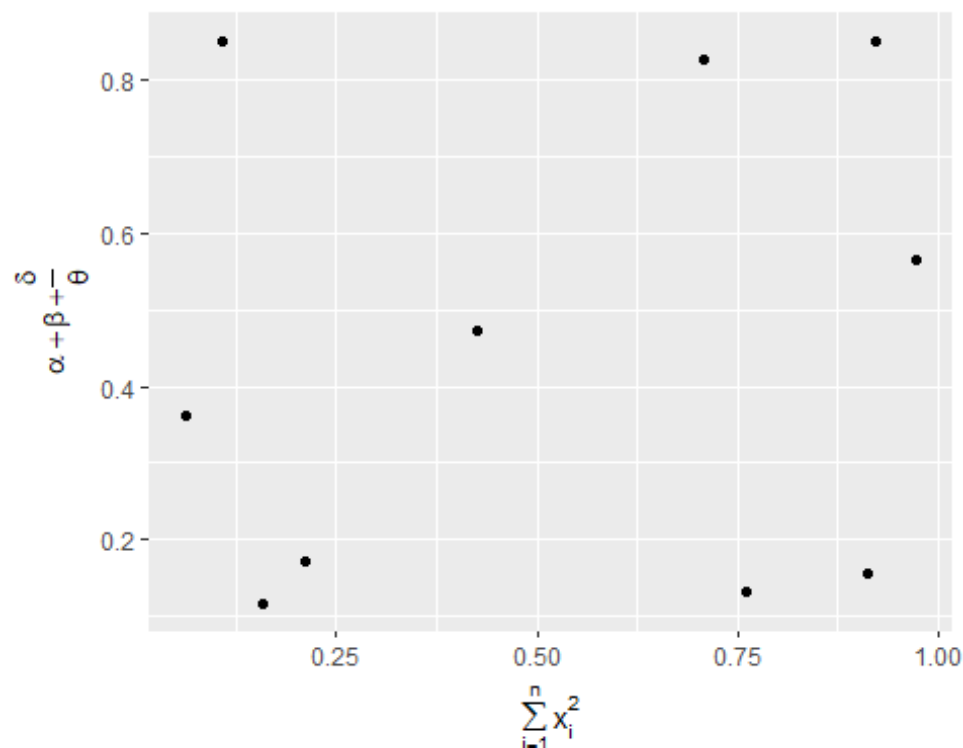


Och även använda matematiska uttryck istället för textsträngar. Byt ut "" mot `quote()` och se `?plotmath` för tillgängliga alternativ:

```

df <- tibble(
  x = runif(10),
  y = runif(10)
)
ggplot(df, aes(x, y)) +
  geom_point() +
  labs(
    x = quote(sum(x[i]^2, i == 1, n)),
    y = quote(alpha + beta + frac(delta, theta))
  )

```

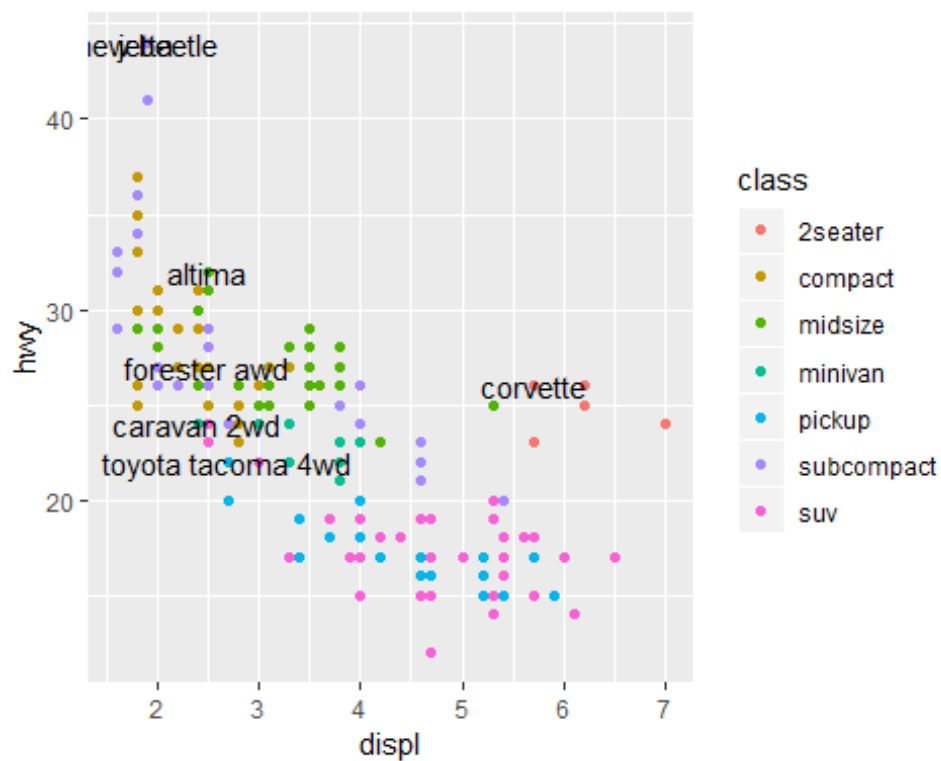
Annoteringar

Det är också möjligt att göra annoteringar på grafen för t.ex. grupper av observationer. Ett viktigt verktyg för detta är `geom_text()` vilket liknar `geom_point` men har ett ytterligare `aes`-argument: `label`.

Texten till sådana etiketter kan genereras på två sätt. Det kan finnas en *tibble* som kan ge texten. Nedanstående graf är kanske inte särskilt meningsfull men illustrativ: lyft fram det mest effektiva märket i varje bil-klass och sätt en etikett på denna i grafen:

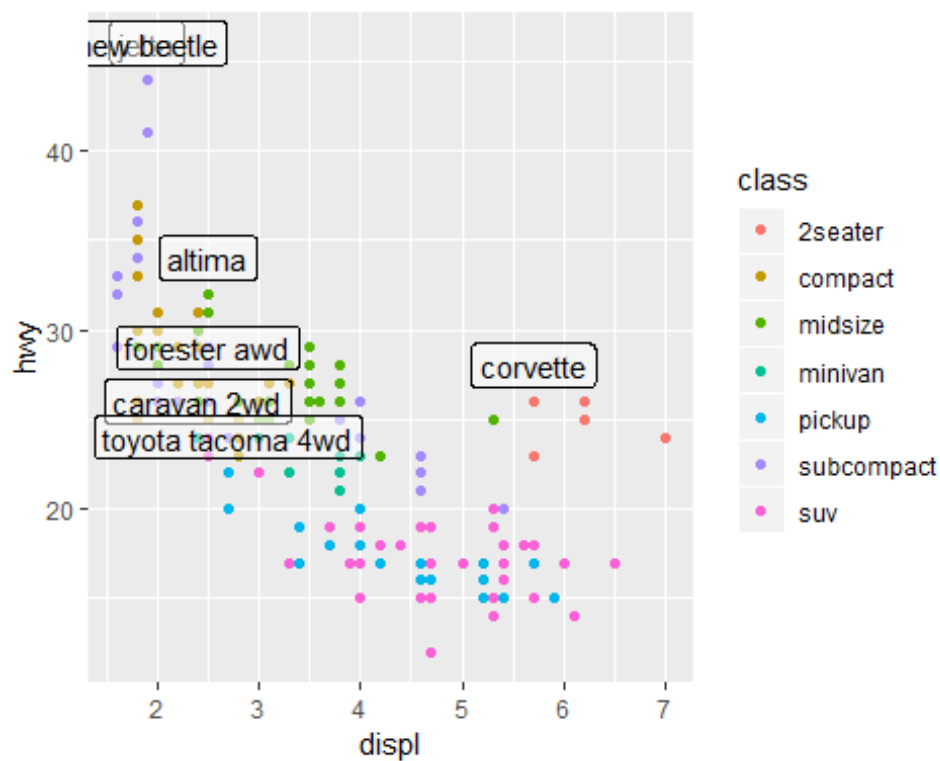
```
best_in_class <- mpg %>%
  group_by(class) %>%
  filter(row_number(desc(hwy)) == 1)

ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_text(aes(label = model), data = best_in_class)
```



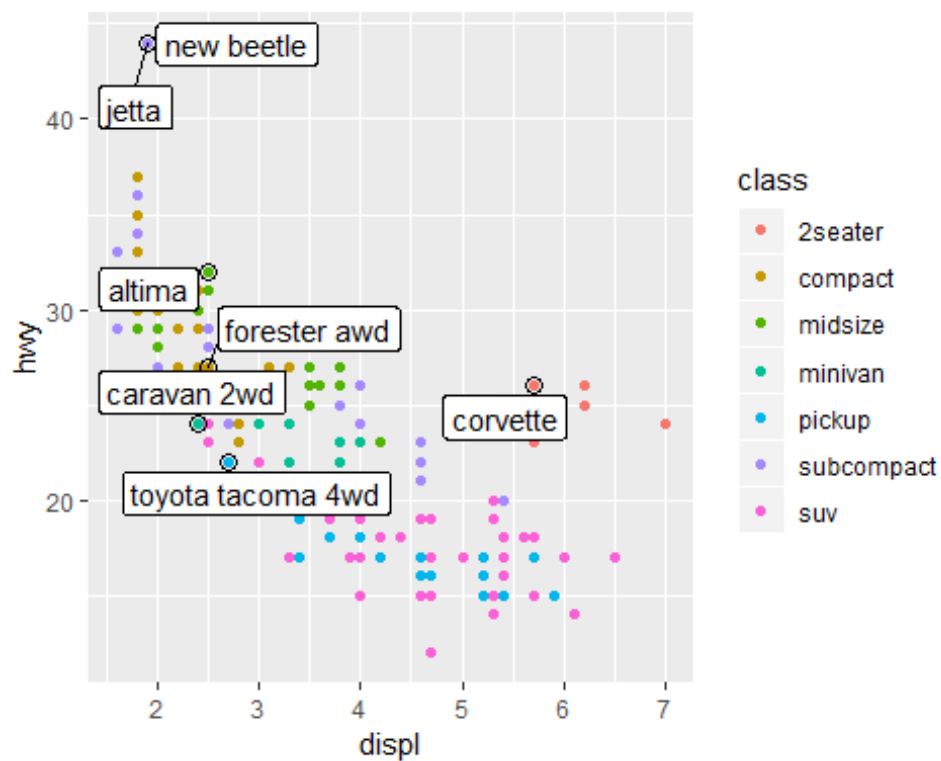
Detta är svårt att läsa av pga överlappande etiketter. Det kan vi delvis avhjälpa genom att använda `geom_label()` som ritar en rektangel bakom texten. Vi kan också använda `nudge_y` för att flytta etiketterna lite över motsvarande punkter:

```
ggplot(mpg, aes(displ, hwy)) +  
  geom_point(aes(colour = class)) +  
  geom_label(aes(label = model), data = best_in_class, nudge_y = 2, alpha = 0.5)
```



Delvis bättre, men fortfarande finns problem upp till vänster som inte kan avhjälpas genom att transformera etiketterna. Istället använder vi funktionen `geom_label_repel()` i modulen `ggrepel`. I denna modul finns en rad verktyg för att se till att etiketter inte överlappar:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_point(size = 3, shape = 1, data = best_in_class) +
  ggrepel::geom_label_repel(aes(label = model), data = best_in_class)
```



Här används också ett ytterligare lager som markerar den etiketterade punkten.

Scales

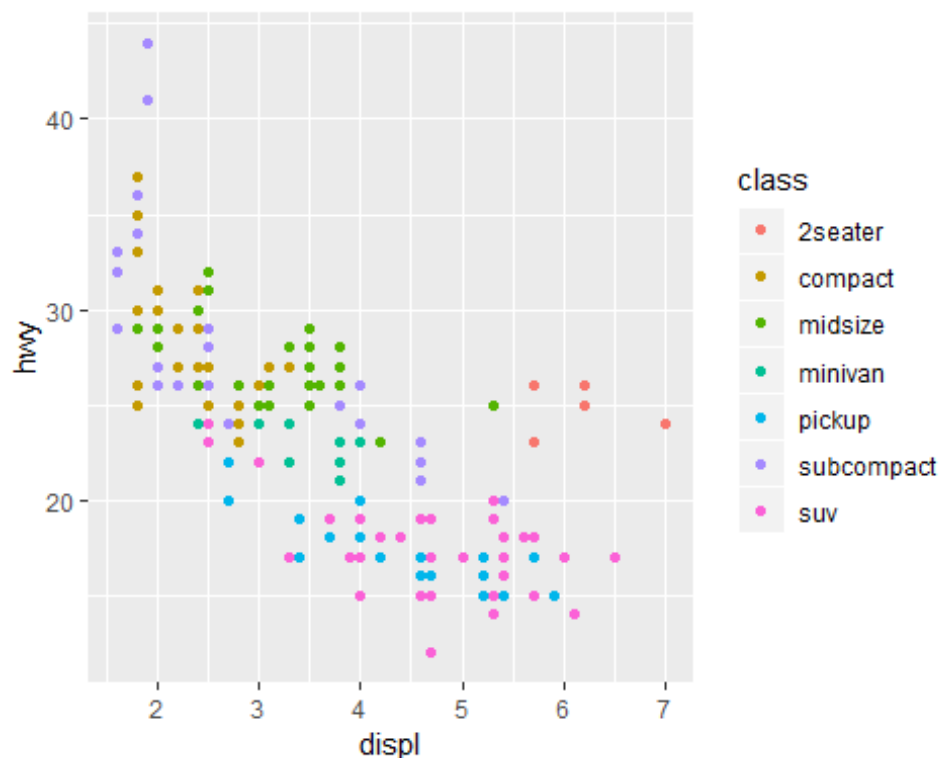
Scales kontrollerar mappningen av data. Typiskt lägger ggplot2 automatiskt till scales. Till exempel:

```
ggplot(mpg, aes(displ, hwy)) +  
  geom_point(aes(colour = class))
```



ggplot2 lägger automatiskt till scales :

```
ggplot(mpg, aes(displ, hwy)) +  
  geom_point(aes(colour = class)) +  
  scale_x_continuous() +  
  scale_y_continuous() +  
  scale_colour_discrete()
```



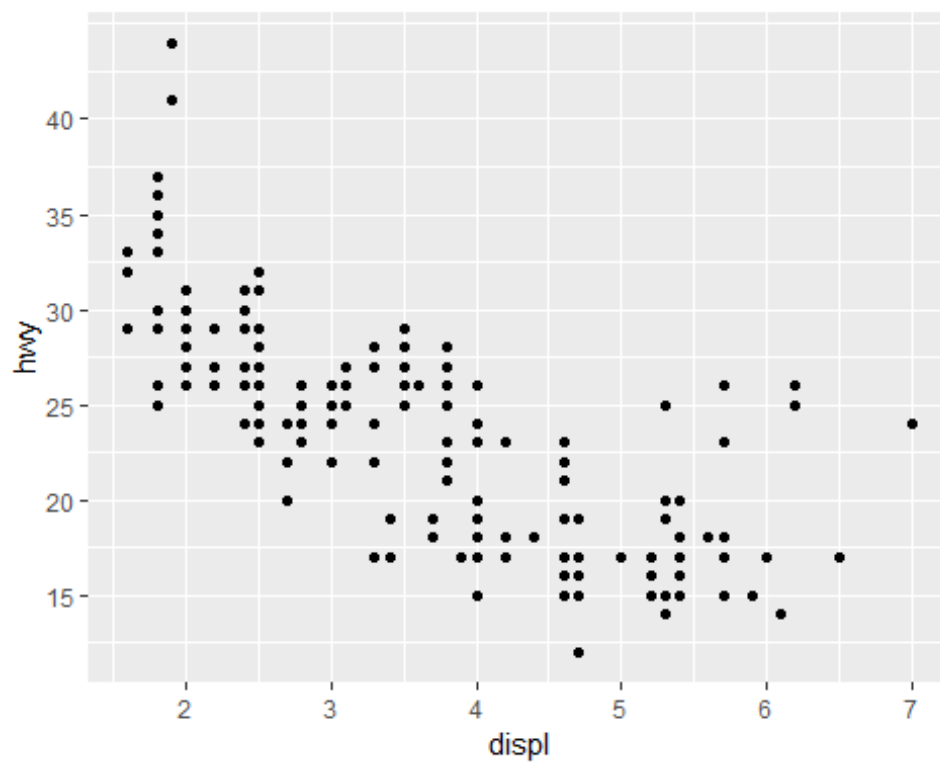
Notera hur benämningen av de specifika scales sker: `scale_` följt av namnet på *aesthetic*, sedan `_`, sedan namnet på scale. Default-namn bestäms av typen av variabler de är associerade med: kontinuerliga, diskreta, datum etc. Oftast fungerar default-valen väl men det finns ibland skäl att välja en annan scale:

1. Du vill tweaka några parametrar av default scale, t.ex. ticks och värden på axlarna eller etiketterna på teckenförklaringen
2. Du vill ersätta en viss scale med en annan som du anser är mer ändamålsenlig.

Axis ticks och teckenförklaringar

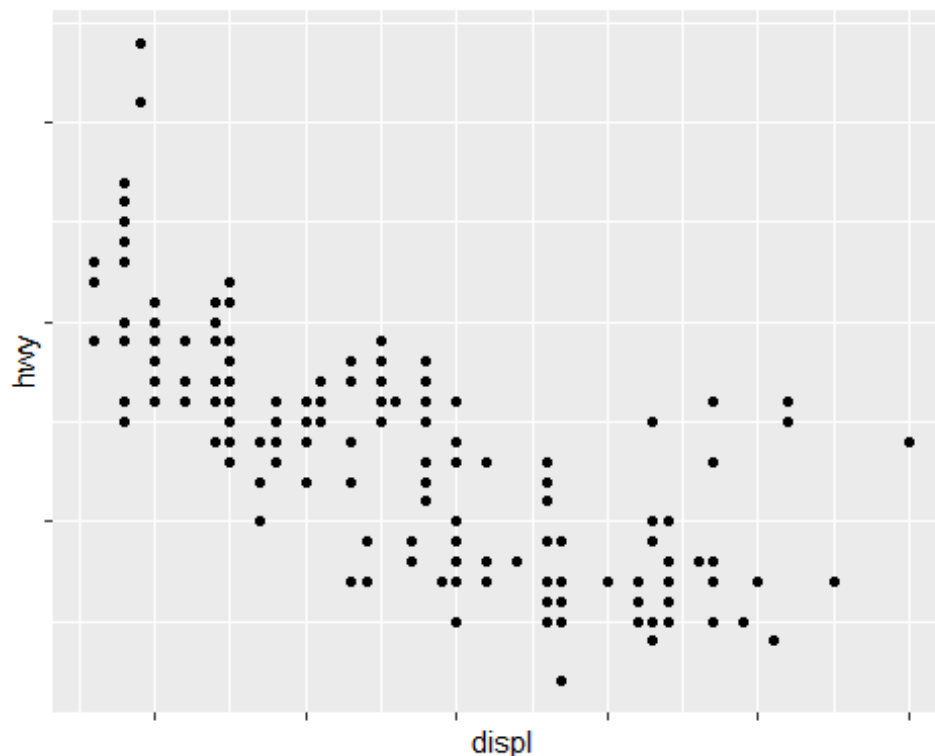
De två primära argumenten för att påverka hur ticks och teckenförklaringar visas är `breaks` och `labels`. `breaks` kontrollerar positionen på ticks eller värdet associerat med de olika teckenkategorierna. `labels` kontrollerar texten på etiketten på respektive ticks eller förklaring:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_y_continuous(breaks = seq(15, 40, by = 5))
```



Du kan använda labels på samma sätt som breaks (dvs en character vektor av samma längd som breaks) men du kan även sätta labels = NULL för att inte ha några labels alls:

```
ggplot(mpg, aes(displ, hwy)) +  
  geom_point() +  
  scale_x_continuous(labels = NULL) +  
  scale_y_continuous(labels = NULL)
```



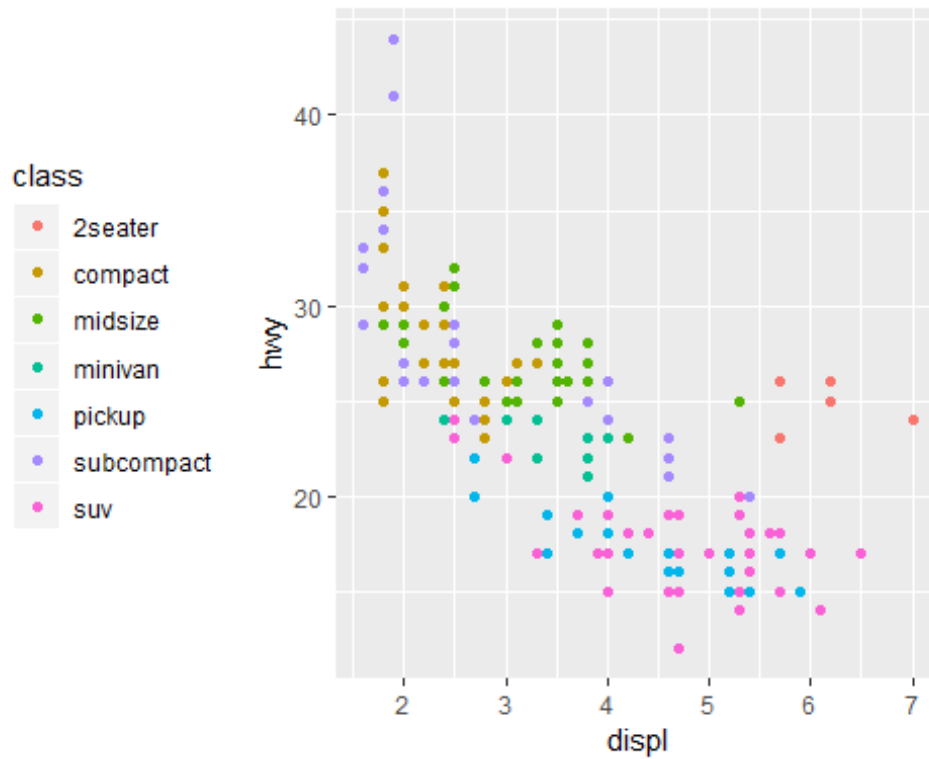
Du kan även använda `breaks` och `labels` för att kontrollera hur teckenförklaringen (`legend`) ser ut. Axlarna och teckenförklaringen kallas tillsammans för *guides*.

Teckenförklaringens (`legend`) layout

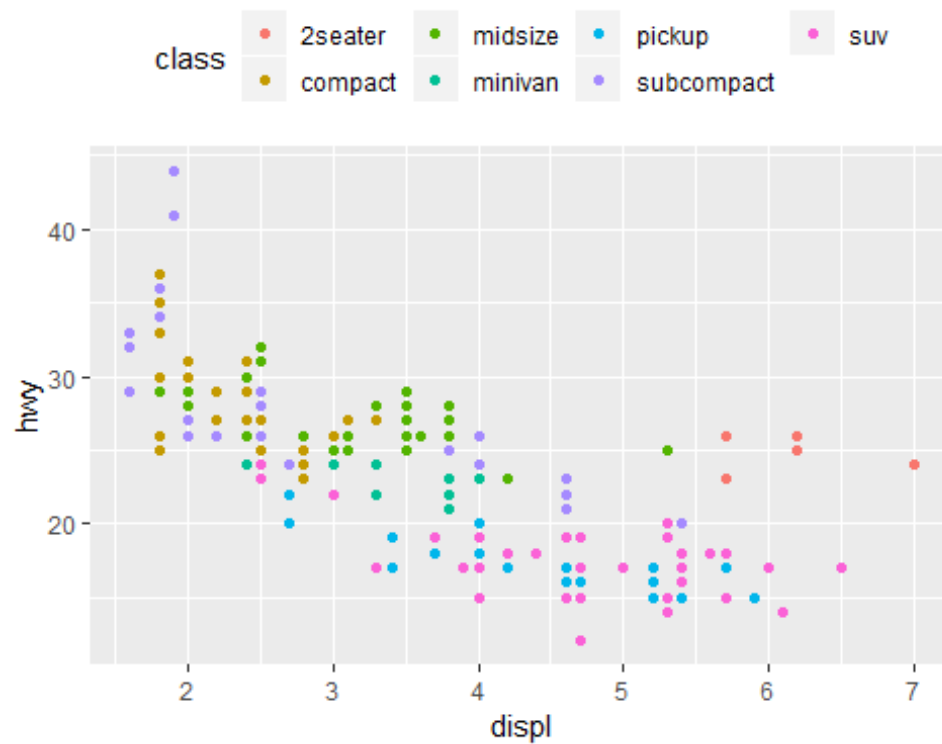
Argumenten `breaks` och `legend` används fr.a. för axlarnas utformning. De fungerar även för teckenförklaringen men det finns några andra verktyg som du sannolikt kommer att använda mer.

För att kontrollera teckenförklaringens position används ett tema, en `theme()`. Dessa styr de komponenter i grafen som inte är data. Argumentet `legend.position` styr vart teckenförklaringen placeras:

```
base <- ggplot(mpg, aes(displ, hwy)) +  
  geom_point(aes(colour = class))  
base + theme(legend.position = "left")
```

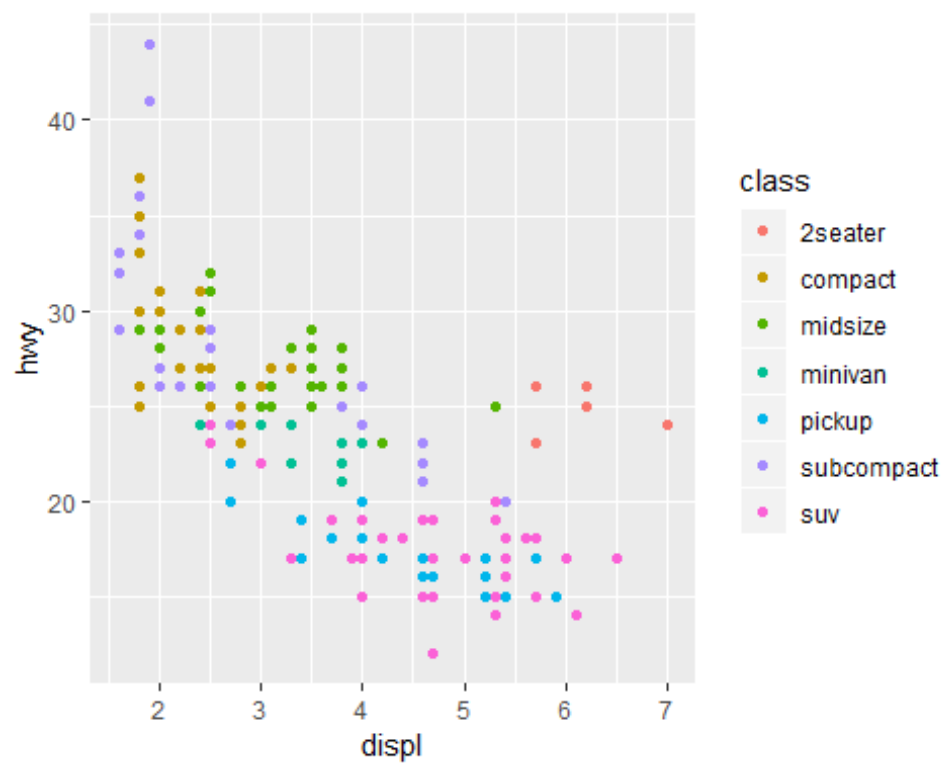
```
base + theme(legend.position = "top")
```



```
base + theme(legend.position = "bottom")
```



```
base + theme(legend.position = "right") # the default
```



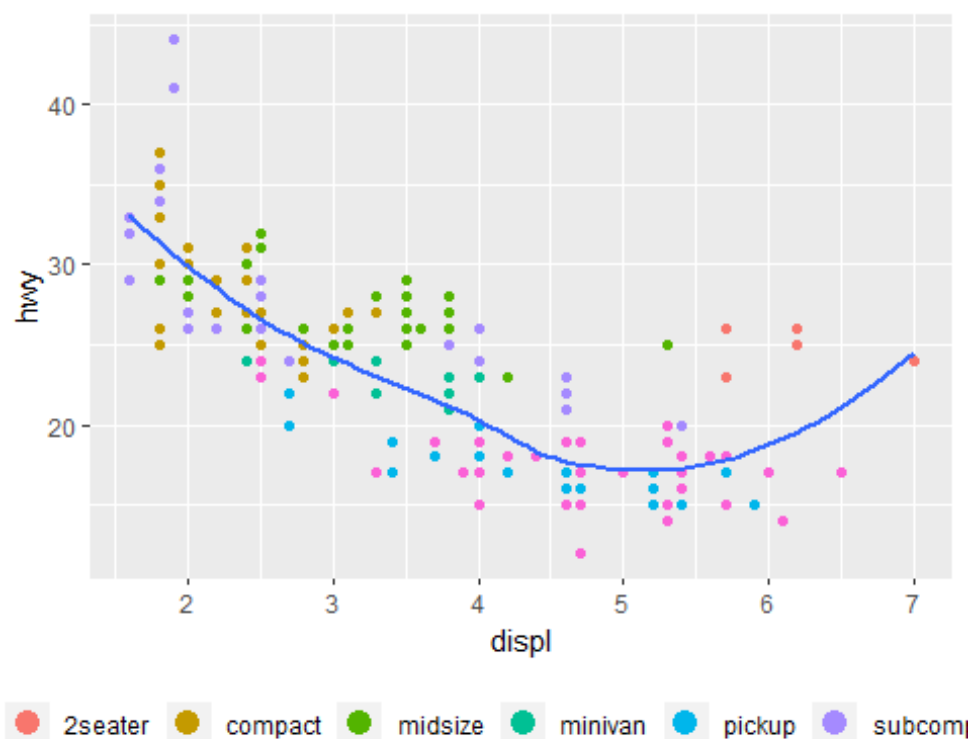
Du kan också använda `legend.position = "none"` för att inte ha någon förklaring alls.

För att styra själva utseendet används `guides()` tillsammans med `guide_legend()` eller `guide_colorbar()`. I exemplet nedan används två andra viktiga verktyg:

– `nrow` används för att kontrollera hur många rader förklaringen använder och – `override.aes` för att justera aesthetics

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_smooth(se = FALSE) +
  theme(legend.position = "bottom") +
  guides(colour = guide_legend(nrow = 1, override.aes = list(size = 4)))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



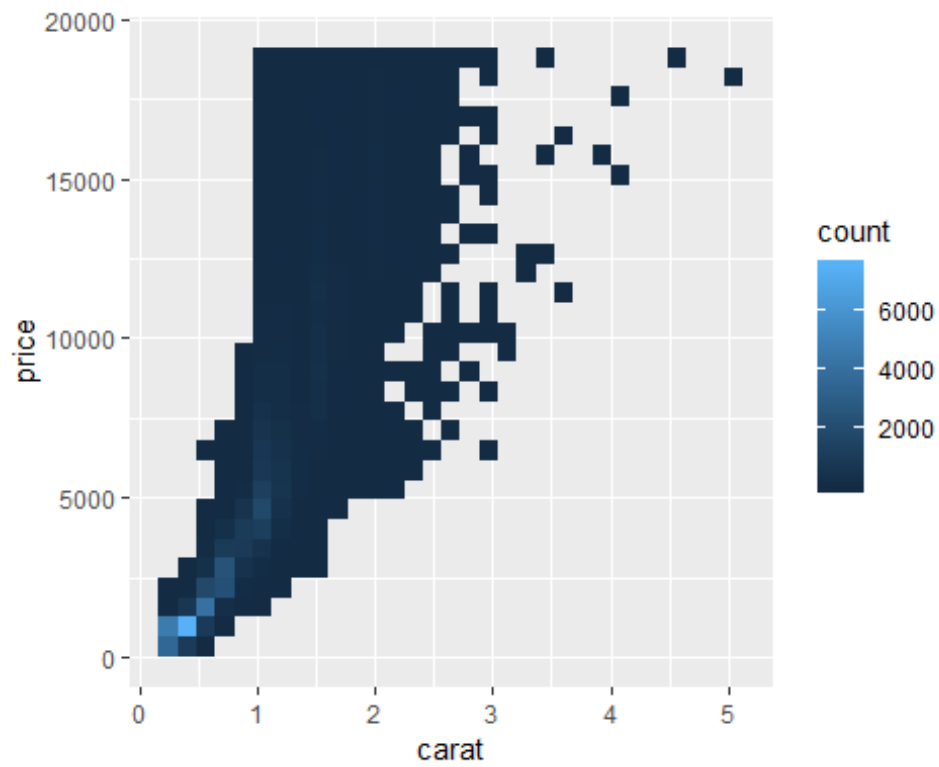
Byta ut en scale

Istället för att skruva litegrann på detaljer kan du byta en *scale* helt och hållet. Det kan vara särskilt intressant beträffande två typer av scales: scales för positionering (*continuous positioning scales*) och färgskalor (*color scales*).

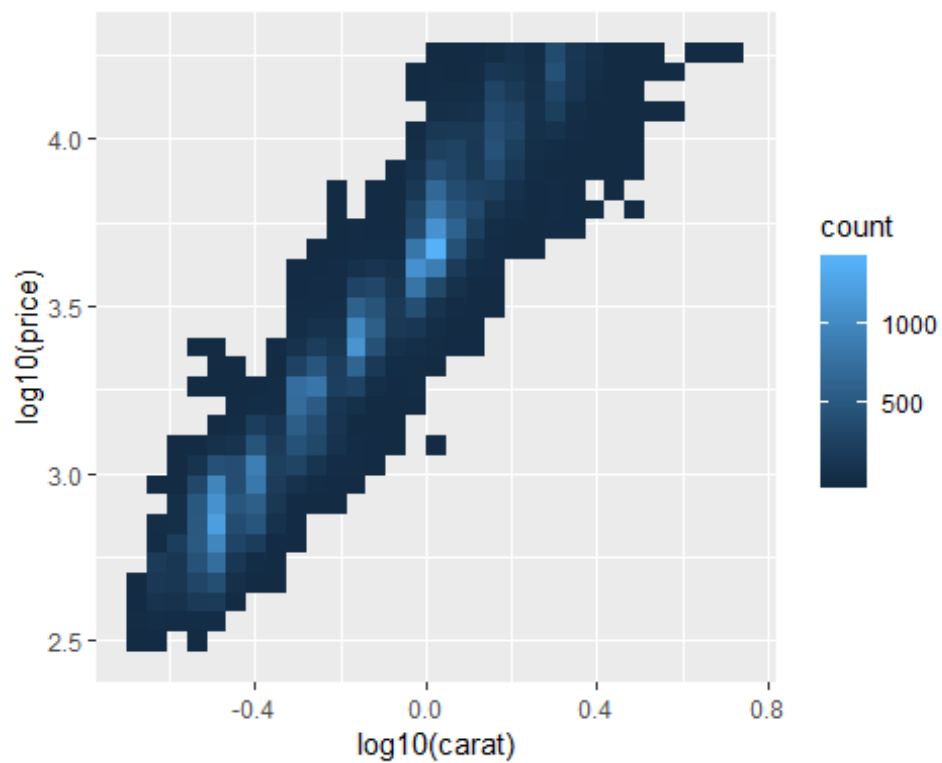
Sättet att byta ut scales fungerar i princip på samma sätt för samtliga scales, så har du lärt dig principerna för dessa två scale replacements så har du lärt dig för samtliga.

Det är ofta bra att visualisera transformeringar av variabler för att lättare se samband mellan dem, så som vi gjorde med `diamonds` tidigare:

```
ggplot(diamonds, aes(carat, price)) +  
  geom_bin2d()
```

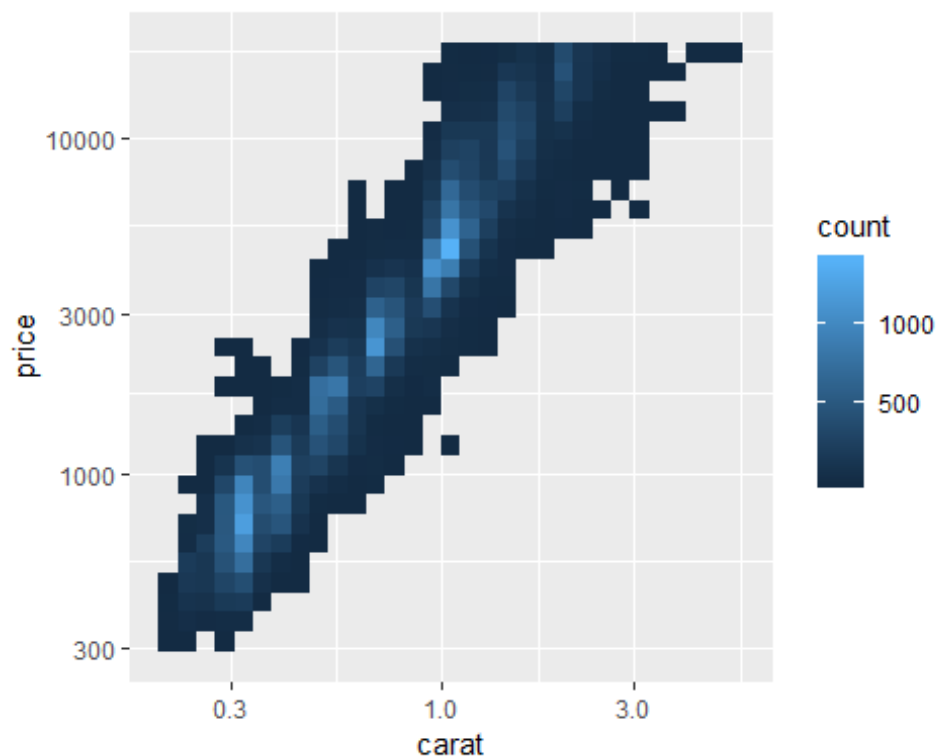


```
ggplot(diamonds, aes(log10(carat), log10(price))) +  
  geom_bin2d()
```



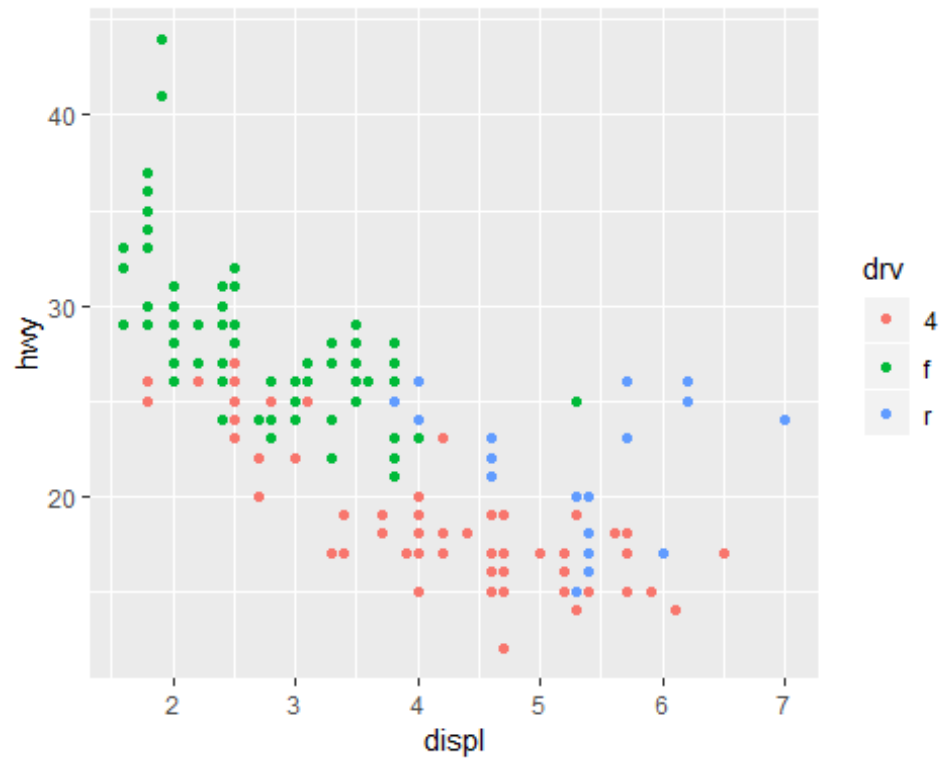
Nackdelen med detta är att axlarna nu är markerade med värdet på de transformerade variablerna vilket kan göra dem mer svårtolkade. Istället för att göra transformeringen i aesthetics-mappningen kan vi göra dem med hjälp av scales. Detta är visuellt samma sak men axlarna innehåller original-värdena:

```
ggplot(diamonds, aes(carat, price)) +  
  geom_bin2d() +  
  scale_x_log10() +  
  scale_y_log10()
```

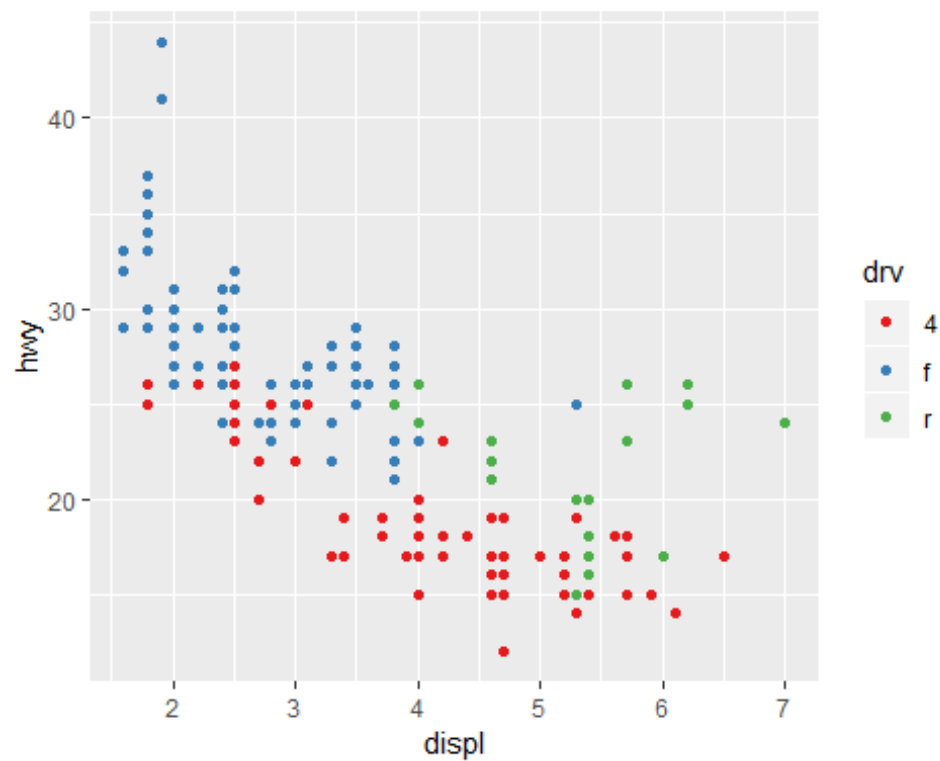


En annan scale som ofta justeras är `colour`. Standard för en kategorisk scale plockar färger jämnt fördelade över färg-hjulet. Användbara alternativ är *ColorBrewer scales* som innehåller en mängd olika möjligheter att färgsätta. Ett exempel är en uppsättning färger som är lättare för personer med färgblindhet att avläsa:

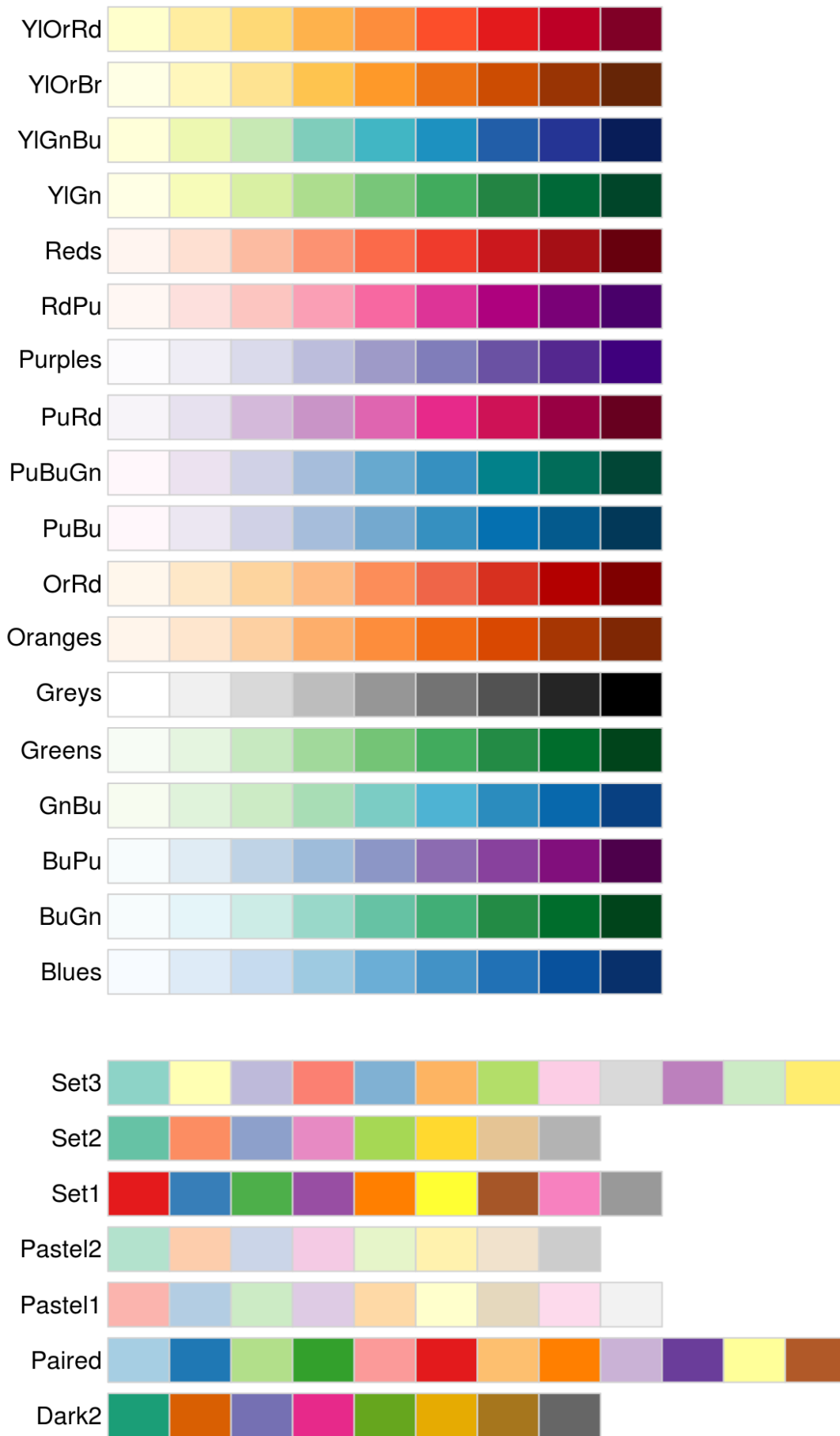
```
ggplot(mpg, aes(displ, hwy)) +  
  geom_point(aes(color = drv))
```



```
ggplot(mpg, aes(displ, hwy)) +  
  geom_point(aes(color = drv)) +  
  scale_colour_brewer(palette = "Set1")
```

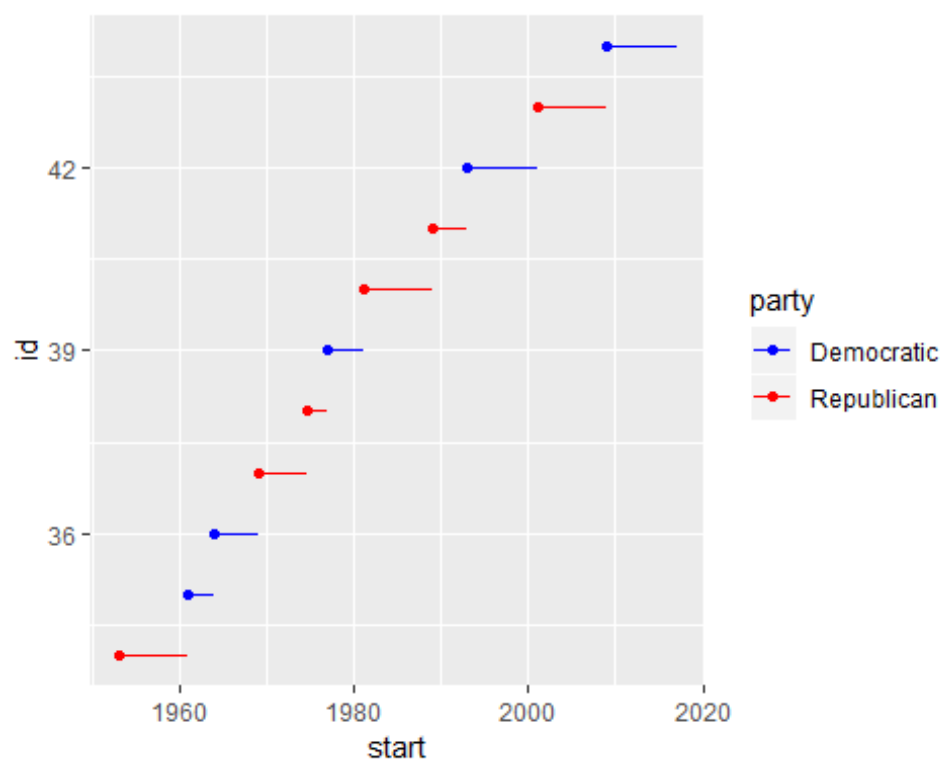


ColorBrewer scales finns dokumenterade på <http://colorbrewer2.org/> och kan användas i R via modeulen RColorBrewer. Här nedan ses alla paletter:



När det finns en förutbestämd koppling mellan kategori och färg kan du använda `scale_color_manual()`. Om vi t.ex. mappar de två största politiska partierna i USA till färg vill vi använda rött för republikanerna och blått för demokrater:

```
presidential %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(start, id, colour = party)) +
  geom_point() +
  geom_segment(aes(xend = end, yend = id)) +
  scale_colour_manual(values = c(Republican = "red", Democratic = "blue"))
```

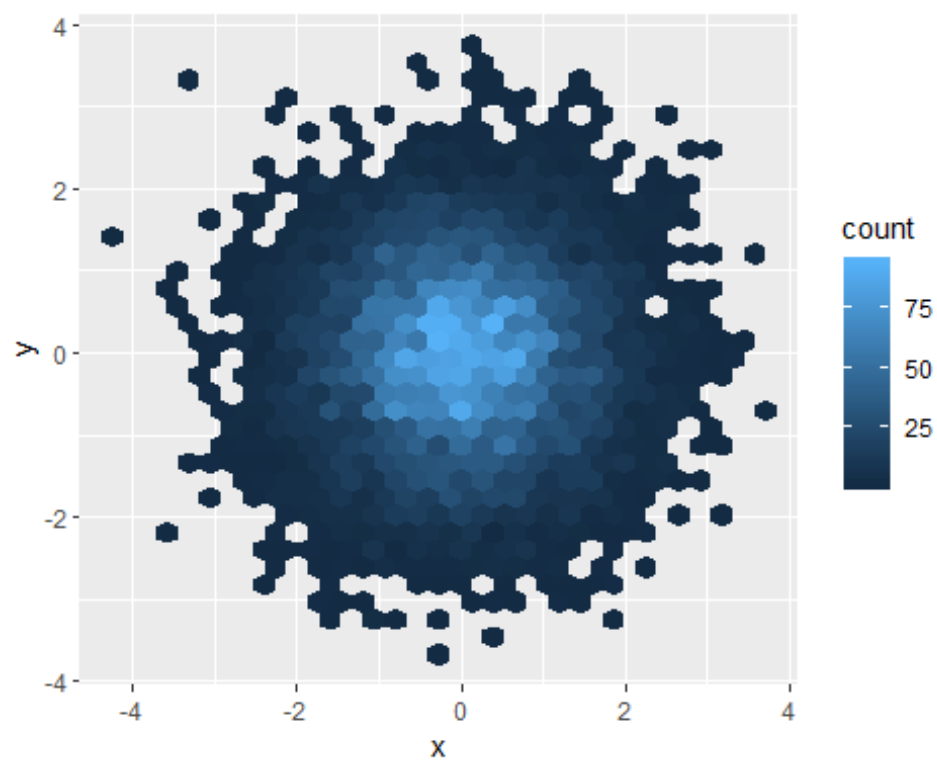


För färg-gradienter kan du använda `scale_color_gradient()` eller `scale_fill_gradient()`. Om du har två divergerande färgskalor, t.ex. för att ange positiva och negativa värden, kan du använda `scale_color_gradient2()`.

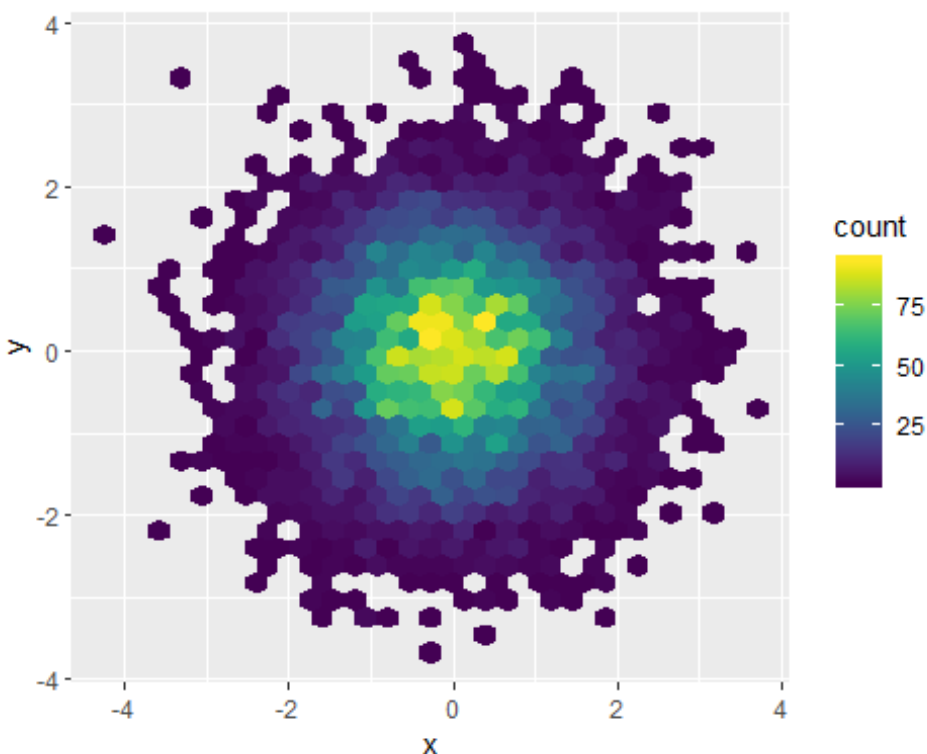
Ett annat alternativ är `scale_color_viridis()` som finns i modulen `viridis`. Det är en kontinuerlig analog till den kategoriska ColorBrewer skalorna:

```
df <- tibble(
  x = rnorm(10000),
  y = rnorm(10000)
)
ggplot(df, aes(x, y)) +
```

```
geom_hex() +  
coord_fixed()
```



```
ggplot(df, aes(x, y)) +  
  geom_hex() +  
  viridis::scale_fill_viridis() +  
  coord_fixed()
```



Notera att alla färgskalor finns i två versioner, en `_fill_` och en `_color_` som matchar *fill* resp *color* aesthetics. Och “color” kan även anges som “colour”.

Zooming

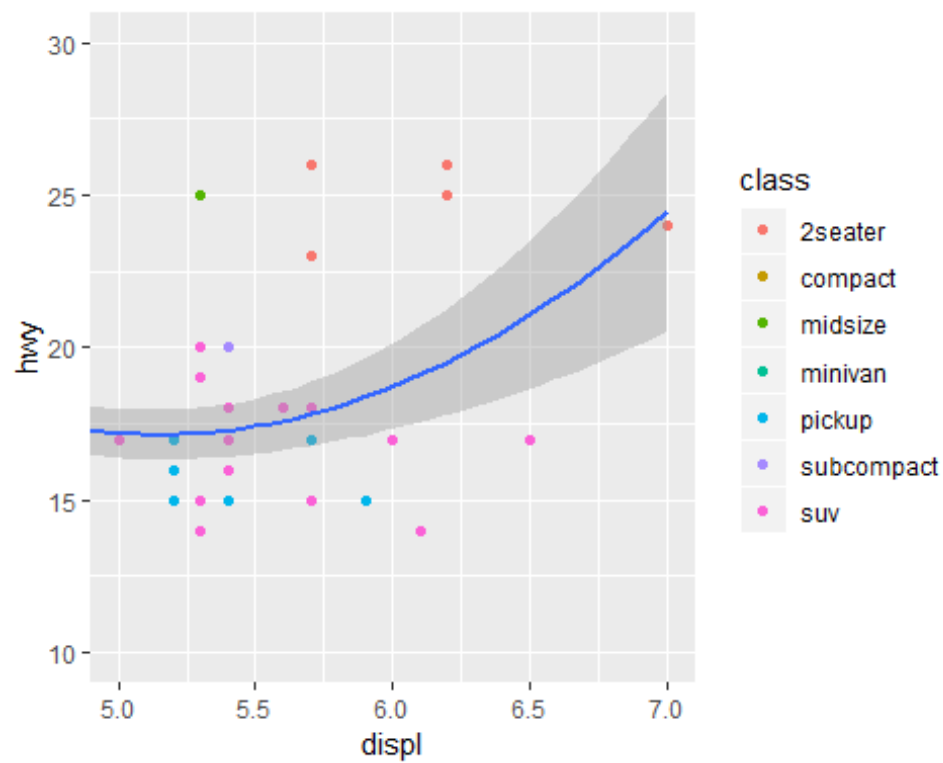
Det finns tre sätt att kontrollera gränserna för grafen :

1. Justera vilka data som ska plottas
2. Definiera gränserna för vardera axeln
3. Definiera `xlim` och `ylim` i `coord_cartesian()`

För att zooma in en begränsad del av en graf är det i regel bäst att använda `coord_cartesian()`. Jämför nedanstående:

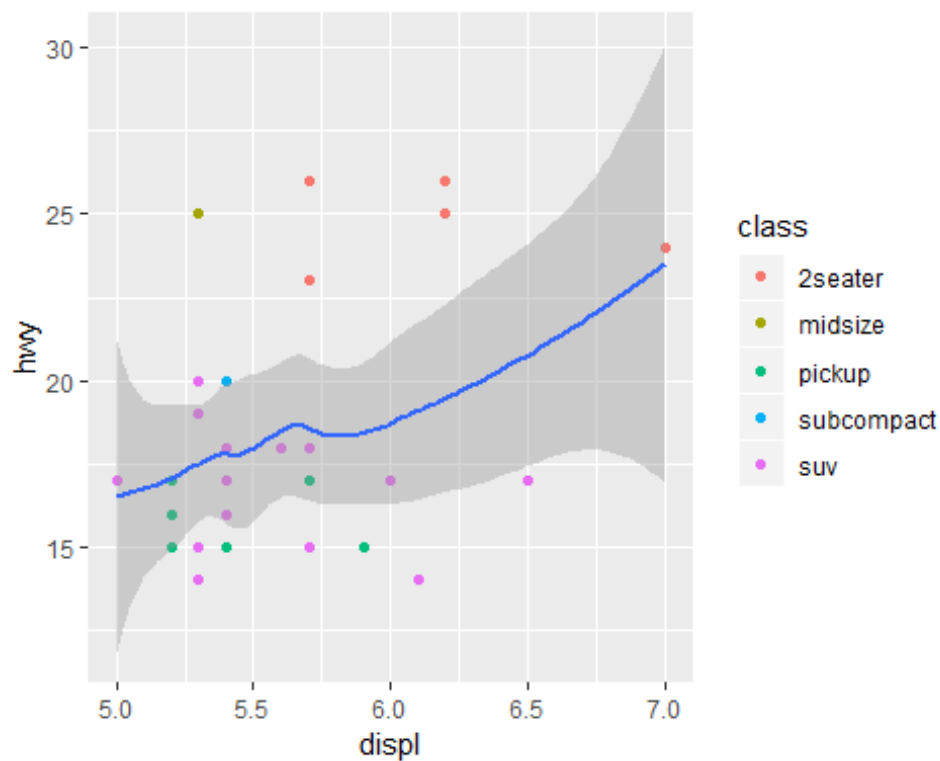
```
ggplot(mpg, mapping = aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth() +
  coord_cartesian(xlim = c(5, 7), ylim = c(10, 30))
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



```
mpg %>%
  filter(displ >= 5, displ <= 7, hwy >= 10, hwy <= 30) %>%
  ggplot(aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth()

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

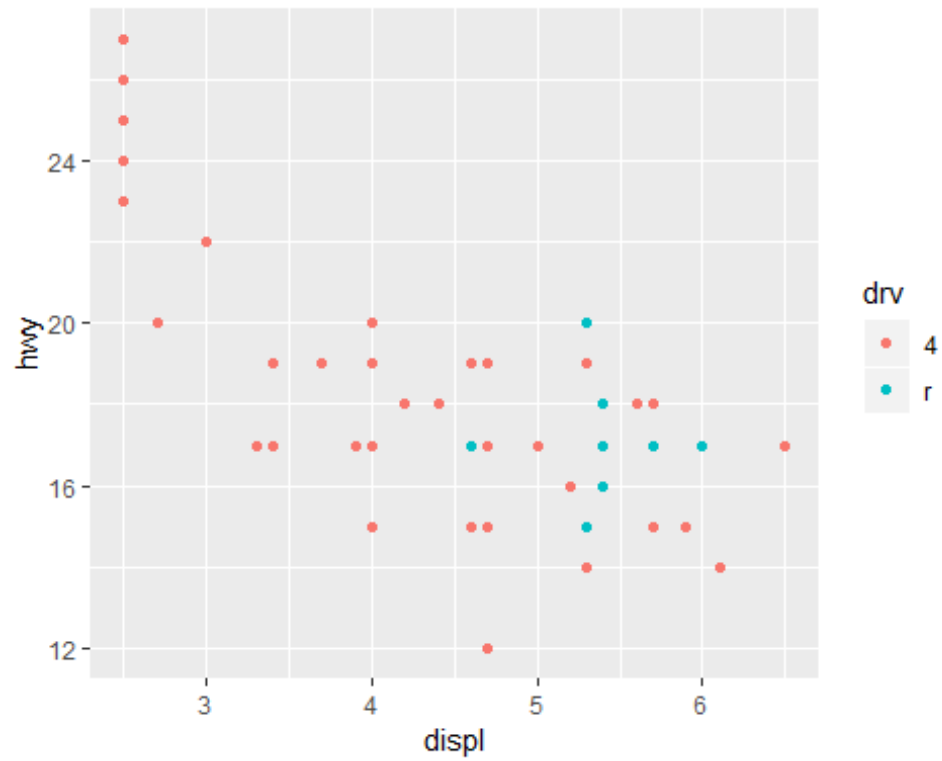


Du kan även definiera limits för de individuella skalorna. Att minska limits innebär i praktiken att göra ett urval av data, *subsetting*. Det är generellt sett mer användbart för att expandera skalorna, t.ex. för att matcha skalorna på två grafer. Om vi t.ex. vill extrahera två klasser av bilar och plotta dem separat, blir det svårt att jämföra graferna eftersom alla tre skalor (x- och y-axlar samt *color-aesthetic* kommer att ha olika omfång:

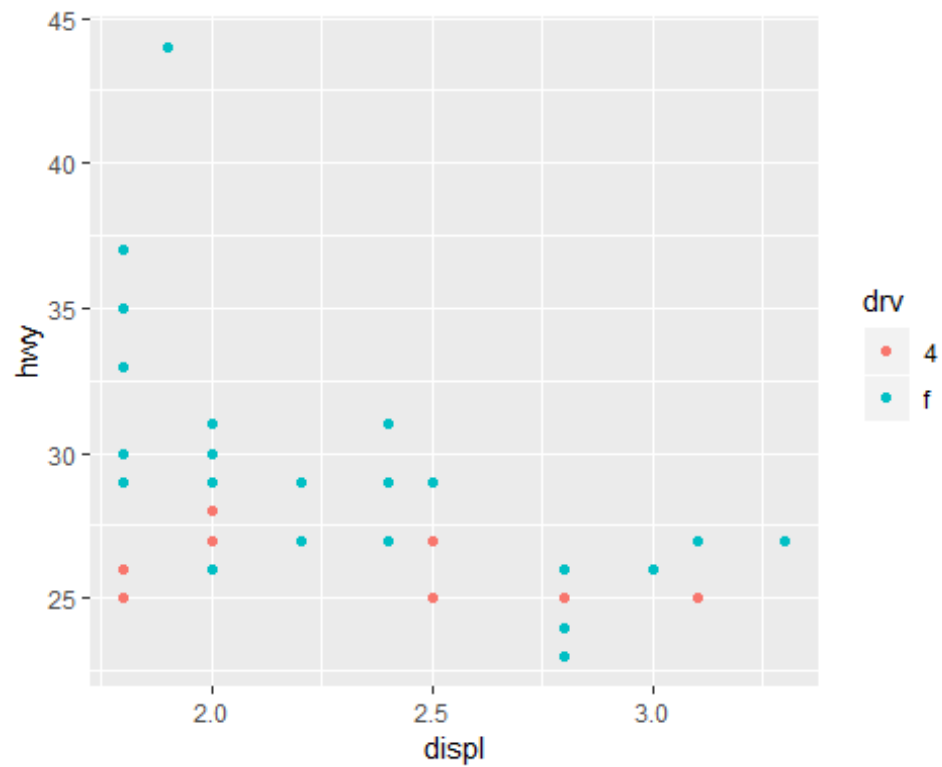
```
suv <- mpg %>% filter(class == "suv")

compact <- mpg %>% filter(class == "compact")

ggplot(suv, aes(displ, hwy, colour = drv)) +
  geom_point()
```



```
ggplot(compact, aes(displ, hwy, colour = drv)) +  
  geom_point()
```



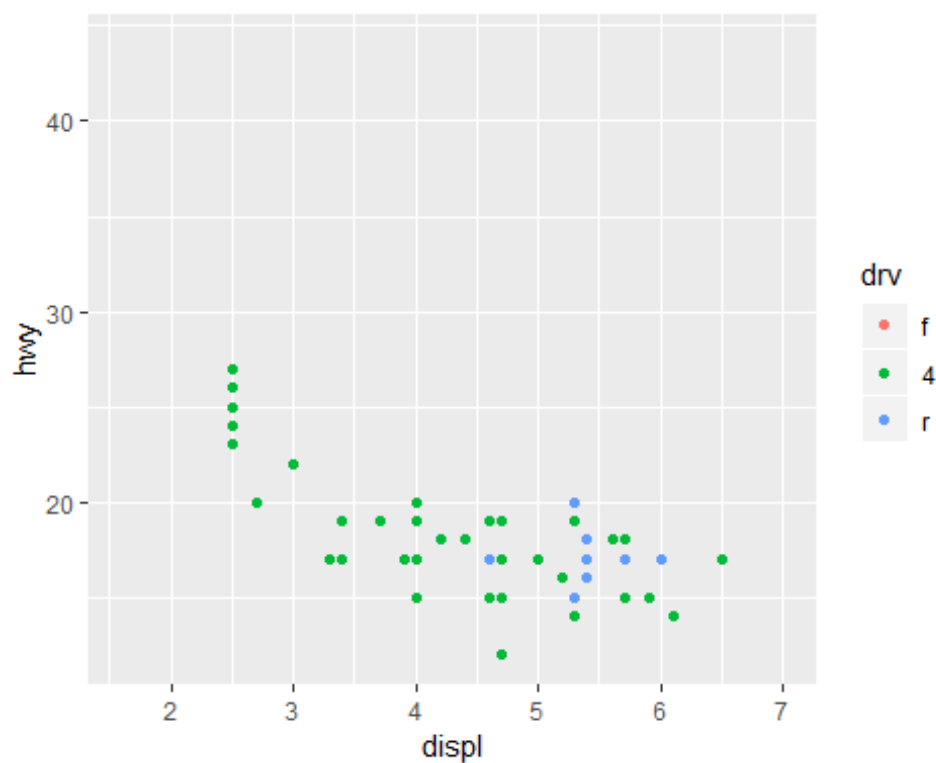
Ett sätt att runda detta problem är att använda samma skala på båda graferna genom argumentet `limits` och ange gränserna för hela datasetet:

```
x_scale <- scale_x_continuous(limits = range(mpg$displ))

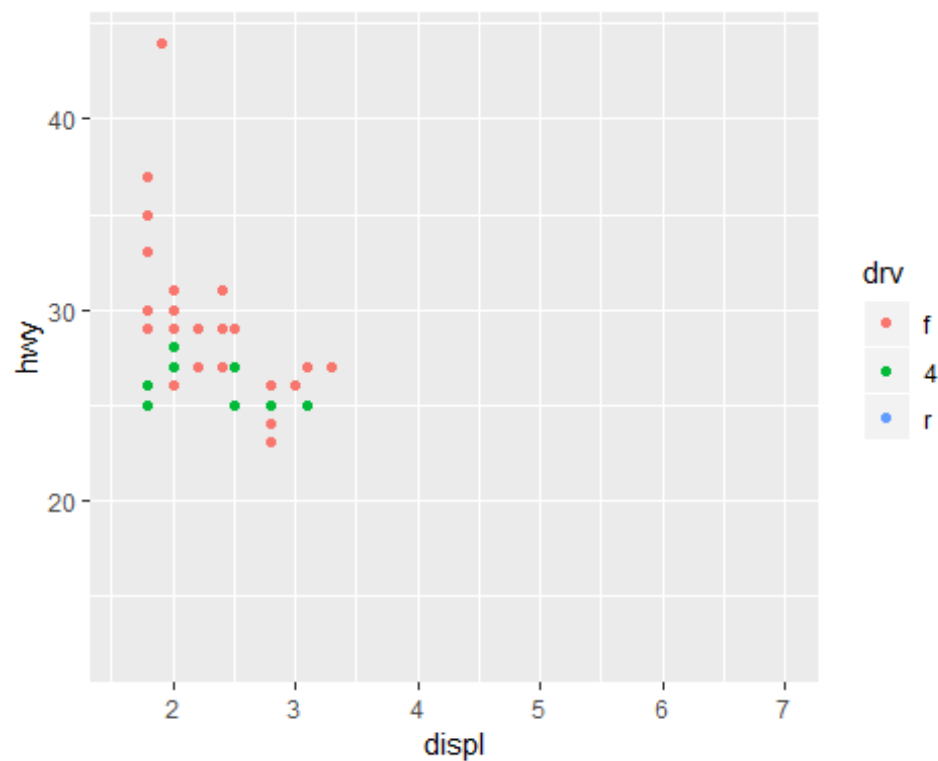
y_scale <- scale_y_continuous(limits = range(mpg$hwy))

col_scale <- scale_colour_discrete(limits = unique(mpg$drv))

ggplot(suv, aes(displ, hwy, colour = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
```



```
ggplot(compact, aes(displ, hwy, colour = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
```

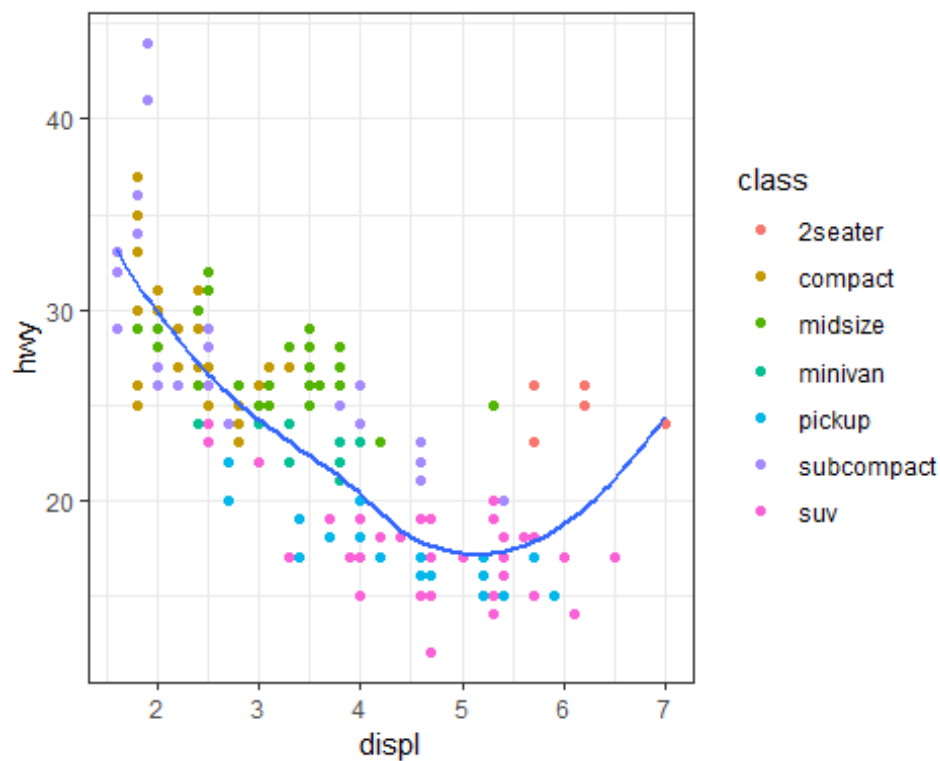
Här kunde du förstås ha kunnat använda `facets`, men denna teknik är generellt användbar, t.ex. om du vill skapa grafer på olika sidor i ett dokument.

Teman (themes)

Till sist, du kan också definiera ickedata-komponenter med hjälp av ett tema:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme_bw()
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



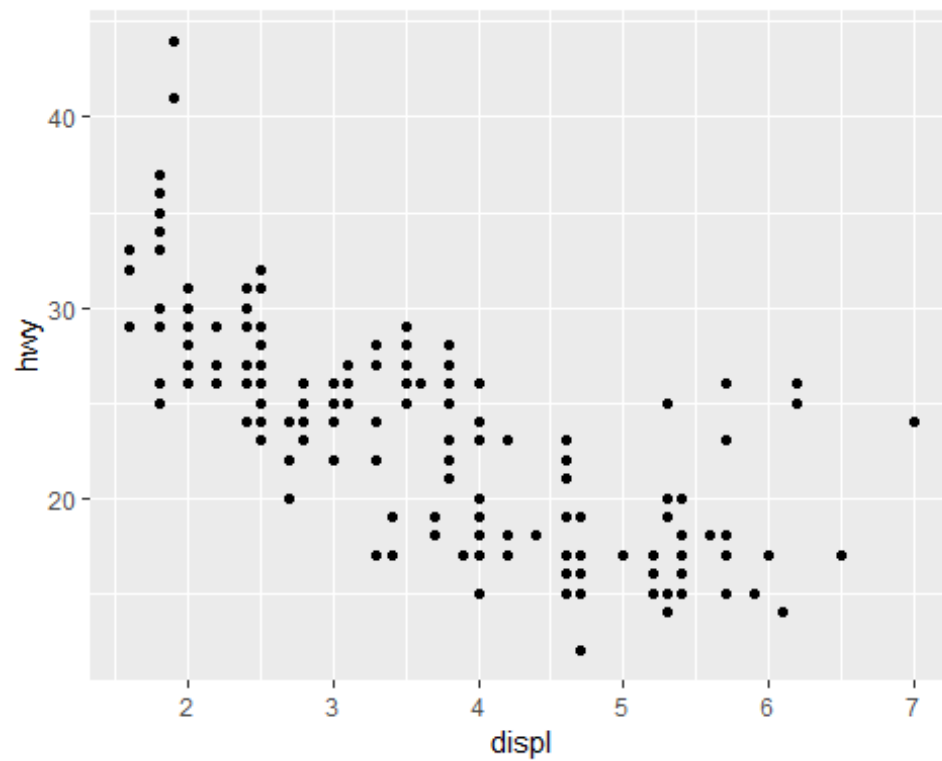
`ggplot2` inkluderar åtta olika teman (se nedan) men det finns en rad andra i tilläggs-moduler såsom `ggthemes` (<https://github.com/jrnold/ggthemes>).

Det är förstås också möjligt att justera individuella delar av ett tema, t.ex. typsnitt, typstorlek. En bra vägledning finns på `ggplot2` hemsidan <https://ggplot2.tidyverse.org/>

Spara graferna

Det finns två sätt att exportera graferna till dokument utanför R-miljön, via `ggsave()` och via `knitr`. `ggsave()` sparar den senaste grafen till disk:

```
ggplot(mpg, aes(displ, hwy)) + geom_point()
```



```
ggsave("my-plot.pdf")
```

```
## Saving 5 x 4 in image
```

Om du inte specificerar `height` och `width` kommer formatet att definieras automatiskt. Om du vill att koden ska vara reproducerbar (och det vill du) bör du ange höjd och bredd.